Объектно-ориентированная разработка на ActionScript 2.0

Основы ActionScript 2.0

СИМВОЛ®

O'REILLY®

Колин Мук

Essential ActionScript 2.0

Colin Moock

O'REILLY®

ActionScript 2.0
Основы

Колин Мук

Санкт-Петербург — Москва
2006

Колин Мук

ActionScript 2.0. Основы

Перевод Н. Шатохиной

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

Мук К.
ActionScript 2.0. Основы. – Пер. с англ. – СПб.: Символ-Плюс, 2006. – 576 с., ил.
ISBN 5-93286-087-1

Руководство посвящено описанию ActionScript 2.0 – последней версии, реализованной во Flash MX 2004 и Flash MX Professional 2004, и адресовано тем, кто работает во Flash и делает первые шаги в программировании, а также тем, кто уже знаком с ООП по таким языкам, как Java или C++, и хочет применить свои знания во Flash.

Рассмотрены основные принципы ООП, его синтаксис и применение в ActionScript 2.0; типы данных, классы, объекты, методы, свойства, наследование, композиция, интерфейсы, пути к классам, пакеты и обработка исключений. Описаны лучшие методики создания объектно-ориентированных проектов, структурирования приложений и обмена кодом с другими разработчиками, овладев которыми, вы научитесь создавать стабильные, масштабируемые и расширяемые приложения. Показано, как применять во Flash паттерны проектирования Observer, Singleton и Model-View-Controller, а также модель делегирования событий, при этом особое внимание уделяется их реализации на ActionScript 2.0. В приложении А содержится справочник по языку (типам данных, классам, объектам, глобальным свойствам и глобальным функциям). Этот материал поможет предупредить возникновение ошибок несоответствия типов при объявлении типов данных.

ISBN 5-93286-087-1
ISBN 0-596-00652-7 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 10.02.2006. Формат 70x100 1/16 . Печать офсетная.
Объем 36 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Посвящается Грэю, чудо-ребенку

Оглавление

Предисловие

Летом 2000 года сразу после окончания колледжа я пришла в компанию Macromedia на должность программиста обеспечения в команду разработчиков Flash. В первые дни моего пребывания в компании команда неутомимо трудилась над выпуском Flash 5, и все были слишком заняты, чтобы загружать меня работой, – разве что помогли мне постичь особенности корпоративной жизни Macromedia. Как же мало я понимала, изучая сложную C++-архитектуру средств разработки Flash, что ActionScript тоже начинал свой путь в индустрии веб-разработки. Flash 5 стал поворотной вехой в истории развития средства разработки Flash: он превратил ActionScript из интерфейса взаимодействия типа «указал и щелкнул» в полноценный язык сценариев, базирующийся на стандарте ECMAScript, с настоящим текстовым редактором. Я приступила к работе как раз в тот момент, когда команда Flash вкладывала реальную мощь сценариев в руки Flash-разработчиков. В следующих двух версиях Flash эта работа была продолжена: сначала я принимала участие в создании отладчика ActionScript во Flash MX и затем в разработке компилятора ActionScript 2.0. Последние несколько лет для меня неразрывно связаны с этим языком, и он сыграл немаловажную роль в моем профессиональном росте, так же как и я способствовала его развитию.

Вначале мое отношение к ActionScript было таким же, как и у многих обычных разработчиков, когда они переходят на какой-то язык программирования. Мне нравилась его гибкость, но расстраивала его ограниченность. Я была счастлива воплотить в жизнь такие возможности как, например, отладчик, потому что он помог Flash реализовать мои собственные ожидания от среды программирования. Мне нравилось шаг за шагом закрывать пробелы в возможностях Flash. Во Flash MX мы достигли больших успехов, существенно улучшив редактор кода и обеспечив пользователю возможность отладки ActionScript. Однако ActionScript 1.0 имел одно неприятное ограничение, которое мы не устранили во Flash MX: возможность написать код, использующий методы ООП, существовала, но сделать это было крайне сложно, и этот код плохо сочетался с реалиями Flash, такими как библиотечные символы.

Создав Flash MX 2004 и ActionScript 2.0, мы прошли еще один основной этап эволюции ActionScript. ActionScript 2.0 предлагает более развитый синтаксис для конструкций ООП, которые ActionScript все-

гда поддерживал. ActionScript 2.0 проще в изучении, чем его предшественники, и ближе к другим промышленным языкам программирования, таким как Java и C#. Он предоставляет разработчикам инфраструктуру, необходимую для создания и обслуживания больших, сложных приложений. Кроме того, наша реализация требовала внесения минимальных изменений во Flash Player, это означает, что ActionScript 2.0 можно экспортировать во Flash Player 6, который на момент выхода Flash MX 2004 применялся уже практически повсеместно.

За короткое время существования ActionScript разработчики успели оценить его исключительную мощь. Flash практически не налагает ограничений на доступ разработчиков к иерархии и объектной модели *MovieClip*, позволяя им делать все, что угодно, везде, где угодно. Эта гибкость побудила наших пользователей к творчеству, позволив им освоить ActionScript и экспериментировать с ним. Однако отсутствие структуры в ActionScript 1.0 усложняет масштабируемость приложений, в результате чего проекты становятся громоздкими и командам трудно обслуживать и организовывать их. Можно было запросто написать плохой код, не говоря уже о том, чтобы разместить код там, где его практически невозможно было найти, если ты плохо знаком с проектом. Создатели ActionScript 2.0 постарались учесть эти просчеты, поддерживая понятную структуру, которой могут придерживаться все разработчики. Более того, компилятор ActionScript 2.0 обеспечивает разработчикам обратную связь по ошибкам, которые в противном случае не были бы обнаружены вплоть до времени выполнения. ActionScript по-прежнему предоставляет широкий и уникальный контроль над графическими элементами. Мы стремимся доказать, что ActionScript является мощным развивающимся языком, не задевая интересов бывалых пользователей языков сценариев.

ActionScript 2.0 также был основой для некоторых замечательных элементов Flash MX 2004.

Все нижеприведенные элементы написаны на ActionScript 2.0:

- Второе поколение компонентов (т. е. компоненты v2)
- Новый инструмент Screens (Экраны), включающий Slides (Слайды) и Forms (Формы) (доступные только во Flash MX Professional 2004)
- Усовершенствованные возможности интеграции данных
- Поддержка многоязыковых ресурсов, предлагаемая панелью Strings (Строки)

Создание важных крупномасштабных функциональных возможностей с помощью ActionScript 2.0 предоставило ценную тестовую и контрольную информацию при работе над компилятором и подсказало многие конструкторские решения. Что еще важнее, эти функциональные возможности обеспечили разработчикам Flash полные рабочие примеры ActionScript 2.0 в действии (см. папку *Macromedia\Flash MX 2004\en\First Run\Classes*). Похожим образом преимущества Ac-

tionScript 2.0 отчетливо проявляются в этих функциях, состоящих из классов, организованных в иерархию класса *mx.**. Кроме того, определить принадлежность кода к компоненту теперь стало просто как никогда, поскольку ActionScript 2.0 сделал возможным устранять ненадежные пережитки прошлого ActionScript, такие как указание транслятору `#initclip` (директива компилятора).

ActionScript начинался с нескольких команд сценариев, вставляемых по щелчку мыши. Пять лет спустя он превратился в полнофункциональный объектно-ориентированный язык, позволяющий создавать большие и сложные приложения. Его синтаксис прост, строен, легок для восприятия и понятен даже новичку. Принимая участие в создании двух версий средств разработки Flash, я шаг за шагом все глубже и глубже узнавала ActionScript и теперь горжусь тем, что в его успехе есть и моя заслуга. Предыдущая книга Колина Мука, «ActionScript for Flash MX: The Definitive Guide», была очень ценной для меня, хотя я уже работала над новой версией ActionScript. Это единственная книга, которую вы найдете на столе каждого специалиста команды разработки Flash. Выхода новой книги, «Essential ActionScript2.0», уже с нетерпением ожидают многие наши сотрудники. И не зря. В ней Мук в который раз продемонстрировал свой глубокий и доступный стиль при рассмотрении сложных тем, излагая не только синтаксис ActionScript 2.0, но также теорию и принципы ООП. Он всесторонне исследовал взаимоотношения между ActionScript 2.0, его предшественниками и другими языками и подробно проиллюстрировал их отличия. Он досконально знает Flash и ActionScript, и это очевидно. Данная книга без сомнения необходима всем, кто желает освоить язык ActionScript 2.0.

– Ребекка Сан (Rebecca Sun),
ведущий разработчик программного обеспечения,
команда разработки Flash компании Macromedia,
март 2004

Введение

В сентябре 2003 года компания Macromedia выпустила в свет Flash MX 2004 и вместе с ним ActionScript 2.0 – существенно расширенную версию языка программирования во Flash.

Для создания Flash-приложений ActionScript 2.0 предлагает формальный синтаксис и методологию *объектно-ориентированного программирования* (ООП). По сравнению с традиционными методиками разработки на базе временной диаграммы основанные на ООП методы разработки ActionScript 2.0 обычно делают приложения:

- Более дружественными к разработчику
- Более стабильными и свободными от ошибок
- Более удобными для повторного использования в проектах
- Более удобными для поддержки, внесения изменений и расширения
- Более пригодными для тестирования
- Более пригодными для совместной разработки двумя или более разработчиками

Это просто исключительные качества. Настолько исключительные, что они превращают эту книгу в своего рода библию. Она призвана привлечь вас в ряды ярых сторонников ActionScript 2.0.

Книга зовет

Она призывает вас избрать ООП инструментом своей ежедневной работы с Flash. Она призывает вас собрать урожай преимуществ ООП – одной из самых важных революций в истории программирования. Она призывает вас постичь ActionScript 2.0 до самых глубин. И она ни перед чем не остановится, чтобы ее призывы достигли цели.

Вот ее план...

В главах части I «Язык ActionScript 2.0» представлены основные идеи, синтаксис и применение ООП. Даже если вы никогда ранее не сталкивались с объектно-ориентированным программированием, прочитав часть I, вы поймете, что это такое и как его использовать. В главе 1 представлен краткий обзор ActionScript 2.0. Глава 2 излагает основы ООП и помогает вам принять решение о том, что подходит для вашего проекта. В главах 3–10 подробно рассмотрены классы, объекты, мето-

ды, свойства, наследование, композиция, интерфейсы, пакеты и огромное количество других понятий ООП. Тем, кто уже знаком с ООП, например, работали с Java или другими объектно-ориентированными языками, эта книга поможет упорядочить имеющийся опыт. Здесь ООП во Flash сравнивается с тем, что уже вам известно. По ходу дела книга вводит ООП в обычную для вас практику через упражнения, демонстрирующие реальное применение ООП во Flash.

В части II «Разработка приложений» вы научитесь создавать целые приложения с помощью ActionScript 2.0. В главе 11 приводятся лучшие примеры планирования и разработки объектно-ориентированного проекта. Из глав 12–13 вы узнаете, как компоненты пользовательского интерфейса и клипы вписываются в хорошо структурированное приложение Flash. В главе 14 показано, как распределить совместно используемый код между разработчиками. Все это поможет создавать более масштабируемые, гибкие, стабильные приложения. И все это входит в план книги.

И наконец, часть III «Примеры паттернов проектирования на ActionScript 2.0» посвящена исследованию различных подходов к разным ситуациям при программировании. Вы увидите, как применять испытанные и широко распространенные стратегии ООП – известные как *паттерны проектирования* (*design patterns*) – во Flash. Паттерны проектирования в части III охватывают два ключевых момента Flash-разработки: передачу событий и управление пользовательским интерфейсом. После введения в шаблоны проектирования в главе 15 в главах 16–19 мы рассмотрим четыре основных паттерна. Поработав с паттернами, представленными в части III, вы поверите в свои силы и перейдете к рассмотрению разнообразных паттернов, представленных в сети или в другой литературе. И получите навык для перехода к другим общепринятым технологиям ООП. Как видите, книга понимает, что не будет с вами вечно. Она должна научить вас находить собственные решения.

Не имеет значения, известно ли вам, что такое «класс», «наследование», «метод», «прототип» или «свойство». Если вы понятия не имеете о том, что такое ООП или почему оно так широко распространено, эта книга рада приветствовать вас. А тех, кто уже имеет опыт ООП-разработок, эта книга хочет сделать лучше. Она хочет снабдить вас исчерпывающим справочным материалом и примерами, необходимыми для максимального повышения производительности в ActionScript 2.0.

Эта книга твердо стремится сделать вас опытным объектно-ориентированным разработчиком. И я уверен, что эта задача будет выполнена.

Чего нет в этой книге

Данная книга посвящена основам ActionScript 2.0 и ООП и поэтому не охватывает все темы, которые могут быть связаны с ActionScript. В частности, здесь нет пространных описаний сопутствующих технологий,

таких как Flash Remoting или Flash Communication Server, как нет здесь и похожего на словарь справочника по языку, какой был в «ActionScript for Flash MX: The Definitive Guide» (O'Reilly).[1] Здесь описываются «родные» функции, свойства, классы и объекты Flash Player, поэтому, прочитав эту книгу, вы научитесь применять эти классы и объекты и встраивать их в свои собственные структуры. Встроенная библиотека классов Flash Player во Flash Player 7 была лишь дополнена, поэтому «ActionScript for Flash MX: The Definitive Guide» остается справочником, актуальным даже для разработчиков, программирующих на ActionScript 2.0. Она представляет собой замечательное дополнение к данной книге.

Здесь не рассматривается инструмент Screens (включая Slides и Forms), поддерживаемый только во Flash MX Professional 2004 и применяемый для визуальной разработки пользовательских интерфейсов (в традициях MS Visual Basic) и для создания слайдовых презентаций (в традициях MS PowerPoint). Несмотря на это, постигнув основы, изложенные в этой книге, вы, без сомнения, будете готовы рассмотреть Screens самостоятельно.

Эта книга также не является руководством по основам программирования, таким как условные операторы (операторы *if*), циклы, переменные, массивы и функции. Основы программирования во Flash рассмотрены в книге «ActionScript for Flash MX: The Definitive Guide».

И наконец, эта книга не обучает использованию среды разработки Flash, за исключением того, что касается разработки приложений на ActionScript 2.0. Для получения справки по среде разработки, например о создании графических объектов или анимации, стоит обратиться к документации или любой хорошей книге, например книге Роберта Хокмана (Robert Hoekman) «Flash Out of the Box», O'Reilly, 2004.

Кому следует и кому не следует читать эту книгу

Вам следует прочитать эту книгу, если вы:

- Программируете на ActionScript 1.0 или JavaScript лучше, чем новичок, но не достигли профессионального уровня, знаете, что такое переменные, циклы, условные операторы, функции, массивы и другие основные понятия программирования.
- Опытный программист на ActionScript 1.0 или ActionScript 2.0, желающий получить достоверную информацию о лучших практических методах ООП в ActionScript 2.0, включая подробный синтаксис и информацию о вариантах использования, индивидуальные особенности языка и примеры структур приложений.

[1] К. Мук «ActionScript для Flash MX. Подробное руководство». – Пер. с англ. – СПб.: Символ-Плюс, 2004.

- Flash-дизайнер, немного занимающийся программированием и интересующийся разработкой приложений.
- Программист, переходящий к разработке в среде Flash с другого языка программирования, например Java, C++, Perl, JavaScript или ASP. (Будьте готовы изучать среду разработки Flash по другим упомянутым ранее источникам. Вам также следует прочитать главу 13 «Клипы» и книгу «ActionScript for Flash MX: The Definitive Guide», которую можно найти в сети по адресу *http://moock.org/asdg/samples*.)

Вам не следует читать эту книгу, если вы являетесь дизайнером/мультипликатором во Flash и имеете небольшой или вообще не имеете опыта программирования. (Лучше начните изучение ActionScript с книги «ActionScript for Flash MX: The Definitive Guide».)

ActionScript 2.0 и ActionScript 1.0

Более подробно ActionScript 2.0 представлен в главе 1, здесь я лишь кратко сориентирую разработчиков, пишущих на ActionScript 1.0.

ActionScript 1.0 и ActionScript 2.0 имеют аналогичный основной синтаксис. Такие основные понятия, как условные инструкции, циклы, операторы и другие не объектно-ориентированные аспекты ActionScript 1.0, могут применяться в ActionScript 2.0 и по-прежнему остаются официальной частью языка. Кроме того, создание объектов, доступ к свойствам и вызов методов в ActionScript 1.0 и ActionScript 2.0 имеют аналогичный синтаксис. Таким образом, вообще говоря, ActionScript 2.0 привычен для разработчиков на ActionScript 1.0. Основное отличие между двумя версиями этого языка программирования заключается в объектно-ориентированном синтаксисе и поддержке средств объектно-ориентированной разработки.

Объектно-ориентированный синтаксис в ActionScript 1.0 непрозрачен и практически не имеет поддержки со стороны средств разработки (например, нет сообщений компилятора, нет структуры файла класса, нет контроля типов, слаба связь между кодом и ресурсами фильма и т. д.). В ActionScript 1.0 объектно-ориентированное программирование было трудным и малопонятным занятием, а в ActionScript 2.0 это дело совершенно естественное. В ActionScript 2.0 средства ООП реализованы в более традиционном ключе, что позволяет применять в нем навыки, полученные при работе с другими объектно-ориентированными языками, и наоборот, переносить навыки работы с ActionScript 2.0 в другие языки.

Те, кто программирует на ActionScript 1.0 и уже применял методы ООП, получат удовольствие от работы с ActionScript 2.0. Тем же, кто работал с ActionScript 1.0 и не применял ООП, не придется учить ООП в ActionScript 1.0, перед тем как обучаться ему в ActionScript 2.0. Вам представилась идеальная возможность изучить и освоить эту важную методику. ООП позволяет повысить производительность, упростить

управление проектами и улучшить качество кода и возможность его повторного использования.

В этой книге не уделяется много времени вопросу обновления кода при переходе с ActionScript 1.0 на ActionScript 2.0, но после ее прочтения у вас не должно быть проблем с этим. Главная задача состоит в том, чтобы сформировать у читателя твердое фундаментальное понимание ActionScript 2.0, и я не хотел слишком отклоняться от этого курса, рассуждая об устаревшем коде ActionScript 1.0. Это означает, что вам придется следить за многочисленными примечаниями, которые выглядят вот так:

В таких примечаниях проводится прямое сравнение методов ActionScript 1.0 с аналогичными методами ActionScript 2.0, так что вы можете увидеть различия между старым и новым усовершенствованным вариантом реализации той или иной задачи.

Наконец, давайте определимся с тем, что я имею в виду под «сравнением программирования на ActionScript 2.0 и ActionScript 1.0». Если вы просто создаете код во временной диаграмме и не используете классы ActionScript 2.0, статические типы данных или другие возможности ООП, то действительно, довольно сложно понять, на чем он написан – на «ActionScript 2.0» или на «ActionScript 1.0». Без особенностей ООП код ActionScript 2.0 совершенно неотличим от кода ActionScript 1.0. Поэтому, когда я говорю «мы научимся программировать на ActionScript 2.0», то я, без сомнения, подразумеваю, что вы займетесь созданием содержательного ООП-приложения, в котором разработаете один или более классов. Например, рассмотрим интерактивную форму, которая просто отправляет сообщения электронной почты. Ее можно было бы полностью реализовать во временной диаграмме Flash, применяя лишь переменные и функции. Если это все, что вы хотите делать в своих приложениях, тогда, откровенно говоря, эта книга вам не нужна. Но она (книга) способна расширить ваш кругозор и превратить вас в высококвалифицированного объектно-ориентированного разработчика, которому под силу справиться и с намного большими проектами. Итак, когда я говорю «программирование на ActionScript 2.0», я имею в виду разработку объектно-ориентированных приложений на ActionScript 2.0. При этом ударение делается на ООП, а не на ActionScript 2.0, поскольку, по существу, ActionScript 2.0 – лишь средство. Вы можете задать вопрос: «Так эта книга о синтаксисе ActionScript 2.0, объектно-ориентированной методологии или объектно-ориентированном программировании?» Ответ: «Обо всем сразу».

Об особенностях ActionScript 2.0 и ActionScript 1.0 применительно к Flash Player 6 и Flash Player 7 рассказано в главе 1.

Расшифровка версий Flash

Представляя семейство продуктов Studio MX, включающее Flash MX, компания Macromedia отказалась от стандартной системы нумерации

версий для своей среды разработки Flash. В названия продуктов, выходящих после Flash MX, Macromedia включала год их выпуска (продукты, выходящие после сентября, маркировались следующим годом). В 2004 году Macromedia разделила Flash на две версии: Flash MX 2004 и Flash MX Professional 2004, как показано в табл. П.1. Для профессиональной версии характерны следующие основные особенности:

- Инструмент Screens (разработка содержимого на базе форм и слайдов)
- Дополнительные средства создания и редактирования видеоинформации
- Средства управления проектами и ресурсами
- Внешний редактор сценариев
- Привязка данных (связывание компонентов с источниками данных, поставляемыми посредством веб-сервисов, XML или наборов записей)
- Усовершенствованные компоненты (однако компоненты Flash MX Professional 2004 замечательно работают во Flash MX 2004)
- Средства разработки для мобильных устройств

Методики, излагаемые в этой книге, могут применяться как во Flash MX 2004, так и во Flash MX Professional 2004, хотя я всегда отмечаю те редкие случаи, когда между двумя версиями есть разница с точки зрения разработки на ActionScript 2.0. В отличие от среды разработки Flash, версии Flash Player по-прежнему нумеруются, и на момент выхода книги последней является Flash Player 7. В табл. П.1 приведены названия версий Flash, применяемые в этой книге.

Таблица П.1. Названия версий Flash, применяемые в этой книге

Название	Значение
Flash MX	Версия среды разработки Flash, выпущенная одновременно с Flash Player 6.
Flash MX 2004	Стандартная версия среды разработки Flash, выпущенная одновременно с Flash Player 7. В общем смысле выражение «Flash MX 2004» применяется и к стандартному изданию (Flash MX 2004), и к версии для профессионалов (Flash MX Professional 2004) этого программного обеспечения. При обсуждении особенностей, характерных для профессиональной версии, эти ограничения в данной книге обозначены явно.
Flash MX Professional 2004	Версия среды разработки Flash для профессионалов, выпущенная одновременно с Flash Player 7. Профессиональное издание включает некоторые возможности, отсутствующие в стандартной версии (см. предыдущую графу таблицы). Для работы с этой книгой и использования ActionScript 2.0 профессиональная версия не является обязательной.

Таблица П.1 (продолжение)

Название	Значение
Flash Player 7	Flash Player, версия 7. Flash Player – это плагин (подключаемый модуль) для основных веб-броузеров Windows и Macintosh. На момент выхода данной книги для Linux был доступен Flash Player 6, а не Flash Player 7. Плагин существует в двух версиях, как элемент управления ActiveX и как версия для Netscape, но я обобщенно называю их «Flash Player 7».
Flash Player *x*.0.*y*.0	Flash Player, в частности версия, определяемая главным номером версии *x* и номером основной сборки *y*, как для Flash Player 7.0.19.0. Второй номер версии и второй номер сборки выпускаемых версий всегда 0.
Standalone Player (автономный проигрыватель)	Версия Flash Player, которая запускается непосредственно как исполняемый файл локальной системы, а не как плагин веб-броузера или элемент управления ActiveX.
Projector (проектор)	Самостоятельный исполняемый файл, который включает и файл *.swf*, и Standalone Player. Проекторы могут создаваться как для операционной системы Macintosh, так и для Windows, с помощью функции File→Publish меню Flash.

Файлы примеров и ресурсы

Официальный веб-сайт этой книги:

http://moock.org/eas2

Файлы примеров для этой книги можно скачать по адресу:

http://moock.org/eas2/examples

Примеры Flash-кода также можно найти в сетевом хранилище кода (Code Depot) книги «ActionScript for Flash MX: The Definitive Guide»:

http://moock.org/asdg/codedepot

Обширный список сетевых ресурсов, касающихся Flash, приведен по адресу:

http://moock.org/moockmarks

Большая коллекция ссылок на сотни ресурсов по ActionScript 2.0:

http://www.actionscripthero.com/adventures

Принятые обозначения

Для представления различных синтаксических компонентов ActionScript в этой книге приняты следующие обозначения:

Пункты меню

Переходы по пунктам меню отображаются с помощью символа →, например File→Open.

`Моноширинный`

Представляет фрагменты кода, имена экземпляров клипа, метки кадров, имена свойств, имена переменных и идентификаторы связывания символов.

Курсив

Представляет имена функций, методов, классов, пакетов, слоев, URL, имена файлов и расширения файлов, например *.swf*. Кроме курсивного начертания, имена методов и функций в тексте сопровождаются круглыми скобками, например *duplicateMovieClip( )*.

`Моноширинный полужирный`

Представляет текст, который при выполнении пошаговой процедуры необходимо ввести дословно. Кроме того, в примерах кода **`Моноширинный полужирный шрифт`** служит для визуального выделения, например подчеркивает важную строку кода в большом примере.

`Моноширинный курсив`

Указывает на код, который необходимо заменить соответствующим значением (например, *здесь расположено ваше имя*). *`Моноширинное курсивное`* начертание также служит для визуального выделения имен переменных, свойств, методов и функций, помещаемых в комментарии в примерах кода.

Это подсказка. Она содержит полезную информацию по рассматриваемой теме, зачастую обращает внимание на важные идеи или лучшие практические методы.

Это предупреждение. Помогает избежать досадных ошибок, обойти трудности или предупреждает о надвигающейся опасности. Внять ему или игнорировать его – ваше дело.

Это примечание об ActionScript 1.0. Здесь сравниваются и противопоставляются ActionScript 1.0 и ActionScript 2.0, что помогает перейти к ActionScript 2.0 и понять важные различия между этими двумя версиями языка программирования.

Примеры кода

Эта книга призвана помочь вам в работе. В общем код, приведенный в данной книге, можно помещать в программы и документацию. Не надо запрашивать у нас разрешения, если только вы не воспроизводите значительные части кода. Например, если в вашей программе присутствует несколько фрагментов кода из этой книги, то разрешение не требуется. Если вы будете отвечать на вопросы, цитируя данную книгу и ссылаясь на примеры кода, разрешение не требуется. А вот для включения существенного количества примеров кода из этой книги в документацию своего продукта разрешение необходимо.

Мы признательны за указание авторства, что обычно включает название, автора, издателя и ISBN, например «Essential ActionScript 2.0 by Colin Moock. Copyright 2004 O'Reilly Media, Inc., 0-596-00652-7», но не требуем этого.

Если, используя примеры кода, вы чувствуете, что выходите за рамки законности или данных выше разрешений, не стесняйтесь обращаться к нам по адресу *permissions@oreilly.com*.

Мы хотели бы знать ваше мнение

Мы тщательно проверили информацию, излагаемую в данной книге, но что-то могло измениться (могли даже вкрасться какие-то ошибки!). Пожалуйста, сообщайте нам о любых обнаруженных ошибках, а также присылайте свои предложения для будущих изданий по адресу:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

(800) 998-9938 (в США или Канаде)

(707) 829-0515 (международный/местный)

(707) 829-0104 (fax)

Для этой книги мы создали веб-страницу, на которой приводим список опечаток, примеры или любую дополнительную информацию. Вы можете найти эту страницу по адресу:

http://www.oreilly.com/catalog/0596006527

Чтобы задать технические вопросы по этой книге или высказаться о ней, пришлите электронное письмо по адресу:

bookquestions@oreilly.com

Более подробная информация о наших книгах, конференциях, программному обеспечению, Центрах ресурсов (Resource Centers) и O'Reilly Network представлена на нашем веб-сайте:

http://www.oreilly.com

Благодарности

Иногда вам предоставляется возможность поблагодарить, но вы осознаете, что не можете выразить глубину своей признательности в полной мере. Вы можете высказать все, что хотите, но в конце концов должны просто поверить, что человек знает, насколько велика ваша благодарность. Я верю, что Ребекка Сан, ведущий разработчик ActionScript 2.0 компании Macromedia, знает это.

Мой ближайший партнер – Дерек Клейтон (Derek Clayton). В течение многих лет мы работали с Дереком над Unity, нашей коммерческой

инфраструктурой для создания многопользовательских приложений (смотрите *http://moock.org/unity*). Дерек был моим наставником с тех пор, как я впервые столкнулся с оператором *if*, а дружим мы еще дольше. Я учусь у него чему-то новому практически каждый день. Эта книга наполнена мудростью, которой он делился со мной все эти годы.

Брюс Эпштейн (Bruce Epstein), мой редактор. Ну что можно сказать? Просто он – лучший. Ни одна гипербола не может преувеличить ни его достоинств, ни справедливости, поэтому я промолчу.

Также я хотел бы поблагодарить всех редакторов, отдел подготовки к печати, дизайнеров, художников, сотрудников служб продаж и маркетинга издательства O'Reilly, включая Гленн Бисигнани (Glenn Bisignani), Клер Клотье (Claire Cloutier), Колина Гормана (Colleen Gorman), Тима О'Рейли (Tim O'Reilly), Роба Романо (Rob Romano), Сару Шерман (Sarah Sherman), Эллен Траутман (Ellen Troutman) и Элли Волькхаузен (Ellie Volckhausen). Также мои благодарности редактору Норме Эмори (Norma Emory) за помощь в подготовке рукописи к печати.

Также я благодарен членам команды разработки Flash компании Macromedia, которые являются постоянным источником вдохновения, знаний и дружбы с момента появления «Flash». Я уверен, что все, кто интересуется компьютерами, в долгу перед всей Flash-командой за то, что они постоянно находят новые формы взаимодействия с помощью компьютера. Прежде всего по гроб жизни я обязан Гари Гроссману (Gary Grossman) за его бесконечную поддержку и доброту, ему мои бесконечные «спасибо» и длительные рукопожатия. Особенно хочется отметить членов Flash-команды, бывших и теперешних, с которыми мне выпала честь вместе работать: Найджела Пег (Nigel Pegg), Майкла Вилльямса (Michael Williams), Эрика Нортона (Erica Norton), Валида Анбара (Waleed Anbar), Денеба Мекета (Deneb Meketa), Метта Вобенсмита (Matt Wobensmith), Майка Чемберса (Mike Chambers), Криса Силджена (Chris Thilgen), Джиллиса Дрю (Gilles Drieu), Найвиша Раджбхандари (Nivesh Rajbhandari), Тею Ота (Tei Ota), Троя Эванса (Troy Evans), Лусьена Бииба (Lucian Beebe), Джона Дауделла (John Dowdell), Бентли Вулфа (Bentley Wolfe), Джеффа Мотт (Jeff Mott), Тайника Уро (Tinic Uro), Роберта Татсуми (Robert Tatsumi), Майкла Ричардса (Michael Richards), Шерон Селдон (Sharon Seldon), Джоди Женг (Jody Zhang), Джима Корбетта (Jim Corbett), Карен Кук (Karen Cook), Джонатана Гей (Jonathan Gay), Пита Сантанджели (Pete Santangeli), Шона Кранзберга (Sean Kranzberg), Майкла Морриса (Michael Morris), Кевина Линча (Kevin Lynch), Бена Чан (Ben Chun), Эрика Виттмана (Eric Wittman), Джереми Кларка (Jeremy Clark) и Джениса Пирса (Janice Pearce).

Мне исключительно повезло иметь действительно замечательных технических редакторов и корректоров при работе над этой книгой. Мудрый взгляд Ребекки Сан охватил весь текст. Гари Гроссман просмотрел ключевые разделы, включая главу 10. Эти проницательные читатели

были моими проводниками при написании книги: Алистар Мак-Лоид (Alistair McLoed), Чафик Казоун (Chafic Kazoun), Джон Виллиамс (Jon Williams), Маркус Дикинсон (Marcus Dickinson), Оуэн Ван Дайджк (Owen Van Dijk), Питер Холл (Peter Hall), Ральф Бокелберг (Ralf Bokelberg), Роберт Пеннер (Robert Penner) и Сэм Нефф (Sam Neff). Особая благодарность Марку Джонкману (Mark Jonkman) и Нику Ридеру (Nick Reader) за их постоянную кропотливую работу по проверке рукописи.

Слова любви моей жене Венди (Wendy), которая поддерживает меня. Моей семье и друзьям. И деревьям за ответы на все вопросы, величие любой мечты и за бумагу, на которой эта книга напечатана.

Колин Мук (Colin Moock)
Торонто, Канада
март 2004

I

Язык программирования ActionScript 2.0

В части I вы познакомитесь с основами объектно-ориентированных понятий, синтаксиса и применения ActionScript 2.0. Даже если вы никогда ранее не сталкивались с ООП, эта часть книги обеспечит вам его понимание. Здесь рассматриваются классы, объекты, методы, свойства, наследование, формирование, интерфейсы, пакеты и масса других основных понятий ООП. Материал этой части книги также поможет вам понять, насколько ООП подходит для ваших проектов и как лучше структурировать классы и их методы.

- Глава 1 «Краткий обзор ActionScript 2.0»
- Глава 2 «Объектно-ориентированный ActionScript»
- Глава 3 «Типы данных и контроль типов»
- Глава 4 «Классы»
- Глава 5 «Создание класса на ActionScript 2.0»
- Глава 6 «Наследование»
- Глава 7 «Создание подкласса в ActionScript 2.0»
- Глава 8 «Интерфейсы»
- Глава 9 «Пакеты»
- Глава 10 «Исключения»

5

Создание класса на ActionScript 2.0

В главе 4 мы познакомились с общим строением классов ActionScript 2.0. В этой главе мы применим эту теорию на практике и напишем на ActionScript 2.0 реальный класс *ImageViewer*. Класс *ImageViewer* создает на экране прямоугольную область для отображения загружаемых в формате JPEG изображений. Мы рассмотрим как проектирование и написание кода самого класса, так и его применение в документе Flash.

Исходные файлы *ImageViewer*, обсуждаемые в этой главе, можно скачать по адресу *http://moock.org/eas2/examples*.

Краткое руководство по созданию класса

Перед тем как перейти к проектированию класса *ImageViewer*, рассмотрим кратко минимальный набор операций, необходимых для создания и применения класса ActionScript 2.0 во Flash. Не переживайте, если некоторые из понятий, фигурирующих в этом обзоре, вам неизвестны. Мы подробно рассмотрим каждое из них далее в этой главе.

Чтобы создать класс на ActionScript 2.0, следуйте этим основным этапам:

1. Создайте новый текстовый файл с расширением *.as* в любом текстовом редакторе или встроенном редакторе Flash MX Professional 2004. Имя файла *.as* должно точно совпадать с именем класса (с учетом регистра).
2. Добавьте в файл *.as* описание класса. Например:

```
class ИмяКласса {
  // Здесь находится тело класса
}
```

Чтобы применить класс ActionScript 2.0 в фильме Flash, делайте следующее:

1. Создайте файл *.fla* с любым именем и поместите его в ту же папку, что и файл *.as* из первого пункта предыдущей операции.
2. Задайте для файла *.fla* кадр экспорта класса, выполнив команду File → Publish Settings → Flash → ActionScript Version → Settings (Настройки) → Export Frame for Classes (Кадр экспорта для классов) (не обязательно). Это определяет, когда класс загружается и когда он становится доступным в фильме. Кадром экспорта обычно назначается некоторый кадр, находящийся после предзагрузчика фильма.
3. С классом можно работать по всему файлу *.fla*, но только после кадра экспорта (если таковой имеется), заданного в пункте 2.
4. Экспортируйте файл *.swf* одним из следующих способов: File → Publish, Control (Управление) → Test Movie (Тестировать фильм) или File → Export (Экспорт) → Export Movie (Экспортировать фильм).

Все это хорошо в небольших проектах, в которых повторное использование кода и распространение не являются основными критериями. Информация по работе с группами классов во многих приложениях представлена в главе 9 и главе 14.

Теперь приступим к проектированию класса *ImageViewer*.

Проектирование класса ImageViewer

Освоившись с синтаксисом первого для вас объектно-ориентированного языка программирования, вы неизбежно осознаете, что основная сложность ООП не в освоении синтаксиса, а в разработке архитектуры приложения, и ежедневно будете сталкиваться с необходимостью принятия конструкторских решений. Какие классы будут образовывать приложение? Как эти классы будут взаимодействовать? Какие открытые методы и свойства они будут иметь? Какие методы и свойства должны оставаться внутренними? Как будут называться все эти классы, методы и свойства?

К счастью, вам не придется оставаться с этими вопросами один на один. В течение многих лет сообщество ООП-разработчиков составляло каталог так называемых *шаблонов проектирования* (*design patterns*), которые описывают общепринятые решения распространенных общих проблем. Как применять шаблоны проектирования во Flash, мы увидим в части III данной книги. Шаблоны проектирования главным образом фокусируются на взаимодействиях классов. Сейчас перед нами стоит более простая задача – разработка одного автономного класса.

Мы уже дали нашему классу имя *ImageViewer* и определили его основное предназначение. Хотите – верьте, хотите – нет, но значительная часть работы по проектированию уже выполнена. Определившись с основным предназначением класса, мы обозначили круг его обязанно-

стей, в которые входят загрузка и вывод изображения. С другой стороны, мы точно установили, что эти функции не относятся к другим классам. То есть в приложении, использующем класс *ImageViewer*, другие классы не будут пытаться самостоятельно загрузить изображения; для этого они будут создавать и использовать экземпляры *ImageViewer*. Таким образом, всего лишь сформулировав круг обязанностей класса *ImageViewer*, мы четко обозначили, как он взаимодействует с другими классами, а это важный аспект проекта любого ООП-приложения. Обозначив круг обязанностей класса, мы, как и были должны, частично определили, как он будет встраиваться в большую структуру.

Теперь, определившись с именем и назначением класса, мы можем перейти к детализации предъявляемых к нему функциональных требований. Вот список функциональных возможностей, которые могут потребоваться от класса *ImageViewer*:

- Загрузка изображения
- Отображение изображения
- Обрезка изображения до размеров определенной прямоугольной «области просмотра»
- Отрисовка рамки вокруг изображения
- Отображение процесса его загрузки
- Перемещение области просмотра
- Изменение размеров области просмотра
- Панорамирование (перемещение) изображения в области просмотра
- Масштабирование (изменение размеров) изображения в области просмотра

Из этого списка мы выберем только то, что совершенно необходимо на настоящий момент, оставляя все остальное на потом. Мы действуем в духе следующего правила экстремального программирования: «нельзя добавлять функциональную возможность раньше времени» (см. *http://www.extremeprogramming.org/rules.html*).

Минимальная функциональность, позволяющая организовать класс *ImageViewer* и запустить его:

- Загрузить изображение
- Показать изображение

С этого короткого списка мы и начнем.

От функциональных требований к коду

Наш первый шаг при переходе от функциональных требований к законченному классу состоит в том, чтобы определить, как класс *ImageViewer* будет использоваться другими программистами. Например, какой метод должен вызывать программист, чтобы изображение появилось на экране? Как осуществляется загрузка изображения? Эти две операции

реализованы как отдельные методы или как один? Определившись с тем, как наш класс должен использоваться другими классами (созданными нами или другими разработчиками), мы можем обратить внимание на то, как он должен работать (т. е. мы можем реализовать предложенные методы). Конечно, в ходе реализации непременно встанет вопрос взаимодействия, оказывающий влияние на способ использования класса и, таким образом, меняющий нашу исходную конструкцию.

Набор открытых методов и свойств, предоставляемых классом, иногда называют *прикладным программным интерфейсом класса*. Внесение изменений в код является естественной частью цикла разработки, но в идеале открытый API не меняется даже при пересмотре внутреннего кода. Термин *реорганизация* (*refactoring*), или *рефакторинг*, подразумевает модификацию внутреннего кода программы без изменения его внешнего (видимого) поведения. Например, можно реорганизовать класс, чтобы повысить его производительность или улучшить код.

В нашем примере мы пытаемся создать API класса *ImageViewer*. Традиционно под «API» подразумевают сервисы, предоставляемые всей библиотекой классов, такие как Java API или Windows API. Однако в данном обычном контексте термин «API» часто употребляется для описания открыто доступной функциональности чего бы то ни было, от отдельного класса до целой группы классов. API класса также иногда называют *общедоступным интерфейсом* (*public interface*), не путайте с графическим интерфейсом пользователя (Graphical User Interface – GUI) и с интерфейсами, которые мы рассмотрим в главе 8.

Вспомним, что первым функциональным требованием для нашего класса *ImageViewer* является загрузка изображения. Эта операция должна быть открыто доступной, т. е. код любого класса должен иметь возможность предписать экземпляру *ImageViewer* загрузить изображение, может быть, несколько раз подряд. Местонахождение в сети (URL) изображения, которое надо загрузить, должно предоставляться извне. Короче говоря, программисту, работающему с экземпляром *ImageViewer*, надо выдать команду «загрузить», а экземпляру *ImageViewer* надо получить от программиста URL, чтобы выполнить эту команду. Похоже, что это неплохая кандидатура на звание метода! Команду «загрузить» назовем *loadImage( )* (загрузить изображение). Вот основная сигнатура метода *loadImage( )*:

```
ImageViewer.loadImage(URL)
```

Имя метода, *loadImage*, играет роль «команды», описывающей выполняемую этим методом операцию. Имя метода должно быть абсолютно понятным и исчерпывающим. Любой, кто увидит его в исходном коде, должен суметь сделать вывод о назначении этого метода без прочтения обширных комментариев. Правильно названные методы не нуждаются в комментариях.

Сложности с подбором имени для метода могут быть вызваны тем, что метод пытается делать слишком многое. Попробуйте разделить его на несколько методов или реструктурируйте класс, особенно если в имени метода присутствует слово «и».

Метод *loadImage( )* принимает один параметр `URL`, который определяет, где в сети находится изображение, которое надо загрузить. Параметр `URL` должен быть строкой, и метод не возвращает никакого значения. Поэтому полная сигнатура и возвращаемый тип метода выглядят так:

```
ImageViewer.loadImage(URL:String):Void
```

Но не забегаем ли мы вперед? Нужен ли вообще отдельный метод *loadImage( )*? Может быть, URL загружаемого изображения должен просто передаваться в конструктор *ImageViewer*, как здесь:

```
var viewer:ImageViewer = new ImageViewer("someImage.jpg");
```

Это более лаконично, чем создание экземпляра *ImageViewer* с последующим вызовом его метода *loadImage( )*. Но без метода *loadImage( )* каждый экземпляр *ImageViewer* может загружать только одно изображение. Если мы хотим, чтобы экземпляры *ImageViewer* последовательно загружали множество изображений, нам нужен метод *loadImage( )*. Возможно, мы должны реализовать и метод *loadImage( )*, и параметр конструктора `URL`, но сделать URL конструктора необязательным. Это интересная возможность, но в нашей нынешней ситуации без нее вполне можно обойтись. С учетом того, что мы без труда можем добавить параметр конструктора без влияния на нашу конструкцию и позже, можно пока спокойно отложить параметр. Это решение мы должны записать как часть логического обоснования создания нашего класса или в официальной спецификации, или просто в комментариях в исходном коде класса. При проектировании и реализации своих классов не забывайте документировать рассматриваемые потенциальные возможности. Кроме того, обязательно фиксируйте возможности, от которых вы отказались (а не просто отложили их реализацию из-за каких-то конструкторских ограничений). Это особенно касается потенциальных возможностей, имеющих неочевидные недостатки или ограничения. Это поможет вам точно вспомнить, почему вы отказались от того или иного конструкторского решения, когда вы или кто-либо другой в следующий раз будет пересматривать код.

Перейдем ко второму функциональному требованию – показать изображение. Когда Flash Player выводит изображение, в этом обязательно участвует по крайней мере один клип (то, во что загружается изображение). Но какой именно клип? Должны ли мы предоставить способ задания существующего клипа в качестве клипа для размещения изображения? Мы могли бы написать метод *setImageClip( )* (задать клип изображения), который должен был бы вызываться перед *loadImage( )*:

```
var viewer:ImageViewer = new ImageViewer();
viewer.setImageClip(someClip_mc);
```

```
viewer.loadImage("someImage.jpg");
```

Это работоспособный код, но он может повредить содержимое, уже находящееся в заданном клипе. Например, если экземпляр *ImageViewer* загрузит изображение в основную временную диаграмму `level0`, то все содержимое Flash Player будет замещено этим изображением! Нехорошо. Более того, если метод *setImageClip( )* применить для изменения клипа с изображением уже после загрузки изображения, то экземпляр *ImageViewer* потеряет ссылку на исходный клип. Потеряв ссылку на клип, экземпляр *ImageViewer* не сможет позиционировать, изменять размеры или любым иным способом управлять изображением, как не сможет он и удалить это изображение перед загрузкой другого.

Чтобы не усложнять (в первой версии все должно быть предельно простым!), мы хотим гарантировать однозначные ассоциации (по одному клипу изображения для каждого экземпляра *ImageViewer*). Следовательно, каждый экземпляр будет создавать клип для размещения изображения. Это гарантирует, что изображение будет загружаться в пустой клип и что каждый экземпляр всегда будет загружать изображение(я) в один и тот же клип. В ActionScript клип может одновременно загружать только одно изображение.

Мы решили, что каждый экземпляр *ImageViewer* будет создавать собственный клип для размещения изображения, но нам по-прежнему надо знать, куда поместить этот клип. То есть мы должны быть оповещены о том, какой из существующих клипов будет играть роль контейнера для создаваемого нами клипа с изображением. Попробуем добавить открытый метод *setTargetClip( )* (задать клип назначения), определяющий клип и глубину, на которую должен быть помещен наш клип с изображением. Код, использующий *ImageViewer*, выглядел бы так:

```
var viewer:ImageViewer = new ImageViewer();
// someClip_mc - клип, который будет содержать новый клип с изображением,
// и 1 - глубина, на которой создается клип с изображением.
viewer.setTargetClip(someClip_mc, 1);
viewer.loadImage("someImage.jpg");
```

Мда. Выглядит несколько громоздко. Неэкономно вызывать два метода для загрузки одного изображения. Более того, мы хотим, чтобы каждый экземпляр *ImageViewer* знал свой клип назначения сразу после создания. Так что давайте перенесем задание клипа назначения в конструктор ImageViewer. Теперь при создании нового экземпляра *ImageViewer* в качестве аргументов его конструктора должны быть предоставлены клип назначения и глубина, и вот как выглядит новая сигнатура конструктора:

```
ImageViewer(target:MovieClip, depth:Number)
```

Код, использующий *ImageViewer*, теперь выглядел бы так:

```
var viewer:ImageViewer = new ImageViewer(someClip_mc, 1);
viewer.loadImage("someImage.jpg");
```

Этот код создает экземпляр *ImageViewer*, который в свою очередь создает новый пустой клип в `someClip_mc` на глубине 1. Затем он загружает файл *someImage.jpg* в этот пустой клип. Будучи загруженным, *someImage.jpg* автоматически появляется на экране (предполагаем, что на тот момент `someClip_mc` видимый).

Теперь сигнатуры и возвращаемые типы конструктора и метода *loadImage( )* нашего текущего проекта класса выглядят так (вспомните, что объявления конструктора никогда не включают возвращаемого типа данных):

```
ImageViewer(target:MovieClip, depth:Number)
ImageViewer.loadImage(URL:String):Void
```

Это кажется довольно разумным. Есть лишь один способ убедиться в этом – написать код.

Реализация ImageViewer (дубль 1)

Каждый класс ActionScript 2.0 должен находиться во внешнем текстовом файле с расширением *.as*. Следовательно, в первую очередь при написании нашего класса *ImageViewer* следует создать текстовый файл *ImageViewer.as*. Работая во Flash MX Professional 2004, можно создавать и редактировать *ImageViewer.as* во встроенном редакторе внешних сценариев. А во Flash MX 2004 (не Professional) текстовый файл *.as* придется создавать в текстовом редакторе сторонних производителей. Но даже пользователи Flash MX Professional 2004, наверное, оценят дополнительные возможности, предлагаемые некоторыми внешними редакторами (управление файлами, выделение синтаксиса и обеспечение подсказок кода).

Особой популярностью пользуются:

SciTE|Flash

http://www.bomberstudios.com/sciteflash

UltraEdit

http://www.ultraedit.com

Macromedia HomeSite

http://www.macromedia.com/software/homesite

TextPad

http://www.textpad.com

PrimalScript

http://www.sapien.com

В каком бы редакторе вы ни работали, создайте на жестком диске новую папку *imageviewer*. Все файлы этого учебного курса мы будем хранить в этой папке.

Во Flash MX Professional 2004 файл *ImageViewer.as* можно создать так:

1. Выберите меню File → New.
2. В диалоговом окне New Document (новый документ) на вкладке General (Общее) в качестве типа (Type) документа выберите ActionScript File (файл ActionScript).
3. Щелкните по кнопке OK. Откроется редактор сценариев с пустым файлом.
4. Выберите File → Save As (Сохранить как).
5. В диалоговом окне Save As задайте `ImageViewer.as` в качестве имени файла (соблюдая регистр) и сохраните файл в созданной вами папке *imageviewer*.

Теперь напишем немного кода и сохраним его в файле *ImageViewer.as*. Мы знаем, что имя нашего класса – *ImageViewer*, поэтому уже можем в общих чертах создать скелет класса. Введите в редактор сценариев код примера 5.1.

Пример 5.1. Скелет класса ImageViewer

```
class ImageViewer {
}
```

Обратите внимание, что имя класса, *ImageViewer*, и имя файла, *ImageViewer.as*, должны совпадать абсолютно (кроме расширения файла *.as*).

Имя класса и имя внешнего текстового файла *.as*, содержащего класс, должны быть идентичными (кроме расширения файла *.as*). Также файл должен иметь расширение *.as*. Не используйте текстовые редакторы, которые сохраняют дополнительные сведения о форматировании, такие как Microsoft Word. Сохраняйте файл как простой текст. По возможности применяйте формат Unicode (кодировка UTF-8). Если ваш редактор не поддерживает Unicode, применяйте ANSI, ASCII или Latin 1.

Если вы назвали свой класс *ImageViewer* (с прописной буквы «V»), но по ошибке присвоили файлу *.as* имя *Imageviewer.as* (с маленькой «v»), Flash не сможет найти класса *ImageViewer*. В этом случае при попытке применить класс *ImageViewer* вы увидите в окне Output следующую ошибку:

```
The class 'ImageViewer' could not be loaded
```
(Класс 'ImageViewer' не может быть загружен)

И наоборот, если вы по ошибке назвали свой класс *Imageviewer* (с маленькой «v»), а файл *.as* называется *ImageViewer.as* (с большой «V»), Flash найдет файл *ImageViewer.as*, но будет жаловаться, что не может найти в файле класс, имя которого совпадает с именем файла. Тогда при попытке использовать класс *ImageViewer* вы увидите следующую ошибку:

```
The class 'Imageviewer' needs to be defined in a file whose relative path is
'Imageviewer.as'
```
(Класс 'Imageviewer' должен быть описан в файле, относительный путь которого – 'Imageviewer.as')

А вместе с именем класса придется менять и имя файла, и наоборот.

Прежде чем перевести скелет нашего класса на усиленное питание, чтобы он немного «поправился», возможно, стоит выделить минутку на настройку своей среды разработки. Во Flash MX Professional 2004 можно включить нумерацию строк (View (Вид) → View Line Numbers (Показать нумерацию строк)) и задать собственные стили шрифтов, время отображения подсказок кода и правила автоматического отступа (Edit (Редактировать) → Preferences (Предпочтения) → ActionScript). Можно даже указать Flash правила автоматического форматирования кода (Edit → Auto Format Options (Опции автоматического форматирования)). Сама возможность автоматического форматирования доступна через Tools (Инструменты)→ Auto Format (Автоматическое форматирование).

Да, так на чем мы остановились? Верно, на скелете класса. На настоящий момент ваш файл *ImageViewer.as* должен содержать:

```
class ImageViewer {
}
```

Теперь вспомним конструкцию нашего класса:

```
ImageViewer(target:MovieClip, depth:Number)
ImageViewer.loadImage(URL:String):Void
```

Эта конструкция показывает базовую структуру конструктора и метода *loadImage( )* класса *ImageViewer*. Теперь давайте заполним эти элементы. Скорректируйте свой файл *ImageViewer.as*, чтобы он соответствовал примеру 5.2.

Пример 5.2. Класс ImageViewer с набросками конструктора и метода loadImage( )

```
class ImageViewer {
  // Функция-конструктор
  public function ImageViewer (target:MovieClip, depth:Number) {
  }

  // Метод loadImage()
  public function loadImage (URL:String):Void {
  }
}
```

Итак, мы добавили в наш класс конструктор и метод *loadImage( )* и можем перейти к написанию кода тела функции-конструктора. Мы уже решили, что конструктор класса *ImageViewer* должен создавать пустой клип для размещения загружаемого изображения. Этот пустой клип должен быть прикреплен к клипу `target` на заданной `depth` следующим образом:

```
public function ImageViewer (target:MovieClip, depth:Number) {
  target.createEmptyMovieClip("container_mc" + depth, depth);
}
```

Обратите внимание, что имя пустого клипа начинается с «container_mc» и заканчивается заданной глубиной (`depth`). Это обеспечивает уникальность имени каждого пустого клипа-контейнера. Например, если глубина `depth` равна 4, то имя пустого клипа будет `container_mc4`. В данный момент времени на заданной глубине может находиться только один клип, и имя каждого пустого клипа должно быть уникальным (если имя и родитель клипа совпадают с именем и родителем предыдущего клипа, доступ к такому клипу в ActionScript организовать невозможно). Таким образом, гарантируя уникальность сгенерированных на основании заданной глубины имен, в `target` может находиться более одного пустого клипа без возникновения конфликтов.

Не беспокойтесь, если вы не учли этого в исходном проекте. Для краткости мы опустили некоторые операции обычного процесса разработки. Обычно в черновом варианте функции-конструктора многие разработчики забывают об уникальности имен каждого клипа-контейнера.

В самом простом случае, когда в один экземпляр *ImageViewer* загружается только одно изображение, это не проблема. Однако как только потребуется создать несколько экземпляров для загрузки нескольких изображений (каждого в отдельный клип), вы сразу увидите недостаток (дефект) своего кода и внесете необходимые коррективы. Со временем вы научитесь предупреждать возможные проблемы проектирования.

Практически во всех случаях конструкция класса должна обеспечивать возможность мирного сосуществования нескольких его экземпляров. Одно исключение составляет случай, когда определяется класс, экземпляры которого никогда не создаются (т. е. который реализует только методы и свойства класса, а не методы или свойства экземпляров). Другим исключением является паттерн проектирования Singleton, обсуждаемый в главе 17.

В данном случае мирное сосуществование означает создание клипов с уникальными именами и размещение их на уникальных глубинах. Заметьте, что исходя из написанного для нашего текущего кода необходимо, чтобы уникальное значение глубины передавал пользователь класса *ImageViewer* (код не пытается автоматически определить уникальную глубину). Следовательно, именно пользователь должен гарантировать, что глубина, передаваемая в конструктор *ImageViewer*, не перекрывает существующие ресурсы; задокументируйте это в комментариях конструктора, как это сделано в примере 5.3.

Ну вот, наш конструктор готов, пора перейти к реализации метода *loadImage( )*, который загружает файл JPEG в пустой клип. В общих чертах код для загрузки изображений выглядит так:

```
theEmptyClip.loadMovie(URL);
```

А вот и проблема: метод *loadImage( )* не может организовать доступ к пустому клипу, созданному конструктором. Мы должны изменить конструктор так, чтобы он сохранял ссылку на пустой клип в свойстве экземпляра. Назовем это свойство `container_mc`. В примере 5.3 показаны новое свойство и скорректированная функция-конструктор (нововведения выделены полужирным шрифтом). Обновите свой файл *ImageViewer.as* в соответствии с примером 5.3 (комментарии включать не обязательно, но их написание должно стать хорошей привычкой). Теперь мы можем использовать свойство `container_mc` для доступа к пустому клипу из метода *loadImage( )*.

Пример 5.3. Класс ImageViewer с его новым свойством

```
class ImageViewer {
  // Вот новое свойство.
  private var container_mc:MovieClip;
  // Функция-конструктор.
  // Вызывающий отвечает за определение
  // уникальной глубины в клипе назначения.
  public function ImageViewer (target:MovieClip, depth:Number) {
    // Сохраняем ссылку на новый пустой клип
    // в свойстве container_mc.
    container_mc = target.createEmptyMovieClip("container_mc" + depth,
                                depth);
  }

  public function loadImage (URL:String):Void {
    container_mc.loadMovie(URL);
  }
}
```

Обратите внимание, что метод *loadImage( )* просто вызывает встроенный во Flash метод *loadMovie( )* (загрузитьФильм). Когда один метод просто вызывает другой, этот процесс называется *обертыванием* (*wrapping*), или созданием *оболочки* (*wrapper*); говорят, что метод *loadImage( )* является *оболочкой* метода *loadMovie( )*. Обычно оболочки для функций и методов (или даже классов) создаются для того, чтобы приспособить их к определенной ситуации. В нашем случае, делая метод *loadImage( )* оболочкой для *loadMovie( )*, мы:

- Делаем наш класс более наглядным (имя *loadImage( )* описывает поведение метода лучше, чем *loadMovie( )*).
- Делаем наш класс более удобным в использовании (`someViewer.loadImage()` удобнее, чем `someViewer.container_mc.loadMovie()`).
- Делаем наш класс более гибким и пригодным для внесения изменений в будущем; позже мы без труда сможем добавить или изменить код метода *loadImage( )*, не меняя особенности работы с ним.

В примере 5.4 показан полный код первой версии класса *ImageViewer* без комментариев. Убедитесь, что ваш файл *ImageViewer.as* соответствует коду примера 5.4.

Пример 5.4. Класс ImageViewer, версия 1

```
class ImageViewer {
  private var container_mc:MovieClip;

  public function ImageViewer (target:MovieClip, depth:Number) {
    container_mc = target.createEmptyMovieClip("container_mc" + depth,
                                                depth);
  }

  public function loadImage (URL:String):Void {
    container_mc.loadMovie(URL);
  }
}
```

Использование класса ImageViewer в фильме

Наш класс *ImageViewer* готов к тестированию, и можно посмотреть, как применять его во Flash-фильме! Начнем с получения файла изображения для загрузки:

1. Найдите в своей системе JPEG-изображение (формат, допускающий только линейную загрузку изображения). Если вы работаете в офисе, убедитесь, что это изображение можно использовать в общественном месте. Если подходящее изображение найти не удается, скачайте вот это *http://moock.org/eas2/examples* вместе с готовым примером *ImageViewer*.
2. Назовите изображение *picture.jpg* и поместите его в ту же папку, что и файл *ImageViewer.as*, созданный вами раньше.

Далее мы создадим документ Flash (файл *.fla*), из которого будем публиковать Flash-фильм (файл *.swf*), содержащий экземпляр нашего класса *ImageViewer*. Пока поместим файлы *.fla*, *.as*, *.swf* и *.jpg* в одну папку, упрощая для каждого из них задачу доступа к другим.

Если файл *.fla* и файл класса (т.е. файл *.as* с описанием класса) находятся в одном каталоге, код файла *.fla* может напрямую по имени ссылаться на файл *.as*. Отсюда следует, что в ActionScript 2.0 проще всего будет работать с классом, если поместить его файл *.as* в одну папку с файлом *.fla*, его использующим. Класс будет автоматически включен в файл *.swf*, экспортируемый из *.fla* (за исключением случаев, когда файл *.fla* вообще не ссылается на класс). В отличие от ActionScript 1.0, для включения класса ActionScript 2.0 в файл *.fla* директива `#include` не нужна.

При повторном использовании классов во многих проектах файлы классов должны храниться централизованно в месте, доступном для каждого проекта. В главах 9 и 14 мы узнаем, как создавать большие проекты и использовать классы многократно.

Теперь создадим Flash-документ, использующий класс *ImageViewer*:

1. В меню среды разработки Flash выберите File → New.

2. В диалоговом окне New Document (Новый документ) на вкладке General в качестве типа (Type) документа выберите Flash Document, затем нажмите кнопку OK.
3. Откройте пункт меню File → Save As, чтобы сохранить Flash-документ как `imageViewer.fla` в ту же папку, что и файл *ImageViewer.as*. (Имена файлов *.fla* начинаются с маленькой буквы. Нет необходимости в соблюдении регистра в именах файла *.fla* и класса. Обычно файл *.fla* использует несколько классов и его имя никак не связано с именами классов или именами файлов *.as*.)
4. На основной временной диаграмме *imageViewer.fla* переименуйте *Layer 1* (слой 1) на `scripts` (сценарии) (весь наш код мы поместим в слой *scripts*).

Теперь мы готовы использовать класс *ImageViewer* из *imageViewer.fla*. Чтобы создать экземпляр *ImageViewer* в кадре 1 основной временной диаграммы *imageViewer.fla*, сделайте следующее:

1. Выполнив команду Window (Окно) → Development Panels (Панели разработки) → Actions (F9), откройте панель Actions.
2. Выберите на основной временной диаграмме *imageViewer.fla* кадр 1.
3. В панели Actions введите следующий код:

```
var viewer:ImageViewer = new ImageViewer(this, 1);
viewer.loadImage("picture.jpg");
```

Обратите внимание, что класс *ImageViewer* доступен глобально и в файле *imageViewer.fla* к нему можно обратиться напрямую по имени из любого кода любого кадра, кнопки или клипа. Кстати, если экспортированный *imageViewer.swf* загружает другой файл *.swf*, класс *ImageViewer* доступен и для этого загружаемого *.swf*. Однако если в загружаемом *.swf*-файле есть класс с именем *ImageViewer*, загружаемая версия *.swf* не перекрывает версии *imageViewer.swf*. Более подробно об использовании классов во время выполнения многими *.swf*-файлами см. в главе 14.

Благодаря волшебному компилятору Flash ссылки на классы ActionScript 2.0 определяются в объекте `_global`, в стиле классов ActionScript 1.0. Чтобы доказать это, после описания нашего *класса ImageViewer* мы можем выполнить:

```
trace(typeof _global.ImageViewer);
```

при этом в окне Output будет выведено «function» (функция). Подробности см. в разделе «ActionScript 1.0 и 2.0 во Flash Player 6 и 7» главы 1.

Если класс с заданным именем уже описан, то в ActionScript 2.0 он не может быть переопределен другим описанием класса ActionScript 2.0. Описание класса можно изменить во время выполнения, лишь непосредственно перезаписав соответствующую глобальную переменную, как показано в данном фрагменте кода:

```
// Замещение описания класса ImageViewer
// строкой делает недоступным класс ImageViewer.
_global.ImageViewer = "Goodbye ImageViewer, nice knowing you.";
```

С помощью аналогичной методики класс ActionScript 2.0 может быть перекрыт классом ActionScript 1.0 следующим образом:

```
_global.ImageViewer = function () {
  // Здесь располагается тело функции-конструктора ActionScript 1.0
}
```

И наконец, момент, которого мы так ждали! Экспортируем наш файл *imageViewer.swf* и протестируем его в режиме Test Movie во Flash следующим образом:

1. Выберем в меню Control → Test Movie. Файл *.swf* должен воспроизводиться, и ваше изображение должно загружаться и появляться на экране.
2. Насладившись вдоволь плодами своего труда, выберите File → Close (Закрыть), чтобы вернуться к файлу *imageViewer.fla*.

Файл *imageViewer.swf* можно экспортировать для воспроизведения в веб-броузере посредством команды File → Publish. Однако если в вашем броузере установлен Flash Player 6, вы увидите, что файл *picture.jpg* не загружается.

Даже несмотря на то, что ActionScript 2.0 компилируется в такой же байт-код, что и ActionScript 1.0, и почти 100% конструкций ActionScript 2.0 поддерживаются Flash Player 6, для обеспечения правильности работы во Flash Player 6 файлы *.swf* должны экспортироваться в формат Flash Player 6. См. главу 1.

Как компилятор экспортирует SWF-файлы

При экспорте файла *.swf* компилятор ActionScript 2.0 составляет список всех классов, необходимых этому *.swf*. В частности, список необходимых классов включает все классы, используемые исходным *.fla*-файлом, и все классы, используемые этими классами. (В нашем случае список необходимых классов состоит из одного *ImageViewer*.) Затем компилятор ищет в системе соответствующие исходные *.as*-файлы и компилирует каждый из них в *.swf*, преобразуя в байт-код, понятный Flash Player. По умолчанию компилятор ведет поиск *.as*-файлов в том каталоге, где находится файл *.fla*, но он также может проверять любые каталоги, приведенные разработчиком в так называемом *пути к классам документа* (*document classpath*) или *глобальном пути к классам* (*global classpath*) (пути к классам мы рассмотрим в главе 9). Файлы классов, существующие в системе, но не используемые *.swf*, не компилируются в *.swf*. А в случае отсутствия необходимого класса возникает ошибка компиляции.

Настройки экспорта *imageViewer.fla* для обеспечения поддержки воспроизведения во Flash Player 6 можно изменить следующим образом:

1. Выберите в меню File → Publish Settings.
2. На вкладке Flash диалогового окна Publish Settings выберите опцию Version (версия) Flash Player 6.
3. Нажмите кнопку OK.

Если определена версия Flash Player 6, то любой файл *.swf*, экспортируемый из *imageViewer.fla* (через File → Publish, Control → Test Movie или File → Export → Export Movie), будет работать во Flash Player 6. Flash Player 5 и более старые версии не поддерживают ActionScript 2.0 независимо от формата файла *.swf*.

Предзагрузка класса ImageViewer

Разве не замечательно было увидеть класс *ImageViewer* в действии? Но есть небольшая проблема. Сейчас класс *ImageViewer* настолько мал, что вряд ли можно заметить, как он загружается. Однако если бы его размер был, скажем, равен 50 Кбайт или 100 Кбайт, то при его загрузке через медленный канал связи задержка была бы ощутимой. По умолчанию все классы загружаются перед отображением кадра 1, что приводит к задержке перед началом фильма. Если время загрузки достаточно велико, возникнет впечатление, что фильм поврежден или «завис». В большинстве случаев отдельные классы невелики, но в некоторых приложениях общий размер всех классов превышает 100 Кбайт. К счастью, Flash обеспечивает нам возможность предварительно определить, когда должны загружаться классы фильма.

Изменим наш файл *imageViewer.fla* так, чтобы используемые им классы не загружались до кадра 10:

1. Выберите в меню File → Publish Settings.
2. В диалоговом окне Publish Settings на вкладке Flash, следующей за ActionScript Version, щелкните Settings.
3. В диалоговом окне ActionScript Settings для Export Frame (Кадр экспорта) для Classes (классы) введите 10.
4. Нажмите кнопку OK, чтобы подтвердить ActionScript Settings.
5. Нажмите кнопку OK, чтобы подтвердить Publish Settings.

Теперь введем в наш файл *imageViewer.fla* самый элементарный предзагрузчик, призванный отображать процесс загрузки класса *ImageViewer*. По завершении загрузки мы переместим головку воспроизведения в кадр 15, где создадим экземпляр *ImageViewer* (как делали это раньше в кадре 1).

Сначала доведем длину временной диаграммы до 15 кадров следующим образом:

1. На основной временной диаграмме *imageViewer.fla* выберем кадр 15 слоя *scripts*.

2. Выберем в меню Insert (Вставка) → Timeline (Временная диаграмма) → Keyframe (Ключевой кадр) (F6).

Далее добавим слой *labels* (метки) с двумя метками кадров, `loading` (загрузка) и `main` (основной). Метки обозначают состояние загрузки и точку запуска приложения соответственно.

1. Выберем в меню Insert → Timeline → Layer (Слой).
2. Дважды щелкнув по имени нового слоя, изменим его на `labels`.
3. К кадрам 4 и 15 слоя *labels* добавим новый ключевой кадр (Insert → Timeline → Keyframe (Ключевой кадр)).
4. Выбрав кадр 4 слоя *labels*, в панели Properties (Свойства) под Frame изменим <Frame Label> (метка кадра) на `loading`.
5. Выбрав кадр 15 слоя *labels*, на панели Properties под Frame изменим <Frame Label> на `main`.

Теперь в слой *scripts* добавим сценарий предзагрузчика:

1. К кадру 5 слоя *scripts* добавим новый ключевой кадр (используя Insert → Timeline → Keyframe).
2. Выбрав кадр 5 слоя *scripts*, в панели Actions введем следующий код:

```
if (_framesloaded == _totalframes) {
  gotoAndStop("main");
} else {
  gotoAndPlay("loading");
}
```

Далее перенесем код, создающий экземпляр *ImageViewer*, из кадра 1 в кадр 15 слоя *scripts*:

1. Выберем кадр 1 слоя *scripts*.
2. В панели Actions вырежем (удалим с помощью клавиш Ctrl-X или Cmd-X) из кадра 1 следующий код:

```
var viewer:ImageViewer = new ImageViewer(this, 1);
viewer.loadImage("picture.jpg");
```

3. Выбрав кадр 15 слоя *scripts*, вставим (с помощью клавиш Ctrl-V или Cmd-V) в панель Actions код, удаленный на шаге 2.

Наконец, добавим сообщение о загрузке, отображаемое во время загрузки класса *ImageViewer*:

1. Выбрав кадр 1 слоя *scripts*, введем в панели Actions следующий код:

```
this.createTextField("loadmsg_txt", 0, 200, 200, 0, 0);
loadmsg_txt.autoSize = true;
loadmsg_txt.text = "Loading...Please wait.";
```

2. Выбрав кадр 15 слоя *scripts*, введем следующий код в конце панели Actions (после кода, введенного на шаге 2 предыдущей процедуры):

```
loadmsg_txt.removeTextField();
```

Вот и все! Протестируйте свой фильм, выбрав в меню пункты Control → Test Movie. Находясь в режиме Test Movie, вы можете наблюдать имитацию скачивания фильма, выключая Bandwidth Profiler (Профилировщик полосы пропускания) (View → Bandwidth Profiler) и затем выбирая View → Simulate Download (Имитировать скачивание). Поскольку наш класс слишком мал, чтобы увидеть сообщение предзагрузки, вам, вероятно, придется выбрать очень малую скорость скачивания. Чтобы изменить скорость скачивания, выберите View → Download Settings (Настройки скачивания).

Реализация ImageViewer (дубль 2)

Теперь у нас есть хоть и простая, но рабочая версия класса *ImageViewer*, и мы можем ввести в него дополнительные возможности. При этом позаботимся о том, чтобы не менять общедоступный API класса. Например, на этом этапе мы не должны менять имя метода *loadImage( )* на *loadAndDisplayImage( )* (загрузить и отобразить изображение). Также мы не должны изменять порядок или тип параметров конструктора или метода *loadImage( )*.

Однако введение дополнений в API класса *ImageViewer* допустимо и нормально. Изменение его закрытых методов или свойств также допустимо, потому что подобные изменения не оказывают влияния на внешний код. Однако внесение изменений в общедоступный API класса опасно и считается плохой практикой, потому что влечет за собой необходимость соответствующего корректирования всего кода, использующего этот класс.

На этапах внутренней разработки (альфа и бета) изменение API представляет собой относительно обычную практику. Но с того момента как класс выходит в свет, его общедоступный API должен оставаться неизменным.

Любые изменения в общедоступном API желательно сопровождать документацией и изменением номера основной версии класса. При дополнении общедоступного API (т. е. введении новых методов) номер основной версии менять не надо, потому что такие дополнения не приводят к нарушению существующего кода.

Мы реализовали две основные возможности (загрузку и показ изображения) из нашего списка функциональных требований, приведенного ранее в разделе «Проектирование класса ImageViewer». Перейдем к третьему и четвертому требованиям: обрезке изображения и отображению рамки вокруг него.

Во Flash нет никаких встроенных средств для изменения загруженного растрового изображения. Поэтому мы не можем просто обрезать наше изображение, а должны применять к нему маску, которая скрывает лишние области изображения из поля зрения. Чтобы создать такую маску, мы сначала создадим пустой клип, затем отрисуем в нем закра-

шенный прямоугольник и применим этот прямоугольник в качестве маски поверх клипа с изображением. В конструктор *ImageViewer* добавим параметры для определения размера маски и местоположения обрезанного изображения.

На этом этапе разработки для управления процессом обрезки мы приняли решение ввести в функцию-конструктор два новых параметра. Не противоречит ли это нашей задаче о неизменности общедоступного API? Не обязательно. Прежде всего, мы пока что не опубликовали наш код, поэтому никто, кроме нас, еще не использует этот класс. Во-вторых, чтобы обеспечить обратную совместимость, мы можем позволить конструктору назначать допустимые принимаемые по умолчанию значения, даже если при вызове конструктора новые аргументы пропущены.

Чтобы обеспечить рамку, мы создадим клип и нарисуем в нем прямоугольный контур. Разместим клип с рамкой визуально поверх изображения. Толщину и цвет рамки будем извлекать из параметров, которые добавим в конструктор *ImageViewer*. Опять же мы решили задавать толщину и цвет рамки во время создания объекта. Другой вариант: можно было бы создать новый метод, посредством которого эти величины могли бы задаваться. Это обеспечило бы вызывающему возможность изменения этих настроек даже после создания объекта.

Для размещения клипов рамки и маски изменим структуру наших экранных ресурсов. Ранее мы создавали единственный клип, `container_mc`, в который загружали изображение.

Для того чтобы сделать изложение более кратким и понятным, я говорю: «клип `container_mc`», на самом деле имея в виду «клип, используемый свойством экземпляра `container_mc`». Строго говоря, имя самого клипа – `container_mcdepth` (где `depth` – глубина, на которой находится клип). А имя свойства экземпляра – `container_mc`.

Вот изначальный код создания клипа:

```
container_mc = target.createEmptyMovieClip("container_mc" + depth, depth);
```

На этот раз `container_mc` выступает исключительно в роли контейнера для трех клипов: клипа маски (`mask_mc`), клипа рамки (`border_mc`) и нового клипа изображения (`image_mc`). Клип `image_mc` мы поместим в самый низ (на глубине 0 в `container_mc`), клип `mask_mc` – посередине (на глубине 1), а клип `border_mc` – сверху (на глубине 2), как показано на рис. 5.1.

Вопрос в том, где в классе *ImageViewer* мы должны создавать клипы рамки, маски, изображения и контейнер? В первой версии класса *ImageViewer* мы создавали клип `container_mc` в функции-конструкторе. Теоретически мы могли бы применять аналогичный подход и сейчас (т. е. создавать клипы рамки, маски и изображения в конструкторе). Но вместо этого в данной версии класса *ImageViewer* для обработки

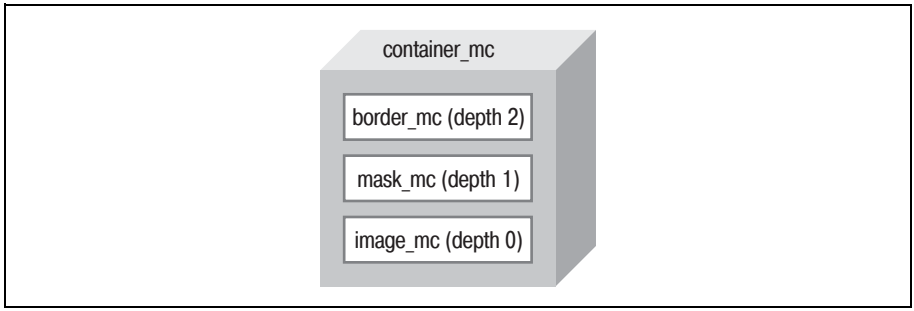

Рис. 5.1. *Структура клипа ImageViewer*

и создания различных клипов мы определим внутренние (закрытые) методы. Распределение работы между разными методами имеет следующие преимущества:

- Это делает код более понятным.
- Это упрощает процесс тестирования (тестировать каждый метод по отдельности проще, чем проводить тестирование большого блока кода, который осуществляет большое количество операций).
- Это делает возможным независимое создание и повторное создание ресурсов (например, можно менять рамку изображения без повторной загрузки самого изображения).
- Это делает возможным независимое изменение или переопределение операций по созданию ресурсов (например, можно написать новую подпрограмму создания рамки, не нарушая остального кода).

Таким образом, мы приняли эти конструкторские решения с целью увеличения гибкости и повышения удобства обслуживания нашего класса.

В табл. 5.1 приведен список новых методов создания клипов, все они объявлены с ключевым словом *private*.

Таблица 5.1. Новые методы экземпляра класса ImageViewer, отвечающие за создание клипов

Имя метода	Описание метода
buildViewer( )	Вызывает отдельные методы для создания клипов контейнера, изображения, маски и рамки.
createMainContainer( )	Создает клип `container_mc`
createImageClip( )	Создает клип `image_mc`
createImageClipMask( )	Создает клип `mask_mc`
createBorder( )	Создает клип `border_mc`

Теперь, когда операции по созданию клипов выведены в отдельные методы, нам надо добавить свойства, обеспечивающие эти методы следующей информацией:

- Ссылку на клип назначения, к которому должен быть прикреплен `container_mc`.
- Список глубин, отражающий порядок визуального расположения в стеке клипов контейнера, изображения, маски и рамки.
- Стиль (ширину и цвет) рамки вокруг изображения.

В табл. 5.2 представлен полный набор свойств экземпляров и класса *ImageViewer*, объявленных как *private*.

Свойства, обозначенные как свойства класса, определяют единственное значение, используемое всеми экземплярами *ImageViewer*. Свойства экземпляра для каждого экземпляра имеют собственное значение.

Таблица 5.2. Свойства класса и экземпляра класса ImageViewer

Имя свойства	Тип	Описание свойства
`container_mc`	Экземпляр	Ссылка на основной клип-контейнер, содержащий все клипы, используемые каждым экземпляром *ImageViewer*
`target_mc`	Экземпляр	Ссылка на клип, который будет содержать клип `container_mc`, как определено конструктором *ImageViewer*
`containerDepth`	Экземпляр	Глубина, на которой в `target_mc` создается `container?mc`, как определено конструктором *ImageViewer*
`imageDepth`	Класс	Глубина, на которой создается `image_mc` в `container_mc`
`maskDepth`	Класс	Глубина, на которой в `container_mc` создается `mask_mc`
`borderDepth`	Класс	Глубина, на которой в `container_mc` создается `border_mc`
`borderThickness`	Экземпляр	Толщина в пикселах рамки вокруг клипа `image_mc`
`borderColor`	Экземпляр	Целочисленный RGB-цвет рамки вокруг клипа `image_mc`

Окончательные изменения класса *ImageViewer* происходят в функции-конструкторе, которая должна определить новые параметры для поддержки новых методов и свойств класса. Мы должны описать параметры для задания:

- Координаты изображения (`x` и `y`)
- Размера маски изображения (`w` и `h`)
- Стиля рамки изображения (`borderThickness` (**толщинаРамки**) и `borderColor` (**цветРамки**))

Кроме этих новых параметров, мы сохраним наши исходные параметры `target` и `depth`. Таким образом, сигнатура конструктора *ImageViewer* теперь выглядит так:

```
ImageViewer(target:MovieClip, depth:Number, x:Number, y:Number,
            w:Number, h:Number, borderThickness:Number, borderColor:Number)
```

ImageViewer (дубль 2), обзор этапа проектирования

Завершив доработку конструктора, мы закончили и изменения нашего класса *ImageViewer* для версии 2.

Вот окончательная конструкция класса для данной версии. Здесь информация из табл. 5.1 и 5.2 повторяется, чтобы продемонстрировать, как можно представить класс на этапе проектирования даже до написания какого-либо кода:

Конструктор:

```
ImageViewer(target:MovieClip, depth:Number, x:Number, y:Number,
            w:Number, h:Number, borderThickness:Number, borderColor:Number)
```

Закрытые свойства:

```
container_mc
target_mc
containerDepth
imageDepth
maskDepth
borderDepth
borderThickness
borderColor
```

Открытые свойства – нет

Закрытые методы:

buildViewer(x:Number, y:Number, w:Number, h:Number)
createMainContainer(x:Number, y:Number)
createImageClip( )
createImageClipMask(w:Number, h:Number)
createBorder(w:Number, h:Number)

Открытые методы: *loadImage(URL:String)*

Реализация ImageViewer (дубль 2)

В примере 5.5 показан код второй версии класса *ImageViewer*. Чтобы полностью понять код, внимательно читайте комментарии. Если с некоторыми специальными методиками ActionScript вы сталкиваетесь впервые (например, с отрисовкой линий или созданием маски клипов), обратитесь к справочнику по языку программирования ActionScript, например к «ActionScript for Flash MX: The Definitive Guide».

Пример 5.5. Класс ImageViewer, дубль 2

```
// Класс ImageViewer, Версия 2
class ImageViewer {
  // Ссылки на клипы
  private var container_mc:MovieClip;
  private var target_mc:MovieClip;

  // Глубины расположения клипов
  private var containerDepth:Number;
  private static var imageDepth:Number  = 0;
  private static var maskDepth:Number   = 1;
  private static var borderDepth:Number = 2;

  // Стиль рамки
  private var borderThickness:Number;
  private var borderColor:Number;

  // Конструктор
  public function ImageViewer (target:MovieClip,
                               depth:Number,
                               x:Number, y:Number,
                               w:Number, h:Number,
                               borderThickness:Number,
                               borderColor:Number) {
    // Задаем значения свойств.
    target_mc = target;
    containerDepth = depth;
    this.borderThickness = borderThickness;
    this.borderColor = borderColor;

    // Настраиваем визуальные ресурсы этого ImageViewer.
    buildViewer(x, y, w, h);
  }

  // Создаем клипы для размещения изображения, маски
  // и рамки. Этот метод передает всю свою работу
  // отдельным методам создания клипов.
  private function buildViewer (x:Number, y:Number,
                                w:Number, h:Number):Void {
      createMainContainer(x, y);
      createImageClip();
      createImageClipMask(w, h);
      createBorder(w, h);
  }

  // Создаем контейнер, в котором располагаются все ресурсы
  private function createMainContainer (x:Number, y:Number):Void {
    container_mc =
      target_mc.createEmptyMovieClip("container_mc" + containerDepth,
                                     containerDepth);
    // Позиционирование клипа-контейнера.
    container_mc._x = x;
    container_mc._y = y;
```

```
}

// Создаем клип, в который фактически загружается изображение
private function createImageClip ():Void {
  container_mc.createEmptyMovieClip("image_mc", imageDepth);
}
// Создаем маску изображения
private function createImageClipMask (w:Number, h:Number):Void {
  // Создаем маску, только если заданы действительные ширина и высота.
  if (!(w > 0 && h > 0)) {
    return;
  }

  // В контейнере создаем клип, выступающий в роли маски изображения.
  container_mc.createEmptyMovieClip("mask_mc", maskDepth);

  // В маске отрисовываем прямоугольник.
  container_mc.mask_mc.moveTo(0, 0);
  container_mc.mask_mc.beginFill(0x0000FF);  // синий цвет для отладки
  container_mc.mask_mc.lineTo(w, 0);
  container_mc.mask_mc.lineTo(w, h);
  container_mc.mask_mc.lineTo(0, h);
  container_mc.mask_mc.lineTo(0, 0);
  container_mc.mask_mc.endFill();

  // Скрываем маску (оставаясь невидимой, она продолжает
  // выполнять свою функцию). Чтобы увидеть маску
  // во время отладки, закомментируем следующую строку.
  container_mc.mask_mc._visible = false;

  // Обратите внимание, что мы еще не подключили маску. Это нужно сделать
  // после начала загрузки изображения, в противном случае изображение
  // при загрузке перекроет маску.
}

// Создаем рамку вокруг изображения
private function createBorder (w:Number, h:Number):Void {
  // Создаем рамку, только если заданы действительные ширина и высота.
  if (!(w > 0 && h > 0)) {
    return;
  }

  // В контейнере создаем клип для размещения рамки изображения.
  container_mc.createEmptyMovieClip("border_mc", borderDepth);

  // Отрисовываем в клипе рамки прямоугольный контур
  // заданного размера и цвета.
  container_mc.border_mc.lineStyle(borderThickness, borderColor);
  container_mc.border_mc.moveTo(0, 0);
  container_mc.border_mc.lineTo(w, 0);
  container_mc.border_mc.lineTo(w, h);
  container_mc.border_mc.lineTo(0, h);
  container_mc.border_mc.lineTo(0, 0);
}
```

```
    // Загружаем изображение
    public function loadImage (URL:String):Void {
      // Загружаем JPEG-файл в клип image_mc.
      container_mc.image_mc.loadMovie(URL);

      // Далее идет отвратительный хакерский прием.
      // Мы избавимся от него в примере 5.6, когда добавим
      // соответствующую поддержку предзагрузки в Версии 3.
      // К тому времени, как один кадр передается,
      // загрузка изображения уже идет, и в этот момент
      // мы благополучно применяем маску к клипу image_mc.
      container_mc.onEnterFrame = function ():Void {
        this.image_mc.setMask(this.mask_mc);
        delete this.onEnterFrame;
      }
    }
  }
```

Использование ImageViewer (дубль 2)

Теперь, когда наш класс *ImageViewer* может обрезать изображение и окружать его рамкой, давайте поставим себя на место разработчика, использующего этот класс (а не создающего и обслуживающего его). Работая с первой версией класса *ImageViewer*, мы помещали в кадр 15 файла *imageViewer.fla* следующий код:

```
var viewer:ImageViewer = new ImageViewer(this, 1);
viewer.loadImage("picture.jpg");
```

Во второй версии *ImageViewer* введены две новые возможности, содержащие довольно много кода. Однако ни один из общедоступных API, описанных в версии 1 *ImageViewer*, в версии 2 не изменился; в версии 2 эти API были лишь дополнены. Метод *loadImage( )* изменился внутри, но никак не изменилась его внешняя сторона. Изменилась изнутри и функция-конструктор, но внешне в нее лишь были добавлены новые параметры. Она сохраняет параметры `target` и `depth` из версии 1 и вводит шесть новых: `x`, `y`, `w`, `h`, `borderThickness` и `borderColor`. Следовательно, пользователи версии 1 *ImageViewer* могут без труда обновить *imageViewer.fla*, чтобы использовать новые возможности версии 2 *ImageViewer*, следующим образом:

1. Заменить старый файл *ImageViewer.as* новым.
2. Изменить код кадра 15 *imageViewer.fla*, включив в него шесть новых параметров, необходимых конструктору *ImageViewer*. Например:

```
var viewer:ImageViewer = new ImageViewer(this, 1, 100, 100, 250, 250, 10, 0xCE9A3C);
viewer.loadImage("picture.jpg");
```

Теперь давайте посмотрим, что произошло бы, если бы мы перешли от версии 1 к версии 2, не изменив кода *imageViewer.fla*. (Для данного сценария предположим, что версия 2 обеспечивает повышение производительности; это и побудило нас перейти к ней.) Мы заменим наш файл

ImageViewer.as, как рекомендовано в шаге 1, но пропустим шаг 2, оставив неизменным код кадра 15:

```
var viewer:ImageViewer = new ImageViewer(this, 1);
viewer.loadImage("picture.jpg");
```

Как повела бы себя версия 2 *ImageViewer*, если бы в создании экземпляра класса принимали участие только два параметра? К счастью, замечательно. В версии 2 предусмотрены специальные средства защиты от недостатка параметров. Методы *createImageClipMask( )* и *createBorder( )* создают клипы маски и рамки только в том случае, если заданы допустимые значения ширины и высоты, как показано в этом фрагменте кода из примера 5.5:

```
if (!(w > 0 && h > 0)) {
  return;
}
```

Таким образом, если конструктору не предоставлены ни высота, ни ширина, версия 2 *ImageViewer* ведет себя точно так же, как и версия 1.

После официального выхода класса обновления в следующих его версиях не должны нарушать кода, использующего предыдущие версии. Изменения самого класса не должны провоцировать изменений кода, использующего этот класс. Достигается это, в частности, тем, что новые классы обеспечивают разумное стандартное поведение в случае отсутствия некоторых аргументов.

Реализация ImageViewer (дубль 3)

Давайте еще раз вспомним потенциальные функциональные возможности класса *ImageViewer*:

- Загрузка изображения
- Показ изображения
- Обрезка изображения до размеров определенной прямоугольной «области просмотра»
- Отрисовка рамки вокруг изображения
- Отображение процесса загрузки изображения
- Перемещение области просмотра
- Изменение размеров области просмотра
- Панорамирование (перемещение) изображения в области просмотра
- Масштабирование (изменение размеров) изображения в области просмотра

На данный момент мы успешно реализовали первые четыре из них. В следующей версии класса *ImageViewer* добавим пятую возможность: отображение процесса загрузки изображения. Остальные воз-

можности оставим нереализованными до тех пор, пока в них не возникнет особой необходимости. (В главе 7 мы вернемся к классу *ImageViewer*, чтобы добавить в него еще две возможности.)

Чтобы реализовать отображение процесса загрузки для нашего класса *ImageViewer*, воспользуемся классом *MovieClipLoader* (ЗагрузчикКлипа), который был добавлен во Flash Player 7 после убедительной просьбы Macromedia улучшить поддержку предзагрузки (любознательные читатели могут найти эту просьбу по адресу *http://moock.org/blog/archives/000010.html*).

С нашей реализацией процесса загрузки связаны следующие основные изменения в классе *ImageViewer*:

- Добавлены два новых свойства:

  `imageLoader` – хранит экземпляр *MovieClipLoader*;

  `statusDepth` – показывает глубину расположения текстового поля для отображения процесса загрузки.

- Изменен конструктор *Image Viewer* – теперь он сначала создает экземпляр *MovieClipLoader*, а затем регистрирует экземпляр *ImageViewer* на получение его событий.
- Изменен метод *loadImage( )* – он теперь:
  - Загружает JPEG-файл, используя *MovieClipLoader*, а не *loadMovie( )*.
  - Создает текстовое поле, в котором отображается процесс загрузки.
- Добавлены три новых метода: *onLoadProgress( )*, *onLoadInit( )* и *onLoadError( )*, предназначенные для обработки событий *MovieClipLoader*.

Если учесть, что мы добавили лишь одну возможность (процесс загрузки изображения), то может показаться, что всех этих конструктивных изменений слишком много. Мужайтесь. Код, необходимый для обеспечения поддержки предзагрузки, может показаться достаточно сложным, если это внове для вас, но, к счастью, он не сильно меняется в зависимости от ситуации. Пару раз реализовав поддержку предзагрузки, вы уже без труда сможете добавлять ее в собственные классы, как мы сделали это здесь.

Рассмотрим каждое из вышеперечисленных изменений по очереди. Начнем с кода новых свойств `imageLoader` (загрузчик изображения) и `statusDepth` (глубина строки состояния):

```
private var imageLoader:MovieClipLoader;
private static var statusDepth:Number = 3;
```

Затем – модифицированный конструктор *ImageViewer* (дополнения выделены полужирным шрифтом):

```
public function ImageViewer (target:MovieClip,
                             depth:Number,
```

```
                                x:Number, y:Number,
                                w:Number, h:Number,
                                borderThickness:Number,
                                borderColor:Number) {
    // Присваиваем значения свойствам.
    target_mc = target;
    containerDepth = depth;
    this.borderThickness = borderThickness;
    this.borderColor = borderColor;

    // Создаем экземпляр MovieClipLoader и сохраняем его
    // в новом свойстве imageLoader.
    imageLoader = new MovieClipLoader();

    // Регистрируем этот экземпляр ImageViewer
    // на получение событий экземпляра MovieClipLoader.
    imageLoader.addListener(this);

    // Настраиваем визуальные ресурсы этого ImageViewer.
    buildViewer(x, y, w, h);
  }
```

Вот скорректированный метод *loadImage( )*. Обратите внимание, что, даже несмотря на то, что все содержимое метода изменилось, способ его применения остался прежним, т. е. общедоступный API класса остался неизменным.

```
  public function loadImage (URL:String):Void {
    // Используем экземпляр MovieClipLoader для загрузки изображения.
    // Эта строка замещает предыдущий вызов loadMovie().
    imageLoader.loadClip(URL, container_mc.image_mc);
    // Создаем текстовое поле состояния загрузки
    // для отображения пользователю процесса загрузки.
    container_mc.createTextField("loadStatus_txt", statusDepth, 0, 0, 0, 0);
    container_mc.loadStatus_txt.background = true;
    container_mc.loadStatus_txt.border = true;
    container_mc.loadStatus_txt.setNewTextFormat(new TextFormat("Arial, Helvetica,
          _sans", 10, borderColor, false, false, false, null, null, "right"));
    container_mc.loadStatus_txt.autoSize = "left";

    // Позиционируем текстовое поле состояния загрузки.
    container_mc.loadStatus_txt._y = 3;
    container_mc.loadStatus_txt._x = 3;

    // Показываем, что идет процесс загрузки изображения.
    container_mc.loadStatus_txt.text = "LOADING";

  }
```

И наконец, в примере 5.6 показаны три метода, которые обрабатывают события загрузки изображения: *onLoadProgress( )*, *onLoadInit( )* и *onLoadError( )*. Метод *onLoadProgress( )* вызывается автоматически при поступлении части изображения. Как только изображение загрузится полностью и инициализируются свойства `_width` и `_height` клипа

image_mc, автоматически активизируется метод *onLoadInit( )*. Метод *onLoadError( )* вызывается автоматически в случае возникновения ошибки загрузки, такой как, например, «File not found» (Файл не найден).

Класс *MovieClipLoader* также предоставляет события *onLoadStart( )* и *onLoadComplete( )*, которые не нужны нашему классу *ImageViewer*. Подробную информацию можно найти в справочной системе Flash (меню ActionScript Dictionary (Словарь ActionScript) → M → MovieClipLoader).

Пример 5.6. Обработчики событий загрузки изображения, добавленные в версию 3 класса ImageViewer

```
public function onLoadProgress (target:MovieClip,
                                 bytesLoaded:Number,
                                 bytesTotal:Number):Void {
  // Отображаем процесс загрузки в текстовом поле. Делим bytesLoaded
  // и bytesTotal на 1024 для получения значений в килобайтах.
  container_mc.loadStatus_txt.text = "LOADING: "
      + Math.floor(bytesLoaded / 1024)
      + "/" + Math.floor(bytesTotal / 1024) + " KB";
}

public function onLoadInit (target:MovieClip):Void {
  // Удаляем сообщение о загрузке.
  container_mc.loadStatus_txt.removeTextField();

  // Применяем маску к загруженному изображению. Это заменяет используемый
  // нами прием onEnterFrame() из метода loadImage() версии 2 из примера 5.5.
  container_mc.image_mc.setMask(container_mc.mask_mc);
}

public function onLoadError (target:MovieClip, errorCode:String):Void {
  // В зависимости от значения errorCode отображаем
  // в текстовом поле соответствующее сообщение об ошибке.
  if (errorCode == "URLNotFound") {
    container_mc.loadStatus_txt.text = "ERROR: File not found.";
  } else if (errorCode == "LoadNeverCompleted") {
    container_mc.loadStatus_txt.text = "ERROR: Load failed.";
  } else {
      // Ловушка для обработки всех возможных кодов ошибок.
    container_mc.loadStatus_txt.text = "Load error: " + errorCode;
  }
}
```

Версия 3 нашего класса *ImageViewer* практически завершена. Однако необходимо учесть еще один вопрос – высвобождение ресурсов перед удалением каждого экземпляра *ImageViewer*.

Высвобождение ресурсов класса

В программировании игра не закончена, пока вы не уберете за собой игрушки. Каждый экземпляр класса *ImageViewer* создает экземпляры клипов, продолжающие существовать во Flash Player до тех пор, пока

они не будут явно удалены, даже если экземпляр *ImageViewer*, создавший их, уже уничтожен! Например, следующий код создает экземпляр *ImageViewer*, загружает изображение и затем уничтожает экземпляр *ImageViewer*:

```
var viewer:ImageViewer = new ImageViewer(this, 1, 100, 100, 250,
                                         250, 10, 0xCE9A3C);
viewer.loadImage("picture.jpg");
delete viewer;
```

После выполнения кода, несмотря на оператор `delete viewer;`, *picture.jpg* загружается и появляется на экране. Почему? Потому что клипы, созданные экземпляром *ImageViewer*, не были удалены перед уничтожением переменной `viewer`.

Оставить клипы на сцене – не единственный способ «осиротить» ресурсы класса *ImageViewer*. При определенных обстоятельствах экземпляр *ImageViewer* также может *самостоятельно* оставить себя в списке слушателей *MovieClipLoader*. Вспомните, что экземпляр *ImageViewer* при создании регистрирует себя в качестве слушателя `imageLoader` с помощью `imageLoader.addListener(this)`. В результате каждый экземпляр *ImageViewer* сохраняется в собственном списке объектов-слушателей `imageLoader`. После этого выполняется следующая строка кода:

```
var viewer:ImageViewer = new ImageViewer(this, 1);
```

Фактически существуют две ссылки на экземпляр *ImageViewer*: одна в переменной `viewer`, а другая в списке объектов-слушателей `viewer.imageLoader`. Если во время операции загрузки экземпляр `viewer` удаляется, экземпляр в списке слушателей `viewer.imageLoader` продолжает существовать.

Конечно, естественно было бы ожидать, что уничтожение `viewer` приводит к уничтожению `viewer.imageLoader` и, следовательно, списка объектов-слушателей `viewer.imageLoader`. В общем это могло бы быть так, но класс *MovieClipLoader* представляет собою особый случай: когда экземпляр *MovieClipLoader* начинает операцию *loadClip( )*, этот экземпляр (вместе со списком объектов-слушателей) хранится во Flash Player до тех пор, пока не будет завершена эта операция или пока не будет удален клип назначения операции загрузки. Например, в следующем фрагменте кода экземпляр `mcl` сохраняется в памяти до тех пор, пока не завершится загрузка *level1.swf* в `box`:

```
var mcl:MovieClipLoader = new MovieClipLoader();
mcl.loadClip("level1.swf", box);
delete mcl;
```

Таким образом, даже несмотря на то, что мы можем благополучно удалить все внешние ссылки на экземпляр *ImageViewer*, он может попрежнему существовать в собственном списке слушателей `imageLoader`!

Поэтому каждый экземпляр *ImageViewer* перед уничтожением должен высвобождать свои ресурсы. Мы осуществляем эту операцию в специальном методе *destroy( )*. Метод *destroy( )* не принимает параметров и должен вызываться перед уничтожением экземпляра *ImageViewer*. Исходный код *destroy( )* прост, но крайне важен и обязателен для выполнения. Он удаляет экземпляр *ImageViewer* из списка слушателей `imageLoader` и уничтожает иерархию клипов, показывающих изображение (прекращая таким образом любой процесс загрузки):

```
public function destroy ():Void {
  // Отменяет уведомления о событиях загрузки
  imageLoader.removeListener(this);
  // Удаляем клипы со сцены (уничтожение container_mc
  // влечет за собой удаление подклипов)
  container_mc.removeMovieClip();
}
```

Теперь, чтобы уничтожить любой экземпляр *ImageViewer*, мы сначала вызываем метод *destroy( )*:

```
// Высвобождаем ресурсы экземпляра
viewer.destroy();
// Удаляем экземпляр
delete viewer;
```

Между прочим, имя метода «destroy» (уничтожить) не является обязательным. С таким же успехом его можно заменить одним из синонимов: die (умертвить), kill (убить) или remove (удалить).

Наконец, следует заметить, что проблема со списком слушателей `imageLoader` не является чем-то из ряда вон выходящим. В любом случае, если объект регистрируется как слушатель другого объекта, перед уничтожением его регистрация должна быть отменена. В примере 5.7 показано, что произойдет, если регистрации объекта перед его уничтожением не будет отменена.

Пример 5.7. Потерянный слушатель

```
var obj:Object = new Object();
obj.onMouseDown = function ():Void {
  trace("The mouse was pressed.");
}
Mouse.addListener(obj);
delete obj;
```

Попробуйте сделать следующее:

1. Поместите код примера 5.7 в кадр 1 фильма.
2. Запустите фильм в режиме Test Movie (Control → Test Movie).
3. Щелкните в пределах Сцены (Stage).

В панели Output должна появиться надпись «The mouse was pressed» (Мышь была нажата), даже несмотря на то, что объект `obj` был удален!

Возможно, ссылка на объект в переменной `obj` была удалена, но другая ссылка в списке объектов-слушателей событий объекта *Mouse* продолжает существовать. Чтобы гарантированно уничтожить объект `obj`, сначала отмените его регистрацию в качестве слушателя *Mouse* следующим образом:

```
Mouse.removeListener(obj);
delete obj;
```

В этом разделе мы обсудили один частный случай, в котором ссылка на объект по неосторожности может сохраниться после удаления экземпляра класса. Невысвобождение ресурсов приводит к расходованию памяти, что со временем может стать причиной снижения производительности или привести к сбою приложения. При написании класса вы должны включать в него подпрограмму, осуществляющую очистку всех занятых экземпляром ресурсов, которую программист может вызывать перед уничтожением экземпляра. Тщательно продумайте, какие ресурсы требуют высвобождения. Не думайте, что, уклонившись от удаления экземпляров, вы избежите возникновения «осиротевших» ресурсов. Необходимо уничтожать экземпляры, если они больше не нужны, и убедиться, что перед этим были высвобождены все ресурсы. Помните: все, что создает ваш код, он же должен и уничтожать.

Окончательный вариант кода ImageViewer

В примере 5.8 приведен окончательный вариант кода версии 3 класса *ImageViewer*. Обновите код в файле *ImageViewer.as* в соответствии с примером. За рекомендациями по применению вернитесь к разделу «Использование ImageViewer (дубль 2)» (в версии 3 по сравнению с версией 2 никаких изменений в общедоступный API класса внесено не было, поэтому с точки зрения работы с ним ничего не изменилось). Если у вас все-таки что-то не заладилось с программой, скачайте исходные файлы для всех трех версий примера ImageViewer по адресу *http://moock.org/eas2/examples*.

Процесс загрузки изображения при загрузке с локального жесткого диска отображаться не будет. Чтобы протестировать *imageViewer.swf*, убедитесь, что разместили свои изображения на веб-сервере и воспроизводите фильм в веб-броузере.

Мы вернемся к классу *ImageViewer* в главе 7.

В примере 5.8 методы класса и функция-конструктор документируются в стиле комментирования JavaDoc. Из исходного файла класса Java автоматически может быть сгенерирована HTML-документация, если комментарии отформатированы согласно условным обозначениям JavaDoc. К сожалению, на момент написания данной книги Flash не поддерживает JavaDoc напрямую, но, может быть, в будущем инстру-

ментальные средства сторонних производителей или сам Flash обеспечат эту возможность. Как бы то ни было, стиль JavaDoc получил широкое признание и способен существенно улучшить читаемость исходного кода. Приведем для справки условные обозначения JavaDoc, используемые в примере 5.8:

@author
: Автор(-ы) класса

@version
: Версия класса

@param
: Имя метода или параметра конструктора и его назначение

Более подробную информацию по JavaDoc см. по адресам:

http://java.sun.com/j2se/javadoc

http://java.sun.com/j2se/javadoc/writingdoccomments

http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/javadoc.html#javadoctags

В примере 5.8 показан окончательный на данный момент вариант класса *ImageViewer*.

Пример 5.8. Класс ImageViewer, версия 3

```
/**
 * ImageViewer, Версия 3.
 * Прямоугольная область для отображения
 * загружаемого изображения.
 * Новые версии на: http://moock.org/eas2/examples/
 *
 * @author: Колин Мук
 * @version: 3.0.0
 */
class ImageViewer {
  // Клип, в котором будут располагаться все ресурсы ImageViewer
  private var container_mc:MovieClip;

  // Клип, к которому будет прикреплен container_mc
  private var target_mc:MovieClip;

  // Глубины расположения визуальных ресурсов
  private var containerDepth:Number;
  private static var imageDepth:Number = 0;
  private static var maskDepth:Number = 1;
  private static var borderDepth:Number = 2;
  private static var statusDepth:Number = 3;

  // Толщина рамки вокруг изображения
  private var borderThickness:Number;
  // Цвет рамки вокруг изображения
  private var borderColor:Number;
```

```
// Экземпляр MovieClipLoader, используемый для загрузки изображения
private var imageLoader:MovieClipLoader;

/**
 * Конструктор ImageViewer
 *
 * @param   target          Клип, к которому будет прикреплен ImageViewer
 * @param   depth           Глубина в target, на которой прикрепляется
 *                          средство просмотра
 * @param   x               Расположение средства просмотра по горизонтали
 * @param   y               Расположение средства просмотра по вертикали
 * @param   w               Ширина средства просмотра в пикселах
 * @param   h               Высота средства просмотра в пикселах
 * @param   borderThickness Толщина рамки изображения
 * @param   borderColor     Цвет рамки изображения
 */
public function ImageViewer (target:MovieClip,
                             depth:Number,
                             x:Number, y:Number,
                             w:Number, h:Number,
                             borderThickness:Number,
                             borderColor:Number) {
  // Присваиваем значения свойств
  target_mc = target;
  containerDepth = depth;
  this.borderThickness = borderThickness;
  this.borderColor = borderColor;
  imageLoader = new MovieClipLoader();

  // Регистрируем этот экземпляр на получение
  // событий от экземпляра imageLoader
  imageLoader.addListener(this);
  // Настраиваем визуальные ресурсы этого ImageViewer
  buildViewer(x, y, w, h);
}

/**
 * Создаем экранные ресурсы этого экземпляра ImageViewer.
 * Иерархия клипа такова:
 *  [d]: container_mc
 *         2: border_mc
 *         1: mask_mc (создание маски для image_mc)
 *         0: image_mc
 * где [d] – глубина, переданная пользователем в конструктор.
 *
 * @param   x   Расположение средства просмотра по горизонтали.
 * @param   y   Расположение средства просмотра по вертикали.
 * @param   w   Ширина средства просмотра в пикселах.
 * @param   h   Высота средства просмотра в пикселах.
 */
private function buildViewer (x:Number, y:Number,
                              w:Number, h:Number):Void {
```

```
    // Создаем клипы для размещения изображения, маски и рамки
    createMainContainer(x, y);
    createImageClip();
    createImageClipMask(w, h);
    createBorder(w, h);
}

/**
 * Создаем клип, container_mc, для размещения
 * визуальных ресурсов экземпляра ImageViewer.
 *
 * @param   x   Расположение клипа container_mc по горизонтали.
 * @param   y   Расположение клипа container_mc по вертикали.
 */
private function createMainContainer (x:Number, y:Number):Void {
  container_mc = target_mc.createEmptyMovieClip("container_mc" +
                                        containerDepth, containerDepth);
  container_mc._x = x;
  container_mc._y = y;
}

/**
 * Создаем клип, в который фактически загружается изображение
 */
private function createImageClip ():Void {
  container_mc.createEmptyMovieClip("image_mc", imageDepth);
}

/**
 * Создаем маску поверх изображения. Обратите внимание,
 * что этот метод на самом деле не подключает маску
 * к клипу изображения, потому что маска клипа перестает
 * действовать при загрузке в него нового содержимого.
 * Следовательно, маска подключается в onLoadInit().
 *
 * @param   w   Ширина маски в пикселах.
 * @param   h   Высота маски в пикселах.
 */
private function createImageClipMask (w:Number,
                                      h:Number):Void {
  // Создаем маску только при условии задания
  // допустимых значений ширины и высоты
  if (!(w > 0 && h > 0)) {
    return;
  }

  // В контейнере создаем клип, выполняющий роль маски изображения
  container_mc.createEmptyMovieClip("mask_mc", maskDepth);

  // Отрисовываем прямоугольник в маске
  container_mc.mask_mc.moveTo(0, 0);
  container_mc.mask_mc.beginFill(0x0000FF); // Для отладки применяем
                                            // синий цвет
```

```
    container_mc.mask_mc.lineTo(w, 0);
    container_mc.mask_mc.lineTo(w, h);
    container_mc.mask_mc.lineTo(0, h);
    container_mc.mask_mc.lineTo(0, 0);
    container_mc.mask_mc.endFill();

    // Скрываем маску (оставаясь невидимой,
    // она продолжает выполнять свою функцию)
    container_mc.mask_mc._visible = false;
  }

  /**
   * Создаем рамку вокруг изображения.
   *
   * @param   w        Ширина рамки в пикселах.
   * @param   h        Высота рамки в пикселах.
   */
  private function createBorder (w:Number,
                                 h:Number):Void {
    // Создаем рамку только при условии задания
    // допустимых значений ширины и высоты.
    if (!(w > 0 && h > 0)) {
      return;
    }

    // В контейнере создаем клип для размещения
    // рамки вокруг изображения
    container_mc.createEmptyMovieClip("border_mc", borderDepth);

    // Отрисовываем прямоугольный контур
    // в клипе рамки заданного размера и цвета
    container_mc.border_mc.lineStyle(borderThickness, borderColor);
    container_mc.border_mc.moveTo(0, 0);
    container_mc.border_mc.lineTo(w, 0);
    container_mc.border_mc.lineTo(w, h);
    container_mc.border_mc.lineTo(0, h);
    container_mc.border_mc.lineTo(0, 0);
  }

  /**
   * Загружаем JPEG-файл в средство
   * просмотра изображений.
   *
   * @param   URL   Локальный или удаленный адрес
   *                предназначенного для загрузки изображения.
   */
  public function loadImage (URL:String):Void {
    imageLoader.loadClip(URL, container_mc.image_mc);

    // Создаем текстовое поле для отображения процесса загрузки
    container_mc.createTextField("loadStatus_txt", statusDepth, 0, 0, 0, 0);
    container_mc.loadStatus_txt.background = true;
    container_mc.loadStatus_txt.border = true;
```

```
  container_mc.loadStatus_txt.setNewTextFormat(new TextFormat("Arial,
                                Helvetica, _sans", 10, borderColor, false,
                                  false, false, null, null, "right"));
  container_mc.loadStatus_txt.autoSize = "left";

  // Позиционируем текстовое поле состояния загрузки
  container_mc.loadStatus_txt._y = 3;
  container_mc.loadStatus_txt._x = 3;

  // Показываем, что идет процесс загрузки изображения
  container_mc.loadStatus_txt.text = "LOADING";
}

/**
 * Обработчик MovieClipLoader. Вызывается объектом
 * imageLoader при поступлении данных.
 *
 * @param   target        Ссылка на клип, процесс загрузки которого
 *                        отображается.
 * @param   bytesLoaded   Количество байт цели, загруженное
 *                        на данный момент.
 * @param   bytesTotal    Общий размер цели в байтах.
 */
public function onLoadProgress (target:MovieClip,
                                bytesLoaded:Number,
                                bytesTotal:Number):Void {
  container_mc.loadStatus_txt.text = "LOADING: "
      + Math.floor(bytesLoaded / 1024)
      + "/" + Math.floor(bytesTotal / 1024) + " KB";
}

/**
 * Обработчик MovieClipLoader. Вызывается объектом
 * imageLoader по окончании загрузки.
 *
 * @param   target   Ссылка на клип, для которого загрузка завершена.
 */
public function onLoadInit (target:MovieClip):Void {
  // Удаляем сообщение о процессе загрузки
  container_mc.loadStatus_txt.removeTextField();

  // Применяем маску к загруженному изображению
  container_mc.image_mc.setMask(container_mc.mask_mc);
}

/**
 * Обработчик MovieClipLoader. Вызывается объектом
 * imageLoader в случае сбоя процесса загрузки.
 *
 * @param   target      Ссылка на клип, для которого
 *                      процесс загрузки дал сбой.
 * @param   errorCode   Строка, сообщающая о причине сбоя загрузки.
 *
 */
```

```
    public function onLoadError (target:MovieClip, errorCode:String):Void {
      if (errorCode == "URLNotFound") {
        container_mc.loadStatus_txt.text = "ERROR: File not found.";
      } else if (errorCode == "LoadNeverCompleted") {
        container_mc.loadStatus_txt.text = "ERROR: Load failed.";
      } else {
        // Ловушка для обработки всех возможных кодов ошибок
        container_mc.loadStatus_txt.text = "Load error: " + errorCode;
      }
    }

    /**
     * Должен вызываться перед уничтожением экземпляра ImageViewer.
     * Дает экземпляру возможность уничтожить все созданные им ресурсы.
     */
    public function destroy ():Void {
      // Отменяем уведомления о событиях загрузки
      imageLoader.removeListener(this);
      // Удаляем клипы из Рабочего поля
      container_mc.removeMovieClip();
    }
  }
```

Назад за парты

Было просто замечательно сделать что-то своими руками. Мы действительно начали понимать, как с помощью ООП можно спроектировать и реализовать приложение (или по крайней мере часть приложения). Теперь пришло время опять вернуться к теории. В следующей главе мы узнаем, как устанавливать взаимоотношение *наследования*, один из типов взаимоотношений между двумя или более классами. Затем, в главе 7, вернемся к практическим занятиям.