Иван Портянкин

# Swing: Эффектные пользовательские интерфейсы

Издательство «Лори»

**Swing: Эффектные пользовательские интерфейсы**
*издание 2-ое*



Верстка *Глазова Т.*



Изд. № : OAI (03)
ЛР № : 07612 30.09.97 г.
ISBN 978-5-85582-305-9

Подписано в печать 20.09.2011 Формат 70 x 100/16
Бумага газетная Гарнитура Баскервиль Печать офсетная
Печ. л. 24.5 Тираж 1000 Заказ №
Цена договорная

Издательство “Лори”
123100, Москва, Шмитовский пер., д. 13/6, стр. 1 (пом. ТАРП ЦАО)
телефон для оптовых покупателей; (499)2560983
размещение рекламы; (499)2590162
www.lory-press.ru

Отпечатано в типографии ООО “Тиль-2004”
Москва, ул. Электрозаводская д. 21

# Содержание

# Введение

Java, «чистокровный» объектно-ориентированный язык с прекрасным набором библиотек, изначально переносимый между различными платформами и практически не зависящий от конкретного производителя, быстро завоевал сердца разработчиков и признание крупнейших производителей программного обеспечения. Это скорее не язык, а платформа. Со времени первого издания книги платформа Java стала еще более популярна и является, пожалуй, самым распространенным решением при создании различных сложных приложений. Она охватывает практически все области, в которых применяется программирование: платформа J2ME (Java 2 Micro Edition) предназначена для создания приложений для мобильных устройств с ограниченными ресурсами, платформа J2EE (Java 2 Enterprise Edition) позволяет создавать распределенные системы самого высокого качества и любого уровня сложности, а также пользовательские интерфейсы Web, но ядром Java можно без сомнения назвать платформу J2SE (Java 2 Standard Edition) — именно с нее все начиналось, и эта часть Java по-прежнему является центральной. С помощью J2SE вы сможете создавать приложения для самых разнообразных нужд пользователей, это могут быть текстовые редакторы, электронные таблицы, клиенты для распределенных систем, созданных по технологии J2EE, апплеты, работающие в браузере как части HTML-страниц, удобные интерфейсы к базам данных JDBC и т. д.

Важнейшую часть любой программы составляет ее пользовательский интерфейс. На самом деле пользователя мало интересуют сложные алгоритмы обработки данных и удачные находки, реализованные внутри программы, также мало его занимают и технологии, примененные для создания самого приложения и его пользовательского интерфейса. Пользователь видит только то, что видит, и именно этот факт следует учитывать при решении всех задач по проектированию и созданию пользовательского интерфейса. Интерфейс должен быть максимально быстрым и максимально удобным для пользователя, который должен также получать и эстетическое удовольствие от работы с вашей программой.

Сегодня особенно популярны пользовательские интерфейсы HTML в сети Web, которые с появлением динамического обновления данных с помощью AJAX во многом вытесняют стандартные приложения операционных систем. Написание таких интерфейсов бывает трудоемким и требует взаимодействия распределенных сетевых систем, в то время как стандартные интерфейсы операционных систем зачастую помогают быстрее создать полнофункциональное приложение, особенно если вам помогает библиотека высокого качества и вам необходим полный доступ к ресурсам системы.

В этой книге мы увидим, на что способны современные версии Java при создании пользовательского интерфейса приложений. В вашем распоряжении окажутся и мельчайшие детали внешнего вида программ, и средства обработки данных, которые должен будет отображать компонент. Java располагает множеством компонентов, способных удовлетворить самый взыскательный вкус и вывести на экран самые замысловатые данные. И как бы сложно ни было то, что вы намереваетесь создать, изучение и освоение инструментов, которые окажутся у вас в руках, не потребует титанических усилий. Поняв основные принципы работы компонентов и разобрав несколько простых примеров, вы осознаете, что получили не обычный набор библиотек для создания пользовательского интерфейса, а настоящую палитру художника, которая позволит нарисовать великолепную картину всего парой несложных мазков.

## Java Foundation Classes

Одной из самых больших и самых важных частей платформы J2SE является набор библиотек под общим названием Java Foundation Classes (JFC). Именно эти библиотеки предназначены для создания эффектных и отточенных пользовательских интерфейсов. Ниже перечислены библиотеки, входящие в набор Java Foundation Classes:

- *Swing*. Важнейшая часть JFC; содержит компоненты для создания пользовательского интерфейса, такие как таблицы и текстовые поля, а также инструменты для работы с этими компонентами. Библиотека Swing, как мы вскоре увидим, великолепно спланирована и реализована, она способна дать все, что может пожелать разработчик пользовательского интерфейса. Если вы создаете лаконичный «деловой» интерфейс, вам понадобится всего пара строк кода. Если же вы «раскрашиваете» ваше приложение всеми красками радуги и создаете для него сложный нестандартный интерфейс, вам понадобится немного больше усилий. Но ничего невозможного для Swing нет.
- *Java2D*. Вторая по важности в JFC библиотека, позволяющая применять в своем приложении современную двухмерную графику, в том числе аффинные преобразования, дробные координаты, сглаживание, расширенные операции с растрами и многое другое. Библиотека Java2D встроена в Swing — все компоненты последней используют ее для вывода своих данных на экран, хотя и неявно.
- *Accessibility*. Набор классов и интерфейсов, следующих промышленному стандарту и наделяющих приложения средствами поддержки пользователей с ограниченными возможностями. С помощью Accessibility вы сможете, например, передать текстовое описание интерфейса системе синтеза речи, что позволит работать с вашей программой пользователям с нарушениями зрения. Замечательно то, что во всех компонентах Swing интерфейсы Accessibility уже реализованы и после небольшой настройки самый обычный интерфейс моментально превращается в специализированный, ориентированный на пользователей с ограниченными возможностями. Для приложений высшего уровня качества это свойство Swing просто незаменимо.
- *Drag'n'Drop*. Дополнение, позволяющее вашему приложению взаимодействовать с приложениями операционной системы пользователя или другими Java-приложениями с помощью технологии перетаскивания (drag and drop). Подобная возможность очень удобна для пользователя и позволяет ему сразу же забыть о том, что приложение написано на Java и не имеет практически никаких связей с его операционной системой. Напрямую с этим дополнением работать, как правило, не приходится, так как Swing берет многие детали процесса на себя.

Ядром Java Foundation Classes без всяких сомнений является библиотека Swing, остальные части набора классов так или иначе встроены в нее или предоставляют для компонентов этой библиотеки дополнительные возможности. Создавать пользовательский интерфейс своих приложений вы будете именно с помощью Swing, и именно эту библиотеку мы будем изучать. Для начала мы познакомимся с основными свойствами Swing, ее общей структурой, а затем перейдем к компонентам библиотеки. Они позволят создавать фантастические пользовательские интерфейсы всего одним мановением руки.

## Структура книги

Большая часть книги посвящена описанию компонентов Swing и примерам их использования. Ну а начнем мы с исследования основных механизмов библиотеки, тех

«винтиков», на которых работают все компоненты Swing. После их изучения секреты библиотеки больше не будут для нас загадкой.

- **Глава 1. Основные концепции.** В этой главе мы взглянем на Swing «с высоты птичьего полета» и раскроем для себя главные замыслы разработчиков библиотеки, которые оказали наибольшее влияние на ее архитектуру. Вы увидите, что в основе Swing лежит библиотека AWT, узнаете, как это влияет на библиотеку Swing и ее компоненты, рассмотрите архитектуру JavaBeans и механизмы разделения модели, вида и контроллера. Не ускользнут от нашего внимания и подключаемые внешний вид и поведение компонентов Swing, а также поддержка пользователей с ограниченными возможностями.

- **Глава 2. Модель событий.** Мы узнаем, как в Swing обрабатываются события. Сначала рассмотрим простые примеры, а затем перейдем к скрупулезному исследованию «начинки» системы обработки событий. Будут рассмотрены вопросы написания слушателей, низкоуровневые события, техника обработки событий и работы с мышью. Этой главы будет вполне достаточно для обычной обработки всех событий.

- **Глава 3. За кулисами системы обработки событий**. В этой главе мы разберем детали системы событий, узнаем про поток рассылки событий EventDispatchThread и очередь событий EventQueue, правила работы с потоками. В конце главы мы узнаем «золотое правило» системы обработки событий Swing; оно совсем не сложно, но позволяет писать по-настоящему быстродействующие программы, невзирая на все доводы скептиков о не слишком высокой производительности пользовательских интерфейсов, созданных с помощью Swing.

- **Глава 4. Рисование в Swing.** Система рисования Swing реализована в основном в базовом классе JComponent. Прежде всего рассмотрен механизм рисования и его применение в реальных приложениях. Мы также коснемся скрытого «двигателя» системы рисования, класса RepaintManager.

- **Глава 5. Внутренние винтики Swing.** Остальные скрытые механизмы Swing, реализованные в базовом классе JComponent, незаметно работают «за кулисами» библиотеки, освобождая вас от рутинной работы. Как правило, нам не надо знать, как и что происходит во внутренних механизмах библиотеки, , но в сложных ситуациях без этого не обойтись. Рассмотрены самые сложные части Swing, реализация которых во многом объясняет работу всех компонентов библиотеки.

- **Глава 6. Контейнеры высшего уровня.** После создания интерфейса его необходимо вывести на экран. Для этого предназначены особые компоненты Swing, единственные, имеющие связь с операционной системой; они называются контейнерами высшего уровня. Помимо обычных свойств они обладают некоторыми особенностями, которые мы досконально обсудим в данной главе. Мы также узнаем о средствах создания многодокументных интерфейсов (MDI).

- **Глава 7. Искусство расположения.** Менеджер расположения — это ассоциированный с контейнером алгоритм, который определяет, как компоненты должны располагаться в контейнере. Стандартные менеджеры расположения просты, но получение с их помощью нужного расположения компонентов часто сродни искусству. В этой главе мы постараемся в полной мере овладеть тайнами этого процесса, а также напишем несколько вспомогательных программных инструментов, призванных облегчить создание сложных расположений компонентов. Сравниваются разные подходы к расположению компонентов на примере элемента реального интерфейса.

- **Глава 8. Вывод вспомогательной информации.** Компоненты, применяемые для создания пользовательского интерфейса программы, можно условно разделить на две части: одни компоненты требуются для получения информации от пользователя и для вывода данных программы, а другие призваны облегчить работу пользователя, обеспечивая его вспомогательной информацией об интерфейсе программы и задачах, которые он решает. В этой главе мы будем говорить именно о компонентах, позволяющих пользователю получить вспомогательную информацию, о надписях JLabel, подсказках и рамках.

- **Глава 9. Элементы управления.** Элементы управления используются в интерфейсе повсюду, с их помощью пользователь влияет на ход выполнения программы. Арсенал предоставляемых Swing элементов управления велик и разнообразен, но пользоваться им просто, в этом вы убедитесь после прочтения данной главы.

- **Глава 10. Меню и панели инструментов.** В меню собраны все команды, необходимые для работы с приложением. Они помогают быстрее ознакомиться с возможностями приложения, поэтому начать работу, имея под рукой хорошо организованную и интуитивно понятную систему команд, гораздо проще. Теперь создание самого сложного меню в Swing перестанет быть проблемой для вас. На «переднем крае» приложения находятся и панели инструментов, которые содержат набор элементов управления для выполнения наиболее часто встречающихся действий. В этой главе мы разберем компонент JToolBar, применяемый в Swing для создания панелей инструментов.

- **Глава 11. Списки.** В этой главе рассматриваются обычные (JList) и раскрывающиеся (JComboBox) списки библиотеки Swing. В обычном списке JList выводятся сразу несколько альтернатив, и пользователь может быстро сравнить их и выбрать то, что ему необходимо, причем, как правило, допускается множественный выбор. Раскрывающийся список JComboBox чаще всего используется для выбора одного варианта из многих (выбранный вариант появляется в поле списка), а также обычно разрешается ввод пользователем собственных значений. Эта глава раскроет все секреты списков Swing, и вы сможете убедиться, что для них нет ничего невозможного. Списки Swing могут отображать самые экзотичные на свете данные.

- **Глава 12. Диапазоны значений.** Эта глава посвящена компонентам, обеспечивающим возможность регулировки (плавной или ступенчатой) данных в некотором диапазоне или наглядно представляющим на экране данные из некоторого диапазона. Ползунки JSlider позволяют выбрать значение в некотором ограниченном диапазоне числовых данных. Индикаторы процесса JProgressBar в наглядной форме показывают, какая часть ограниченной задачи уже завершена. Наконец, счетчики JSpinner дают возможность выбора произвольного значения среди некоторого набора альтернатив, который может быть и неограниченным. Как мы увидим, все эти компоненты Swing обладают завидной гибкостью и способны на многое.

- **Глава 13. Управление пространством.** С недостатком места в контейнере призваны справляться специальные компоненты, которые и рассматриваются в данной главе. К таким компонентам относятся панель с вкладками JTabbedPane, разделяемая панель JSplitPane и панель прокрутки JScrollPane. Панель с вкладками обеспечивает эффективное размещение в одном контейнере нескольких вспомогательных панелей с компонентами, разделяемая панель дает возможность динамически, по мере необходимости, перераспределять пространство контейнера между двумя компонентами. Панель прокрутки, незаменимый участник любого пользовательского интерфейса, с легкостью вмещает в себя компоненты даже самых гигантских размеров и позволяет мановением руки (или строкой кода) перемещаться к любой их части.

- **Глава 14. Стандартные диалоговые окна.** Во всех современных графических системах имеется набор так называемых стандартных диалоговых окон, позволяющих быстро выводить для пользователя разнообразную информацию или же получать эту информацию от него. В этой главе мы увидим, какие стандартные диалоговые окна предлагает нам Swing. Мы подробно рассмотрим класс JOptionPane, десятки методов которого позволят быстро и эффектно вывести на экран несложные сообщения, ввести нужные данные или запросить у пользователя подтверждение операции. Компоненты JFileChooser и JColorChooser незаменимы для выбора файлов и цветов.

- **Глава 15. Уход за деревьями.** Иерархические отношения данных принято отображать в специальных компонентах пользовательского интерфейса, называемых деревьями. Деревья в Swing, реализованные компонентом JTree, обладают впечатляющими возможностями, и данная глава полностью посвящена им. Мы изучим далеко не самый простой процесс создания модели дерева, овладеем всеми тайнами стандартных узлов и стандартной модели деревьев Swing — эти модели чаще всего и применяются при создании интерфейсов. Далее нас ждет подробное изучение модели выделения дерева и настройка всех аспектов отображения и редактирования узлов деревьев Swing.

- **Глава 16. Текстовые компоненты.** Одной из самых впечатляющих частей библиотеки Swing является пакет javax.swing.text, обеспечивающий ее средствами для работы с текстом. Благодаря этому пакету в вашем арсенале появятся несколько текстовых компонентов, реализующих любые механизмы для ввода и редактирования текста, от самых простых до чрезвычайно изощренных. В этой главе мы познакомимся со всеми текстовыми компонентами и их основными возможностями. Кроме того, мы затронем вопросы, относящиеся к внутренней реализации текстовых компонентов Swing: изучим основные свойства модели текстовых компонентов Document и многое другое.

- **Глава 17. Таблицы.** Пожалуй, настоящая «звезда» библиотеки Swing — таблица JTable. Таблица JTable дает вам возможность с легкостью выводить двухмерную информацию, расположенную в виде строк и столбцов, без особых усилий настраивать и сортировать данные для таблицы, выводить их в любом необходимом виде, управлять заголовками таблицы и ее выделенными элементами и с небольшими усилиями делать еще очень многое. Мы подробно рассмотрим процесс создания модели таблицы и варианты применения стандартных моделей, изучим модели выделения и весьма полезную модель столбцов, узнаем способы самой тонкой настройки внешнего вида столбцов, ячеек, а также увидим, как настраивается процесс редактирования ячеек таблицы.

- **Глава 18. Круговорот данных**. Данные — это самое драгоценное для пользователей, и они очень ценят, когда приложение бережно к ним относится. В этой главе мы узнаем, как обеспечить обмен данными приложений Swing и остальных приложений системы с помощью буфера обмена и визуального перетаскивания, а также как включить в приложение поддержку отмены и повтора операций.

Книга позволит получить именно ту информацию, которая вам необходима. Если вы занимаетесь созданием не самых сложных интерфейсов и вам просто нужны сведения о том или ином компоненте Swing, обращайтесь к соответствующей главе книги. Если у вас возникнут вопросы об устройстве одного из внутренних механизмов библиотеки, вы всегда сможете найти ответ в одном из разделов первых нескольких глав книги. Ну а полностью овладеть всей магией и мощью, которую способна предоставить библиотека Swing, вы сможете, прочитав последовательно от начала и до конца всю книгу.

## Для кого предназначена книга

Эта книга — не для новичков в программировании. Подразумевается, что вы имеете некоторый опыт программирования на Java, скорее всего, уже пробовали создавать на этом языке первые пользовательские интерфейсы с помощью AWT, Swing или средств быстрой разработки (Rapid Application Development, RAD), таких как Eclipse или NetBeans. Поэтому представленные в книге конструкции языка Java, а также термины, применяемые при программировании и создании пользовательских интерфейсов, не введут вас в заблуждение. Замечательно, если вы владеете навыками объектно-ориентированного проектирования и анализа, в этом случае понимание архитектуры библиотеки придет гораздо быстрее и работа с ней станет истинным удовольствием. Отсутствие подобных навыков не помешает вам в полной мере изучить Swing, но только наличие некоторого опыта в объектно-ориентированном проектировании даст возможность оценить всю элегантность библиотеки.

## Интерактивная документация

Материал книги ни в коем случае не повторяет и не копирует информацию, которая содержится в интерактивной документации JDK для библиотеки Swing (описание всех классов, методов и полей библиотеки, как правило, довольно краткое). Вместо этого мы изучим основы Swing, рассмотрим главные механизмы библиотеки, а затем, разрабатывая небольшие и понятные примеры, овладеем всеми тайнами компонентов Swing. Так вы составите для себя полную картину возможностей Swing, а интерактивная документация станет вашим верным компаньоном, способным подсказать правильное название метода и список его параметров. На самом деле при работе с библиотекой Swing необходимость в документации возникает не часто, поскольку библиотека удачно спроектирована, а названия свойств и методов выбраны очень точно (чему немало способствует архитектура JavaBeans, которую мы рассмотрим в главе 1). При чтении глав, посвященных конкретным компонентам Swing, рекомендуем иметь «под рукой» интерактивную документацию, с ее помощью вы еще быстрее познакомитесь со всеми возможностями компонента.

## Язык шаблонов проектирования

Везде, где возможно, архитектура библиотеки Swing и ее компонентов описывается с помощью языка *шаблонов проектирования* (design patterns), прекрасного средства описания любого (обобщенного или подробного) объектно-ориентированного решения. Шаблоны проектирования представляют собой набор наиболее удачных и проверенных временем схем объектно-ориентированного проектирования и анализа, позволяющих быстро, элегантно и максимально гибко решать часто возникающие задачи. Библиотека Swing буквально «прошита» шаблонами проектирования, и это еще раз подтверждает высокий уровень, на котором она спроектирована. Владея шаблонами проектирования, вы быстро сможете их обнаружить практически в каждом аспекте Swing, и это поможет работать с библиотекой максимально эффективно. Нет ничего страшного, если вы незнакомы с шаблонами проектирования, — вы сможете пользоваться всей мощью Swing и без них, но лучше незамедлительно приступить к их изучению.

## Примеры

Книга содержит большое количество примеров, призванных максимально ускорить и упростить знакомство со всеми аспектами работы библиотеки Swing и ее компонентов. Примеры намеренно сделаны очень простыми и по возможности лаконичными: так они акцентируют ваше внимание на том, *как* что-то можно сделать, а не потрясают вас великолепным интерфейсом, который можно создать с помощью более сложного

и объемного кода. Создание великолепных сложных интерфейсов после прочтения книги не составит для вас труда, а примеры книги помогут понять все то, о чем в ней рассказывается. Эти примеры будут служить неплохим справочником и после прочтения книги: открыв страницу с нужным примером, вы мгновенно поймете, как работать с тем или иным компонентом Swing.

Все примеры можно найти на сайте книги www.IPSoftware.ru. С компиляцией и запуском примеров не должно быть никаких проблем — этот процесс намеренно сделан настолько элементарным, насколько это вообще может быть. Вам нужно зайти в папку с номером главы и вызвать компиляцию всех исходных файлов (javac *.java), после чего их можно запускать. Никаких сред, средств сборки, дополнительных инструментов, кроме пакета JDK, не требуется. Лишь в нескольких примерах, которые можно пересчитать на пальцах одной руки, могут понадобиться отдельные библиотеки, и об этом обязательно упоминается при описании примера.

По мере возможности в различных главах книги мы создаем полезные инструменты и новые расширенные компоненты, которые могут сослужить неплохую службу при создании собственных интерфейсов. Все подобные инструменты размещены в особом пакете com.porty.swing. Специально для вашего удобства в состав исходных текстов книги входит архив в формате JAR (booktools.jar) со всеми инструментами, созданными в данной книге. Вы сможете включить этот архив в список своих классов и легко применять созданные нами в книге инструменты в своих программах. Найти этот архив также можно на сайте www.IPSoftware.ru.

## *Снимки экрана*

Ряд примеров проиллюстрирован изображением экранов. Все снимки были сделаны с внешним видом Java по умолчанию — Metal, который, увы, не отличается элегантностью и неотразимостью внешнего вида. Но пусть это вас не пугает, поменять внешний вид на один из доступного нам широкого выбора, к примеру JGoodies Looks или Subtance, легко. Достаточно указать дополнительный параметр для виртуальной машины Java, -Dswing.defaultlaf=<имя класса внешнего вида>, подставить туда ваш любимый внешний вид, и примеры будут выглядеть намного лучше.

## *Java 7*

Новый пакет разработки Java 7 (JDK 1.7) во время написания книги находился в завершающей стадии, для написания примеров использовались его предварительно доступные версии. Везде, где было возможно, были упомянуты и описаны новые возможности Swing в Java 7.

## *Изменения во втором издании*

Первое издании книги получило множество самых замечательных отзывов от читателей, которые предложили осветить во втором издании дополнительные интересующие их вопросы и присылали свои идеи, за что им огромное спасибо. Структура книги осталась прежней, и самым главным остается пошаговое изучение краеугольных камней, которые лежат в основе Swing, с помощью скрупулезных объяснений и подробных, простых примеров.

Вся информация была заново просмотрена и проверена, все новое, появившееся в Swing за время, прошедшее после выхода первого издания, было добавлено в соответствующие главы и разделы. Особенно тщательно рассматривается работа в многозадачном окружении, поскольку это жизненно необходимо и для высокой скорости Swing-приложения, и для его безошибочной работы.

Многие читатели просили добавить в книгу снимки экрана (screenshots), которые можно увидеть при запуске примера, это было сделано там, где необходимо.

Ну и конечно, в книге есть новые главы, открывающие новые горизонты при работе со Swing и с пользовательскими интерфейсами в общем. Они позволят создать действительно эффектные приложения.

## *Благодарность команде Sun (или Oracle)*

Хочется выразить свою огромную благодарность команде разработки библиотеки Swing компании Sun (впрочем, на момент окончания книги она уже стала частью компании Oracle). Большая часть этой команды находится у нас в России, в Санкт-Петербурге, ее участники рецензировали самые важные и сложные части этой книги и внесли множество неоценимых предложений и замечаний, а также помогли автору с некоторыми затруднениями. Особенно хочется поблагодарить Александра Поточкина, который к тому же ведет свой дневник о разработке приложений Swing в Интернете и является автором прекрасного инструмента JXLayer, который стал частью JDK и описан в этой книге в главе 6. Адрес его дневника, где размещено множество полезной информации, вы можете узнать на сайте книги.

## *Книга в Интернете*

Свои отзывы о материале книги, предложения о дополнениях, комментарии, информацию о найденных ошибках вы сможете разместить на сайте www.IPSoftware.ru. Там же вы сможете задать свои вопросы автору, получить необходимые консультации, найти все исходные тексты данной книги, вспомогательные инструменты для работы с компонентами Swing, а также узнать о будущих планах и новых разработках в области Swing и Java Foundation Classes. Сайт книги к тому же описывает список самых интересных на данный момент дневников (блогов), библиотек, дополнительных внешних видов и инструментов, посвященных библиотеке, зная которые можно немало добавить к своим приложениям Swing.

# Глава 1. Основные концепции

С этой главы мы начнем свое путешествие по библиотеке Swing, которая является самой большой и самой важной частью набора классов Java Foundation Classes, составляя ядро этого набора. Мы узнаем, чем руководствовались разработчики библиотеки, какие цели они преследовали, как эти цели были ими достигнуты. Сначала мы обсудим внутреннее устройство библиотеки и наиболее важные моменты ее реализации, не останавливаясь на деталях — так нам будет проще оценить масштабы и возможности Swing, которые поистине безграничны и вызывают искреннее восхищение. Затем, в последующих главах, мы подробнее остановимся на компонентах Swing.

Итак, что же такое Swing? Если говорить кратко, то это набор графических компонентов для создания пользовательских интерфейсов приложений и апплетов, а также вспомогательные классы и инструменты для работы с этими компонентами. В принципе, это лаконичное заявление довольно точно описывает библиотеку Swing, но оно не должно вводить вас в заблуждение. Легкость программирования, мощь тогда, когда она нужна, и возможность настроить все по своему вкусу — все это относится к Swing как к никакой другой библиотеке.

Обычно, если кто-то заявляет о безграничных возможностях новой библиотеки, это значит, что ее изучение, освоение и последующая работа будут непростыми. Это правило не срабатывает в случае с библиотекой Swing — работать с ней можно, не зная ровным счетом ничего о том, как она устроена, а ведь внутри нее скрыты совсем непростые механизмы. Например, внешний вид компонентов Swing можно легко изменить одной строчкой кода, и для этого не нужно знать подробностей. Только если вы соберетесь придать компонентам некий особенный (нестандартный) внешний вид, вам потребуются детали. Можно годы работать с библиотекой Swing, просто создавая компоненты, добавляя их в окна и обрабатывая события, и ничего не знать о скрытом в ней мощном механизме программирования моделей. Одним словом, Swing соответствует уровню каждого: и новичку, и опытному программисту будет казаться, что эта библиотека создана специально для него.

Последовательно читать все то, что написано в этой и двух следующих главах, вовсе не обязательно. Это особенно верно для тех, кто начал программировать на Java не так давно. Можно просто выбирать тот материал, который в данный момент интересует вас больше всего, читать и тут же применять новые знания в своих приложениях. Однако при достижении некоторого уровня знаний лучше остановиться и попытаться осмыслить библиотеку в целом. Именно этим мы сейчас и займемся.

Поначалу библиотеки Swing в Java вообще не было. Вместо нее использовалась библиотека AWT, общая идея которой была весьма неплохой. Но вот реализация этой идеи подразумевала довольно ограниченный набор компонентов, что очень сильно вредило Java как языку создания *настольных приложений* (desktop applications). Многие решения, принятые разработчиками Swing, были направлены на то, чтобы расширить излишне лаконичный набор компонентов AWT и сделать библиотеку Swing пригодной для создания с минимальными усилиями современных пользовательских интерфейсов. По признанию большинства программистов, команде Swing это удалось с блеском. Впрочем, по прошествии некоторого времени «шишки» AWT забылись, и сама она теперь лишь является верной помощницей Swing.

Важнейшим отличием Swing от AWT является то, что компоненты Swing вообще не нуждаются в поддержке операционной системы и поэтому гораздо более стабильны и быстры. Такие компоненты в Java называются *легковесными* (lightweight), и понимание основных принципов их работы во многом объяснит работу Swing. К сожалению, в документации и большей части изданных книг этим компонентам уделяется очень мало внимания (чаще всего определение легковесных компонентов состоит в том, что они «не используют код, зависящий от платформы»). Мы постараемся увидеть и «пощупать» легковесные компоненты, чтобы понять, почему же они появились в Java.

Также мы узнаем о таком «волшебном» свойстве Swing, как подключаемые внешний вид и поведение. Это свойство позволяет компонентам Swing вести себя максимально гибко, приспосабливаясь к любым условиям, принимать необходимый вид, а также предоставляет программистам такой удобный инструмент, как *модели*. Реализованы подключаемые внешний вид и поведение в Swing на основе уже ставшей классической архитектуры MVC (см. далее). После этого мы узнаем об еще одном «секретном оружии» Swing — поддержке специальных средств для пользователей с ограниченными возможностями, причем поддержка эта полностью реализована в самой библиотеке и не требует от программиста дополнительных усилий. Итак, начнем.

## *В начале было... AWT*

Основой библиотеки Swing, тем тонким слоем, что лежит между ней и зависящим от платформы кодом, является библиотека AWT (Abstract Window Toolkit — инструментарий для работы с различными оконными средами). В отличие от библиотеки Swing, которая появилась в Java версии 1.1 как нестандартное дополнение и стала частью платформы только с выходом Java 2, пакет java.awt входил в Java с самого первого выпуска. Поначалу именно он предназначался для помощи в создании пользовательских интерфейсов.

Исходное назначение AWT — предоставить набор графических компонентов, который вобрал бы в себя наиболее характерные черты современных элементов управления и позволил бы однократно создавать пользовательские интерфейсы, подходящие для любой платформы. Компоненты AWT на самом деле не выполняют никакой работы и очень просты — это просто «Java-оболочки» для элементов управления той операционной системы, на которой работает пользователь. Все запросы к этим компонентам незаметно перенаправляются к операционной системе, которая и выполняет всю работу. Чтобы сделать классы AWT независимыми от конкретной платформы, каждому из них соответствует своеобразный *помощник*[1] (peer), который и работает с этой платформой. Для того чтобы встроить в AWT поддержку новой платформы, нужно просто переписать код этих помощников, а интерфейс основных классов остается неизменным[2].

На рис. 1.1 показана иерархия классов AWT. Диаграмма следует формату унифицированного языка моделирования (Unified Modeling Language, UML): сверху располагаются базовые классы, а ниже классы, унаследованные от базовых.

---

[1] Дословный перевод слова «peer» — «равный, имеющий те же права», и так оно и есть — классы AWT и их помощники разделяют обязанности поровну: первые предоставляют услуги программистам-клиентам Java, а вторые незаметно связывают эти услуги с операционной системой. Но с точки зрения программиста-клиента (а мы стоим именно на такой позиции) удобнее использовать слово «помощник».

[2] Эту концепцию, использованную в AWT, можно описать и языком шаблонов проектирования. Это не что иное как мост (bridge) — программист взаимодействует с одной иерархией классов, а работа фактически выполняется в другой, параллельной иерархии. Последняя часто скрывается от глаз программиста, что позволяет легко изменять ее.

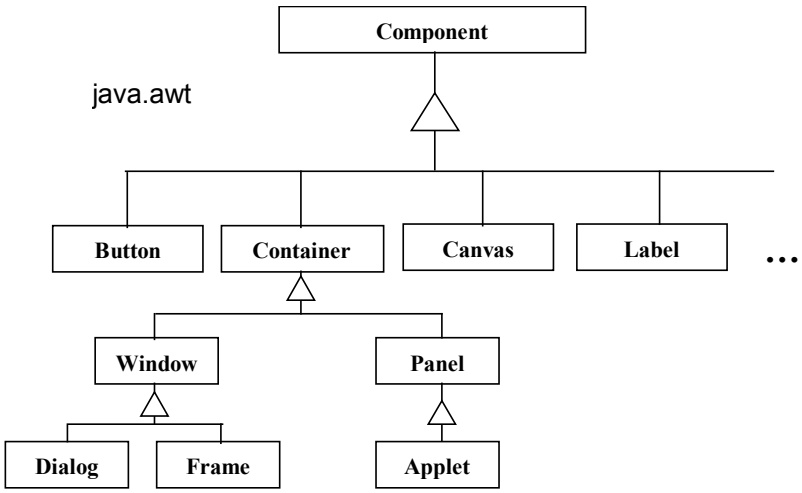



**Рис. 1.1.** Исходная иерархия классов AWT

Как видно, базовые свойства и поведение всех графических компонентов описаны в абстрактном классе Component, который является ядром библиотеки AWT. Почти все компоненты пользовательского интерфейса (за исключением меню) унаследованы от этого класса. Важным частным случаем компонента в AWT является так называемый *контейнер*, отличительные черты которого описаны в классе Container (также абстрактном). Контейнер отличается от обычных компонентов тем, что может содержать в себе другие компоненты (а значит и другие контейнеры, что позволяет организовывать пользовательские интерфейсы любой сложности). Одним из самых необычных свойств AWT можно назвать способность контейнеров располагать компоненты по определенному алгоритму, определяемому *менеджером расположения* (layout manager)[3]. *Контейнерами высшего уровня*, то есть контейнерами, содержащими все остальные элементы пользовательского интерфейса, служат окна с рамкой, диалоговые окна, а также апплеты, запускаемые из браузера. Они представлены классами Frame, Dialog и Applet.

Классы AWT, представляющие собой компоненты пользовательского интерфейса, такие как кнопки (Button) и надписи (Label), — очень небольшие и простые в использовании, как того и хотели создатели AWT. Вы можете смело работать с ними, зная, что на любой платформе компоненты вашего пользовательского интерфейса будут выглядеть так, как решит операционная система, то есть неотличимо от других ее приложений. Как мы уже знаем, об этом заботятся помощники компонентов. Их обязанности описаны в соответствующих интерфейсах (имя каждого можно получить, добавив к имени компонента слово «Peer», например ButtonPeer). Для поддержки библиотекой AWT новой платформы оставается только написать реализующие эти интерфейсы классы, работающие с данной платформой (например, в пакете JDK для Windows кнопки создаются с помощью класса WButtonPeer, реализующего интерфейс ButtonPeer). Такой механизм позволяет обеспечить переносимость графических приложений, написанных на Java.

Впрочем, несмотря на то что AWT реальна и действительно позволяет, как может показаться, писать все те же пользовательские интерфейсы, ей пользуются крайне мало, и служит она лишь основой для библиотеки Swing. Все дело в том, что компоненты AWT

[3] Менеджер расположения — это особая служба, даже «стратегия», если думать на языке шаблонов, мы подробно обсудим, как и почему она используется в Java, в главе 7, которая полностью посвящена этому вопросу.

на самом деле являются компонентами операционных систем, причем самыми простыми, такими, чтобы на любой системе они могли действовать примерно одинаково. Однако системы сильно различаются, и это «пересечение множеств» интерфейсов различных операционных систем, которому старается следовать AWT, слишком нефункционально для современных интерфейсов.

Почему же AWT является основой Swing? Связующим звеном, соединяющим Swing с реальным миром, являются легковесные компоненты AWT. Сами компоненты AWT имеют связь с операционной системой, на которой работает ваше приложение. Даже если вы создаете собственный компонент, наследуя от любого класса компонента AWT, вы все равно остаетесь в жестких рамках этих взаимоотношений, потому что ваш компонент представляет собой маленькое окно, полностью принадлежащее операционной системе. Как мы уже знаем, это влечет за собой массу проблем, прежде всего в плане гибкости ваших компонентов. Однако Java — полноценный и весьма мощный язык программирования, и для создания чего-либо на нем вовсе не обязательно обращаться к ресурсам операционной системы. Итак, *легковесный компонент* — это просто область в пространстве экрана, занимаемом вашим Java-приложением. Главные его атрибуты — это координаты в окне и размер. Для операционной системы легковесный компонент вообще не существует, потому что представляет собой всего лишь часть какого-то окна. Всю работу по поддержке легковесных компонентов берут на себя библиотека AWT и виртуальная машина Java. Для программиста не существует никакого отличия между легковесными и обычными компонентами, которые по аналогии стали называть *тяжеловесными* (heavyweight), то есть имеющими связь с операционной системой и представленными в своем собственном окне. Программирование абсолютно одинаково как для первых, так и для вторых. Конечно, где-то сказка должна заканчиваться, и связь с системой все же существует. Связь эту стали обеспечивать тяжеловесные контейнеры (обычно окна и апплеты), в которых вы размещали свои легковесные компоненты. Именно они и встали на конце цепочки, прорисовывая легковесные компоненты в своем пространстве и не требуя для этого никаких дополнительных усилий от программиста.

Благодаря избавлению от сомнительных связей с разнообразными операционными системами легковесные компоненты полностью находятся во власти программиста на Java и могут следовать всем его указаниям. Они не ограничены тем или иным набором функций извне. Конечно, мы лишаемся части уже реализованной функциональности операционных систем (уж кнопки-то были написаны не один десяток лет назад) и вынуждены переписать базовые основы с нуля (точнее компания Sun была вынуждена это сделать при написании Swing), даже скопировать внешний вид компонентов[4], но гибкость и 100% реализация на Java полностью оправдывает затраты, позволяя придать компонентам любой внешний вид и предоставить самый современный, широчайший набор функций.

Легковесные компоненты позволили совершенно безболезненно перейти от работы с прямоугольными компонентами, залитыми цветом фона, к работе с компонентами абсолютно любой, даже самой фантастичной, формы. Действительно, прорисовка легковесного компонента — это всего лишь прорисовка области окна вашего приложения, и ни операционная система, ни виртуальная машина Java не заставляли закрашивать эту область одним цветом. Вы вообще могли не рисовать свой компонент, несмотря даже на то, что он присутствовал на экране. Такая гибкость, позволяющая, в частности, создавать прозрачные компоненты и компоненты любой формы, дорогого стоила — ведь все эти возможности доставались программисту практически «даром», не требуя хитроумных приспособлений для своей реализации.

[4] Представьте, насколько увлекательно было перерисовывать уже давно нарисованные кнопки и диалоги Windows и Solaris на языке Java. Работа не из приятных, поэтому до сих пор трудно сказать, что имитация внешнего вида операционных систем в Swing верна на все сто.

Итак, мы плавно подошли к самой библиотеке Swing. Уже понятно, что компоненты этой библиотеки, будучи основаны на AWT, являются легковесными. Давайте обсудим столпы Swing немного поподробнее.

## Компоненты Swing — это легковесные компоненты AWT

Как вы уже поняли, создатели новой библиотеки пользовательского интерфейса Swing не стали «изобретать велосипед» и в качестве основы для своей библиотеки выбрали AWT. Хоть порка AWT и стала очень популярным занятием с момента выхода самой первой версии Java (и не безосновательно), определенные достоинства у этой библиотеки все же были. Вспомогательная иерархия помощников позволяла без особых усилий переносить AWT практически на любую платформу, а это весьма достойное качество.

Конечно, речь не шла об использовании конкретных тяжеловесных компонентов AWT (представленных классами Button, Label и им подобными). Они уже достаточно скомпрометировали себя близкой связью с операционной системой, которая не позволяла Java-приложениям и библиотекам в полной мере управлять этими компонентами. Нужную степень гибкости и управляемости обеспечивали только легковесные компоненты.

Коротко обсудив библиотеку AWT, мы с вами выяснили, что же такое легковесные компоненты и чем они хороши. Создать легковесный компонент в AWT можно двумя способами: унаследовать класс своего компонента от абстрактного класса Component (и получить обычный легковесный компонент) или унаследовать свой компонент от уже существующего наследника класса Component, другого абстрактного класса Container (и получить контейнер, тоже легковесный). По диаграмме наследования, представленной на рис. 1.2, видно, какой путь избрали создатели Swing.

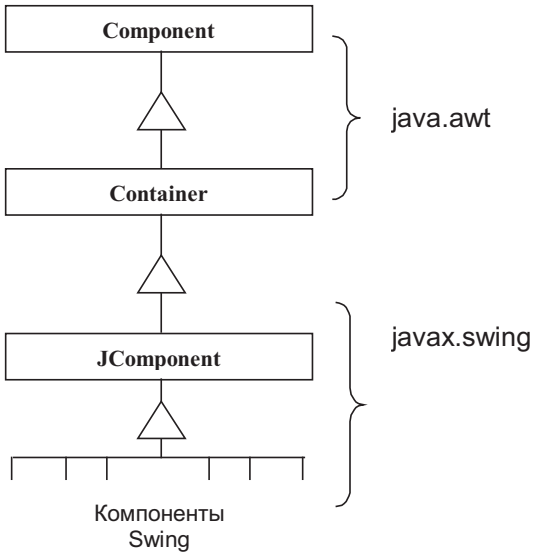



**Рис. 1.2.** Диаграмма наследования компонентов библиотеки Swing

Легко видеть, что библиотека Swing обзавелась новым базовым классом, который стал называться JComponent и который был унаследован от абстрактного класса Container, определяющего поведение контейнеров AWT. Таким образом, все знания разработчиков о компонентах и контейнерах AWT автоматически переходили в Swing, что значи-

тельно облегчало знакомство с новой библиотекой, особенно для тех разработчиков, которые уже пытались создавать Java-приложения с помощью старых библиотек. Создатели Swing постарались максимально облегчить переход от AWT к новой библиотеке, практически полностью сохранив в ней имена классов AWT и их методов. Все компоненты AWT имеют своих наследников в Swing, и имена классов этих компонентов отличаются лишь префиксом «J» (например, тяжеловесная кнопка Button имеет в Swing свой легковесный аналог — кнопку JButton). Новые классы Swing также имели полный набор методов старых классов AWT[5], что делало процесс перехода с AWT на Swing простым и безболезненным (достаточно было импортировать пакет javax.swing и добавить к именам классов букву «J»). Все это обусловило молниеносное распространение новой библиотеки и еще сильнее подогрело интерес к ней.

После знакомства с иерархией классов AWT может показаться странным, что в иерархии Swing класс JComponent унаследован непосредственно от базового класса контейнеров Container. Почему разработчики Swing не создали два разных подкласса: один для обычных компонентов, с названием JComponent, и второй для контейнеров, с названием JContainer? Возможно, так было бы проще понимать разницу между ними, к тому же это позволило бы сократить число ненужных методов в обычных компонентах и предотвратить ошибки, связанные с добавлением компонентов туда, куда ничего не следует добавлять (например, в надпись или флажок). Как оказывается, ответ на этот вопрос, как и на многие другие вопросы в Swing, во многом кроется в принципе работы легковесных компонентов.

Как вы помните, легковесный компонент — это просто область в пространстве окна вашего Java-приложения, полностью находящаяся в вашей власти и не воспринимаемая операционной системой как нечто самостоятельное (это верно и для простых компонентов, и для легковесных контейнеров). Помимо тех преимуществ, что мы уже обсуждали (прозрачность, быстродействие, переносимость), это к тому же означает, что *все* легковесные компоненты, какими бы сложными они ни были, в любом случае не смогут «сбежать» с пространства, занимаемого окном вашего приложения. А это дает возможность заранее подготовить и оптимизировать их вывод на экран (прорисовать их во внеэкранном буфере, проверить, какие из компонентов видны на экране, какие из них «повреждены»[6], и только потом обновить нужную область на экране). Подобный процесс (*двойная буферизация*) одинаков как для компонентов, так и для контейнеров, поэтому естественно было включить его в какой-то один базовый класс.

Нетрудно догадаться, что описанная работа выполняется в базовом классе JComponent, и то, что он представляет собой одновременно и компоненты, и контейнеры Swing, оказывается на самом деле очень удобным (не приходится «раскидывать» один и тот же код по двум классам и координировать их действия). Эффективная прорисовка компонентов Swing на экране — одна из самых важных обязанностей класса JComponent, и выполняет он ее очень качественно. Включив в базовый класс Swing двойную буферизацию, разработчики библиотеки избавили эту библиотеку от проблем, связанных с мерцанием и медленной скоростью вывода на экран.

Помимо того что базовый класс JComponent обеспечивает механизм эффективной прорисовки компонентов на экране, он обеспечивает поддержку наиболее важных свойств Swing, характерных для всех компонентов этой библиотеки. В класс JComponent

[5] Конечно, компоненты Swing не ограничивались набором методов из AWT — они имели просто гигантские возможности и соответственно в несколько раз больше методов. Названия методов AWT были сохранены для облегчения перехода с AWT на Swing, но вот перейти со Swing на AWT практически невозможно (слишком велики возможности Swing по сравнению с минималистским прикладным программным интерфейсом библиотеки AWT, да и компонентов в Swing гораздо больше).

[6] Поврежденная область (damaged area) — это термин компьютерной графики, обозначающий область экрана, нуждающуюся в обновлении. Аналогом может быть dirty area — загрязненная область.

встроена поддержка *всплывающих подсказок* (tool tips), *рамок* (borders), *средств для пользователей с ограниченными возможностями* (accessibility), *клавиатурных действий* (keyboard actions) и многого другого. Все это мы подробно обсудим немного позднее в соответствующих главах.

Таким образом, можно смело сказать, что класс JComponent является настоящим ядром библиотеки Swing, и львиная доля возможностей последней обеспечивается этим классом, который можно назвать настоящей «рабочей лошадкой» библиотеки. При создании обычных приложений вам вряд ли придется глубоко вникать в механизмы работы этого класса, однако чтобы «выжать» из своего приложения максимум возможностей, необходимо выяснить, что происходит за кулисами библиотеки. В главе 5 мы подробно исследуем внутреннюю работу этого базового класса Swing (пожалуй, самого важного класса библиотеки).

Не все компоненты Swing унаследованы от класса JComponent. Когда мы обсуждали легковесные компоненты, то выяснили, что должен иметься тяжеловесный контейнер, который будет отвечать за прорисовку всех содержащихся в нем легковесных компонентов. В AWT такими контейнерами чаще всего служили окна Frame и Dialog, а также класс апплетов Applet. Можно пытаться добавлять компоненты Swing в эти контейнеры, однако работать все будет не очень хорошо (особенно это касается меню Swing, которые представляют собой обычные компоненты, и места в контейнерах AWT для них не предусмотрено). Поэтому Swing предоставляет свои, слегка измененные тяжеловесные контейнеры высшего уровня: окна JWindow, JDialog и JFrame, а также апплет JApplet. Перечисленные классы имеют всю необходимую поддержку для компонентов Swing, которую обеспечивает так называемая *корневая панель* (root pane) — особый контейнер Swing (мы подробно обсудим его в главе 6).

Подводя предварительный итог, мы можем сказать, что, хотя Swing и основана на AWT, разница между двумя этими библиотеками велика. Возможности Swing гораздо обширней возможностей AWT, и поначалу с трудом верится в то, что первая появилась в результате доработки второй. Единственная их связь — базовые классы Component и Container, позволяющие Swing абстрагироваться от связей с операционной системой и называться библиотекой, полностью написанной на Java и не требующей каких бы то ни было ресурсов конкретной платформы. Так что при создании приложений и собственных компонентов прежде всего следует рассмотреть возможность использования в качестве основной библиотеки Swing, которая сделает много рутинной работы за вас, и лишь в особых случаях обращаться к AWT.

## Совместное использование компонентов AWT и Swing

Начиная с выпуска Java 2, стандартными библиотеками языка Java являются как библиотека AWT (которая считается устаревшей), так и основанная на ней библиотека Swing. Поэтому никаких препятствий для совместного использования графических компонентов этих двух библиотек нет. Вы можете спокойно добавлять компоненты AWT и Swing в свой контейнер, потому что в основе их (как и в основе любой библиотеки пользовательского интерфейса, даже от стороннего производителя) лежат базовые классы Component и Container. Вопрос в том, будет ли подобная конструкция правильно работать.

Долгое время компоненты AWT и Swing не могли находиться вместе в одном контейнере, если их области перекрывались. Особенно болезненно это было для многодокументных интерфейсов (Multi-Document Interface, MDI) Swing и вспомогательных компонентов, таких как панели прокрутки. Страдали и легковесные всплывающие меню. Проблема состояла в том, что легковесные компоненты ни при каких обстоятельствах не могли находиться на экране *выше* тяжеловесных, даже если это было явно указано.

Только в версии JDK 1.6 (малой версии 10) и во всех последующих версиях, конечно включая Java 7, проблема была решена, и тяжеловесные компоненты теперь могут находиться «под» легковесными. Надеемся, что вам не придется использовать старые выпуски JDK по причинам совместимости, в противном случае лучше всего избегать совместного использования легковесных и тяжеловесных компонентов (Swing позволяет создать любой интерфейс, не прибегая к AWT). Если вы работаете со старой версией и совместить компоненты все же необходимо, следует следить за тем, чтобы они не размещались в легковесных контейнерах (типа панели прокрутки JScrollPane или внутреннего окна JInternalFrame) и не перекрывали легковесные меню. Но при малейшей возможности следует обновить версию JDK, и все проблемы решатся сами собой.

## *Архитектура JavaBeans*

Одним из самых популярных течений в современном программировании является так называемое *компонентное*, или *модульное*, *программирование*. Идея его очень проста: вы берете необходимое количество «компонентов» из имеющегося у вас набора, настраиваете их свойства, обеспечиваете совместную работу и в результате получаете готовое приложение. Вообще этот процесс очень напоминает складывание строения из кубиков в детском конструкторе. Компонентное программирование недаром стало очень популярным — использование проверенных и качественных компонентов от хорошего производителя значительно ускоряет и упрощает разработку даже самого сложного приложения. В идеале количество уже разработанных компонентов должно быть достаточным для быстрого и практически безошибочного производства самого сложного приложения. В общем и целом компонент можно определить как завершенный фрагмент кода (*черный ящик*), предоставляющий некоторые услуги программисту (например, компонент может представлять собой графический элемент управления) и написанный в соответствии с некоторыми правилами, позволяющими определить свойства компонента. Объектно-ориентированное программирование также является компонентным, и компонентами в нем служат классы и объекты. В данный момент компонентное программирование выходит на новый уровень: такие технологии как SOAP (Simple Object Access Protocol), UDDI (Universal Description, Discovery, and Integration) и WSDL (Web Services Description Language) позволяют создавать компоненты (*веб-службы*) в глобальном масштабе (получить доступ к ним можно будет из любой точки, имеющей выход в Интернет).

Мы говорим о создании пользовательского интерфейса, где компоненты чаще всего представляют собой разнообразные элементы этого интерфейса (кнопки, списки и т. п.). Так как компоненты эти графические, ими удобно манипулировать визуально, так чтобы постоянно наблюдать, что получается в результате. Первопроходцем визуального программирования пользовательских интерфейсов стал язык Visual Basic (VB). Программирование для Windows никогда не было «приятной прогулкой», но с введением в VB графических компонентов и возможности визуально располагать эти компоненты на форме все изменилось. Разработка графических приложений стала требовать гораздо меньше времени и усилий, и визуальное программирование мгновенно вознеслось на вершину популярности. Проблемой VB было то, что компоненты представляли собой элементы ActiveX, и написать на VB собственный компонент было непросто (приходилось обращаться к более мощным, но и более сложным языкам). Как следствие стали появляться средства быстрой разработки приложений (RAD) следующего поколения, и самым ярким их представителем является среда Delphi. В ней процесс создания компонента практически не отличается от процесса написания обычной программы, благодаря чему было создано множество разнообразных компонентов для этой среды. Более того, само понятие компонента в Delphi не является синонимом графического элемента, компонент в этой среде может предоставлять и другие услуги (например, сетевые соединения).

Программист может настраивать поведение компонента, изменяя его свойства и обрабатывая события. *Свойства* (properties) компонента определяют его внешний вид и поведение. *События* (events) позволяют узнавать о действиях пользователя, изменении свойств, а также обеспечивают взаимодействие с другими компонентами. В визуальном средстве, таком как Delphi, список свойств постоянно находится у вас под рукой, и менять их можно буквально на «лету», без перекомпиляции и до получения нужного результата, после чего остается лишь обработать нужные вам события.

Базовой архитектурой всех, графических, вычислительных или любых иных компонентов Java является схема JavaBeans. Компоненты, написанные в соответствии с этой архитектурой, являются обычными классами языка Java, легко переносятся и следуют единому стандарту, позволяющему определить, какими свойствами обладает компонент и какие события он поддерживает. Главными составляющими этой архитектуры являются *соглашение об именах* и, что крайне важно именно для компонентов пользовательского интерфейса, *система обработки событий*, которая, впрочем, также основана на соглашении об именах.

## Соглашение об именах

Прежде чем включить компонент в средство быстрой разработки приложений, необходимо предоставить какой-то способ, позволяющий распознавать его свойства и события. В различных средах и системах используются разные подходы: среди наиболее распространенных можно назвать специальные записи в общедоступном системном реестре (компонент регистрирует себя при установке, после чего средство разработки находит его и добавляет в палитру инструментов) и специальные сопровождающие *библиотеки типов* (type library), содержащие описание свойств, методов и событий компонента. Эти подходы не идеальны — нет гарантии того, что ваше средство RAD совместимо с компонентом, сами компоненты компилируются в обычные двоичные файлы и перенести их на другую платформу почти невозможно.

Однако Java является платформенно-переносимым языком, и поэтому ваши программы не компилируются под конкретную платформу, а записываются в специальный байт-код, который затем выполняется виртуальной машиной Java. Благодаря этому программы сохраняют изначальную структуру классов, в том числе сохраняются имена переменных и методов (вы можете полностью исследовать содержимое класса с помощью общеизвестного любому программисту на Java механизма отражения из пакета java.lang.reflection). Создатели JavaBeans решили использовать это уникальное в своем роде свойство Java для того, чтобы максимально упростить создание компонентов. Действительно, достаточно сигнализировать о наличии свойств и событий особыми именами методов и переменных.

Итак, для того чтобы создать компонент (любого типа, в том числе и графический), вам понадобится всего лишь создать новый класс. Конечно, у такого компонента не будет ни одного свойства и события, и вот здесь вступает в действие соглашение об именах[7]. Давайте посмотрим, какие правила необходимо соблюдать для объявления свойств и событий.

1. Каждому свойству с именем xxx необходимо предоставить метод для считывания значения этого свойства со стандартным именем getXxx() и метод для записи нового значения этого свойства с именем setXxx(). Например, если вы создаете графический компонент и хотите, чтобы у него было свойство color (цвет), необходимо сначала объявить закрытую переменную нужного типа (в нашем случае переменная имеет тип Color):

[7] В спецификации JavaBeans эти соглашения об именах почему-то названы шаблонами проектирования (design patterns), с чем вряд ли можно согласиться.

```
private Color color;
```

Затем нужно написать два метода для считывания и записи этого свойства:

```
public Color getColor() { /* … */ }
public void setColor(Color c) { /* … */ }
```

Когда средство разработки программ при анализе вашего класса обнаружит пару этих методов, оно добавит в список свойств вашего компонента свойство color, и разработчик, использующий ваш компонент, сможет менять это свойство. Можно даже предоставлять только метод для считывания (getXxx()) и таким образом помечать свойство компонента как предназначенное только для чтения. Заметьте, что имена методов должны следовать рекомендациям Sun, поэтому после слов get или set наличие прописной буквы обязательно.

2. Если какое-то свойство компонента имеет булев тип (boolean), можно использовать альтернативный вариант именования метода для получения значения этого свойства. Например, рассмотрим свойство вида:

   ```
   private boolean enabled;
   ```

   Методы для этого свойства могут иметь следующие названия:

   ```
   public boolean isEnabled() { /* ... */ }
   public void setEnabled(boolean b) { /* ... */ }
   ```

   То есть метод для записи нового значения не меняется, в методе для получения значения вместо приставки get используется is. Это сделано исключительно для того, чтобы людям было проще читать названия таких методов.

3. Методы компонента, которые не предназначены для считывания и записи свойств, но служащие для выполнения каких-то важных действий, должны быть объявлены открытыми (public).

4. Для каждого события, которое способен генерировать компонент, необходимо предоставить пару методов для добавления и удаления объектов, заинтересованных в этом событии (эти объекты называют *слушателями*). Мы подробнее разберемся с событиями чуть позже, в главе 2.

5. Класс компонента необходимо объявить открытым (public) и определить в нем конструктор без параметров. Тогда средство разработки программ сможет создать экземпляр такого класса, не обладая дополнительной информацией о типе и предназначении параметров, передающихся в конструкторе. Если вашему компоненту все же нужны параметры для корректной работы, всегда можно сделать их свойствами и предоставить в классе методы для считывания и записи значений этих свойств.

В результате мы получим класс, единственное отличие которого от других классов состоит в том, что его методы следуют стандартной схеме именования JavaBeans. Этого будет вполне достаточно для того, чтобы средство визуальной разработки приложений смогло распознать в нем компонент и вывести список его свойств и событий. Только вдумайтесь, насколько просто звучит такое утверждение и какие у него потрясающие последствия. Вы создаете обычный класс, называя методы по незамысловатым правилам, компилируете его, после чего класс становится доступным для всего мира, причем использовать его можно как угодно — достаточно просто создать объект в своей программе и настроить в визуальном средстве или в распределенном окружении так, как вам нужно.

Архитектура JavaBeans стала настоящим открытием в компонентном программировании и придала ему новый импульс. С ее появлением отпала необходимость изучать дополнительные библиотеки и разрабатывать сопровождающие программы только для того, чтобы построить компонент. Вы строите компоненты, просто создавая классы, что вместе с переносимостью Java дает программисту очень большие возможности. Неудивительно, что после выхода пакета JDK 1.1 и появления в нем спецификации JavaBeans началось настоящее «извержение» компонентов от разных производителей. Более того, создатели Java переделали многие классы стандартных библиотек так, чтобы они удовлетворяли требованиям новой архитектуры. Сейчас представить себе Java без JavaBeans уже невозможно: все связанные с платформой технологии и спецификации используют JavaBeans.

Кроме того что технология JavaBeans дает возможность работать визуальным средствам с любыми компонентами от любых производителей, она может быть полезна и при «ручном» написании программ. Зная набор простых правил, которым подчиняются названия методов любого компонента, можно предположить, каким свойством обладает компонент, и вызывать соответствующие этому свойству методы, не обращаясь к документации или тратя на ознакомление с ней минимум времени. Если создатели компонента удачно подбирают названия его свойств, знакомство программиста с таким компонентом проходит быстро и безболезненно (компоненты библиотеки Swing — прекрасный пример).

## Расширенные возможности

Архитектура JavaBeans не ограничивается соглашением об именах и новой системой событий, и предоставляет программисту дополнительные способы настройки и расширения своих компонентов. В их числе можно назвать привязанные и ограниченные свойства, собственные редакторы свойств, дополнительные описания компонентов и многое другое. Чтобы получить представление о возможностях JavaBeans, обсудим эти возможности, не вдаваясь в детали, а подробности всегда можно узнать в интерактивной документации Java.

### Привязанные свойства

*Привязанные* (bound) свойства используются в тех случаях, когда о смене значения свойства необходимо сообщить другому компоненту или объекту. В пакете java.beans имеются несколько вспомогательных классов, позволяющих реализовать привязанные свойства без особых усилий. Все сводится к тому, что в классе компонента появляется еще одна пара методов для добавления и удаления слушателей события типа PropertyChangeEvent, однако событие это не относится к работе компонента, а возникает при смене значения некого свойства. Заинтересованным в смене значения свойства объектам нужно лишь следить за этим событием. При обработке события можно узнать, какое свойство изменилось, каким стало новое значение этого свойства, каким было его старое значение. Свойства почти всегда делают привязанными, по нескольким причинам. Во-первых, визуальные средства разработки часто используют этот механизм, чтобы контролировать, когда следует обновлять внешний вид компонентов и список свойств, ну а во-вторых, сами компоненты следят за состоянием своих или чужих свойств, чтобы выглядеть или вести себя соответственно.

### Ограниченные свойства

*Ограниченные* (constrained) свойства помогают контролировать правильность смены свойств компонента в визуальном средстве разработки. Для реализации ограниченных свойств в класс компонента добавляются методы, позволяющие добавлять и удалять слушателей типа VetoableChangeListener. При смене значения ограниченного свойства этим слушателем посылается запрос, они проверяют, можно ли установить

новое значение свойства, и если новое значение не подходит, возбуждают исключение PropertyVetoException. Визуальное средство при возникновении такого исключения сообщает пользователю, что введенное им новое значение свойства применять нельзя.

## Индексированные свойства

Бывают ситуации, когда использовать обычные свойства и методы для работы с ними не совсем удобно, например, если компонент хранит множество однотипной информации (список). В JavaBeans специально для этого предусмотрены *индексированные* (indexed) свойства. Отличие их от обычных свойств заключается в том, что вместо одного объекта в методах set/get передается массив объектов, и в дополнение к этим двум методам предоставляются методы, изменяющие одну из позиций в этом массиве. Выглядеть это может так:

```
public Color[] getColors();
public void setColors(Color[] c);
public Color getColors(int index);
public void setColors(int index, Color c);
```

Здесь имя индексированного свойства — colors, и для него предоставлено по 2 метода get/set. Первая пара методов манипулирует списком целиком, вторая позволяет менять одну из позиций списка. Индексированные свойства могут быть привязанными и ограниченными, так же как обычные.

## Редакторы свойств

По умолчанию считается, что визуальное средство разработки способно обеспечить правильное редактирование свойств. Для наиболее часто встречающихся свойств так оно и есть (любое средство справится со свойствами, заданными в виде строк, чисел или цветов). Однако если ваш компонент обладает свойством неизвестного и довольно сложного типа, визуальное средство не сможет помочь программисту, и ему придется настраивать ваш компонент вручную, что вообще-то противоречит принципу визуального программирования. Архитектура JavaBeans решает эту проблему с помощью *редакторов свойств* (property editors). Редактор свойств — это дополнительный компонент, позволяющий настраивать свойства определенного типа, он чаще всего поставляется вместе с компонентом. Создать его довольно просто — надо расширить класс PropertyEditorSupport, написать на его основе редактор свойств и зарегистрировать его. Проще всего зарегистрировать редактор с помощью класса PropertyEditorManager из пакета java.beans. Статический метод registerEditor() этого класса позволяет сопоставить тип свойства и класс его редактора.

## Описание компонента

Если средство разработки исследует компонент, опираясь только на соглашение об именах, то у него оказывается минимум информации (названия свойств и событий, используемых в классе компонента, и ничего больше). Иногда этого мало, и в таких случаях для компонентов придется писать подробную документацию с описанием каждого свойства, события и их действия. В JavaBeans предусмотрен механизм, называемый *описанием компонента* (bean info), который дает возможность ассоциировать с компонентом дополнительную информацию о его свойствах и событиях. Этот механизм позволяет создателю компонента точно указать, как должно называться свойство или событие (совпадение с названиями в коде не обязательно), кратко описать его предназначение, отделить обычные свойства от свойств, контролирующих более сложные аспекты работы компонента.

Для того чтобы создать описание компонента, необходимо определить класс со специальным именем: имя класса компонента плюс слово «BeanInfo». Этот класс должен либо реализовывать интерфейс BeanInfo, либо расширять класс SimpleBeanInfo (который также реализует интерфейс BeanInfo и служит для упрощения реализации этого интерфейса). Реализуя методы интерфейса BeanInfo, программист может весьма подробно описать свой компонент. Затем полученный класс упаковывается в архив вместе с компонентом, и средство разработки, загружая этот архив, извлекает дополнительную информацию о компоненте.

### Постоянство

При работе с компонентом программист меняет его свойства и пишет код, обрабатывающий события. Эти изменения отражаются средством быстрой разработки в коде создаваемой программы: по мере необходимости оно генерирует код, который после компиляции и запуска программы приведет компонент к нужному состоянию. Однако такой подход не всегда удобен и не всегда возможен, например, если информация внутри компонента очень важна или смена состояния посредством вызова методов занимает слишком много времени и памяти (наиболее вероятна такая ситуация при удаленном вызове методов компонента).

Поэтому в компонентном программировании появилось понятие *постоянства* (persistence). Говорят, что компонент поддерживает постоянство, если он обладает способностью вынести свое состояние во внешнее хранилище (файл или базу данных), а после при необходимости восстановить его.

Поначалу архитектура JavaBeans поддерживала постоянство посредством встроенного в Java механизма *сериализации* (serialization). Суть этого механизма состоит в том, что объект, реализующий интерфейс Serializable, может быть представлен в двоичной форме (фактически производится побитовое копирование полей объекта). Поэтому для компонентов рекомендовалось реализовывать интерфейс Serializable на тот случай, если придется хранить свое состояние во внешней среде. Однако такой подход не всегда обеспечивает хороший результат — сериализованные объекты без специальных усилий (серийных номеров) несовместимы друг с другом (из-за любых, даже минимальных, изменений в классе объекта или в виртуальной машине), формат сериализованных данных неудобен для чтения в случае отладки или локализации приложения.

С появлением JDK 1.4 нашлось еще одно весьма достойное решение. В этом пакете разработки состояние компонента стало возможным хранить в файле формата XML. К очевидным преимуществам нового подхода следует отнести то, что сохраняется не детализированная информация о компоненте, а лишь минимальная последовательность действий, необходимая для приведения компонента в нужное состояние. Это решение вполне может обеспечить необходимую степень совместимости, простоты и переносимости. Учитывая популярность XML (Extension Markup Language — расширенный язык разметки) и разнообразие средств для работы с этим универсальным языком, можно сказать, что компоненты JavaBeans имеют приличный механизм обеспечения постоянства. Новый механизм поддерживается классами XMLEncoder и XMLDecoder из пакета java.beans. Данные классы неплохо документированы, практически все их действия легко переопределить в соответствии со своими нуждами, а на сайте Sun/Oracle вы найдете немало дополнительной информации о них.

## Компоненты Swing — это компоненты JavaBeans

Теперь, когда мы увидели, как много привнесла в Java спецификация JavaBeans, нетрудно догадаться, что библиотека Swing проектировалась в полном соответствии с ней. Классы библиотеки Swing, представляющие собой компоненты, созданы в соответствии с соглашением об именах. Все возможности компонентов Swing представлены в виде свойств

и имеют соответствующий набор методов get/set. События компонентов также используют схему JavaBeans. Поддерживается и новый механизм постоянства (на основе XML).

Однако компоненты Swing не злоупотребляют возможностями JavaBeans и реализуют лишь то, что действительно необходимо для работы (эта библиотека специально разрабатывалась в расчете на то, что ей будет максимально просто пользоваться). Все свойства компонентов являются привязанными (это действительно удобно и позволяет легко узнавать об изменениях в компоненте), однако ограниченных свойств, требующих к себе повышенного внимания, в библиотеке почти нет (их всего несколько, и все они сосредоточены в классе внутренних окон JInternalFrame). Индексированных свойств также немного (вместо них рекомендуется использовать *модели*, о которых мы еще поговорим). Описания компонентов в Swing есть, но обычно эти описания требуются только в визуальных средствах. И вообще при применении библиотеки не чувствуется, что компоненты спроектированы согласно архитектуре JavaBeans и могут свободно использоваться в визуальных средствах — с ними одинаково просто работать и как с визуальными компонентами, и как с обычными классами.

То, что компоненты Swing являются компонентами JavaBeans, важно скорее для средств RAD, для любого из которых сейчас характерна полная поддержка Swing. В этой книге мы не рассматриваем вопросы и проблемы визуального программирования, однако знание JavaBeans иногда оказывается очень удобным. Создателям Swing удалось удачно подобрать названия свойств компонентов, и при написании программы зачастую удается просто угадывать названия нужных методов. Например, довольно очевидно, что у текстового поля (JTextField) должны быть свойства «текст» (text), имея в виду весь напечатанный текст, и «выделенный текст» (selected text). Для получения значения этих свойств можно попытаться вызвать методы getText() и getSelectedText(). Как оказывается, методы именно с такими именами присутствуют в классе JTextField. Зная методы для получения значений свойств, легко понять, какие методы используются для изменения свойств: setText() и setSelectedText(). Вы будете приятно удивлены тем, как часто срабатывает в Swing такой трюк, и как это ускоряет работу. Конечно, это не рецепт на все случаи жизни, но, по крайней мере, методы с такими понятными названиями и запоминать гораздо проще.

## *Подключаемые внешний вид и поведение*

Обсуждая библиотеку AWT, мы выяснили, что компоненты этой библиотеки не отвечают за свой внешний вид на экране. Их помощники направляют запросы к операционной системе, которая и обеспечивает появление на экране элементов управления. Благодаря этому внешний вид Java-приложений и обычных приложений для конкретной операционной системы не должен отличаться (по крайней мере, в теории). Увы, это не совсем так. Виртуальные машины Java от разных производителей даже на единственной платформе вполне могут придать AWT-приложениям различающийся внешний вид, похожий на что угодно, но только не на приложения, к которым привык пользователь платформы.

Когда стало окончательно ясно, что компоненты AWT хоть и работают, но обеспечить более или менее приличный интерфейс не могут (по крайней мере, без дополнительных усилий), взоры обратились в сторону легковесных компонентов. Мы уже знаем, что легковесные компоненты не зависят от платформы и полностью находятся во власти Java-приложения. Это прекрасно, но здесь есть ловушка. Кто-то должен обеспечить им нужный внешний вид и взаимодействие с пользователем. Операционная система больше не участвует в создании компонентов, и получается, что разработчику компонента нужно делать все самому, начиная от подбора цветов и заканчивая поддержкой клавиатуры.

Пока библиотека Swing находилась на ранней стадии разработки, появились первые библиотеки легковесных компонентов от сторонних производителей. Создатели этих библиотек пошли по пути наименьшего сопротивления — прорисовка компонентов и их поведение записывались в один класс. С одной стороны, было понятно, что каждый компонент

рисует себя сам, и для изменения внешнего вида компонента нужно было вызывать кое-какие его методы. С другой стороны, получалось, что в классе компонента было сосредоточено гигантское количество методов, и все они предназначались для разных целей: для настройки внешнего вида, для настройки функциональности, для получения состояния и событий, и т. д. Работать с такими классами очень неудобно. К тому же, если требовалось внести малейшее изменение в работу компонента, необходимо было наследовать от него и настраивать его внутренние механизмы (а это всегда нелегко, особенно с компонентами от других производителей). Был у такого подхода и еще один недостаток. Создавая компонент, его разработчик, сознательно или нет, стремился приблизить его вид и поведение к виду и поведению компонентов той платформы, на которой он работал. Получалось, что большинство библиотек имели Windows-подобные компоненты. Они так выглядели и на Unix, и на Mac, и работать с ними пользователям этих систем было крайне неудобно.

Перед создателями Swing стояла другая задача. Главным преимуществом языка Java является его переносимость и поддержка наиболее популярных платформ. Во внешнем виде и поведении Java-приложений нужно было как-то учитывать требования всех этих платформ, чтобы их пользователям не приходилось тратить лишнее время на освоение новых приложений. Поэтому было принято решение вообще отказаться от какого-то конкретного внешнего вида компонентов библиотеки Swing и предоставить механизм, позволяющий достаточно просто, не меняя функциональности компонентов, менять их внешний вид. Прежде всего, нужно было отказаться от совмещения функций компонента и его внешнего вида и поведения. Такая задача — отделение постоянной составляющей от изменяющейся, встречается в программировании очень часто, в том числе и в программировании библиотек пользовательского интерфейса. Разработчики Swing обратились к решению, проверенному годами. Посмотрим, что это за решение.

## Архитектура MVC

Довольно давно (по меркам компьютерной индустрии вообще в доисторических временах), около 1980 года, появилась очередная версия объектно-ориентированного языка Smalltalk, названная Smalltalk-80. В этой версии возникла архитектура, предназначенная для создания легко расширяемых и настраиваемых пользовательских интерфейсов. Эту архитектуру назвали *модель – вид — контроллер* (Model/View/Controller, MVC). По сути дела, появление в Smalltalk данной архитектуры привело к возникновению пользовательского интерфейса таким, каким мы его знаем сегодня, ведь именно тогда родились основные концепции и внешний вид большинства известных компонентов. Идея этих компонентов затем была использована в Macintosh, после чего перешла к многочисленным последователям Macintosh. Несмотря на то, что с момента появления MVC прошло уже немало времени, эта архитектура остается одним из самых удачных объектно-ориентированных решений, и поэтому часто используется и сегодня.

Как нетрудно догадаться по названию, MVC состоит из трех частей.

- *Модель* (model) хранит данные компонента и позволяет легко, не обращаясь к самому компоненту, изменять или получать эти данные. Например, раскрывающийся список позволяет вывести на экран перечень элементов (обычно это строки). Вместо того чтобы включать методы для манипуляции элементами списка в класс раскрывающегося списка, можно предоставить отдельный класс, работающий исключительно с данными. Такой подход позволит разработчику сосредоточиться именно на той задаче, которой он занимается в данный момент: можно сначала подготовить данные (считать их из файла или сетевого соединения, отсортировать, локализовать и т. п.), а потом уже передать их раскрывающемуся списку для вывода на экран. Хранение данных отдельно от самого компонента также позволяет изменять структуру данных модели, не меняя функций компонента. Самое же главное состоит в том, что отдельная модель позволяет появляться «хорошим» классам, которые описывают данные в языке реша-

емой задачи, а не в языке той или иной библиотеки пользовательского интерфейса. Если нам понадобится список книг в библиотеке, мы назовем класс именно так, а не будем встраивать его в мудреные классы, подгоняя под нужды интерфейса.

- *Вид* (view) выводит данные на экран для представления их пользователю. Отделение вида от данных позволяет представлять одни и те же данные совершенно разными способами. Например, текст формата HTML (Hypertext Markup Language – гипертекстовый язык разметки) можно вывести в разном виде: провести разметку документа, разместить изображения и ссылки, использовать различные шрифты, а можно показать HTML-документ как код, который состоит из набора тегов и текста среди них. Между тем данные для этих разных видов требуются одни и те же (текст формата HTML). Вспоминая пример с раскрывающимся списком, можно сказать, что он является видом, представляющим на экране набор элементов. Данные раскрывающегося списка можно было бы представить и в другом виде, например, в таблице.
- *Контроллер* (controller) определяет, как должны реагировать вид и данные модели в ответ на действия пользователя. Наличие в MVC контроллера позволяет использовать одни и те же данные и виды в разных целях. HTML-страница, например, может быть показана в браузере или в визуальном средстве создания страниц. Браузер может задействовать контроллер, который при щелчке на ссылке переходит на страницу, указанную в ссылке (полностью меняет данные модели, загружая в нее новую порцию HTML-текста), а визуальное средство, скорее всего, использует контроллер, вызывающий при щелчке на ссылке редактор свойств этой ссылки (который меняет лишь часть данных модели, относящихся к ссылке). Раскрывающемуся списку также не помешает пара контроллеров: один для списка, не позволяющего редактирование элементов, а другой для редактируемого списка. Не всегда, но, тем не менее, довольно часто, контроллер также выполняет надзирательные функции и не позволяет пользователям портить модели некорректными данными, проводя проверку их на правильность перед отсылкой в модель.

Окончательно прояснит работу MVC и взаимодействие трех участников этой архитектуры рис. 1.3.

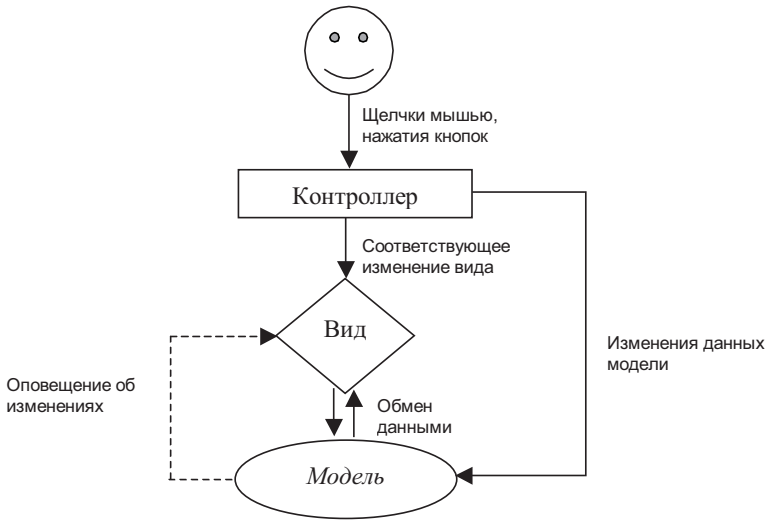



**Рис. 1.3.** Взаимодействия между моделью, видом и контроллером

Пользователь взаимодействует с программой, совершая различные действия (нажимая клавиши, перемещая мышь, щелкая ее кнопками и т. п.); информация о его действиях поступает в контроллер. Контроллер определяет, как обработать эти действия, и посылает сообщения модели и/или виду[8]. При этом используются следующие обозначения: сплошные стрелки — это обращение к конкретному объекту с известным типом, а пунктирные стрелки — это оповещения[9] заранее неизвестных объектов об изменении ситуации в системе. Предпочтительнее задействовать механизм оповещения (пунктирные стрелки), так как он позволяет избежать сильной связи между объектами, а это повышает гибкость системы. Особенно важен механизм оповещения для модели, что и показано на нашей диаграмме. Как видно, модель на самом деле не знает ни о присоединенном к ней виде, ни об используемом контроллере. Она пассивна: ее данные меняет контроллер (или вы сами), а информацию об этих изменениях она рассылает заинтересованным объектам, которые заранее неизвестны. Благодаря этому модель по-настоящему независима от остальных частей MVC, и ее легко использовать с разными видами, контроллерами и несколькими компонентами. В качестве слушателя оповещений, которые рассылает модель при изменении своих данных, чаще всего выступает вид (он может быть не один), обновляющий изображение компонента в соответствии с новыми данными. Например, при нажатии пользователем клавиши в текстовом поле контроллер (если он допускает ввод текста) вставляет в модель новый символ, соответствующий нажатой клавише, модель оповещает присоединенный к ней вид об изменении, а вид отображает новый символ на экране.

Кроме такого, наиболее естественного способа работы с компонентом (все изменения происходят в ответ на действия пользователя), можно непосредственно взаимодействовать с отдельными частями архитектуры MVC. Иногда это может быть очень кстати. Например, вы можете использовать данные модели, не затрагивая контроллер и вид, манипулировать ими, и все изменения, что в них происходят, автоматически появятся на экране. Можно не ждать событий от пользователя, а программно генерировать их в контроллере (реакция на них со стороны модели и вида будет такой же) — это может быть весьма удобным при автоматизированном тестировании интерфейса или при обучении. Очень сильной стороной архитектуры MVC является также ее динамичность: никто не запрещает вам менять виды, контроллеры и модели прямо во время работы программы. Одним словом, архитектура MVC совсем не тривиальна, и возможности ее действительно велики.

С появлением в Smalltalk архитектуры MVC стало возможным создавать очень гибкие пользовательские интерфейсы. У программиста имелся набор различных видов, моделей и контроллеров, используя их в нужном сочетании, можно было добиться практически любого эффекта. Упрощено было и создание новых элементов пользовательского интерфейса: функции моделей, контроллеров и видов были четко разделены, их легко было наследовать от уже существующих классов, немного изменяя реализацию в соответствии со своими нуждами.

Как вы помните, перед создателями Swing стояла задача отделения внешнего вида компонентов от их функций, так чтобы было возможно легко изменять внешний

[8] Не удивляйтесь, контроллер может взаимодействовать и с моделью, и с видом, и сразу с обоими. Как мы увидим далее, контроллер — на самом деле довольно запутанная часть MVC, и поэтому от него зачастую просто-напросто избавляются. К примеру, с видом контроллер может взаимодействовать в современных интерфейсах с обратной визуальной связью, которая служит лишь эстетическим целям и удобству, но никак не связана с моделью.

[9] Механизм оповещения, использованный в MVC, — это ни что иное, как шаблон проектирования под названием наблюдатель (observer), один из самых известных, самых простых и самых полезных шаблонов в мире объектно-ориентированного проектирования (все гениальное по-прежнему просто). Мы подробно рассмотрим этот шаблон в главе 2, посвященной системе событий Swing, которая фактически представляет собой реализацию этого шаблона.

вид компонентов в соответствии с используемой платформой. Вне всяких сомнений, MVC прекрасно справится с такой задачей: меняя вид и контроллер, легко можно добиться изменения поведения и внешнего вида компонента. В теории все выглядит прекрасно.

## Все ли так хорошо в MVC?

Ответ кажется очевидным: «Конечно, все просто замечательно!». Действительно, архитектура MVC хороша — гибкая, легко настраиваемая, расширяемая и простая в понимании, она позволяет делать многое. Однако хоть это и кажется удивительным, использовать ее в оригинальном варианте оказывается не совсем удобно. Проблема здесь кроется не только и не столько в самой архитектуре — скорее накладывают свои ограничения те задачи, которые мы хотим решить с ее помощью, и те инструменты, которыми мы эти задачи собираемся решать.

Слабым звеном оказывается контроллер. Посмотрите, какие обязанности возлагаются на этот элемент системы: он должен преобразовывать действия пользователя (нажатия клавиш, движения мыши и т. п.) в соответствующие изменения данных модели. Очевидно, что между ними возникает довольно близкая связь. Контроллеру просто необходимо знать хотя бы что-то о том, какие данные хранит модель, как она это делает и как эти данные меняются. Например, нажатие клавиши для текстового поля означает вставку символа в документ, для кнопки — изменение состояния на нажатое, для раскрывающегося списка — выбор нового элемента. Каждый раз контроллер меняет довольно специфичные данные модели, и основное преимущество от использования его в качестве отдельного элемента системы — возможность заменять одни контроллеры другими — просто теряется (как поведет себя контроллер кнопки в таблице?).

То же самое можно сказать и о связи контроллера и вида. Эта связь сильная: вид хранит ссылку на контроллер, а контроллер на вид. Можно попытаться избежать подобной связи, полагаясь только на связь контроллера с моделью и дальнейшее оповещение моделью вида. Но такое взаимодействие сделает интерфейс неудобным: оно практически не в состоянии учитывать некоторые операции при изменении данных, которые также нужно отображать: например, модель текстового документа не должна во избежание излишней сложности хранить информацию о мерцающем курсоре, выделенном тексте, операциях перетаскивания и других подобных вещах. Все это должны решить между собой контроллер и вид, и без отражения подобных операций интерфейс будет крайне неудобным. В итоге получается, что хотя формально MVC отделяет контроллер, последний фактически «намертво» привязан к определенному виду и модели этого вида. Если компонент предполагает несколько сложных реакций на действия пользователя (к таким компонентам можно отнести, например, текстовые компоненты), контроллер в той или иной форме можно оставить, в противном случае он лишь вносит дополнительные сложность и путаницу.

Решение напрашивается само собой — надо просто соединить в единое целое контроллер и вид, образовав визуально-поведенческую (look and feel) часть компонента. Это целое и будет общаться с моделью. Такой подход не просто больше подходит для Java, но еще и более удобен для программирования: например, если компонент нужно отключить, то логичнее вызвать какой-то метод компонента (который сразу отключит обработку событий и сменит внешний вид), а не тасовать контроллеры и виды.

Кстати, описанная в этом разделе проблема с контроллерами характерна не только для Java, но и для большинства других языков, и уже довольно давно. Многие библиотеки пользовательских интерфейсов используют MVC, но немногие реализуют каноническую архитектуру. Большинство объединяют контроллер и вид, чтобы упростить библиотеку и улучшить взаимодействие ее частей. Более того, еще в самой первой версии

MVC в языке Smalltalk-80 применялось их объединение, называемое инструментарием (*tool*) для поддержки модели. Впрочем, есть и библиотеки, строго разделяющие контроллер, модель и вид.

## Решение Swing — представители пользовательского интерфейса

Итак, мы выяснили, что классическая архитектура MVC не очень хорошо подходит для Java: разделение контроллеров, моделей и видов не совсем оправдано. Гораздо более простым в реализации и дальнейшем использовании оказывается решение, совмещающее вид и контроллер в одно целое. Именно такое решение было выбрано разработчиками Swing. Посмотрим, что у них получилось (рис. 1.4).

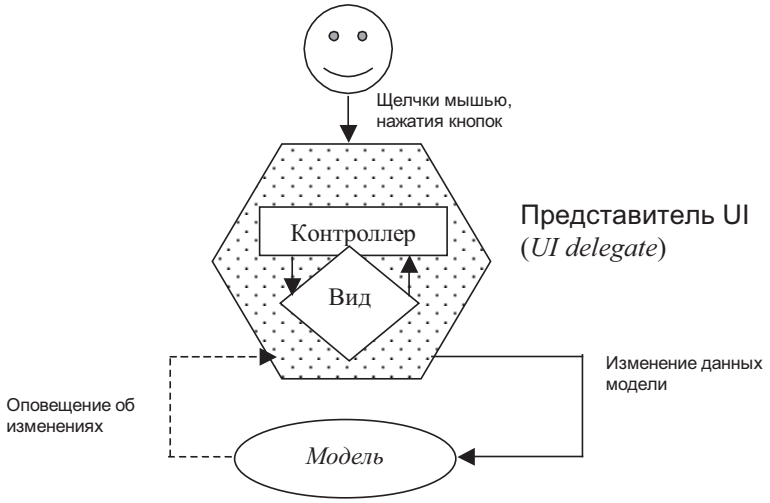



**Рис. 1.4.** Организация представителей

Как видно из диаграммы, разработчики Swing объединили вид и контроллер в новый элемент, который назвали *представителем* (delegate) *пользовательского интерфейса* (User Interface, UI). Теперь все действия пользователя поступают не в контроллер, определяющий реакцию на них, а в этот новый элемент, в котором происходит значительная часть работы. Он определяет, нужно ли реагировать на них (так как контроллер теперь находится внутри, исчезает необходимость переделывать его, чтобы отключить реакцию на действия пользователя — свойство «включено/выключено» стало свойством представителя) и, если нужно, то сразу же без генерации каких-либо событий и изменения данных меняет вид (это происходит быстро — вид и контроллер находятся в одном месте и имеют исчерпывающую информацию друг о друге, благодаря этому пользовательский интерфейс быстро реагирует на любые изменения и позволяет легко воспроизвести самые сложные операции по изменению данных), а уже после этого представитель говорит модели о том, что данные изменились, и их необходимо обновить. Модель обновляет хранящиеся в ней данные и оповещает заинтересованных субъектов (чаще всего того же представителя) об изменениях. В ответ внешний вид компонента окончательно обновляется, чтобы соответствовать новым данным.

Если рассматривать диаграмму подробней, оказывается, что схема работы новой системы очень проста, но весьма эффективна. Отдельного контроллера теперь нет, и ему не

придется прикладывать титанических усилий, чтобы оставаться многократно используемым и полезным элементом системы, способным работать с любой моделью. Вид четко знает, какой элемент он представляет на экране, то есть данные модели, как бы она ни была реализована, он выведет на экран надлежащим образом (с видом вообще нет проблем). Находящийся теперь с ним в «одном флаконе» контроллер может проще решить свою задачу обработки событий пользователя: когда нужно, он сразу изменяет вид (это необходимо для смены каких-либо аспектов вида, не связанных с данными модели, как правило, эстетических, во имя удобства пользователя). Взаимодействие контроллера и вида реализовано внутри представителя пользовательского интерфейса, поэтому происходит быстро. Завершающим этапом является отправка совмещенным видом-контроллером сообщения модели о необходимости обновления данных (представитель знает, как работает отображаемая им модель) и рассылка моделью оповещения о новых данных. Это оповещение дает возможность всем присоединенным к модели видам прорисовать новые данные, а остальным слушателям узнать, что пользователь внес в данные изменения.

Надо сказать, что разделение функций компонента только на две части (UI-представителя и модель) как нельзя лучше подходит для Swing. Разработчики Swing не стремились создать какой-то новый тип пользовательского интерфейса, они, прежде всего, должны были обеспечить поддержку компонентами всех известных платформ таким образом, чтобы Java-приложения по возможности внешне не отличались от приложения для конкретной платформы. UI-представители позволяют легко реализовать такое поведение. Оставляя функции компонента и модели данных в стороне, нужно лишь изменить UI-представителя, чтобы он реагировал на действия, характерные для какой-то платформы, и выводил компонент на экран в привычном для пользователей этой платформы виде.

Таким образом, слегка изменив и во многом упростив изначальную архитектуру MVC, создатели Swing сумели сохранить ее основные достоинства: простое изменение внешнего вида и поведения компонентов (это осуществляется заменой UI-представителя компонента), а также мощь модельного программирования (программист по-прежнему может использовать различные модели для одного компонента, менять данные и манипулировать ими, не заботясь об обновлении вида и о типе компонента, описывать данные в терминах решаемой задачи). Не совсем правильно говорить, что в Swing задействована архитектура MVC (в библиотеке реализована архитектура из двух частей, а не из трех), поэтому часто говорят об использовании в Swing отношения *модель-представитель* (model-delegate), или об *архитектуре с разделенной моделью* (separable model architecture).

## Как все работает

Кажется, мы дошли до сути механизма, обеспечивающего компонентам Swing различные поведение и внешний вид. Имеется UI-представитель, обрабатывающий события пользователя и рисующий компонент на экране; есть модель, хранящая данные компонента. Непонятно одно — как этот механизм взаимодействует с классами библиотеки Swing, такими как кнопки (JButton) или списки (JList). Работать с ними просто — вы создаете экземпляр класса кнопки и добавляете его в контейнер. Где же UI-представитель? Очевидно, что не в классах компонентов, иначе менять их внешний вид и поведение было бы невозможно (пришлось бы переписывать все эти классы для поддержки другого внешнего вида).

Рисунок 1.5 иллюстрирует роль, которую играет класс компонента во взаимоотношениях UI-представителя, модели и конечных пользователей (к ним относятся программисты-клиенты Swing). Можно сказать, что класс компонента — это точка приложения сил архитектуры «модель-представитель», в нем сосредотачивается информация о том, как UI-представитель взаимодействует с некоторой моделью. Модель не знает, с каким UI-представителем она сотрудничает и какой компонент ее использует, все что известно о модели — это то, что она есть. Раз модель существует, на нее должна быть

ссылка. Хранится эта ссылка в классе компонента. UI-представитель связывается с моделью только через класс компонента. Прямой связи нет, и это главное условие гибкости и взаимозаменяемости. Таким образом осуществляется обмен данными между видом и моделью. Программисту, намеревающемуся использовать некоторый компонент, не придется думать о том, какой UI-представитель задействован в данный момент и как его соединить с моделью. От него требуется лишь настроить модель (или применить модель по умолчанию) и передать ее компоненту. Компонент знает, какого UI-представителя нужно использовать (он получает эту информацию от менеджера внешнего вида UIManager, о котором мы вскоре поговорим), и готов к работе[10].

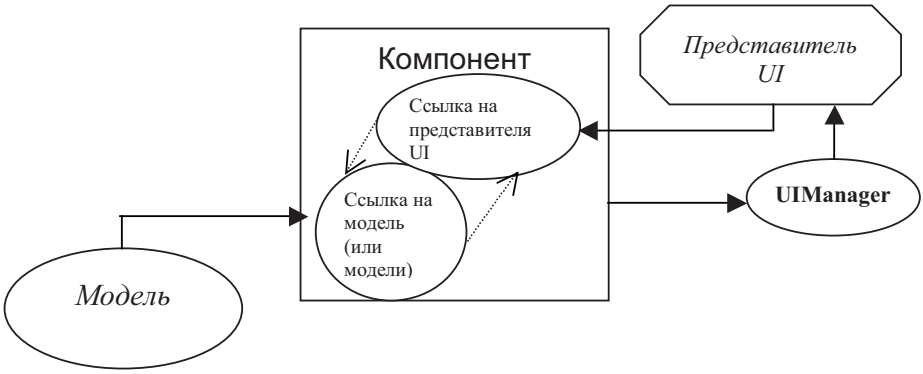



**Рис. 1.5.** Взаимоотношения UI-представителя и модели

Следует четко осознать, что именно классы компонентов (такие как JButton и JTable) являются основной частью библиотеки Swing. Может показаться, что они не так уж важны: ведь в них не происходит ни прорисовка компонента, ни обработка событий, ни манипуляция данными. Однако это не так: UI-представители и модели являются лишь частью внутреннего механизма библиотеки, а сами они не представляют большого интереса. Компонент просто *делегирует* к ним запросы: представитель осуществляет прорисовку и обработку событий, а модель хранит данные. Как мы уже выяснили, это обеспечивает великолепную гибкость библиотеки. Главными остаются компоненты Swing — все свойства (название кнопки, данные таблицы и т. п.) принадлежат им, они являются компонентами JavaBeans, ими вы манипулируете в своей программе или в визуальном средстве, именно они получают события от операционной системы и ресурсы для прорисовки, которые затем передают для выполнения действий в UI-представителей.

## Управление внешним видом и поведением программы

В библиотеке Swing довольно много компонентов, и каждый из них имеет своего UI-представителя, ответственного за обработку событий и прорисовку компонента на экране. Рано или поздно настает момент, когда внешний вид и поведение вашего Java-приложения приходится менять (например, чтобы оно выглядело одинаково с приложениями той платформы, на которой ему приходиться работать). Если бы разработчику

[10] Если использовать терминологию шаблонов проектирования, то можно сказать, что с точки зрения UI-представителей и моделей классы компонентов Swing действуют как посредники (mediators), обеспечивая слабую связанность системы. С точки же зрения программистов-клиентов Swing классы компонентов являются фасадами (facade) для архитектуры «модель-представитель» (применяя компонент, не обязательно задумываться о том, что происходит внутри него).

пришлось менять UI-представителя индивидуально для каждого компонента, это было бы не только утомительно и долго, но и внесло бы множество ошибок.

Поэтому в Swing управление внешним видом осуществляется в специальном классе UIManager. Он позволяет вам установить внешний вид и поведение для всех компонентов библиотеки сразу. Для этого нужно лишь вызвать статический метод этого класса setLookAndFeel() и передать в него объект класса LookAndFeel. Объект LookAndFeel — это хранитель информации об определенном внешнем виде и поведении программы, в нем содержится информация о UI-представителях, название внешнего вида, а также методы, упрощающие работу класса UIManager. По умолчанию компоненты автоматически «выбирают» себе UI-представителя именно с помощью класса UIManager.

Внешние виды для компонентов Swing, которые поставляются с пакетом разработки JDK 1.6 последних ревизий (и список не планируется дополнять в новой версии 1.7), перечислены в табл. 1.1.

**Таблица 1.1.** Доступные внешние виды компонентов Swing

| Название | Местонахождение | Предназначение |
|---|---|---|
| Внешний вид и поведение приложений на Java (внешний вид Metal, его современный подвид называется Ocean, или внешний вид, не зависящий от платформы) | Пакет javax.swing.plaf.metal<br><br>Класс MetalLookAndFeel | Именно этот внешний вид используется в Swing по умолчанию в версиях до 1.7 (если вы явно не установите другой). Специально разработан создателями Swing, для того чтобы придать приложениям на Java собственный уникальный вид. Подходит для всех платформ. Позволяет до некоторого предела менять цвета, шрифты и другие аспекты внешнего вида, но не слишком сильно — для этого предназначены так называемые «темы». |
| Полностью настраиваемый внешний вид Synth | Пакет javax.swing.plaf.synth<br><br>Класс SynthLookAndFeel | Пакет, позволяющий менять внешний вид приложений с помощью изменяемых «обоев» (skins) или цветов. Сам он не определяет никакого вида, это система для поддержки реализации внешнего вида с помощью изображений или цветов. Может быть настроен как через классы, так и через файл настроек в формате XML. |
| Платформенно-независимый внешний вид Nimbus, предположительно замена для Metal | Пакет com.sun.java.swing.plaf.nimbus<br><br>Класс NimbusLookAndFeel | Внешний вид нового поколения, построенный на базе настраиваемого вида Synth. Преимуществом данного вида без сомнения является возможность легко и самыми малыми штрихами менять внешний вид компонентов, «подгоняя» их под свои нужды. Внешне более тяготеет к Unix-стилю. Еще одним плюсом является использование векторной графики и соответственно независимость от разрешения экрана и возможность менять размеры компонентов. |

**Таблица 1.1** *(продолжение)*

| Название | Местонахождение | Предназначение |
|---|---|---|
| Внешний вид и поведение Windows-приложений | Пакет com.sun.java.swing.plaf.windows<br><br>Класс WindowsLookAndFeel | Этот внешний вид предназначен для эмуляции Windows-приложений. Его можно использовать только при работе под Windows. Отличается при работе на Windows XP и других версиях некоторыми визуальными отличиями и эффектами. Внешний вид Windows Vista пока что не реализован. |
| Внешний вид и поведение Unix-приложений | Пакет com.sun.java.swing.plaf.motif<br><br>Класс MotifLookAndFeel | Позволяет приложениям на Java выглядеть аналогично Unix-приложениям с использованием среды CDE/Motif. Подходит для любых платформ (не только для Unix) |

Кроме перечисленных внешних видов, входящих в стандартный инструментарий JDK, компании Apple и Sun также разработали внешний вид Macintosh (Mac Look & Feel). Если ваше приложение будет работать на платформе Mac, и вы хотите, чтобы оно выглядело соответственно, то можно загрузить этот внешний вид с сайта java.sun.com, либо он по умолчанию будет включен в пакете JDK для Macintosh. Однако внешний вид Mac, так же как и внешний вид Windows, можно использовать только на соответствующей платформе (таковы требования корпораций Apple и Microsoft). Специально для получения внешнего вида, соответствующего платформе, на которой работает приложение, в классе UIManager определен метод getSystemLookAndFeel(). При работе под управлением Windows этот метод вернет вам внешний вид Windows, а при работе под Unix – внешний вид Unix.

Лучше всего менять внешний вид и поведение перед тем, как на экране появится окно вашего приложения. Хотя никто не запрещает вам менять внешний вид прямо во время работы программы, в таком случае компоненты не изменятся автоматически, и вам придется вызывать специальный метод класса SwingUtilities, чтобы обновить их. К тому же при изменении внешнего вида прямо во время работы программы могут возникнуть проблемы с размерами компонентов и их расположением в контейнере – разные внешние виды придают компонентам разные размеры, и то, что прекрасно смотрится во внешнем виде Metal, может выглядеть ужасным при переходе к внешнему виду Motif. Кстати, это типичная ситуация для начинающих работать со Swing программистов: они настолько воодушевляются возможностью тасовать внешние виды и менять поведение своего приложения, что поначалу только этим и занимаются, не обращая внимания на то, что в результате внешний вид приложения сильно страдает.

В принципе, оптимальным для приложения является использование одного внешнего вида и одного варианта поведения. Такое заявление может показаться странным: как же отказываться от великолепного механизма, позволяющего одной строчкой кода полностью сменить внешность и реакцию приложения? Разнообразие еще никому не вредило, но как показывают время и уже созданные Java-приложения, широкий выбор внешних видов не позволяют получать по-настоящему эффектные приложения. Дело в том, что при разработке пользовательского интерфейса первоклассных программ учитываются рекомендации создателей компонентов этого интерфейса, что дает возможность добиваться наилучших результатов. Но невозможно сделать то же самое с помощью внешних видов для конкретных платформ: если вы создадите эффектное приложение с внешним видом Windows, полностью следуя рекомендациям Microsoft, вы не

сможете перенести его на Unix, потому что использовать внешний вид Windows на других платформах запрещено (а рекомендации Microsoft для интерфейса Unix не подходят, и это еще мягко сказано). Разрабатывая приложение под Mac, вы будете следовать рекомендациям Apple, и, сменив внешний вид приложения, можете сильно удивиться результатам.

Здесь на передний план выходит независимый от платформы внешний вид, Nimbus или Metal, специально созданный для Java-приложений. Компания Sun разработала для него ряд рекомендаций, выполняя которые, можно получить по-настоящему красивые интерфейсы. Мы подробно рассмотрим эти рекомендации и этапы воплощения их в жизнь в главе 7, когда будем говорить о размещении компонентов в контейнере. Создав интерфейс специально для независимого внешнего вида, вы с легкостью перенесете его на любую платформу. Все сказанное не стоит воспринимать как совет отказаться от внешних видов, эмулирующих известные платформы, но, как показывает практика, их использование все равно не обеспечивает полного соответствия «родным» приложениям этих платформ. Дело в том, что Swing всегда находится на шаг позади (сначала меняется интерфейс конкретной платформы, команда Swing разрабатывает внешний вид, эмулирующий этот интерфейс, обновленный внешний вид выходит в новом пакете JDK, а в это время интерфейс конкретной платформы опять меняется, пусть даже и ненамного). За изменения же внешнего вида Java можно не беспокоиться, потому что он меняется одновременно со Swing.

Некоторые вопросы вызывает внешний вид компонентов в стиле Nimbus и особенно Metal. Многие, мягко говоря, не находят их привлекательными (следует признать, что причины для недовольства имеются — слишком уж «топорно» выглядит этот лаконичный интерфейс по сравнению с современными системами пользовательских интерфейсов). Но вы вовсе не ограничены внешними видами, созданными в Sun. На данный момент имеется умопомрачительное количество разнообразных внешних видов, некоторые их которых определенно стоят того, чтобы на них обратили внимание. Вы вполне можете задействовать вместо него один из интерфейсов от стороннего производителя, по-прежнему выполняя рекомендации для интерфейсов от Sun, потому что большинство сторонних внешних видов являются просто производными от Metal/Synth/Nimbus, оставляя без изменения поведение, размеры компонентов и пр. и меняя лишь изображения. Для начала можно зайти на сайт http://www.javootoo.com/, где найдутся все наиболее популярные внешние виды для Swing, например очень популярный и используемый во многих коммерческих продуктах с применением Swing внешний вид JGoodies Looks (при использовании в интерфейсе рекомендаций для внешнего вида Metal и компонентов во внешнем виде JGoodies Looks получаются приложения, способные «обставить» самые продуманные и изысканные пользовательские интерфейсы). В качестве неплохой и бесплатной замены внешнему виду Metal/Nimbus хорошо подходит внешний вид Substance, Liquid и многие другие.

С другой стороны, новый внешний вид Nimbus, который возможно станет внешним видом по умолчанию в версии JDK 1.7, намного приятнее и современнее своего предшественника Metal. Его цветовая гамма очевидно стремится к Unix, однако, как легко узнать из его документации, цвета Nimbus легко меняются несколькими настройками (управляют основными эффектами три базовых цвета), а тот факт, что Nimbus построен на Synth, позволяет подменять изображения, цвета или процедуру прорисовки любого компонента. Это может помочь создать намного более приятный внешний вид, чем тот, что доступен по умолчанию, с достаточно маленькими затратами.

Но в целом, подключаемые внешний вид и поведение (Pluggable Look And Feel, PLAF) — одно из самых мощных свойств Swing. Никакая другая библиотека или операционная система не позволяет осуществить подобные масштабные действия настолько просто и быстро. Вы можете разработать для своего приложения любой вид, не задумываясь о платформах и их различиях, создать совершенно особый колорит, подчерки-

вающий предназначение вашего приложения (конечно, это довольно долго и недешево, но всегда можно использовать внешний вид от стороннего производителя). Никто не запрещает вам создать абсолютно новаторский трехмерный интерфейс или интерфейс, основанный только на системе синтеза и распознавания речи. Причем вам не нужно будет изменять ни строчки кода в вашем приложении.

## *Специальные средства для пользователей с ограниченными возможностями*

Практически в любой современной системе создания пользовательского интерфейса поддерживаются специальные средства для работы людей, которые по каким то причинам (травмы или болезни) не способны работать с компьютером обычным образом (они могут не видеть экран, им может быть неудобно манипулировать мышью и клавиатурой). Зачастую единственным, что связывает таких людей с окружающим миром, становится компьютер. Поэтому очень важно обеспечить приложения механизмом, позволяющим специальными средствами передавать информацию пользователям в нужном виде.

Надо сказать, что, создавая приложение с помощью Swing, вы уже выполняете большую часть работы, необходимой для поддержки специальных средств. Разработчики Swing учли важность этой поддержки и встроили во все компоненты особую информацию. Эта информация (она может быть передана специальным средствам, которые обработают ее надлежащим образом) описана во внутренних классах компонентов, использующих библиотеку Accessibility (эта библиотека относится к набору Java Foundation Classes).

Итак, в каждом компоненте библиотеки Swing имеется внутренний класс, имя которого составляется из названия класса компонента и слова «Accessible», например, в классе кнопки (JButton) имеется внутренний класс AccessibleJButton. В классах AccessibleXXX содержится исчерпывающая информация о компонентах Swing (набор «ролей», которые исполняют графические компоненты в пользовательском интерфейсе; действия, которые можно совершать над компонентом, основные свойства компонента, например, текст надписи). Если на компьютере установлено специальное средство, оно находит эту информацию (вызывая для этого метод getAccessibleContext(), встроенный в общий класс всех компонентов Swing JComponent) и выводит ее в надлежащем виде (например, для человека с расстройством зрения интерфейс будет «прочитан» путем синтеза речи). Практически все известные средства для «чтения» интерфейсов или работы через специальные устройства поддерживают библиотеку Accessibility, к примеру, известная программа JAWS.

Вся работа фактически уже сделана разработчиками Swing: например, если вы создаете кнопку с надписью (не предпринимая никаких дополнительных усилий), специальное средство сразу получит информацию о том, что в пользовательском интерфейсе программы имеется кнопка, название кнопки и данные о том, что кнопку можно «нажать». Что делать с этой информацией дальше, зависит от ситуации и от того, как именно пользователю удобно получать информацию. Самое главное здесь — это то, что вы просто пишите приложение, совершенно не задумываясь о том, что в дальнейшем оно может быть использовано человеком с ограниченными возможностями, и, тем не менее, вся необходимая информация будет на месте. Учитывая другие достоинства Swing: независимость от конкретной платформы, способность динамически менять внешний вид и поведение (что особенно важно для специальных средств, при использовании которых возможно придется увеличивать размеры компонентов, по-особому обрабатывать действия, озвучивать их), простую локализацию и поддержку Unicode, можно сказать, что для специальных средств библиотека Swing подходит как нельзя лучше.

Если вам вдруг понадобиться проверить, как Swing справляется с поддержкой специальных средств, это всегда можно сделать. Загляните на официальный сайт java.sun.com, там вы найдете несколько инструментов для тестирования, а также ссылки на реальные специальные средства с поддержкой библиотеки Accessibility.

## Локализация приложений

В современном мире, который уже практически полностью сдался на милость глобализации и информатизации, возможность быстро и без особых усилий подготовить приложение для работы на других языках и в других условиях невероятно ценна. В общем-то, разрабатывая приложение для более или менее широкого круга пользователей, особенно если оно разрабатывается в большой корпорации, можно сразу же начинать готовиться к тому, что его нужно будет выпускать сразу на нескольких языках.

Библиотека Swing великолепно справляет с задачей представления приложений на нескольких языках, во многом потому, что сам язык Java создавался как язык, способный корректно работать в таких ситуациях. Вынос всего текста, используемого в приложении, во внешние файлы .properties, поддержка этими файлами всего спектра символов Unicode, а значит всех мировых языков, хранение всех строк в самой программе как Unicode, позволяют безболезненно переводить программу на любые языки. Поддержка часовых поясов, дат, времени, чисел всех мировых культур также встроена в Java и доступна в Swing. Легко изменяемый внешний вид, реакция на действия пользователей, доступная нам благодаря подключаемому внешнему виду и поведению, позволит адаптировать или даже «замаскировать» приложение, так что его легко можно будет принять за другое, или написанное специально для определенных компаний или условий.

Все это еще раз доказывает, что удачная модель библиотеки Swing, вкупе с возможностями самого языка Java, позволяет создавать приложения самой сложной структуры и рассчитывать на их легкую поддержку и широкую масштабируемость.

## Резюме

Итак, мы узнали, что основой Swing является библиотека AWT. Тем не менее, возможности библиотеки Swing гораздо богаче, к тому же она по праву «носит звание» библиотеки, полностью написанной на Java. Благодаря технологии JavaBeans в визуальных средствах разработки программ удалось упростить использование компонентов Swing и классов, поля и методы которых формируются по простым правилам. Подключаемые внешний вид и поведение компонентов, а также встроенная поддержка специальных возможностей придают Swing необычайную гибкость и избавляют нас от многих проблем и рутинной работы.

# Глава 2. Модель событий

Графический пользовательский интерфейс (GUI) относится к *системам, управляемым по событиям* (event-driven systems). При запуске программы вы создаете пользовательский интерфейс, а затем ждете наступления некоторого события: нажатия клавиши, движения мыши или изменения компонента системы. При наступлении события программа выполняет необходимые действия, а затем снова переходит к ожиданию. Программа, использующая для создания пользовательского интерфейса библиотеку Swing, не является исключением. В этой главе мы увидим, как обрабатывать события в Swing, и рассмотрим основные типы событий, общие для всех графических компонентов.

Для любой библиотеки пользовательского интерфейса очень важно качество используемой в ней системы обработки событий. Как бы библиотека ни была хороша внешне или функционально, неудачно реализованная обработка событий сведет все ее преимущества «на нет». Библиотека AWT из первого выпуска JDK — хороший пример. Несмотря на то, что она не блистала качеством и внешним видом компонентов, основным нападкам подвергалась именно неудачная система обработки событий, и, будь она получше, возможно, у AWT было бы более светлое будущее. Однако система обработки событий AWT была не слишком приспособлена к созданию масштабных программ — код, обрабатывающий события, находился прямо в классах компонентов: приходилось наследовать от них, искать нужное событие, пользуясь уже набившими оскомину операторами if и switch, и смешивать пользовательский интерфейс с деловой логикой программы. В итоге получалась программа, которую иногда было тяжело поддерживать и обновлять. Все это привело к тому что библиотека AWT была принята не слишком-то тепло.

К счастью, нам не придется возвращаться во времена старой системы обработки событий, и вспомнили мы ее лишь из уважения к истории развития библиотек пользовательского интерфейса в Java. Библиотека Swing использует систему обработки событий JavaBeans, и система эта действительно хороша. С одной стороны, она проста и понятна, с другой, предоставляет множество способов обработки событий, и вам остается лишь выбрать из них наиболее подходящий. Основным достоинством системы обработки событий Swing следует признать то, что как бы вы ни писали свою программу, какие бы способы создания интерфейса ни применяли, код, отвечающий за создание интерфейса, будет отделен от кода, обрабатывающего события. Это позволит и легко обновлять программу, и легко понимать ее.

Прежде чем перейти к описанию системы обработки событий Swing, мы узнаем, на базе какого решения она была создана. Во времена прежней системы существовало мнение, что хорошая система обработки событий в Java не будет создана до тех пор, пока в языке не появятся указатели на методы, что позволило бы использовать технику *обратных вызовов* (callbacks). Эта техника довольно характерна для обработки событий — вы передаете графическому компоненту ссылку на метод, который будет обрабатывать событие, и при возникновении этого события компонент вызывает ваш метод по полученной ссылке. Элегантное решение было найдено, когда создатели новой системы обработки событий решили использовать опыт объектно-ориентированного программирования, обратившись к шаблонам проектирования.

## *Наблюдатели*

В программировании довольно часто возникает ситуация, когда при изменении данных в одном месте программы необходимо тут же обновить данные в другом месте программы. Если мы используем объектно-ориентированный язык программирования, это означает, что при изменении состояния одного объекта необходимо каким-либо образом изменить другой объект. Самое простое решение — организация непосредственной связи между объектами, то есть поддержание ситуации, когда объекты хорошо осведомлены о существовании друг друга. Первый объект знает, что изменение его состояния интересует другой объект, у него есть ссылка на этот объект, и каждый раз при изменении своего состояния он сообщает об этом другому объекту. В такой ситуации второй объект (получающий сообщения об изменениях) чаще всего нуждается в дополнительной информации о первом объекте (в котором происходят изменения), поэтому у него есть ссылка на первый объект. В результате получается то, что мы называем *сильно связанными* объектами — два объекта хранят ссылки друг на друга.

Однако это простое решение оказывается весьма ограниченным. Любое последующее изменение в программе приведет к тому, что придется основательно ее переписывать. Если изменениями в первом объекте интересуется какой-то объект, вполне возможно, что в будущем этими изменениями заинтересуется еще один объект, а потом еще несколько. Единственным способом обеспечить их информацией об изменениях в объекте станет переписывание исходного кода программы, чтобы включить в него ссылки на новые объекты. Однако делать это каждый раз при изменении количества заинтересованных объектов неудобно, долго и чревато ошибками. А если до запуска программы количество заинтересованных объектов вообще неизвестно, то наше простое решение совсем не подойдет.

Описанная ситуация встречается довольно часто, так что неудивительно, что наилучший вариант решения уже был найден в форме наблюдателя — одного из самых известных шаблонов объектно-ориентированного проектирования. Наблюдатель определяет, как следует организовать взаимоотношения между объектами, чтобы избежать сильной связи между ними и таким образом добиться необходимой гибкости. Давайте посмотрим, что этот шаблон из себя представляет.

Итак, объект, за которым ведется наблюдение, называется *субъектом* (subject). Объект, заинтересованный в изменениях субъекта, называются *наблюдателем* (observer). В обязанности субъекта входит добавление наблюдателей (в идеале количество наблюдателей произвольно), отправка наблюдателям сообщений об изменениях своего состояния и отсоединение ранее добавленных наблюдателей. Наблюдатели проще субъектов: они определяют методы, которые следует вызывать субъекту, для того чтобы сообщить о своих изменениях (рис. 2.1).

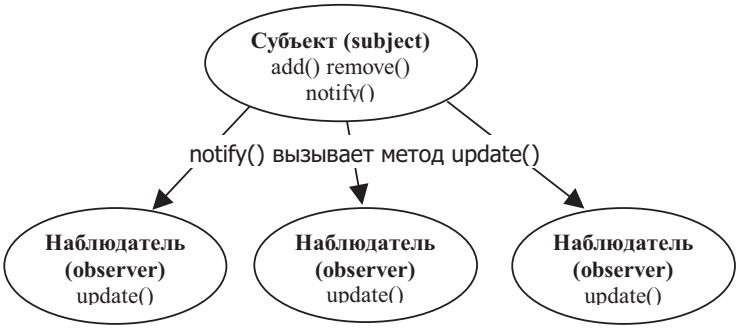



**Рис. 2.1.** Взаимоотношения наблюдателей и субъектов

Рисунок показывает, что субъект обладает тремя методами: метод add() позволяет добавить очередного наблюдателя (как правило, наблюдатели хранятся в виде списка, что позволяет иметь произвольное их количество и легко манипулировать ими); метод remove() позволяет удалить ранее добавленного наблюдателя; метод notify() сообщает наблюдателям, добавленным ранее методом add(), о смене состояния субъекта. Для этого он вызывает определенный во всех объектах-наблюдателях специальный метод, в нашем случае это метод update(). В свою очередь наблюдатели знают, что очередной вызов метода update() означает смену состояния субъекта, и выполняют в этом методе все необходимые действия. Чтобы субъекты и наблюдатели смогли работать друг с другом, их функции описываются в базовых классах (или интерфейсах), и объекты, которым необходимо знать друг о друге, но которым нежелательно быть сильно связанными, наследуют от этих базовых классов или реализуют интерфейсы и начинают работать как субъекты и наблюдатели.

Ничего сложного в субъектах и наблюдателях нет, тем не менее, они позволяют объектам просто и чрезвычайно элегантно узнавать друг о друге. Объекты, изменения в которых могут быть интересны окружающим, становятся субъектами (то есть просто поддерживают список наблюдателей), а заинтересованным в их изменениях объектам нужно лишь определить методы наблюдателей и зарегистрировать свой интерес к субъектам. Никто не запрещает субъектам самим быть наблюдателями, и наоборот. В Java субъекты и наблюдатели могут работать особенно эффективно — возможность определять внутренние классы позволяет одному классу одновременно иметь информацию о нескольких совершенно разных субъектах и эффективно обрабатывать ее.

Нетрудно заметить, что шаблон наблюдателя должен прекрасно работать в системах обработки событий, происходящих в пользовательском интерфейсе. Действительно, субъектами являются различные графические компоненты, из которых состоит интерфейс — кнопки, списки, текстовые поля, а программист определяет, что в них происходит (щелчки на кнопках, перемещения мыши), описывает объекты-наблюдатели и решает, как поступить при смене состояния интересующего его компонента. По сути, все действует по законам знакомой нам по первой главе архитектуры «модель-вид-контроллер», только на более высоком уровне. Компонент представляет собой вид, в модели хранятся данные, так, как это удобно программе, ну а наблюдатель является контроллером, только уже более высокого уровня, который определяет что будет происходить в программе в ответ на происходящие события, то есть контроллером всей программы или ее части.

Интересно, что вся библиотека Swing буквально «напичкана» субъектами и наблюдателями. Мало того, что эта концепция используется при обработке событий, она еще позволяет моделям и UI-представителям Swing иметь самую свежую информацию друг о друге. И модели, и UI-представители одновременно представляют собой субъектов и наблюдателей — при изменении данных модели она уведомляет об этом UI-представителя (выступающего в качестве наблюдателя), и тот обновляет внешний вид компонента в соответствии с новыми данными. Если же в ответ на действие пользователя меняется внешний вид компонента (а за это отвечает UI-представитель), то уже модель становится наблюдателем и получает уведомление о том, что данные необходимо изменить.

Но вернемся к системе обработки событий Swing. Она на самом деле основана на отношении вида субъект-наблюдатель. Субъектами являются компоненты Swing (кнопки JButton, списки JList и т. п.), а наблюдателями — специальные объекты, которые называют *слушателями*. Для того чтобы узнать о каком-либо событии, надо написать соответствующего слушателя и присоединить его к компоненту.

## *Слушатели*

*Событие* (event) в пользовательском интерфейсе — это либо непосредственное действие пользователя (щелчок или движение мыши, нажатие клавиши), либо изменение состояния какого-либо компонента интерфейса (например, щелчок мыши может при-

вести к нажатию кнопки). *Источником события* (event source) в Swing может быть любой компонент, будь то кнопка, надпись с текстом или диалоговое окно. Для того чтобы узнавать в своей программе о происходящих в компоненте событиях, нам необходимо сообщить компоненту о своей заинтересованности. Сделать это можно, передав компоненту *слушателя* (listener) определенного события. Слушатель — этот тот самый наблюдатель (разница только в названиях), которому компонент будет сообщать о происходящих в нем событиях.

Каждому типу события соответствует свой слушатель. Так гораздо проще следить именно за теми событиями, что нас интересуют, а не за всеми подряд — код, обрабатывающий щелчки мыши, будет отделен от кода, ответственного за нажатия клавиш. Обязанности слушателей (то есть методы, которые в них должны быть определены) описаны в соответствующих интерфейсах. Вы реализуете в своем классе нужный вам интерфейс слушателя, передаете его в интересующий вас компонент и спокойно ждете наступления события. Давайте рассмотрим простой пример, в котором мы попытаемся узнать о нажатиях клавиш при активном окне.

```
// FirstEvents.java
// События - нажатия клавиш на клавиатуре
import javax.swing.*;
import java.awt.event.*;

public class FirstEvents extends JFrame {
    public FirstEvents() {
        super("FirstEvents");
        // при закрытии окна - выход
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        // регистрируем нашего слушателя
        addKeyListener(new KeyL());
        // выводим окно на экран
        setSize(200, 200);
        setVisible(true);
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(
         new Runnable() {
          public void run() { new FirstEvents();  } });
   }
}
// этот класс будет получать извещения о событиях
class KeyL implements KeyListener {
    // печать символа
    public void keyTyped(KeyEvent k) {
        System.out.println(k);
    }
    // нажатие клавиши
```

```
    public void keyPressed(KeyEvent k) {
        System.out.println(k);
    }
    // отпускание нажатой клавиши
    public void keyReleased(KeyEvent k) {
        System.out.println(k);
    }
}
```

Пример очень прост — мы создаем класс, унаследованный от окна JFrame, устанавливаем для него размер методом setSize(), указываем, что при закрытии окна следует завершить работу приложения (методом setDefaultCloseOperation(), подробнее об этом методе мы узнаем в главе 4, посвященной окнам) и выводим окно на экран. Правда сам запуск приложения происходит через некий таинственный класс SwingUtilities, однако такова архитектура Swing. Очень скоро мы узнаем, почему не стоит вызывать конструктор окна напрямую, а сейчас гораздо интереснее посмотреть, как создается слушатель события.

Прежде всего, необходимо написать класс, реализующий интерфейс слушателя. Если вы просмотрите интерактивную документацию Java, то увидите, что для получения информации о нажатиях клавиш используется интерфейс KeyListener, именно его мы и реализовали в классе KeyL. В этом интерфейсе определены три метода, каждый из которых вызывается при наступлении определенного события: keyPressed() и keyReleased() — при нажатии и отпускании клавиши, keyTyped() — при печати символа (когда нажимается и отпускается клавиша, соответствующая печатному символу[1]). В качестве параметра каждому методу передается объект KeyEvent, который используется для получения дополнительной информации о событии (кода клавиши, источника события и т. д.). Чтобы сделать программу максимально простой, мы просто передаем эту информацию в стандартный поток вывода.

Последний этап — регистрация нашего слушателя в интересующем нас компоненте. Компонент у нас в программе только один — это наше окно. Для регистрации слушателя событий от клавиатуры мы вызываем метод addKeyListener(), в который передаем ссылку на объект класса KeyL. После этого остается только запустить программу и посмотреть, какие сообщения она выводит в стандартный поток вывода при нажатиях клавиш (конечно, события от нажатий клавиш будут возникать только при активном окне).

Подобным образом обрабатываются все события в Swing. Когда вы собираетесь обработать какое-то событие, то, прежде всего, выясняете, какой слушатель получает сообщения о событиях этого типа, создаете класс, реализующий обязанности слушателя, и регистрируете его в интересующем вас компоненте. Процесс очень прост, но удивительно эффективен: он позволяет идентифицировать и обрабатывать лишь те события, которые нас интересуют, не выискивая нужное в потоке всех событий; причем события обрабатываются отдельно от компонентов, где эти события происходят, что позволяет точно отделять части программы, связанные с созданием пользовательского интерфейса, от частей, реализующих деловую логику программы.

---

[1] Метод keyTyped() — более высокого уровня, чем остальные два слушателя KeyListener. Он сообщает программе, что пользователь нажал и отпустил или удерживает клавишу с печатным символом (печатные символы — это все буквы, цифры, а также пробел и клавиши Enter и Esc), однако эти печатные символы могут не иметь точного отображения на одну клавишу клавиатуры, в то время как другие два метода сообщают именно о нажатии физических клавиш.

## *Схема именования событий JavaBeans*

Система обработки событий в Swing является частью архитектуры JavaBeans, которая позволяет создавать переносимые и легко используемые графические компоненты для визуальных средств разработки программ. В главе 1 мы узнали, что все компоненты Swing являются компонентами JavaBeans. Как вы помните, основой JavaBeans является соглашение об именах, которое позволяет визуальным средствам легко узнавать, какими свойствами обладает компонент. Для этого компонент определяет набор методов со специальными именами get/set. Методы эти служат для считывания и записи значений свойств компонента.

События в JavaBeans обрабатываются с помощью слушателей; как это происходит, мы только что увидели. Нетрудно понять, что для того чтобы визуальное средство смогло определить, какие события могут возникать в компоненте, также необходим какой-то механизм для выявления этих событий. Таким механизмом является специальная схема именования событий и обрабатывающих их слушателей.

У каждого события есть имя. Например, в примере из предыдущего раздела этим именем было слово «Key» (клавиша) — неудивительно, ведь это событие происходит при нажатии клавиш на клавиатуре. События, которые происходят в окнах, называются «Window» (окно); в общем случае будем считать, что названием события являются просто символы «XXX». Чтобы событие стало доступно визуальному средству разработки, необходимо проделать описанную ниже процедуру.

1. Определить класс, в котором будет храниться информация о произошедшем событии (что это будет за информация, определяет создатель события). Класс должен быть унаследован от базового класса java.util.EventObject и иметь название вида XXXEvent, где «XXX» — это название нашего события. В предыдущем примере мы видели, что информация о событиях от клавиатуры хранилась в классе с названием KeyEvent.

2. Создать интерфейс слушателя, в который будет приходить информация о событии. Это должен быть именно интерфейс, а не класс. Название интерфейса должно иметь следующий вид: XXXListener — в предыдущем примере мы использовали слушателя с именем KeyListener. Этот интерфейс должен быть унаследован от базового интерфейса всех слушателей событий java.util.EventListener (это пустой интерфейс без методов, он просто помечает то, что унаследованный от него интерфейс является слушателем). В интерфейсе может быть определено сколь угодно много методов, единственное требование к этим методам — наличие параметра типа XXXEvent. Никаких других параметров у методов быть не должно.

3. Включить поддержку события в класс компонента, в котором это событие может происходить. Чтобы сделать это, необходимо определить два метода: один для присоединения слушателей, другой для их отсоединения. Названия методов должны выглядеть следующим образом: addXXXListener() — для метода, присоединяющего слушателей, и removeXXXListener() — для метода, отсоединяющего слушателей. Если вспомнить пример, то там присоединение слушателя происходило как раз с помощью метода addKeyListener(), определенного в классе окон JFrame (на самом деле этот метод определен в базовом классе всех компонентов Component, и добавлять слушателей клавиатуры можно к любому компоненту).

Если данные требования выполнены, то визуальное средство легко найдет все типы событий, поддерживаемые компонентом, соответствующих им слушателей и определит методы для работы с этими слушателями. Это прекрасно, и мы уже от-

мечали, насколько JavaBeans упрощает создание компонентов. Сейчас для нас гораздо важнее то, что, зная правила, по которым создаются события JavaBeans (а компоненты Swing используют именно эти события), мы можем очень просто определить всю интересующую информацию о событии, не перерывая документацию. Предположим, нам захотелось узнать, что происходит в окне нашего приложения. Скорее всего, события, происходящие в окнах, описаны в классе WindowEvent. Если это так, то согласно схеме именования нам нужно реализовать интерфейс WindowListener и зарегистрировать его в нашем окне с помощью метода addWindowListener(). Как оказывается, именно такие классы и интерфейсы позволяют обрабатывать оконные события, и, хотя мы не знали об этом, а просто предполагали, наши предположения полностью подтвердились!

Система обработки событий JavaBeans удивительно прозрачна и понятна: достаточно один раз понять простые принципы, лежащие в ее основе, как она мгновенно становится совершенно незаменимым инструментом в работе, помогающим справиться с большинством проблем, которые возникают при обработке событий.

## *Стандартные события*

Теперь, когда мы познакомились с тем, как слушатели позволяют обрабатывать события, и рассмотрели правила, по которым составляются имена событий, слушателей и методов, можно переходить к изучению стандартных событий, поддерживаемых всеми графическими компонентами.

Надо сказать, что события в Java условно разделяются на *низкоуровневые* (low-level events) и *высокоуровневые* (high-level events). К низкоуровневым событиям относят те, что происходят непосредственно в результате действий пользователя: это движения мыши, передача фокуса ввода от одного приложения другому, нажатия клавиш и т. п.[2] Они поступают в Java-программу от операционной системы или от внутренних механизмов виртуальной машины. Высокоуровневые события происходят в результате изменения состояния компонента. Такие события поступают не от операционной системы, а создаются самим компонентом. Процесс создания события еще называют *запуском* (fire). Во многих компонентах Swing вы можете увидеть методы с именами вида fireXXX(); именно в таких методах создаются объекты с информацией о событиях, которые затем рассылаются слушателям. Часто события высокого уровня возникают после того, как происходят несколько событий низкого уровня (например, кнопка сообщает о своем нажатии, после того как над ней была нажата и отпущена кнопка мыши).

Начнем мы с низкоуровневых событий (табл. 2.1). Эти события могут возникать в любом графическом компоненте, унаследованном от класса java.awt.Component (правда, есть несколько исключений).

**Таблица 2.1.** Основные низкоуровневые события

| Краткое описание события | Методы слушателя | Источник события |
|---|---|---|
| Событие от клавиатуры. Описано в классе KeyEvent. Возникает, когда пользователь нажимает клавишу | keyPressed(KeyEvent), keyReleased(KeyEvent), keyTyped(KeyEvent) | Все компоненты (наследники класса java.awt.Component) |

[2] Низкоуровневые события легко отличить от высокоуровневых — все они унаследованы от особого базового класса AWTEvent.

**Таблица 2.1** *(продолжение)*

| Краткое описание события | Методы слушателя | Источник события |
|---|---|---|
| Нажатия и отпускания кнопок мыши и попадание курсора в область компонента. Класс MouseEvent позволяет следить за состоянием кнопок мыши и контролировать нахождение указателя мыши в определенной области | mouseClicked(MouseEvent), mouseEntered(MouseEvent), mouseExited(MouseEvent), mousePressed(MouseEvent), mouseReleased(MouseEvent) | Все компоненты |
| Перемещение мыши. Класс события тот же — MouseEvent, но вот слушатель называется по-другому — MouseMotionListener[3] | mouseDragged(MouseEvent), mouseMoved(MouseEvent) | Все компоненты |
| Прокрутка колесика мыши. Класс MouseWheelEvent | mouseWheelMoved(MouseWheelEvent) | Все компоненты |
| Передача фокуса ввода. Класс FocusEvent | focusGained(FocusEvent), focusLost(FocusEvent) | Все компоненты |
| Добавление или удаление компонентов в контейнере. Класс ContainerEvent | componentAdded(ContainerEvent), componentRemoved(ContainerEvent) | Все контейнеры (наследники класса java.awt.Container) |
| Изменения состояния окна. Класс WindowEvent. Позволяет узнать о перемещении, сворачивании, закрытии окна и т. д. | windowActivated(WindowEvent), windowClosed(WindowEvent), windowClosing(WindowEvent), windowDeactivated(WindowEvent), windowDeiconified(WindowEvent), windowIconified(WindowEvent), windowOpened(WindowEvent) | Окна (наследники класса java.awt.Window), например, окна с рамкой (JFrame) или диалоговые окна (JDialog) |

Вы видете, что в таблице нет названий слушателей и методов, предназначенных для добавления и удаления этих слушателей. Такие названия легко получить самим: достаточно вспомнить простые правила именования, которые мы рассмотрели в предыдущем разделе. Используя данную таблицу, можно написать слушатель для любого низкоуровневого события (в таблице описаны не все, а только наиболее важные низкоуровневые события; компоненты поддерживают еще несколько событий, которые используются крайне редко, вы можете познакомиться с ними в интерактивной документации).

Чтобы окончательно убедиться в том, что события эти на самом деле возникают, и компоненты сообщают о них слушателям, рассмотрим небольшой пример:

```
// LowLevelEvents.java
// Наблюдение за основными низкоуровневыми событиями
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
```

[3] Это единственное исключение из правила именования событий JavaBeans. По идее, для класса MouseEvent слушатель должен был бы называться MouseListener (как в предыдущей строке таблицы), но разработчики решили разбить этот слушатель на два.

```
public class LowLevelEvents extends JFrame {
    // сюда мы будем выводить информацию
    private JTextArea out;
    public LowLevelEvents() {
        super("LowLevelEvents");
        // при закрытии окна - выход
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        // добавим текстовое поле
        add(new JScrollPane(out = new JTextArea()));
        // и кнопку
        JButton button = new JButton("Источник событий");
        add(button, "South");
        // регистрируем нащего слушателя
        OurListener ol = new OurListener();
        button.addKeyListener(ol);
        button.addMouseListener(ol);
        button.addMouseMotionListener(ol);
        button.addMouseWheelListener(ol);
        button.addFocusListener(ol);
        // выводим окно на экран
        setSize(400, 300);
        setVisible(true);
    }
    // внутренний класс - слушатель событий
    class OurListener implements MouseListener, KeyListener,
        MouseMotionListener, MouseWheelListener,
            FocusListener {

        public void mouseClicked(MouseEvent e)
        {   out.append(e.toString() + "\n"); }
        public void mousePressed(MouseEvent e)
        {   out.append(e.toString() + "\n"); }
        public void mouseReleased(MouseEvent e)
        {   out.append(e.toString() + "\n"); }
        public void mouseEntered(MouseEvent e)
        {   out.append(e.toString() + "\n"); }
        public void mouseExited(MouseEvent e)
        {   out.append(e.toString() + "\n"); }
        public void keyTyped(KeyEvent e)
        {   out.append(e.toString() + "\n"); }
        public void keyPressed(KeyEvent e)
```

```
        {   out.append(e.toString() + "\n"); }
        public void keyReleased(KeyEvent e)
        {   out.append(e.toString() + "\n"); }
        public void mouseDragged(MouseEvent e)
        {   out.append(e.toString() + "\n"); }
        public void mouseMoved(MouseEvent e)
        {   out.append(e.toString() + "\n"); }
        public void focusGained(FocusEvent e)
        {   out.append(e.toString() + "\n"); }
        public void focusLost(FocusEvent e)
        {   out.append(e.toString() + "\n"); }
        public void mouseWheelMoved(MouseWheelEvent e)
        {   out.append(e.toString() + "\n"); }
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(
         new Runnable() {
          public void run() { new LowLevelEvents(); } });
    }
}
```

В этом примере мы создаем окно, добавляем в центр текстовое поле (помещенное в панель прокрутки JScrollPane, подробнее о ней мы узнаем в главе 13), а в нижнюю часть окна помещаем простую кнопку JButton, которая и будет служить источником событий. Далее к кнопке присоединяются слушатели разнообразных событий.

Интерфейсы слушателей реализованы в классе OurListener. В примере этот класс реализует обязанности сразу пяти слушателей, так что методов в нем довольно много. Каждый из этих методов просто выводит информацию о происшедшем событии в текстовое окно, где вы и можете ее изучить. Запустив приложение и совершая различные действия с кнопкой, вы сможете увидеть, когда, как и какие события в ней происходят (это как раз тот случай, когда вместо изучения исходного текста примера, лучше посмотреть, как он работает).

После того как вы насладитесь этим примером, мы познакомимся с самыми важными высокоуровневыми событиями. Они обрабатываются точно так же, как низкоуровневые (помните, разделение событий на две категории условно, их обработка ничем не отличается, разными являются лишь источники этих событий). Высокоуровневые события создаются (запускаются) самими компонентами и обозначают наиболее важные изменения в этих компонентах. У каждого компонента чаще всего имеются собственные события (у кнопки — это нажатие, у флажка — установка, у списка — выбор нового элемента), поэтому высокоуровневых событий очень много. Рассматривать их в отдельности от компонентов не имеет смысла, так что каждый раз при знакомстве с новым компонентом мы будем говорить и о том, какие события могут в нем возникать.

Однако есть несколько высокоуровневых событий, которые настолько часто используются в компонентах, что мы можем рассмотреть их сейчас (табл. 2.2). Это, прежде всего, события, которые позволяют компонентам сообщать об изменениях своих свойств.

**Таблица 2.2.** Наиболее часто используемые высокоуровневые события

| Краткое описание события | Методы слушателя | Источник события |
|---|---|---|
| Событие PropertyChangeEvent. Обеспечивает работу механизма привязанных свойств JavaBeans | propertyChange (PropertyChangeEvent) | Практически все графические компоненты JavaBeans (в том числе все компоненты Swing) |
| Событие ChangeEvent. Запускается некоторыми компонентами и моделями для сообщения о своих изменениях | stateChanged(ChangeEvent) | Некоторые компоненты Swing. Многие модели используют это событие для связи с UI-представителями |
| Событие ActionEvent. Сообщает о действии над компонентом | actionPerformed(ActionEvent) | Компоненты Swing, у которых есть какое-то «главное» действие (например, у кнопки — нажатие) |

Описанные в таблице события PropertyChangeEvent и ChangeEvent в обычных программах почти не используются, однако именно с их помощью происходит взаимодействие между компонентами, их UI-представителями и моделями. Событие PropertyChangeEvent — это основополагающее событие архитектуры JavaBeans, оно позволяет следить за тем, как и какие свойства меняются в компоненте (так называемые *привязанные* свойства). Мы уже упоминали в главе 1, что все свойства компонентов Swing являются привязанными. Это не только позволяет использовать их в визуальных средствах, но и дает возможность моделям и UI-представителям эффективно взаимодействовать друг с другом. Более того, иногда и в стандартных программах только такие события позволяют отследить изменение некоторых свойств компонентов, когда отдельные события не предусмотрены. Событие ChangeEvent — это более простое событие, которое также дает знать об изменениях состояния компонента или модели. Довольно часто модели запускают это событие, сообщая об изменениях в данных.

В таблицу наиболее часто используемых событий также попало событие ActionEvent. Оно на самом деле встречается практически в каждом приложении, в основном благодаря тому, что почти в каждом приложении есть кнопки и/или меню. Впрочем, это событие возникает не только при щелчке на кнопке, оно часто используется и тогда, когда нужно сообщить о каком-то важном действии (выбор элемента в раскрывающемся списке, конец ввода в текстовом поле, срабатывание таймера и т. п.)

Содержимое таблиц 2.1 и 2.2 не нужно запоминать, но эта информация может пригодиться, если вам понадобится обработать событие, которое прежде обрабатывать не приходилось. Тогда вы сможете заглянуть в таблицы и посмотреть, какой слушатель требуется для обработки этого события.

## *Техника написания слушателей*

Использование слушателей для обработки событий очень удобно не только потому, что позволяет разделять места возникновения событий (пользовательский интерфейс) и места их обработки. Слушатели также позволяют обрабатывать события так, как вам нужно, подстраивать схему обработки событий под свою программу, а не проектировать программу с тем расчетом, что в ней придется обрабатывать события. Вы не думаете о том, что придется делать при обработке событий, а просто пишете код, и в нужный момент используете один из доступных вариантов обработки события. В этом разделе мы рассмотрим наиболее популярные варианты обработки событий с помощью слушателей.

## Адаптеры

Если вы посмотрите на интерфейсы некоторых слушателей, то обнаружите в них не один, а несколько методов. В некоторых слушателях методов довольно много (например, в слушателе оконных событий WindowListener). Это вполне логично — каждый метод отражает свое небольшое изменение в компоненте. Создание для каждого простого события отдельного слушателя привело бы к появлению невообразимого количества интерфейсов с расплывчатым предназначением, а обработка сложного события в одном методе неудобна — придется предварительно выяснять, что же именно произошло.

Однако оказывается, что использование интерфейсов слушателей с несколькими методами тоже не совсем удобно. Чаще всего при обработке события какого-либо типа нас интересует нечто конкретное (например, щелчок мыши или закрытие окна), а не все возможные варианты происходящего события. Тем не менее, при реализации интерфейса Java обязывает определить все методы этого интерфейса, даже те, которые нас вообще не интересуют. Это не только неудобно, но и вносит некоторую путаницу в код, в котором появляются «пустые», ничего не выполняющие методы.

Чтобы избежать этих неудобств, в дополнении к слушателям библиотека предоставляет специальные классы, называемые *адаптерами* (adapters). Адаптер — это просто класс, который реализует интерфейс определенного слушателя, а значит, все его методы. Однако методы слушателя адаптер оставляет пустыми, без всякой реализации, что позволяет нам вместо реализации интерфейса слушателя наследовать адаптер от класса адаптера. При наследовании в дочерний класс переходят все методы базового класса, и остается переопределить только те из них, которые нас интересуют. Попробуем использовать адаптеры в следующем примере.

```
// Adapters.java
// Использование адаптеров вместо интерфейсов
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class Adapters extends JFrame {
    public Adapters() {
        super("Adapters");
        // при закрытии окна - выход
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        // регистрируем слушателя
        addMouseListener(new MouseL());
        // выводим окно на экран
        setSize(200, 200);
        setVisible(true);
    }
    // наследуем от адаптера
    class MouseL extends MouseAdapter {
        // следим за щелчками мыши в окне
        @Override
        public void mouseClicked(MouseEvent e) {
```

```
            System.out.println(e);
        }
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(
         new Runnable() {
          public void run() { new Adapters(); } });
    }
}
```

В примере создается небольшое окно, к которому добавляется слушатель событий от мыши MouseListener. Однако реализовывать интерфейс слушателя мы не стали, а создали новый внутренний класс MouseL, унаследованный от класса адаптера MouseAdapter. Вместо того чтобы определять все методы слушателя (а их ни много, ни мало — целых пять, можете убедиться в этом, заглянув в табл. 2.1), мы переопределили только один. Здесь нас интересовало только, когда пользователь щелкает кнопками мыши в окне, так что нам пригодился метод mouseClicked(). Остальные четыре метода остались в стороне, как будто их и не было.

Для всех низкоуровневых событий из пакета java.awt.event, слушатели которых состоят более чем из одного метода, имеются адаптеры. Чаще всего они и используются при обработке событий, и только в тех редких случаях, когда программу интересуют все события определенного рода, задействуются интерфейсы. Узнать название класса адаптера очень просто: если слушатель называется XXXListener, то имя адаптера выглядит как XXXAdapter[4]. Но вот для высокоуровневых событий компонентов Swing имеются только слушатели, адаптеров для них вы не найдете. Видимо, разработчики решили, что если хоть какое-то событие от компонента обрабатывается, то информация о нем нужна полная. Довольно странное решение.

Единственный недостаток адаптеров — меньший контроль со стороны компилятора и отсюда большая вероятность возникновения ошибки при работе с ними. Если методы интерфейса должны быть определены (иначе программа не будет компилироваться), то при наследовании от класса адаптера никаких ограничений нет, например, в только что рассмотренном примере в классе MouseL можно было определить следующий метод:

```
MouseClicked(MouseEvent e)
```

Этот метод очень похож на правильный метод слушателя, предназначенный для получения сообщений о щелчках на кнопках мыши, но с его помощью вы не обработаете никакие события. Можно щелкать мышью что есть силы, искать ошибку в тысяче мест, а она оказывается скрытой в названии метода — первая буква оказалась прописной, а надо было писать строчную. До появления в Java аннотаций это могло быть реальной проблемой, однако аннотация @Override прекрасно справляется с задачей, не давая коду компилироваться, если метод не переопределен из родительского класса. Всегда помечайте методы адаптеров этой аннотацией, и вы будете спокойны за результат.

## Каждому событию — по слушателю

Хорошо известно, что программа читается гораздо чаще, чем пишется. Процесс поддержки программ вообще занимает львиную долю времени разработки, поэтому удобочитаемость кода очень важна. А одним из важнейших условий удобочитае-

[4] Это похоже на схему именования, но на самом деле адаптеры не являются частью спецификации JavaBeans, это просто удобный способ сократить объем ненужного кода. У слушателя с несколькими методами вполне может не быть адаптера.

мости кода является четкое разделение его на объекты, каждый из которых занимается своим делом и не затрагивает другие. Однако когда дело касается создания пользовательских интерфейсов, соблюсти это правило не так-то просто. Программе приходится работать по событиям пользовательского интерфейса, учитывать особенности этих событий, деловая логика программы смешивается с графическим интерфейсом, что приводит к совершенно неудобоваримому и плохо поддерживаемому коду.

Система событий Swing позволяет полностью избежать этих неприятностей. Каждому событию сопоставляется свой слушатель, который может быть описан весьма далеко от места возникновения события. Каждый слушатель представляет собой класс — обычный, внутренний или анонимный, в котором сосредотачивается обработка события. При таком подходе программа становится простой и понятной: с одной стороны имеются классы и объекты, представляющие деловую логику программы, с другой — объекты и классы, реализующие ее пользовательский интерфейс. Отдельно от тех и других находятся объекты, обрабатывающие события, именно они увязывают происходящее в пользовательском интерфейсе с изменениями в деловой логике, и именно такой подход к обработке событий является оптимальным и наиболее естественным: обрабатывая событие, создавайте класс, и в этом классе объединяйте деловую логику и графический интерфейс. Следуя этой элементарной технике, можно четко структурировать код и обеспечить его высокое качество, не прилагая никаких сверхусилий.

## Внутренние классы

Чаще всего обработка событий происходит во внутренних классах, относящихся к классу, в котором создается основная часть пользовательского интерфейса. Причину этого понять несложно: когда происходит событие, нас интересует не только сам факт появления этого события, но и то, в каком состоянии находятся компоненты пользовательского интерфейса (каков текст в текстовых полях, сколько выбрано элементов в списке и т. п.). Конечно, более заманчиво выглядела бы обработка событий в отдельных классах (так было бы еще проще определять, где и какие события обрабатывает программа, а также обновлять эти классы и наследовать от них). Но в таком случае возникла бы сильная связь между этими классами и классами пользовательского интерфейса, а это никому не нужно, потому что требует дополнительной работы и вносит путаницу. В предыдущих примерах мы уже применяли внутренние классы для обработки событий, давайте сделаем это еще раз.

```
// InnerClassEvents.java
// Внутренние классы для обработки событий
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class InnerClassEvents extends JFrame {
    private JTextField text;
    private JButton button;
    public InnerClassEvents() {
     super("InnerClassEvents");
     // при закрытии окна - выход
     setDefaultCloseOperation(EXIT_ON_CLOSE);
```

```
        // последовательное расположение
        setLayout(new FlowLayout());
        // добавим текстовое поле
        add(text = new JTextField(10));
        // и кнопку
        add(button = new JButton("Нажмите"));
        // будем следить за нажатиями кнопки
        button.addActionListener(new ButtonL());
        // выводим окно на экран
        pack();
        setVisible(true);
    }
    // класс - слушатель нажатия на кнопку
    class ButtonL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            System.out.println(text.getText());
        }
    }
    public static void main(String[] args) {
           SwingUtilities.invokeLater(
            new Runnable() {
             public void run() { new InnerClassEvents(); } });
    }
}
```

В примере показана классическая ситуация: имеется текстовое поле и кнопка — мы добавили их в наше окно, предварительно установив для него последовательное расположение компонентов (подробнее про расположение компонентов будет рассказано в главе 7, а про окна в главе 6). Пользователь что-то вводит в поле и щелкает на кнопке, а программа должна обработать введенные им данные. Использование внутреннего класса для обработки события щелчка на кнопке (слушателя ActionListener) дает нам возможность без помех получить доступ к текстовому полю text и содержащийся в нем текст. Используй мы отдельный класс, нам пришлось бы каким-то образом заполучить ссылку на объект нашего окна, более того, в классе окна InnerClassEvents нам пришлось бы либо объявить текстовое поле открытым для доступа (public), либо добавить новый метод, возвращающий текст, набранный в текстовом поле.

Таким образом, внутренние классы — это практически оптимальный механизм обработки событий, позволяющий одновременно отделить место обработки события от места его возникновения и иметь полный доступ к элементам пользовательского интерфейса. (Появление в Java версии 1.1 внутренних классов во многом было связано с необходимостью иметь механизм, упрощающий обработку событий JavaBeans.) От внутренних классов можно наследовать почти так же, как и от обычных, так что при создании новой версии своего приложения не нужно залезать в уже работающий и отлаженный код, а достаточно просто унаследовать от внутреннего класса и немного подправить обработку какого-либо события. В любом случае при обработке события прежде всего следует рассмотреть возможность использования отдельного внутреннего класса.

Конечно, в больших приложениях может произойти не совсем приятная ситуация накопления огромного количества внутренних классов внутри одного из больших элементов системы, например главного окна приложения. Но, во-первых, использование современных средств разработки (IDE) позволяет легко перемещаться по структуре класса, которая будет четко выражена и отделена от логики, ну а во-вторых, при разработке большого приложения нужно сделать усилие и разделить пользовательский интерфейс на логические модули, которые не обязательно соединять в одном классе посредством внутренних классов. Все это позволит оставить приложение элегантным и хорошо читаемым, несмотря на свои размеры.

## Быстро и грязно

Заголовок этого раздела может показаться странным, но техника, с которой мы сейчас познакомимся, никаких других ассоциаций и не вызывает: код программы превращается в пеструю смесь, словно по нему прошелся кто-то в гигантских сапогах, оставляя повсюду расплывчатые грязные следы. Внутренние классы можно определять внутри класса, просто вкладывая один класс в другой, но существует и другой способ создавать классы. Он самый быстрый и самый неопрятный, а создаваемые классы называются *анонимными* (anonymous classes). В том месте программы, где вам понадобится какой-то класс, вы не создаете его в отдельном файле с отдельным именем, а пишете начинку этого класса (методы и т. п.) прямо на месте. В результате скорость написания программы немного возрастает, хотя страдает ее читабельность (впрочем, как мы увидим, с этим можно бороться). Никто не запрещает использовать анонимные классы и для создания слушателей событий. Рассмотрим пример:

```
// AnonymousClassEvents.java
// Анонимные классы для обработки событий
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class AnonymousClassEvents extends JFrame {
    public AnonymousClassEvents() {
        super("AnonymousClassEvents");
        // анонимный класс присоединяется прямо на месте
        // выход из приложения при закрытии окна
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        // добавим кнопку
        JButton button = new JButton("Нажмите меня");
        getContentPane().add(button);
        // слушатель создается в методе
        button.addActionListener(getButtonL());
        // выводим окно на экран
```

```
        pack();
        setVisible(true);
    }
    // этот метод создает слушателя для кнопки
    public ActionListener getButtonL() {
        return new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("ActionListener");
            }
        };
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(
         new Runnable() {
            public void run() { new AnonymousClassEvents(); } });
    }
}
```

В этом очень простом примере создается окно, в которое помещается кнопка. Для обработки закрытия окна мы создаем собственного слушателя оконных событий WindowEvent, и делаем это с помощью анонимного класса, который наследует от класса WindowAdapter и при закрытии окна (методом windowClosing()) завершает работу приложения. Все происходит прямо на месте: и регистрация слушателя, и создание слушателя, и его описание. Пожалуй, быстрее обработать событие невозможно. Однако видно, что получавшийся код весьма запутан и плохо управляем: нет никакой возможности получить ссылку на объект-слушатель, нельзя унаследовать от него, анонимный класс не может получить доступ к переменным внешней области видимости, которые не были объявлены неизменными (final). Есть более удобный способ работы с анонимными слушателями — их можно создавать в специальных методах. В нашем примере — это метод getButtonL(), возвращающий слушателя нажатий кнопки (который просто выводит сообщение о нажатии в стандартный поток вывода). Он предоставляет чуть больше возможностей и удобства: класс находится в отдельном методе, метод легко найти, его можно переопределить.

Однако анонимные классы в любом случае ведут к запутыванию кода, создаются ли они прямо на месте или с помощью специальных методов. Их можно использовать, если вы хотите быстро набросать простую программу и забыть о ней. Но в больших и сложных программах, склонных расти и совершенствоваться и которые надо будет поддерживать, анонимные классы могут превратить вашу жизнь в ад. От «раскиданных» по разным местам фрагментов классов, что-то зачем-то делающих, чтение и отладка кода проще не станут. В таких случаях лучше предпочесть «настоящие» именованные внутренние классы или использовать специальные приемы, повышающие гибкость программы. Некоторые наиболее популярные из них мы сейчас рассмотрим.

## Диспетчеризация

Вне всяких сомнений, техника снабжения каждого события собственным слушателем, располагающимся в отдельном классе, является самой распространенной и по праву: она действительно разделяет места возникновения и обработки события и по-

зволяет создавать кристально чистый код. Но справедливости ради стоит отметить, что эта техника не единственная, и не всем она по душе (хотя она идеально вписывается в парадигму объектно-ориентированного программирования). Есть и другой способ обработки событий, в котором используется противоположная идея: обработка событий происходит в одном классе (или в нескольких, но не в таком умопомрачительном количестве, как в предыдущих вариантах). Техника эта называется *диспетчеризацией* (dispatching), или *перенаправлением* (forwarding), и довольно часто используется в визуальных средствах разработки интерфейса.

Суть этой техники такова: вы обрабатываете схожие события в одном слушателе, не плодя море классов, сортируете происходящие события по источникам (местам, где события происходят) и вызываете для каждого события обрабатывающий его метод (то есть слушатель не обрабатывает событие, а выступает диспетчером, отправляющим его в нужное место, отсюда и название техники). Чтобы все стало окончательно ясно, рассмотрим небольшой пример.

```
// ForwardingEvents.java
// Техника диспетчеризации событий
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ForwardingEvents extends JFrame {
 public ForwardingEvents() {
     super("ForwardingEvents");
     // при закрытии окна - выход
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     // последовательное расположение
     getContentPane().setLayout(new FlowLayout());
     // добавим пару кнопок
     button1 = new JButton("OK");
     button2 = new JButton("Отмена");
     getContentPane().add(button1);
     getContentPane().add(button2);
     // будем следить за нажатиями кнопок
     Forwarder forwarder = new Forwarder();
     button1.addActionListener(forwarder);
     button2.addActionListener(forwarder);
     // выводим окно на экран
     pack();
     setVisible(true);
 }
 JButton button1, button2;
 // класс - слушатель нажатия на кнопку
 class Forwarder implements ActionListener {
     public void actionPerformed(ActionEvent e) {
```

```
            // рассылаем события по методам
            if ( e.getSource() == button1 ) onOK(e);
            if ( e.getSource() == button2 ) onCancel(e);
        }
    }
    // обработка события от кнопки "OK"
    public void onOK(ActionEvent e) {
        System.out.println("onOK()");
    }
    // обработка события от кнопки "Отмена"
    public void onCancel(ActionEvent e) {
        System.out.println("onCancel()");
    }

    public static void main(String[] args) {
           SwingUtilities.invokeLater(
            new Runnable() {
             public void run() { new ForwardingEvents(); } });
    }
}
```

В данном примере создается окно, в которое помещено две кнопки. К каждой кнопке присоединен слушатель событий Forwarder, следящий за нажатиями кнопок, причем слушатель этот создается только один раз (что без сомнений позволяет экономить память). В самом слушателе проделывается немудреная работа: при возникновении события от кнопки выясняется, в какой именно кнопке произошло это событие, после чего вызывается метод с соответствующим названием. Слушатель Forwarder можно расширить, чтобы он поддерживал гораздо большее число кнопок, и при этом не придется создавать новые классы — достаточно будет лишь определить новые методы. Если в дальнейшем понадобится модифицировать работу приложения, это будет несложно: надо унаследовать новый класс от класса окна и переопределить интересующие нас методы, например onOK().

Диспетчеризация имеет свои преимущества: код получается более компактным и в некотором смысле более привычным для тех программистов, что перешли на Java с других языков программирования, где обработка событий осуществляется именно в методах, а не в отдельных классах. Именно такая иллюзия и создается в результате использования такой техники: мы видим несколько методов, вызываемых при возникновении событий, и реализуем обработку этих событий, переопределяя методы. Однако есть здесь и свои ограничения: то, как события рассылаются по методам, целиком зависит от классов слушателей, подобных Forwarder, и если событие от какого-то компонента не обрабатывается этим классом, вам остается лишь развести руками и писать слушатель самому. Если компонентов в интерфейсе достаточно много, и для каждого из них создается свой метод, обрабатывающий некоторое событие, получится гигантский класс, «битком набитый» методами, а работать с такими классами всегда неудобно; более того, появление таких классов свидетельствует о нарушении основополагающего правила объектно-ориентированного программирования: каждый класс решает собственную небольшую задачу.

Если вы используете какое-либо визуальное средство создания интерфейса, и в нем для обработки событий требуется диспетчеризация, прекрасно. Задействуйте методы, которые это средство генерирует, и в них обрабатывайте события, по крайней мере, до тех пор, пока код сохраняется более или менее чистым и управляемым. Но если вы пишете код для обработки событий самостоятельно, создавайте для слушателей событий отдельные классы. Это прекрасно структурирует код и избавляет вас от дополнительной скучной работы (написания диспетчера, хранения бесполезных ссылок и создания множества новых методов).

## Проблема висячих ссылок

Обработка событий в Swing реализована на очень высоком уровне, в чем вы уже не раз могли убедиться: события обрабатываются просто и так, как вам необходимо. Однако в системе обработки событий есть свойство, которое иногда необходимо учитывать, чтобы не привести программу к катастрофе.

Рассмотрим ситуацию, в которой программа создает компоненты и добавляет их в интерфейс динамически, прямо во время работы, заранее не зная, сколько их будет. Хорошим примером является любое средство визуальной разработки пользовательского интерфейса: во время создания интерфейса в контейнер добавляется и удаляется множество компонентов, и к каждому из них присоединяются некоторые слушатели (чаще всего это слушатели привязанных свойств компонентов). Пользователь такого средства может экспериментировать с интерфейсом, создавать все новые и новые обычные и диалоговые окна, добавлять в них новые компоненты. К каждому из таких компонентов визуальное средство добавляет слушателя событий (или даже нескольких слушателей). Затем пользователь может удалить эти компоненты, закрыть текущий контейнер и снова открыть его — все это приведет к удалению компонентов, но ссылки на слушатели в них *останутся*, потому что визуальное средство не вызвало метод removeXXXListener(), отсоединяющий ранее присоединенных слушателей. Такие ссылки и называются *висячими*.

Теперь надо вспомнить, как в Java работает сборщик мусора. Хорошо известно, что созданные объекты в Java не нужно явно удалять: об этом заботится сборщик мусора. Он работает в фоновом режиме параллельно с программой, периодически включается и производит удаление объектов, на которые не осталось явных ссылок. Здесь нас и поджидает сюрприз — все те графические компоненты, которые визуальное средство удалило из контейнера, не удаляются сборщиком мусора, потому что в них еще имеются явные ссылки на слушателей событий, ранее присоединенных визуальным средством. И чем больше будет работать программа, чем интенсивнее пользователь будет создавать интерфейсы, тем меньше останется памяти, и в конце концов все может завершиться аварийным завершением программы с потерей несохраненных данных.

Описанная ситуация возникает не так уж и редко, потому что Swing прекрасно подходит для программ с динамическим пользовательским интерфейсом: нет ничего проще добавления компонентов в контейнер и присоединения к ним слушателей прямо «на ходу». Поэтому, если вы пишете программу с заранее неизвестным количеством компонентов, не забывайте отсоединять от удаляемых компонентов ранее присоединенных слушателей методом removeXXXListener(), особенно если эти слушатели используются многократно и привязаны к каким-либо еще компонентам системы, которые еще активны и не стали мусором. В обычных же программах, вроде тех, что мы рассматривали в этой главе, нет необходимости строго следить за слушателями — количество компонентов ограничено, и все они остаются на месте на время работы программы.

Кстати, висячие ссылки — это проблема не системы обработки событий Swing, а побочный эффект использования шаблона проектирования «наблюдатель». Везде, где он

применяется, в том или ином виде возникают висячие ссылки (субъект и наблюдатель чаще всего имеют разное время жизни), и с ними приходится бороться. Тем не менее, дефект это не такой уж серьезный, и достоинства наблюдателя перевешивают его недостатки.

## *Создание собственных событий*

Компоненты библиотеки Swing настолько разнообразны и функциональны, что их возможностей с лихвой хватает для нужд большей части приложений. Поэтому львиную долю времени нам приходится обрабатывать события уже имеющихся компонентов, и мы только что подробно рассмотрели, как это можно делать.

Впрочем, как бы ни были хороши стандартные компоненты библиотеки, рано или поздно возникают ситуации, когда нужные нам возможности они обеспечить не могут. В таком случае придется создать собственный компонент, унаследовав его от какого-либо компонента библиотеки или написав «с нуля» (то есть унаследовав от базового компонента JComponent библиотеки Swing или, если вы хотите создать компонент «с чистого листа», от базового класса Component библиотеки AWT). У вашего нового компонента, если он выполняет не самые простые функции, наверняка будут какие-то события, и программистам-клиентам компонента (если компонент окажется удачным, возможности для его многократного использования обязательно найдутся) необходимо предоставить способ обработки этих событий. Для компонента, соответствующего архитектуре JavaBeans, это означает наличие интерфейса слушателя, класса события и пары методов для присоединения и удаления слушателей. Чуть раньше мы кратко обсудили систему именования событий JavaBeans и правила, которым подчиняются слушатели и классы событий. Давайте попробуем создать свой компонент и новый тип события. Следование архитектуре JavaBeans к тому же позволит использовать новый компонент и его события в визуальных средствах разработки.

В качестве примера рассмотрим простую кнопку: небольшой прямоугольник с рамкой, при нажатии которого происходит некоторое событие. Написание собственно компонента (его процедуры прорисовки) выльется всего в несколько простых строк кода, гораздо интереснее процесс создания события (нажатия кнопки) для этого компонента.

Прежде всего необходимо создать класс события. Как вы помните из описания схемы событий JavaBeans, этот класс должен быть унаследован от класса java.util.EventObject и иметь название вида XXXEvent, где XXX — название события. Вот что получается для нашей кнопки:

```
// com/porty/swing/event/ButtonPressEvent.java
// Событие (нажатие) для кнопки
package com.porty.swing.event;

import java.util.EventObject;

public class ButtonPressEvent extends EventObject {
    // Конструктор. Требует задать источник события
    public ButtonPressEvent(Object source) {
        super(source);
    }
}
```

В нашем событии не будет храниться никакой дополнительной информации, так что класс события чрезвычайно прост. Заметьте, что конструктор класса требует указать источник события; как правило, это компонент, в котором событие произошло. Источник события нужно задавать для любого события, унаследованного от класса EventObject, а получить его позволяет метод getSource() того же базового класса. Таким образом, при обработке любого события JavaBeans вы можете быть уверены в том, что источник этого события всегда известен. И, наконец, обратите внимание на то, что класс нашего нового события разместился в пакете com.porty.swing.event, и к нему проще организовать доступ. При создании других событий вам вовсе не обязательно делать их классы такими же простыми: вы можете добавлять в них подходящие поля и методы, которые сделают работу с событием более комфортной.

Далее нам нужно описать интерфейс слушателя нашего события. Данный интерфейс будут реализовывать программисты-клиенты компонента, заинтересованные в нажатиях кнопки. Интерфейс слушателя, следующего стандарту JavaBeans, должен быть унаследован от интерфейса java.util.EventListener. В последнем нет ни одного метода, он служит «отличительным знаком», показывая, что наш интерфейс описывает слушателя событий. Итак:

```
// com/porty/swing/event/ButtonPressListener.java
// Интерфейс слушателя события нажатия кнопки
package com.porty.swing.event;

import java.util.EventListener;

public interface ButtonPressListener
            extends EventListener {
    // данный метод будет вызываться при нажатии кнопки
    void buttonPressed(ButtonPressEvent e);
}
```

В интерфейсе слушателя мы определили всего один метод buttonPressed(), который и будет вызываться при нажатии нашей кнопки. В качестве параметра этому методу передается объект события ButtonPressEvent, так что заинтересованный в нажатии кнопки программист, реализовавший интерфейс слушателя, будет знать подробности о событии. Интерфейс слушателя, так же как и класс события, разместился в пакете com.porty.swing.event.

Теперь нам остается включить поддержку события в класс самого компонента. Для этого в нем нужно определить пару методов для присоединения и отсоединения слушателей ButtonPressListener, эти методы должны следовать схеме именования событий JavaBeans. В нашем случае методы будут именоваться addButtonPressListener() и removeButtonPressListener(). Слушатели, которых программисты регистрируют в данных методах, будут оповещаться о нажатии кнопки. Существует два основных способа регистрации слушателей в компоненте и оповещения их о происходящих событиях.

- Регистрация *единичного* (unicast) слушателя. В классе компонента определяется единственная ссылка на слушателя события, так что узнавать о событии может только один слушатель одновременно. При присоединении нового слушателя старый слушатель, если он был, перестает получать оповещения о событиях, так как ссылка на него теряется.
- Регистрация *произвольного* (multicast) количества слушателей. В компоненте заводится список, в котором и хранятся все присоединяемые к компоненту слушатели.

При возникновении события все находящиеся в списке слушатели получают о нем полную информацию.

В подавляющем большинстве ситуаций используется гораздо более гибкий и удобный второй способ с произвольным количеством слушателей, практически все события стандартных компонентов Swing (это же относится и к компонентам AWT), низкоуровневые и высокоуровневые, поддерживают произвольное количество слушателей. Для нас не составит особого труда реализовать список слушателей в нашем компоненте, так что мы тоже выбираем второй способ. Все готово для того, чтобы приступить к созданию самого компонента:

```
// com/porty/swing/SimpleButton.java
// Пример компонента со своим собственным событием
package com.porty.swing;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.porty.swing.event.*;

public class SimpleButton extends JComponent {
 // список слушателей
 private ArrayList<ButtonPressListener>
             listenerList = new ArrayList<ButtonPressListener>();
 // один объект-событие на все случаи жизни
 private ButtonPressEvent event =
            new ButtonPressEvent(this);

    // конструктор - присоединяет к кнопке слушателя
 // событий от мыши
 public SimpleButton() {
    addMouseListener(new PressL());
    // зададим размеры компонента
    setPreferredSize(new Dimension(100, 100));
 }
 // присоединяет слушателя нажатия кнопки
 public void addButtonPressListener(
            ButtonPressListener l) {
    listenerList.add(l);
 }
 // отсоединяет слушателя нажатия кнопки
 public void removeButtonPressListener(
            ButtonPressListener l) {
```

```
        listenerList.remove(l);
    }
    // прорисовывает кнопку
    public void paintComponent(Graphics g) {
        // зальем зеленым цветом
        g.setColor(Color.green);
        g.fillRect(0, 0, getWidth(), getHeight());
        // рамка
        g.setColor(Color.black);
        g.draw3DRect(0, 0, getWidth(), getHeight(), true);
    }
    // оповещает слушателей о событии
    protected void fireButtonPressed() {
           for (ButtonPressListener l:
                   listenerList) {
               l.buttonPressed(event);
           }
       }
    // внутренний класс, следит за нажатиями мыши
    class PressL extends MouseAdapter {
        // нажатие мыши в области кнопки
        public void mousePressed(MouseEvent e) {
            // оповестим слушателей
            fireButtonPressed();
        }
    }
}
```

Мы создаем очень простой компонент SimpleButton: это прямоугольник с рамкой, который тем не менее обладает собственным событием ButtonPressEvent. Компонент унаследован нами от базового класса JComponent библиотеки Swing, так что все действия, связанные с его прорисовкой, мы поместили в метод paintComponent() (подробнее о базовом компоненте Swing и реализованной в нем системе прорисовки мы узнаем в главе 3). В конструкторе происходит настройка основных механизмов компонента: мы присоединяем к нему слушателя событий от мыши, реализованного во внутреннем классе PressL, а также задаем размер нашего компонента методом setPreferredSize() (по умолчанию размеры компонента считаются нулевыми). В слушателе PressL мы будем отслеживать нажатия кнопок мыши, для этого нам понадобится метод mousePressed(). Как только пользователь щелкнет кнопкой мыши в области, принадлежащей нашему компоненту, будет вызван метод fireButtonPressed(), обязанностью которого является оповещение слушателей о событии. Кстати, название вида fireXXX() (или fireXXXEvent()) является неофициальным стандартом для методов, «запускающих» высокоуровневые события, хотя такие методы и не описаны в спецификации JavaBeans. Мы еще не раз встретим методы с такими названиями при работе с событиями различных компонентов Swing и их моделями.

Слушатели ButtonPressListener будут храниться в списке ArrayList, адаптированном только под хранение объектов ButtonPressListener. Благодаря мощи стандартного

списка ArrayList методы для присоединения и отсоединения слушателей совсем просты: им нужно лишь использовать соответствующие возможности списка. Также просто и оповещение слушателей о событиях: для каждого элемента списка мы вызываем метод buttonPressed(), передавая ему в качестве параметра объект-событие. Объект-событие у нас один на компонент, и именно он передается всем слушателям: на самом деле, зачем создавать для каждого слушателя новое событие, если в нем не хранится ничего, кроме ссылки на источник события, то есть на сам компонент. Мы не включили в компонент поддержку многозадачности: если регистрировать слушателей будут несколько потоков одновременно, у нашего компонента могут возникнуть проблемы (для эффективной работы список ArrayList рассчитан на работу только с одним потоком в каждый момент времени). Но исправить это легко: просто объявите методы для присоединения и отсоединения слушателей как synchronized. Аналогичное действие проделайте и с методом fireButtonPressed() (он тоже работает со списком). Есть и еще один способ включить для компонента поддержку многозадачного окружения: с помощью класса java.util.Collections и статического метода synchronizedList(). Последний метод вернет вам версию списка ArrayList со встроенной поддержкой многозадачности. Впрочем, как мы узнаем из следующей главы, работать с компонентами Swing из нескольких потоков без специальных усилий нельзя, так что смысла в добавлении слушателей из нескольких потоков как правило никакого нет.

Ну а теперь давайте проверим работу нашего нового компонента и посмотрим, будет ли обрабатываться принадлежащее ему событие. Вот простой пример:

```
// SimpleButtonTest.java
// Обработка события нового компонента
import javax.swing.*;
import com.porty.swing.*;
import com.porty.swing.event.*;

import java.awt.*;

public class SimpleButtonTest extends JFrame {
    public SimpleButtonTest() {
        super("SimpleButtonTest");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        // создаем кнопку и присоединим слушателей
        SimpleButton button = new SimpleButton();
        // анонимный класс
        button.addButtonPressListener(
            new ButtonPressListener() {
                public void buttonPressed(ButtonPressEvent e) {
                    System.out.println("1!");
                }
            });
        // внутренний класс
        button.addButtonPressListener(new ButtonL());
```

```
        // добавим кнопку в окно
        JPanel contents = new JPanel();
        contents.add(button);
        setContentPane(contents);
        // выведем окно на экран
        setSize(400, 300);
        setVisible(true);
    }
    class ButtonL implements ButtonPressListener {
        public void buttonPressed(ButtonPressEvent e) {
            System.out.println("2!");
        }
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(
         new Runnable() {
          public void run() { new SimpleButtonTest(); } });
    }
}
```

В примере мы создаем небольшое окно, в панели содержимого которого разместится наш новый компонент, простая «кнопка» SimpleButton (в качестве панели содержимого используется отдельная панель JPanel). К нашей кнопке мы присоединяем два слушателя ButtonPressListener: один слушатель описан прямо на месте, в виде анонимного класса, а второй реализован во внутреннем классе. После добавления кнопки в окно последнее выводится на экран. Запустив программу с примером, вы увидите (посмотрев на консольный вывод), как слушатели оповещаются о событии.

Подобная цепочка действий повторяется для любого нового события JavaBeans: вы описываете класс события и интерфейс его слушателя, следуя хорошо известным правилам, и добавляете в компонент пару методов для регистрации слушателей и их отсоединения. Для хранения слушателей используется подходящий список, все хранящиеся в нем слушатели оповещаются о возникновении события.

Обратите также внимание, что хранение слушателей в простых коллекциях данных, таких как список ArrayList, накладывает на слушателя определенные ограничения. В самом деле, слушатель вызывается в момент перечисления элементов списка, и удалить себя или другого слушателя в этот момент не сможет, так как список на это не рассчитан. Модификацию списка слушателей лучше проводить не из кода слушателя.

## Список EventListenerList

В пакете javax.swing.event, который предназначен для хранения событий и слушателей компонентов библиотеки Swing, есть интересный инструмент — специализированный список EventListenerList. Этот список позволяет хранить в одном месте произвольное количество слушателей любого типа, лишь бы они были унаследованы от «отличительного знака» слушателей — интерфейса EventListener. Для того чтобы можно было легко различать слушателей разных типов, при добавлении в список каждому слушателю требуется сопоставить его «тип», который представлен объектом Class. Таким образом

список EventListenerList состоит из пар вида «объект Class – слушатель EventListener». Для добавления в список новой такой пары служит метод add(). Если бы мы в только что разобранном нами примере использовали для хранения слушателей вместо ArrayList список EventListenerList, то для присоединения слушателей нам понадобилось бы написать следующий код:

```
listenerList.add(ButtonPressListener.class, l);
```

Получить слушателей определенного типа позволяет метод getListeners(), которому необходимо передать объект Class, определяющий тип слушателей. Данный метод возвращает массив слушателей нужного нам типа. Для получения слушателей ButtonPressListener пригодился бы такой код:

```
EventListener[] listeners =
    listenerList.getListeners(ButtonPressListener.class);
```

С полученным массивом слушателей легко работать: для каждого элемента массива (все элементы массива имеют базовый тип EventListener) вы можете смело провести преобразование к указанному вами в методе getListeners() типу слушателя (у нас это ButtonPressListener) и вызвать определенные в интерфейсе слушателя методы, которые и сообщат слушателям о том, что в компоненте произошли некоторые события. Помимо метода getListeners() можно также использовать метод getListenersList(), возвращающий все содержащиеся в списке пары Class – EventListener в виде массива. С массивом, полученным из метода getListenersList(), работать сложнее: приходится самостоятельно проверять, принадлежит ли слушатель нужному нам типу и учитывать тот факт, что каждый слушатель хранится вместе с объектом Class. Метод getListeners() без сомнения проще и удобнее.

Практически все компоненты библиотеки Swing, обладающие собственными событиями, и стандартные модели с поддержкой слушателей используют для хранения слушателей список EventListenerList. Если в вашем компоненте или модели имеется событие только одного типа (и соответственно, слушатель одного типа), проще задействовать обычный список, вроде ArrayList. Но если событий и слушателей несколько, преимущества EventListenerList выходят на первый план: достаточно единственного экземпляра такого списка для хранения всех регистрируемых в вашем компоненте или модели слушателей; не нужно понапрасну расходовать память на несколько списков и выполнять излишнюю работу. Требуется лишь правильно указывать, слушателя какого типа вы собираетесь добавить в список, удалить из него (для этого предназначен метод remove()) или получить.

## *События от мыши и волшебный метод contains()*

Хотя мы только начали знакомиться с событиями, сразу же стоит упомянуть одну особенность работы с событиями от мыши. Дело в том, что они весьма зависимы от области на экране, которую занимает интересующий вас компонент: только в том случае, когда курсор мыши находится в области компонента, мы можем сказать, что происходят перемещения курсора, щелчки или перетаскивания. Заведует же определением местонахождения той или иной точки экрана в области компонента метод contains(), определенный в базовом классе всех компонентов Java Component.

По умолчанию нам, как правило, ничего не приходится делать – область, которую выделяет компоненту контейнер (панель, окно или апплет) и является той зоной, в которой происходят все события от мыши. Это понятное и прозрачное поведение. Но бывает, что нам нужно модифицировать это поведение, особенно если компонент не прямоугольной формы или имеет какие-то «окна», сквозь которые события от мыши проникают к компонентам, лежащим ниже в контейнере. Ситуация в Swing не такая уж

редкая, и мы увидим примеры в главе 4. В качестве быстрого и понятного примера можно посмотреть на такую кнопку:

```
// ContainsTest.java
// Изменение поведения мыши и метод contains()
import javax.swing.*;
import java.awt.*;


public class ContainsTest extends JFrame {
    public ContainsTest() {
     super("ContainsTest");
     // при закрытии окна - выход
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     // добавим кнопку и переопределим метод
     JButton button = new JButton("Невидима") {
            @Override
            public boolean contains(int x, int y) {
                // не содержим ни одной точки
                return false;
            }
        };
     // настроим панель содержимого и выведем окно на экран
     setLayout(new FlowLayout());
     add(button);
     setSize(300, 200);
     setVisible(true);
 }
 public static void main(String[] args) {
        SwingUtilities.invokeLater(
         new Runnable() {
          public void run() { new ContainsTest(); } });
 }
}
```

В этом чрезвычайно простом примере мы создаем маленькое окно и размещаем в нем кнопку. Кнопка ничем не отличается от своих собратьев за исключением метода contains() — этот метод для переданных ему координат точки должен сообщить, принадлежит ему точка или нет. Мы говорим, что нам чужда любая точка из нашей области, и запустив пример, вы увидите, что происходит. Кнопка не нажимается мышью, хотя она доступна и включена, но свободно нажимается с клавиатуры.

Из мелочей следует помнить, что необходимо переопределять именно метод, которому передаются два целых числа, координаты компонента X и Y, а не второй вариант метода с объектом Point, он системой не вызывается. Ну и координаты исчисляются в системе самого компонента — (0,0) обозначает начало компонента,

и максимум не будет превышать длину и ширину компонента, так работает система рассылки событий.

## *Резюме*

Итак, события могут возникать в любом компоненте Swing; для того чтобы узнать о них, надо создать соответствующего слушателя событий и зарегистрировать его в компоненте. В большинстве случаев обработка событий не заходит дальше этой простой цепочки действий, и в этой главе мы рассмотрели все ее детали, так что создание и присоединение слушателей не является для нас проблемой. Однако при работе с системой событий возникают и более сложные ситуации, как правило, это связано с появлением при обработке событий различных потоков выполнения. Более сложную часть обработки событий мы рассмотрим в следующей главе.

# Глава 3. За кулисами системы обработки событий

В главе 2 мы рассмотрели основной механизм обработки событий — слушателей. Присоединяя их к компоненту, мы можем узнавать обо всех действиях пользователей, а простые примеры показали прекрасную работу слушателей. Однако иногда возникают более сложные задачи, требующие более изощренного контроля над происходящими событиями, например, если вам понадобится отфильтровать некоторые события или перехватить их раньше, чем они дойдут до слушателей. В этой главе мы познакомимся с «начинкой» системы обработки событий Swing и посмотрим, как она функционирует. Разумеется, эту главу не имеет смысла читать, не ознакомившись с ее основами (см. главу 2).

Программы, написанные на Java, выполняются виртуальной машиной Java (Java Virtual Machine, JVM), которая обеспечивает независимость от конкретной платформы. Виртуальная машина использует ресурсы операционной системы для решения своих задач, это утверждение верно и для событий пользовательского интерфейса. Низкоуровневые события (движения мыши, нажатия клавиш, закрытие окон) возникают именно в операционной системе, виртуальная машина перехватывает их и перенаправляет компонентам пользовательского интерфейса Java-приложения. Посмотрим теперь, как низкоуровневые события попадают к компонентам Java (к которым относятся и компоненты Swing).

## Поток EventDispatchThread и очередь событий EventQueue

Вспомним, как запускается Java-программа: вызывается статический метод main(). При этом виртуальная машина создает новый поток выполнения (который так и называется «main») и передает ему управление. Самое интересное происходит, когда в методе main() создается объект, каким-либо образом связанный с графической системой Java, то есть унаследованный от базового класса java.awt.Component. При этом происходит целая череда событий.

1. Инициализируется динамическая библиотека Java, связывающая графическую подсистему Java с операционной системой, когда в этом возникает необходимость, как правило, с первыми же операциями внутри библиотеки AWT.

2. Создаются необходимые помощники компонентов AWT, отвечающие за их связь с операционной системой.

3. Компоненты выводятся на экран, и операционная система начинает посылать им события (своим «родным» тяжеловесным компонентам). Эти события преобразуются в объекты Java, которые помещаются в очередь событий EventQueue.

4. Очередь событий проверяет наличие потока рассылки событий, и, так как приложение только начинает работу, впервые запускает поток выполнения EventDispatchThread, который связывается с очередью событий EventQueue и начинает рассылать события, хранящиеся в этой очереди событий, по соответствующим компонентам AWT.

Если с первыми двумя пунктами все более или менее понятно (если вы помните, мы обсуждали помощников компонентов AWT в главе 1), то последние два пункта стоит рассмотреть поподробнее. Итак, что же такое очередь событий? Это экземпляр класса EventQueue из пакета java.awt, и на самом деле класс этот очень прост. Он представляет собой контейнер, работающий по принципу FIFO (First-In, First-Out — первый пришел, первый вышел), то есть по принципу очереди (отсюда и название). В этой очереди графическая система Java хранит все приходящие в программу низкоуровневые события. Происходит это следующим образом: пользователь совершает действие, операционная система сообщает об этом, виртуальная машина Java создает соответствующий событию объект (например, объект класса MouseEvent, если пользователь щелкнул мышью) и добавляет его в конец очереди EventQueue. Класс EventQueue — это *одиночка* (singleton)[1], сколько бы обычных окон, диалоговых окон или компонентов приложение не создавало, очередь событий всегда присутствует в единственном экземпляре. Единственное, что можно сделать, — это заменить существующую очередь событий своей собственной. В том, что очередь событий всегда хранится только в одном экземпляре и реализована в виде очереди, имеется немалый смысл. В очередь событий может поступать громадное количество разнообразных событий, и все из них в той или иной степени изменяют состояние компонента. Если бы события, пришедшие позже, обрабатывались перед событиями, пришедшими раньше, компонент мог бы прийти в противоречивое состояние. Использование дисциплины FIFO позволяет этого избежать. То же самое относится и к наличию только одной очереди: будь их несколько, неизвестно, по каким правилам следовало бы распределять события по очередям и выбирать их оттуда.

Далее вступает в действие специальный поток выполнения EventDispatchThread. Он циклически «подкачивает» (pump) события из очереди событий EventQueue (это делает метод pumpEvents()), и, если события в очереди имеются, он извлекает их и рассылает по соответствующим компонентам AWT (это выполняет метод dispatchEvent() класса EventQueue). Так как поток EventDispatchThread один, очередь событий одна, и события хранятся в порядке их поступления, обеспечивается последовательная обработка событий в соответствии с очередностью их поступления, то есть события обрабатываются одно за другим.

Роль потока EventDispatchThread в любой графической программе на Java трудно переоценить, это настоящий «властелин» этих программ. Именно в этом потоке происходит вся работа, и он никогда не останавливается. Мы отмечали, что графическая программа в промежутке между событиями простаивает, это так лишь с нашей стороны, с другой стороны всегда работает поток EventDispatchThread, вызывающий наш код только при наступлении событий (в графической программе наш код весьма пассивен). Стоит ему остановиться, как программа тут же прекратит свою работу. В этом предложении ответ на часто возникающий вопрос, почему программа, использующая графический интерфейс, не заканчивается после выполнения метода main(), как это делают обычные (консольные) программы. Вот маленький пример:

```
public class ExitTest {
  public static void main(String[] args) {
    new javax.swing.JFrame().pack();
```

[1] Одиночкой класс EventQueue является с точки зрения библиотеки AWT, которая одновременно задействует только один его экземпляр. Однако вы можете создать сколь угодно много объектов этого класса, и заменить используемый AWT объект EventQueue своим собственным (это может быть и объект специальным образом унаследованного класса с новой функциональностью). Тем не менее, *используется* только один объект.

```
    }
}
```

Если вы запустите эту программу, то увидите, что она и не подумает закончить работу после того, как создаст окно JFrame, установит для него оптимальный размер методом pack(), и метод main() завершится. В спецификации виртуальной машины Java сказано, что программа завершает свою работу, когда в ней не останется работающих потоков выполнения (не являющихся демонами). В графической программе такой поток выполнения остается — это как раз поток распределения событий EventDispatchThread. Прямого доступа к нему нет, так что выход из программы приходится осуществлять «грубой силой»: вызывая метод System.exit()[2].

Более того, поток EventDispatchThread рассылает не только события, возникающие в результате действий пользователя, в его обязанности входит рассылка событий даже такого экзотического типа, как PaintEvent и InvocationEvent. Вы не сможете обработать в своей программе события этого типа, они используются внутри графической подсистемы: событие PaintEvent означает необходимость перерисовать компонент, целиком или частично, а событие InvocationEvent позволяет вмешаться в «плавное течение» потока EventDispatchThread и произвести некоторые дополнительные действия, ненадолго приостановив его.

Чуть позже мы вернемся к роли потока выполнения EventDispatchThread в графических программах, а сейчас давайте все-таки посмотрим, как события, происходящие в результате действий пользователя, добираются до компонентов Swing. Надо сказать, что низкоуровневые события, возникающие в операционной системе, рассылаются только тяжеловесным компонентам AWT, потому что легковесных компонентов (к которым относятся и компоненты Swing) операционная система не видит. «Прекрасно, — говорите вы, саркастически улыбаясь, — мы договорились до того, что легковесные компоненты Swing не получают уведомления о событиях. Что же, слушатели работают по мановению волшебной палочки?» На самом деле легковесные компоненты узнают о событиях, только не напрямую, а через один скрытый механизм. Сейчас мы с ним познакомимся.

## Доставка событий методам processXXXEvent()

Итак, когда поток выполнения EventDispatchThread обнаруживает, что в очереди событий появилось новое событие, он извлекает его и проверяет тип этого события, чтобы определить, что делать с ним дальше. Если извлеченное из очереди событие относится к графическому компоненту (это может быть событие, относящееся к прорисовке, или более интересное нам событие, возникшее в результате действия пользователя), то поток EventDispatchThread вызывает метод dispatchEvent() очереди событий EventQueue. Этот метод не делает ничего экстраординарного, а просто узнает, к какому компоненту относится событие (то есть определяет источник события — об этом сообщает операционная система) и вызывает метод dispatchEvent() этого компонента.

Все компоненты AWT наследуют метод dispatchEvent() от своего базового класса java.awt.Component, но ни один из них не в состоянии вмешаться в процесс первоначальной обработки событий, потому что метод этот является неизменным (final) и переопределить его невозможно. Сделано это неспроста. Именно в методе dispatchEvent() происходит самая важная внутренняя работа по распределению событий, в том числе координация действий с помощниками компонентов, преобразование событий PaintEvent в вызо-

[2] Правда в последних версиях JDK есть и другой способ закончить графическую программу: необходимо удалить из системы все окна их методом dispose(). Если активных окон не останется, поток рассылки завершит свою работу.

вы методов paint() и update()[3], а также целая череда действий, вмешиваться в которые нет смысла, потому что они относятся к внутренней реализации библиотеки и постоянно меняются без всякого предупреждения. Для нас гораздо важнее то, что, в конце концов, метод dispatchEvent() передает событие (если это было событие в ответ на действие пользователя) методу processEvent().

Метод processEvent() — это следующий этап на пути следования событий к слушателям. Ничего особенного в нем не происходит, его основная функция — передать событие согласно его типу одному из методов processXXXEvent(), где «XXX» — название события, например Focus или Key. Именно методы processXXXEvent() распределяют приходящие события по слушателям. Например, метод processKeyEvent() определяет, что именно за событие от клавиатуры пришло, и вызывает соответствующий метод слушателя KeyListener (если конечно, такие слушатели были зарегистрированы в компоненте). После этого мы наконец-то получаем уведомление о событии в свою программу (посредством слушателя) и выполняем необходимые действия. Так заканчивается путь события, начинавшийся в очереди событий. В целом все выглядит так, как показано на рис. 3.1.

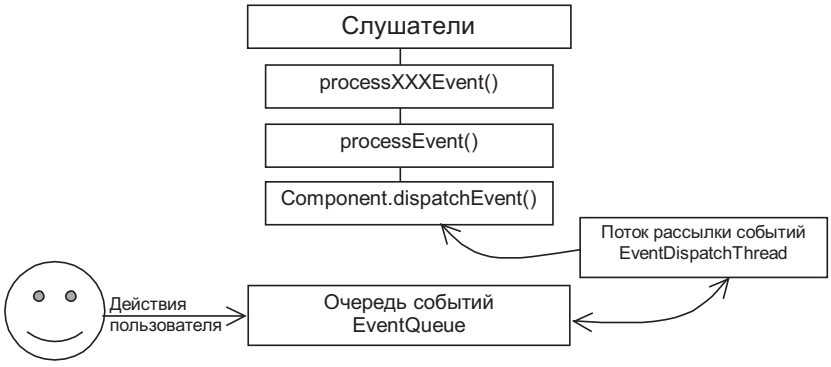



**Рис. 3.1.** Путь события к слушателям

Вся описанная цепочка действий относится к тяжеловесным компонентам AWT, которые «видны» операционной системе и которым она посылает низкоуровневые события. Как же легковесные компоненты получают уведомления о событиях? Оказывается, что происходит это с помощью тяжеловесного контейнера (обычно окна или апплета), в котором расположены легковесные компоненты. Он вмешивается

[3] Благодаря непрерывной работе потока EventDispatchThread и событиям PaintEvent графический интерфейс программы способен вовремя перерисовываться, если какая-либо часть его была закрыта другим окном, а затем снова представлена глазам пользователя. Операционная система сообщает Java-приложению о необходимости перерисовки части окна, виртуальная машина создает соответствующее этому сообщению объект-событие PaintEvent, а поток рассылки событий мгновенно, сразу после обнаружения в очереди событий EventQueue нового события PaintEvent, передает это событие в принадлежащий ему компонент (как правило, перерисовка выполняется для главного окна приложения), где данное событие преобразуется в вызов метода paint(). Именно в методе paint() производится прорисовка любого компонента. Программисту-клиенту не нужно знать всех подробностей движения события PaintEvent: довольно знания того, что код, прорисовывающий компонент, необходимо поместить в метод paint().

в цепочку обработки событий еще на уровне метода dispatchEvent()[4] и, прежде чем продолжить обработку события обычным образом, «ворошит» все содержащиеся в нем легковесные компоненты, проверяя, не принадлежит ли пришедшее событие им. Если это так, то обработка события тяжеловесным контейнером прекращается, а событие передается в метод dispatchEvent() соответствующего легковесного компонента. Эти действия встроены в базовый класс контейнеров java.awt.Container, и любой унаследованный от него контейнер может быть уверен, что содержащиеся в нем легковесные компоненты получат необходимую поддержку. Для программистов-клиентов Swing все эти хитрости остаются незаметными, и цепочка обработки событий кажется одинаковой как для тяжеловесного, так и для легковесного компонента.

Мы упомянули здесь методы processXXXEvent() не для «красного словца», они могут быть полезными и в обычных программах. Чаще всего эти методы используют там, где необходимо выполнить некоторое действие перед поступлением события к слушателю или вообще запретить распространение события. Для этого необходимо переопределить метод processXXXEvent(), соответствующий нужному нам событию. Рассмотрим простой пример, в котором мы попытаемся вмешаться в цепочку обработки событий от мыши:

```
// PreProcessMouse.java
// Перехват событий от мыши до их поступления к слушателям
import javax.swing.*;
import java.awt.event.*;

public class PreProcessMouse extends JFrame {
    PreProcessMouse() {
        super("PreProcessMouse");
        // при закрытии окна - выход
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        // добавим слушателя событий от мыши
        addMouseListener(new MouseL());
        // выводим окно на экран
        setSize(200, 200);
        setVisible(true);
    }
    // перехват событий от мыши
    public void processMouseEvent(MouseEvent e) {
        if ( e.getClickCount() == 1 ) {
            // один щелчок не пропускаем к слушателям
            return;
        }
```

[4] Несмотря на то, что метод dispatchEvent() является неизменным (final), вмешаться в его работу контейнер все-таки может, потому что метод dispatchEvent() просто вызывает метод dispatchEventImpl(), в котором и выполняется вся работа. Метод dispatchEventImpl() обладает видимостью в пределах пакета, так что все классы из пакета java.awt могут привнести в него что-то новое.

```
        // иначе вызываем метод базового класса
        else super.processMouseEvent(e);
    }
    // в этом слушателе будем следить за щелчками мыши
    class MouseL extends MouseAdapter {
        @Override
        public void mouseClicked(MouseEvent e) {
            System.out.println(
                "ClickCount: " + e.getClickCount());
        }
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(
         new Runnable() {
          public void run() { new PreProcessMouse(); } });
    }
}
```

В примере мы создаем небольшое окно JFrame, при его закрытии приложение будет заканчивать свою работу. К этому окну (которое является обыкновенным компонентом, унаследованным от базового класса Component), мы присоединяем слушателя событий от мыши MouseL, который унаследован от адаптера MouseAdapter и следит за щелчками мыши в окне (о них сообщается в метод слушателя mouseClicked()). Однако слушатель этот — не единственное заинтересованное в щелчках мыши «лицо»: мы переопределили метод processMouseEvent() нашего окна JFrame. Как нам теперь известно, именно в этом методе происходит рассылка событий от мыши слушателям, в нем мы также отслеживаем щелчки мыши еще до поступления события к слушателям. Посмотрите, что происходит: в методе processMouseEvent() мы выясняем, сколько раз пользователь щелкнул мышью (не важно, правая эта кнопка или левая), и если количество щелчков равно единице, заканчиваем работу метода, что равносильно игнорированию всех присоединенных слушателей (рассылка событий происходит в методе базового класса, и если мы его не вызываем, то слушатели ничего не узнают). В противном случае, если количество щелчков больше единицы, мы вызываем метод базового класса, в обязанности которого входит рассылка события слушателям.

Запустив программу с примером, вы (глядя на консольный вывод) сможете увидеть, какие щелчки мыши добираются до слушателя событий MouseL, а какие щелчки, несмотря на то, что вы их старательно производите в окне, так и не появляются в этом слушателе. В методы processXXXEvent() попадают всевозможные события одного типа, для мыши это щелчки, перетаскивание, нажатия, вход в область компонента и выход из нее, и многое другое. Определяя, какие из этих событий вам нужны, вы можете встроить в свои графические интерфейсы поддержку самой тщательной фильтрации событий и тонко настроить их поведение. Слушатели находятся на самом верху пирамиды рассылки событий и не знают, что событие до них может не добраться: они просто терпеливо ожидают, когда оно произойдет. Вы, со своей стороны, всегда можете опуститься ниже по цепочке обработки событий, и чем ниже вы будете опускаться, тем более тонкие возможности будут оказываться в ваших руках.

Тем не менее, как бы заманчиво не выглядела перспектива фильтрации событий до их поступления к слушателям, возможности этого подхода весьма ограничены. Вспомните еще раз, как происходит рассылка событий: все основные действия происходят

в недоступных для нас механизмах класса Component и Container, и в итоге событие попадает в наши руки уже после того, как система определила, какому компоненту оно принадлежит. То есть фактически фильтровать события мы можем только для того компонента, которому они принадлежат, а смысла в этом немного: чаще всего фильтрация событий имеет смысл для контейнеров, содержащих другие компоненты. Фильтруя или особым образом обрабатывая события для контейнера, к примеру, отключая все щелчки правой кнопки мыши, мы, как правило, хотим, чтобы подобная фильтрация работала и для всех содержащихся в контейнере компонентов. Но реализовать такое поведение, переопределяя методы processXXXEvent(), нельзя: события обрабатываются для каждого компонента отдельно. Создатели Swing учли ситуацию и добавили в контейнеры высшего уровня особый компонент, *прозрачную панель*; она позволяет фильтровать и предварительно обрабатывать события сразу для всего пользовательского интерфейса программы. Прозрачную панель мы рассмотрим в главе 6, там будет упомянут и похожий по смыслу компонент JXLayer.

Напоследок стоит упомянуть о том, что события от клавиатуры (KeyEvent) заслуживают особого внимания. По многим причинам они не вписываются в общие схемы: среди основных причин — необходимость поддержки механизмов передачи фокуса ввода и клавиатурных сокращений. Фокус ввода определяет, какие компоненты получают события от клавиатуры первыми, а также то, в какой последовательности происходит обработка этих событий. На самом деле, операционная система не видит легковесных компонентов, так что с ее точки зрения все события от клавиатуры происходят только в контейнере высшего уровне, в котором однако может находиться множество легковесных компонентов. Система передачи фокуса ввода должна четко определять, какой компонент получит событие от клавиатуры. Эта система встроена в работу метода dispatchEvent() базового класса Component и благодаря этому всегда имеет шанс перехватить любое необходимое для передачи фокуса ввода событие от клавиатуры. Клавиатурные сокращения — один из основополагающих инструментов Swing, поддерживаются благодаря особой реализации метода processKeyEvent() базового класса библиотеки JComponent. Поэтому переопределение метода processKeyEvent() для специальной обработки или фильтрации событий от клавиатуры нужно производить с осторожностью и всегда вызывать метод базового класса super.processKeyEvent(), а также осознавать, что некоторые нажатия клавиш до этого метода могут так и не добраться. Мы будем подробно обсуждать систему передачи фокуса ввода и клавиатурные сокращения в главе 5, и там же узнаем, какие дополнительные способы специальной обработки событий от клавиатуры у нас есть.

## Маскирование и поглощение событий

При окончательной «шлифовке» системы событий ее создатели заметили, что методы компонентов processEvent() и processXXXEvent(), служащие для рассылки событий, могут становится «узким местом» графических программ: в них вполне способна «затеряться» некоторая часть процессорного времени. Исследование показывает, что в компоненты приходят уведомления абсолютно обо всех происходящих с ними низкоуровневых событиях, будь то события от мыши, клавиатуры, изменения в размерах контейнера или что-либо еще, причем приходят такие уведомления всегда, независимо от того, заинтересован компонент в этих событиях или нет, и в легковесные, и в тяжеловесные компоненты. В итоге получалось, что и для сложных компонентов, таких как таблицы, по-настоящему нуждающихся в информации о большей части событий, и для простых вспомогательных компонентов, подобных панелям, которые по большому счету нуждаются только в своевременной перерисовке, система присылала полную информацию обо всех происходящих событиях. И это не приводило к блестящей производительности.

Решением стало *маскирование событий* (event masking). С каждым компонентом графической системы, способным получать уведомления о событиях, теперь ассоциирована специальная маска в виде длинного (long) целого, которая и определяет, какие события компонент получает. По умолчанию отключено получение всех событий, кроме уведомлений о необходимости перерисовки. Но, как только вы (или другой компонент) присоединяете к компоненту слушателя события определенного типа, например, слушателя событий от мыши MouseListener, компонент тут же включает в свою маску признак обработки данного события (от мыши), так что система обработки событий начинает присылать ему все связанные с этой маской события. Принцип прост: нет слушателей — нет и событий, как только слушатели появляются — автоматически появляются события.

Маскирование скорее относится к внутренним механизмам системы обработки событий, и чаще всего вам достаточно того, что при присоединении слушателей маска компонента автоматически обновляется. Однако вы можете и вручную управлять маскированием, быстро включая (или отключая) доставку интересующих вас событий. Это менее гибкий способ управления событиями, чем, к примеру, методы processXXXEvent(), но иногда он позволяет добиться желаемого гораздо быстрее. Одним из самых распространенных вариантов «ручного» маскирования можно назвать как раз переопределение методов processXXXEvent(): если вы не присоедините к компоненту слушателей, а попытаетесь сразу же получить информацию о событиях из методов processXXXEvent(), у вас ничего не выйдет. Вспомните, по умолчанию события в маску не включаются. Это распространенная ошибка, а решение ее просто: надо просто включить в маску нужное вам событие, после чего метод processXXXEvent() заработает «как по маслу».

Добавить событие в маску компонента или удалить его оттуда позволяет пара методов enableEvents() и disableEvents(). Правда, вызвать их напрямую не так то просто: методы эти описаны как защищенные (protected), то есть они доступны только подклассам компонента. В качестве параметров им требуется передать набор объединенных логическим «ИЛИ» констант из класса AWTEvent, которые и определят, какие низкоуровневые события вы включаете или отключаете. Рассмотрим небольшой пример:

```
// MaskingEvents.java
// Маскирование событий
import java.awt.*;
import javax.swing.*;


public class MaskingEvents extends JFrame {
 public MaskingEvents() {
     super("MaskingEvents");
     // при закрытии окна - выход
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     // отключим события от окна
     disableEvents(AWTEvent.WINDOW_EVENT_MASK);
     // добавим особую кнопку
     JPanel contents = new JPanel();
     contents.add(new CustomButton("Привет!"));
     setContentPane(contents);
     // выведем окно на экран
     setSize(400, 300);
     setVisible(true);
```

```
    }
    // особая кнопка
    class CustomButton extends JButton {
        public CustomButton(String label) {
            super(label);
            // отключаем события с клавиатуры и от мыши
            disableEvents(AWTEvent.KEY_EVENT_MASK);
            disableEvents(AWTEvent.MOUSE_EVENT_MASK);
        }
    }
    public static void main(String[] args) {
            SwingUtilities.invokeLater(
             new Runnable() {
              public void run() { new MaskingEvents(); } });
    }
}
```

Здесь мы создаем небольшое окно с рамкой JFrame и сразу же указываем ему, что при закрытии окна нужно будет завершить работу приложения. В нашем примере все меняется: мы пытаемся властно приказать окну (методом disableEvents(), он нам доступен, потому что класс в примере унаследован от окна JFrame, а оно, в свою очередь, от базового класса Component) исключить из маски события от окна. Однако после запуска примера вы убедитесь, что окно не обратило внимания на нашу маску и закрывается так же, как и обычное окно. В первом издании книги пример работал, и окно не закрывалось, однако это было сочтено нарушением безопасности пользователей, особенно при работе из апплетов, и маскирование оконных событий теперь отключено. Это первая особенность системы маскирования событий.

Пример также демонстрирует, как управлять маскированием событий от клавиатуры; мы отключаем их для особой кнопки, унаследованной от обычной кнопки JButton библиотеки Swing. Запустив программу с примером, вы увидите, что созданную кнопку легко нажать мышью, но невозможно активизировать с клавиатуры (с помощью клавиши Enter или пробела в зависимости от используемых внешнего вида и поведения). Все работает как обычно, слушатели и UI-представители готовы к обработке событий, но они просто не доходят до них: маскирование выполняется на самых нижних уровнях системы обработки событий, и если событие не включено в маску компонента, обработать его оказывается невозможно.

Однако тут же маскирование демонстрирует еще одну особенность, постойте-ка, что мы только что сказали? «Кнопку легко нажать мышью» - но глянув на код примера, совершенно очевидно, что мы отключили события от мыши. Это вторая особенность маскирования событий — событие маскируется только в том случае, если оно не включено в маску событий *и* если для событий данного типа нет слушателей. Однако наша кнопка самаприсоединила слушателя, потому что желает знать, когда на нее нажмут, и придуманная нами маска снова оказывается не у дел.

## ВНИМАНИЕ

> Маскирование событий не так уж и очевидно, как кажется, — оно не работает, если событие ожидает хотя бы один слушатель, неважно, кем и когда он добавлен, и не работает для оконных событий.

Маскирование редко требуется в обычных приложениях, однако его можно применить в специальных целях, таких как автоматизированное тестирование интерфейса или проверка работоспособности приложения в условиях отказа одного или нескольких видов событий. Впрочем, стоит помнить, что, так же как и в случае с методами processXXXEvent(), маскирование событий работает только для одного компонента, что может быть неудобно. В таких случаях нам снова может пригодиться прозрачная панель или инструмент JXLayer, которые мы будем обсуждать в главе 6.

При обработке низкоуровневых событий вы при необходимости можете *поглощать* (consume) их. Это означает буквально следующее: вы при обработке события в слушателе или методе processXXXEvent() приходите к выводу, что обработку данного события можно завершить прямо сейчас, то есть передавать его дальше для какой-либо дополнительной обработки больше не следует. Осуществить это и позволяет механизм поглощения событий. Пример привести нетрудно: в слушателе событий от клавиатуры, присоединенном к текстовому полю, появляется событие нажатия некоторой клавиши. Ваша программа особым образом обрабатывает его, но не хочет, чтобы этот символ был передан текстовому полю обычным порядком, что приведет к появлению символа на экране. В своем слушателе вы поглощаете событие, и оно не попадает к текстовому полю.

Поглощение события выполняет метод consume(), определенный в базовом классе всех низкоуровневых событий AWTEvent. Поглощение работает только для низкоуровневых событий (имеются в виду события от клавиатуры и мыши), да и то со многими оговорками: после вызова метода consume() событие не доберется до самого компонента, если он каким-то образом переопределил методы processXXXEvent(), и операционная система не сможет обработать его обычным образом (если этот компонент тяжеловесен и обрабатывает события). Однако до всех зарегистрированных в компоненте слушателей событие все равно доберется, и будут ли они обрабатывать уже «поглощенное» событие или нет (было ли событие поглощено, позволяет узнать метод isConsumed()), зависит только от их «доброй воли». Таким образом, для библиотеки Swing (компоненты которой легковесны и не обладают помощниками) поглощение не слишком полезно, если они следят за событиями с помощью слушателей. Небольшой пример проиллюстрирует механизм поглощения:

```
// ConsumingEvents.java
// Поглощение событий
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ConsumingEvents extends JFrame {
 public ConsumingEvents() {
     super("ConsumingEvents");
     // при закрытии окна - выход
     setDefaultCloseOperation(EXIT_ON_CLOSE);
        // слушатель, поглощающий печатание символов
        KeyListener kl = new KeyAdapter() {
            @Override
            public void keyTyped(KeyEvent e) {
                e.consume();
            }
```

```
        };
        // добавляем текстовые поля
        setLayout(new FlowLayout());
        JTextField swingField = new JTextField(10);
        swingField.addKeyListener(kl);
        add(swingField);
        TextField awtField = new TextField(10);
        add(awtField);
        awtField.addKeyListener(kl);
        // кнопка
        JButton button = new JButton("Жмите!");
        add(button);
        // слушатель пытается поглотить события от мыши
        button.addMouseListener(new MouseAdapter() {
            @Override
            public void mousePressed(MouseEvent e) {
                e.consume();
            }
    });
        // выводим окно на экран
    setSize(300, 200);
    setVisible(true);
 }
 public static void main(String[] args) {
        SwingUtilities.invokeLater(
         new Runnable() {
          public void run() { new ConsumingEvents(); } });
 }
}
```

Мы создаем небольшое окно, в котором у нас будет два текстовых поля — по одному из библиотек Swing и AWT и кнопка. Для начала мы пробуем проверить, будут ли исправно поглощаться события нажатия клавиш на клавиатуре, если мы вызовем для них consume(). Запустив программу, вы увидите, что будут, а это значит текстовые компоненты Swing получают события от клавиатуры не от слушателей, а на более ранних этапах и поглощение работает. Так как поле AWT по сути является системным компонентом, оно также не получит поглощенные нами события. С кнопкой все сложнее — несмотря на наши попытки «съесть» событие, она все равно реагирует на мышь, а значит, получает события через слушатели, и мы не можем так просто это ей запретить.

Компоненты Swing являются легковесными, то есть полностью написанными на Java, и часто они обрабатывают события с помощью обычных слушателей, поэтому поглощенные события до них в основном все равно дойдут. Если вы хотите гарантированно перехватить некоторое событие и «не пустить» его к легковесному компоненту, используйте подходящий метод processXXXEvent(), возможности окон Swing, которые мы разберем в главе 6, или наследование от класса компонента. Поглощение вам в этом не поможет.

## Работа с очередью событий

Очередь EventQueue является настоящим ядром системы обработки событий: все события, так или иначе возникающие в вашей программе, проходят через нее, включая даже такие экзотические события, как уведомления о перерисовке PaintEvent. Тем примечательнее тот факт, что вы имеете доступ к очереди событий, используемой в вашей программе, и более того, можете унаследовать от класса EventQueue, переопределить некоторые его методы, и использовать в программе свой вариант очереди событий, получая над системой обработки событий практически неограниченную власть.

Получить используемую в данный момент очередь событий позволяет метод getSystemEventQueue() класса Toolkit, а получить объект Toolkit можно методом getToolkit(), который имеется в каждом унаследованном от класса Component компоненте (кстати, класс Toolkit — это *абстрактная фабрика*, используемая для создания основных частей AWT). Полученный экземпляр очереди событий позволяет проделать многое: вы сможете узнать, какие события находятся в данный момент в очереди событий, вытащить их оттуда, поместить в очередь новые события (которые могут быть созданы вами вручную, а всем компонентам будет «казаться», что события эти возникли в результате действий пользователя). Помещение в очередь сгенерированных вами событий — прекрасный способ автоматизированного тестирования вашего интерфейса или демонстрации некоторых его возможностей. Приложение при этом будет вести себя точно так же, как если бы с ним работал самый настоящий пользователь. Рассмотрим небольшой пример:

```
// UsingEventQueue.java
// Использование очереди событий
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class UsingEventQueue extends JFrame {
 public UsingEventQueue() {
     super("UsingEventQueue");
     // выход при закрытии окна
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     // кнопка и ее слушатель
     JButton button = new JButton("Генерировать событие");
     button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
             // генерируем событие закрытия окна
             getToolkit().getSystemEventQueue().postEvent(
                 new WindowEvent(UsingEventQueue.this,
                     WindowEvent.WINDOW_CLOSING));
         }
     });
     // добавим кнопку в панель содержимого
     setLayout(new FlowLayout());
```

```
        add(button);
        // выведем окно на экран
        setSize(400, 300);
        setVisible(true);
    }
    public static void main(String[] args) {
           SwingUtilities.invokeLater(
            new Runnable() {
             public void run() { new UsingEventQueue(); } });
    }
}
```

В примере мы создаем небольшое окно, при закрытии которого программа будет завершать свою работу. В панель содержимого окна (для нее мы устанавливаем последовательное расположение компонентов FlowLayout) добавляется кнопка JButton с присоединенным к ней слушателем. При нажатии кнопки мы создаем событие WindowEvent типа WINDOW_CLOSING, именно такое событие генерируется виртуальной машиной, когда пользователь пытается закрыть окно. Созданное событие (помимо типа ему нужно указать источник, то есть окно, которое закрывается пользователем) мы помещаем в очередь событий, используя для этого метод postEvent(). Запустив программу с примером, вы увидите, что при нажатии кнопки приложение заканчивает свою работу, точно так же, как если бы мы нажали кнопку закрытия окна. Система обработки событий «играет почестному» — те события, которые виртуальная машина генерирует в ответ на действия пользователя, вы можете с тем же успехом генерировать самостоятельно — результат будет точно таким же.

Заменить стандартную очередь событий собственным вариантом очереди позволяет метод push() класса EventQueue. Ему нужно передать экземпляр вашей очереди событий, унаследованной от класса EventQueue. Наибольший интерес при наследовании представляет метод dispatchEvent(), который, как мы знаем, вызывается при распределении событий всех типов. Переопределив данный метод, вы получите исчерпывающую информацию о том, как, когда и какие события распределяются в вашей программе. Это может быть весьма ценно при отладке и диагностике вашей программы или при реализации особого поведения ваших компонентов[5].

## Влияние на программы потока EventDispatchThread

Теперь, когда мы в подробностях обсудили детали системы обработки событий, настала пора вернуться к потоку распределения событий EventDispatchThread и той роли, которую он играет в любой графической программе. Мы уже знаем, что этот поток распределяет по компонентам события, находящиеся в очереди событий EventQueue, и запускается той самой очередью, когда в нее попадают первые события графической системы. Таким образом, если ваша программа использует графические компоненты, она автоматически оказывается в многозадачном окружении, и вам придется принимать во внимание вопросы совместного использования ресурсов и их синхронизации. «Почему же программа оказывается в многозадачном окружении?» — спросите вы.

[5] Есть чуть менее сложный способ узнать обо всех распределяемых в вашей программе низкоуровневых событиях — использовать особый слушатель AWTEventListener. Вы присоединяете его к системе обработки событий с помощью класса Toolkit, указывая при этом, о событиях каких типов вы хотите знать. После присоединения слушателя AWTEventListener события всех указанных вами типов будут перед распределением попадать этому слушателю.

«Ведь в ней остается только один поток, поток рассылки событий? С кем он может конфликтовать?». Во-первых, даже в простейших программах, вроде тех, что мы уже разбирали, кроме потока рассылки событий всегда есть еще один поток выполнения с названием main. В нем выполняется, как нетрудно догадаться, метод main(), и он вполне может конфликтовать с потоком рассылки событий. Ну а, во-вторых, важнейшим свойством любого пользовательского интерфейса является его *отзывчивость*. Если программа замирает хотя бы на несколько секунд, не показывая при этом признаков жизни, это приводит пользователя в самую настоящую ярость. Если не использовать для выполнения длинных сложных задач отдельные потоки (так чтобы не блокировать рассылку событий и работу программы), обеспечить отзывчивость интерфейса будет невозможно.

Прежде всего выясним, когда именно запускается поток рассылки событий. Если брать в расчет библиотеку AWT, основу Swing, то там считается, что поток рассылки событий запускается, как только какой-либо компонент переходит в *реализованное* (realized) состояние. Реализованное состояние компонента означает, что он становится видимым для операционной системы, начинает получать от нее сообщения о событиях и занимает некоторое место на экране или в памяти экрана (даже если он все еще невидим). При получении событий очередь событий запускает поток рассылки. Легковесные компоненты (к которым относятся и компоненты Swing), а их операционная система не видит, переходят в реализованное состояние одновременно с переходом в реализованное состояние тяжеловесных контейнеров (окон или апплетов), в которых они содержатся. В свою очередь, тяжеловесные контейнеры переходят в реализованное состояние после вызова одного из следующих методов:

- pack() — служит для придания окну оптимального размера;
- setVisible(true) или show() — выводит окно на экран.

Если вы вызвали один из этих методов, можете быть уверены в том, что поток рассылки событий уже начал свою работу, и вы находитесь в многозадачном окружении. До вызова одного из этих методов графические компоненты AWT являются просто объектами, и вы можете работать с ними из любого потока.

Вопросы синхронизации с потоком рассылки событий становятся особенно важны при работе с компонентами библиотеки Swing, потому что последние практически полностью лишены каких бы то ни было средств для работы в многозадачном окружении. Это сделано не случайно: при разработке библиотеки рассматривалось множество альтернатив, и было выяснено, что наделение компонентов механизмами поддержки многозадачного окружения приведет не только к их усложнению, но и к значительному падению скорости их работы. Поэтому с компонентами Swing работать имеет смысл *только* из потока рассылки событий, то есть только из слушателей или методов, служащих для обработки событий. Но тут возникает резонный вопрос: «А как же обеспечить отзывчивость интерфейса? Ведь длительные вычисления в потоке рассылки событий блокируют остальной интерфейс программы. И неужели придется забыть обо всех удобствах, которые дает параллельное выполнение нескольких потоков?».

На самом деле все не так плохо. Во-первых, у вас есть несколько методов, которые можно вызывать из любого потока выполнения, не опасаясь соревнования с потоком рассылки событий. Они обладают встроенной синхронизацией и всю работу с компонентами все равно вызовут из потока рассылки событий. Вот эти методы:

- repaint() — служит для перерисовки компонента;
- revalidate(), validate(), invalidate() — позволяют заново расположить компоненты в контейнере и удостовериться в правильности их размеров.

Таким образом, вы (в отдельном потоке выполнения) можете провести какие-либо сложные вычисления, изменить некоторые данные, а затем попросить компонент обновить себя, так чтобы он смог вывести на экран новую информацию. И это не приведет к конфликтам.

Ну и, наконец, есть самый гибкий способ изменить компонент из другого потока. Если ваша задача, выполняющаяся в отдельном потоке, так или иначе приводит к необходимости изменить что-либо в компоненте, и в этом вам не могут помочь безопасные вызовы, остается только одно. Во избежание неприятностей с потоком рассылки событий и неожиданного тупика все действия с компонентами *все равно* нужно выполнять из потока рассылки событий. И у вас есть возможность выполнить некоторое действие в потоке рассылки событий. Для этого предназначены методы invokeLater() и invokeAndWait() класса EventQueue. Данным методам нужно передать ссылку на интерфейс Runnable, метод run() этого интерфейса будет выполнен потоком рассылки событий, как только он доберется до него. (Переданная в метод invokeLater() или invokeAndWait() ссылка на интерфейс Runnable «оборачивается» в событие специального типа InvocationEvent, обработка которого сводится к вызову метода run().) В итоге вы действуете следующим образом: выполняете в отдельном потоке сложные долгие вычисления, получаете некоторые результаты, а действия, которые необходимо провести после этого с графическими компонентами, выполняете в потоке рассылки событий с помощью метода invokeLater() или invokeAndWait(). Рассмотрим небольшой пример:

```
// InvokeLater.java
// Метод invokeLater() и работа с потоком рассылки событий
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class InvokeLater extends JFrame {
 public InvokeLater() {
     super("InvokeLater");
     // при закрытии окна - выход
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     // добавим кнопку со слушателем
     button = new JButton("Выполнить сложную работу");
     button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
             // запустим отдельный поток
             new ComplexJobThread().start();
             button.setText("Подождите...");
         }
     });
     // настроим панель содержимого и выведем окно на экран
     setLayout(new FlowLayout());
     add(new JTextField(20));
     add(button);
     setSize(300, 200);
```

```
        setVisible(true);
    }
    private JButton button;
    // поток, выполняющий "сложную работу"
    class ComplexJobThread extends Thread {
        public void run() {
            try {
                // изобразим задержку
                sleep(3000);
                // работа закончена, нужно изменить интерфейс
                EventQueue.invokeLater(new Runnable() {
                    public void run() {
                        button.setText("Работа завершена");
                    }
                });
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
    public static void main(String[] args) {
            SwingUtilities.invokeLater(
             new Runnable() {
              public void run() { new InvokeLater(); } });
    }
}
```

Мы создаем небольшое окно, в панели содержимого которого размещается кнопка JButton и вспомогательное текстовое поле. При нажатии кнопки будет вызван слушатель ActionListener, и мы предполагаем, что работа, которая предстоит слушателю, довольно сложна и займет приличное время, поэтому выполнять ее нужно в отдельном потоке (на самом деле, выполнение долгих вычислений в слушателе, который был вызван потоком рассылки событий, приведет к блокированию остальных событий, ждущих в очереди рассылки тем же потоком, и в итоге интерфейс программы станет неотзывчивым). Отдельный поток, выполняющий долгие сложные вычисления, реализован во внутреннем классе ComplexJobThread, унаследованном от базового класса всех потоков Thread. Проблема состоит в том, что по окончании вычислений нам нужно сменить надпись на кнопке, а мы к этому моменту будем находиться в отдельном потоке. Менять надпись из потока, отличного от потока рассылки событий, не стоит: это может привести к неопределенности данных в компонентах и ошибкам, и даже к тупикам и взаимным блокировкам потоков (мы уже знаем, что компоненты Swing не обладают встроенной синхронизацией). В момент обращения другого потока к компоненту последний вполне может получить событие о щелчке мыши от потока рассылки событий и прийти в нестабильное состояние.

Здесь нам и пригодится статический метод invokeLater() класса EventQueue. Он позволяет выполнить некоторый фрагмент кода из потока рассылки событий. В качестве

параметра данному методу нужно передать ссылку на объект, который реализует интерфейс Runnable: метод run(), определенный в этом интерфейсе, и будет выполнен потоком рассылки событий. Так мы и поступаем в примере: код, меняющий надпись на кнопке, будет выполнен из потока рассылки событий, так что можно не опасаться возникновения конфликтов. Запустив программу с примером и нажав кнопку, вы увидите, что во время вычислений пользовательский интерфейс доступен (вы сможете набрать что-либо в текстовом поле или нажать кнопку еще раз). Сразу по завершении вычислений вы узнаете об этом по изменению надписи не кнопке, и никаких конфликтов при этом не возникнет.

Помимо метода invokeLater() в вашем распоряжении также имеется метод invokeAndWait(). Он аналогичным образом позволяет выполнить фрагмент кода из потока рассылки событий, но в отличие от invokeLater(), делает это *синхронно*: приостанавливая работу потока, из которого вы его вызвали, до тех пор, пока поток рассылки событий не выполнит ваш код. С другой стороны, invokeLater() работает *асинхронно*: он немедленно возвращает вам управление, так что вы можете продолжать работу, зная, что рано или поздно ваш фрагмент кода будет выполнен потоком рассылки событий. В большинстве ситуаций предпочтительнее метод invokeLater(), а метод invokeAndWait() стоит использовать только там, где немедленное выполнение потоком рассылки фрагмента вашего кода обязательно для дальнейших действий. Работая с методом invokeAndWait(), следует быть внимательнее: поток, из которого вы его вызвали, будет ожидать, когда поток рассылки событий выполнит переданные ему фрагмент кода, а в этом фрагменте кода могут быть обращения к ресурсам, принадлежащим первому потоку, тому самому, что находится в ожидании. В итоге возникнет *взаимная блокировка*, и программа окажется в *тупике*. Метод invokeLater() позволяет всего этого избежать.

Практически все компоненты Swing не обладают встроенной синхронизацией, но благодаря описанным двум методам класса EventQueue вы всегда сможете выполнить некоторые действия с компонентами из потока рассылки событий. Правда, есть несколько исключений, к примеру, текстовые компоненты Swing, такие как многострочные поля JTextArea или редакторы JEditorPane, позволяют изменять свой текст из другого потока (и это очень удобно, особенно при загрузке больших текстов). Таких исключений немного, и если компонент может работать в многозадачном окружении, вы увидите упоминание об этом в его интерактивной документации и в этой книге.

Принципы работы потока рассылки событий и рассмотренный только что пример плавно подводят нас к «золотому правилу» Swing: *пишите слушатели короткими и быстрыми*. Здесь все просто: рассылка событий и вызов соответствующих слушателей, в которых они обрабатываются, происходят из одного потока выполнения, потока рассылки событий EventDispatchThread. Как только какой-либо слушатель начинает производить долгие вычисления, все остальные события «застревают» в очереди событий, и интерфейс программы перестает отзываться на любые действия пользователя, а работать с такой неповоротливой программой — удовольствие ниже среднего.

## СОВЕТ

Если в слушателе или методе обработки событий выполняются достаточно длинные вычисления, выносите их в отдельный поток. Манипулировать графическими компонентами из другого потока выполнения вы всегда сможете с помощью методов класса EventQueue или класса SwingUtilities из пакета javax.swing. Последний позволяет избежать импорта ненужных пакетов AWT и именно его рекомендуется применять в Swing .

Следуя этому нехитрому совету, с помощью Swing вы всегда будете писать эффектные и скоростные пользовательские интерфейсы. Зачастую незнание этого правила приводит к написанию чудовищно неповоротливых программ, а вину за это их создатели сваливают на Swing и Java, которые, тем не менее, позволяют разрабатывать интерфейсы, работающие не менее быстро, чем интерфейсы приложений «родной» операционной системы, что неоднократно доказано на практике даже очень большими приложениями.

## Отзывчивость программы и SwingWorker

Мы уже поняли, что выполнение долгих вычислений и логики в отдельном потоке позволяет обеспечить быстрый отклик компонентов Swing и всего интерфейса к большому удовольствию его пользователей, а проблему конфликтов позволяет решить исполнение задач в потоке рассылки событий с помощью класса EventQueue или SwingUtilites. Однако это все потребовало от нас изучить изнанку системы событий Swing и напрямую манипулировать потоками.

В библиотеке Swing имеется класс, способный облегчить нашу участь, и скрыть некоторые детали процесса за своими понятными методами. Он называется SwingWorker, и задача его состоит в том, что элегантно разделить исполнение основной задачи, обновление промежуточных результатов в пользовательском интерфейсе (конечно же, из потока рассылки событий) и позволить выполнить какие-либо действия с интерфейсом после окончания вычислений.

Давайте попробуем переделать предыдущий пример с «долгими» вычислениями с помощью SwingWorker. Вот что у нас получится:

```
// UsingSwingWorker.java
// Класс SwingWorker для отзывчивости интерфейса
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.List;

public class UsingSwingWorker extends JFrame {
    private JButton button;
    public UsingSwingWorker() {
     super("UsingSwingWorker");
     // при закрытии окна - выход
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     // добавим кнопку со слушателем
     button = new JButton("Выполнить сложную работу");
     button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
             // запустим отдельную долгую работу
             new ComplexJob().execute();
             button.setText("Подождите...");
         }
     });
```

```
        // настроим панель содержимого и выведем окно на экран
        setLayout(new FlowLayout());
        add(new JTextField(20));
        add(button);
        setSize(300, 200);
        setVisible(true);
    }
    // класс, выполняющий "сложную работу"
    class ComplexJob extends SwingWorker<String,String> {
           // здесь выполняется работа, это отдельный поток!
           public String doInBackground() throws Exception {
               Thread.sleep(2000);
               // публикуем промежуточные результаты
               publish("Половина работы закончена...");
               Thread.sleep(2000);
               return "";
           }
           // обработка промежуточных результатов
           // это поток рассылки событий!
           protected void process(List<String> chunks) {
               button.setText(chunks.get(0));
           }
           // окончание работы - и вновь это поток рассылки
           public void done() {
               button.setText("Работа завершена");
           }
       }
    public static void main(String[] args) {
           SwingUtilities.invokeLater(
            new Runnable() {
             public void run() { new UsingSwingWorker(); } });
    }
}
```

Мы вновь создаем небольшое окно, в панели содержимого которого размещается кнопка JButton и вспомогательное текстовое поле. При нажатии кнопки нас ожидает тяжелая, долгая, неповоротливая задача, и во имя наших пользователей мы обязаны выполнить ее вне потока рассылки событий. Теперь нам помогает класс SwingWorker, специально предназначенный для этих задач.

Прежде всего, стоит отметить, что класс этот абстрактный и требует чтобы от него всегда наследовали, а также он позволяет настроить типы данных, с которыми он будет работать. В нашем примере мы выбираем для данных просто строки. Первый тип данных — это то, что будет возвращать в качестве результата наша задача. Так как у нас пример умозрительный, мы ничего не возвращаем и используем пустую строку. Второй

тип — это промежуточные результаты, у нас это текст для кнопки, которые мы выводим на экран.

Для запуска задачи в отдельном потоке применяется метод execute(). Можно запускать экземпляры SwingWorker и в своих собственных потоках, так как класс этот реализует интерфейс Runnable. Далее все интересное происходит в переопределенных нами методах, все они вызываются внутренними механизмами класса SwingWorker. Сама долгая работа выполняется в методе doInBackground(), он исполняется в другом потоке, и в нем ни в коем случае не следует работать с интерфейсом программы. Зато из этого метода мы может вызывать метод publish(), передавая туда список данных, этот список попадает в метод process(), и вызывается он уже из потока рассылки событий, где мы вольны так обновить интерфейс, как нам пожелается. Мы просто обновляем текст на кнопке. Ну а метод done() вызывается после окончания работы метода doInBackground() и его можно использовать для окончательного обновления интерфейса после окончания задачи.

Преимуществ использования класса SwingWorker вместо ручного создания потоков несколько. Во-первых, его методы несут больше смысловой нагрузки по сравнению с беспорядочными вызовами очереди событий и созданием кучи объектов Runnable, и понять что именно делает задача, обладая нехитрым знанием SwingWorker, очень легко. Ну а главное — это оптимизация SwingWorker. Он не позволит вам расплодить очень много маленьких потоков (на данный момент максимальное количество потоков в нем ограничено 10), ставя новые потоки в очередь ожидания, попытается оптимизировать вызовы publish(), да и просто использование системного класса предполагает что поддержка программ с ним в будущем всегда проще, случись вдруг в Swing какие-то архитектурные перемены. В системных классах поддержка перемен всегда быстра, достаточно загрузить новую версию пакета JDK.

## Почему мы так запускаем Swing-приложения

Начиная с первых глав, мы то и дело пишем маленькие примеры, чтобы лучше разобраться во всем, что происходит, и нетрудно заметить, особенно теперь, когда мы знаем о существовании потока рассылки и очереди событий, что мы с самого начала метода main() создаем объекты своих интерфейсов в этом самом потоке. Когда-то пользователи Swing наивно полагали, что создавать графические компоненты в других потоках до того момента, как те перейдут в реализованное состояние (помните, мы обсудили что это за состояние чуть ранее), вполне безопасно. Ведь так оно и есть для любого легковесного компонента AWT. Но, только если это не компонент библиотеки Swing.

Впрочем, зачем нам слова, если можно увидеть собственными глазами. Рассмотрим пример, который все проиллюстрирует:

```
// StartingEventThread.java
// Проверка момента запуска потока рассылки событий
import javax.swing.*;
import java.awt.*;

public class StartingEventThread {
    public static void main(String[] args) {
        // заменяем системную очередь событий своей
        Toolkit.getDefaultToolkit().
          getSystemEventQueue().push(new CustomQueue());
        // создаем окно
```

```
        JFrame frame = new JFrame("Тест");
        System.out.println("(1) JFrame()");
        // добавляем флажок
        JCheckBox checkBox = new JCheckBox("Тест");
        frame.add(checkBox, "South");
        System.out.println("(2) Добавлен флажок");
        // создаем список
        DefaultListModel model = new DefaultListModel();
        JList list = new JList(model);
        frame.add(list);
        System.out.println("(3) Добавлен список");
        // обновляем модель
        model.addElement("Тест");
        System.out.println("(4) Обновление модели");
        // окончательно выводим интерфейс на экран
        frame.setVisible(true);
        System.out.println("(5) Интерфейс построен");
    }
    // специальная очередь событий, сообщающая
    // отладочную информацию о событиях и потоках
    static class CustomQueue extends EventQueue {
        // метод кладет событие в очередь
        public void postEvent(AWTEvent event) {
            System.out.println("post(), поток: " +
              Thread.currentThread().toString());
            System.out.println("post(), событие: " + event);
            super.postEvent(event);
        }
        // метод распределяет событие по компонентам
        protected void dispatchEvent(AWTEvent event) {
            System.out.println("dispatch(), поток: " +
              Thread.currentThread().toString());
             System.out.println("dispatch(), событие: " + event);
            super.dispatchEvent(event);
        }
    }
}
```

В примере мы используем тот факт, что очередь событий можно заменить на свою реализацию, используя метод фабрики AWT Toolkit. Так как изобретать заново велосипед мы не желаем, мы просто унаследуем от стандартной очереди событий, разбавив ее реализацию отладочными сообщениями, которые покажут нам, кто и когда, а самое главное, из какого потока, кладет в очередь события. Нам понадобится два метода —

postEvent() применяется, когда в очередь кладется событие высокого уровня, например, события внутренних систем Swing, в метод же dispatchEvent() попадают события уже для окончательной обработки, некоторые из них могут и не попасть в postEvent() в текущей реализации очереди событий.

Далее мы, нарушая архитектуру остальных примеров и программ книги, начинаем создавать и настраивать графические компоненты Swing прямо в методе main(), которые запускается в своем отдельном потоке. После каждого действия, произведенного с компонентами пользовательского интерфейса, в консоль выводится диагностическое сообщение, которое даст нам знать, на каком этапе создания интерфейса мы находимся. С другой стороны, наша специальная очередь, внедренная в самое сердце Swing, расскажет нам, откуда появятся первые события и кем они будут поставлены в обработку. Самих действий не так много — создается окно JFrame, флажок, список JList с моделью по умолчанию, и производится изменение этой модели, что согласно модели MVC, должно привести к оповещению списка о необходимости перерисовки, хотя он еще и не виден на экране.

Все, что произойдет далее, мы увидим в консоли. По сути, результаты могут разниться в зависимости от вашей версии JDK, операционной системы и любых других факторов, способных повлиять на внутренние механизмы виртуальной машины JVM и реализации AWT. В момент написания этой главы первые строчки на консоли выглядели следующим образом:

```
(1) JFrame()
(2) Добавлен флажок
```

Таким образом, создание окна и флажка не вызвало добавление в очередь новых событий пользовательского интерфейса, и потоку рассылки событий запускаться нет смысла. Но затем происходит следующее:

```
(3) Добавлен список
post(), поток: Thread[main,5,main]
post(), событие: java.awt.event.InvocationEvent[...]
```

По некой причине добавление списка в окно добавляет в очередь новое событие InvocationEvent, а такие события обычно приходят от внутренних механизмов библиотеки Swing. Причем, как мы видим, событие было положено из потока main, потому что добавление списка произошло из метода main(). Все это означает, что поток рассылки событий запустился и вскоре доставит событие компонентам. Но мы еще не закончили создавать интерфейс и теперь с легкостью можем столкнуть поток рассылки и поток main. Посмотрим, что происходит далее:

```
(4) Обновление модели
dispatch(), поток: Thread[AWT-EventQueue-1,6,main]
dispatch(), событие: java.awt.event.InvocationEvent[...]
post(), поток: Thread[main,5,main]
post(), событие: java.awt.event.InvocationEvent[...]
dispatch(), поток: Thread[AWT-EventQueue-1,6,main]
dispatch(), событие: java.awt.event.InvocationEvent[...]
post(), поток: Thread[AWT-Windows,6,main]
post(), событие: java.awt.event.InvocationEvent[...]
post(), поток: Thread[main,5,main]
post(), событие: java.awt.event.InvocationEvent[...]
(5) Интерфейс построен
```

Теперь ситуация совсем вышла изпод контроля и может причинить нам массу неудобств. Обновление модели снова заставляет некие механизмы Swing посылать все новые события в очередь событий, и эти события рассылает (через метод dispatch()) другой поток рассылки событий. Все это видно в нашем консольном выводе. Любое случайное столкновение потоков приведет к проблемам — в легком случае может нарушиться внешний вид приложения, а в тяжелом оно запросто может зависнуть, потеряв все бесценные данные тяжелой работы пользователя.

**СОВЕТ**

> Не следует создавать или заранее подготавливать компоненты Swing в отдельном потоке, отличном от потока рассылки событий. Реализация компонентов Swing подразумевает любые манипуляции с ними из потока рассылки событий, даже если компоненты еще не добавлены на экран.

Последний вопрос — что же это за внутренняя работа Swing, которая пробуждает поток рассылки событий, даже если компоненты еще не добавлены на экран. Как мы узнаем из главы 5, это в основном сделано для нашего с вами удобства, так как любое обновление моделей и сложных компонентов автоматически обновляет их на экране, что и приводит к запуску потока рассылки (так называемая автоматическая проверка корректности и перерисовка). За удобство приходится платить, в том числе и таким образом: немного вычурной конструкцией запуска приложений Swing в методе main.

## *Отладка потоков в системе событий*

Как мы уже поняли, наиболее «щекотливым» вопросом при работе с системой событий Swing является необходимость практически для всех видов приложений использовать отдельные потоки для выполнения действий и при этом не переходить дорогу потоку рассылки событий. Даже если программу вы пишете сами и старайтесь все делать правильно и без ошибок, но при написании потоков настолько естественно вызывать методы компонентов Swing или их моделей напрямую, не думая о вызове через очередь событий, что очень легко немного промахнуться и подставить программу под удар.

В таких ситуациях удобно постоянно проверять себя, и всех своих коллег, если вы работаете не в одиночку, с помощью вспомогательных инструментов. Как мы вскоре узнаем в главе 4, в Swing все действия с компонентами приводят в объект RepaintManager, где можно проверить, из какого потока проводится работа с компонентами и найти проблемные ситуации. Писать с нуля нам ничего не понадобится, так как хорошие реализации уже есть.

Прежде всего, это хорошо известный сторонний внешний вид Swing, называемый Substance, который по умолчанию запрещает работу с компонентами Swing из «неправильных» потоков, создавая исключения. Таким образом, небрежно написанная программа вообще не имеет шансов стабильно работать в этом внешнем виде. Даже если вы используете другой внешний вид, вы можете тестировать свою программу во внешнем виде Substance для поиска возможных проблем с множественными потоками. Это не единственный вариант: существует также целая библиотека для модульного тестирования Swing-приложений под названием FEST, которая вызовом одного метода включает режим запрета работы с компонентами Swing из потоков, отличных от потока рассылки событий. В противном случае возникает исключение. Библиотека FEST очень полезна для автоматизированного тестирования Swing-приложений, попробуйте найти ее в Интернете, после ознакомления со Swing.

## *Резюме*

Система обработки событий Swing по-настоящему элегантна и прозрачна, она позволяет создавать гибкие и легко расширяемые программы. Чем более сложные программы и пользовательские интерфейсы вы будете создавать, тем больше вы будете ценить разнообразие и гибкость способов, которыми можно обработать события. События можно обрабатывать даже на самом низком уровне, вмешиваясь в сокровенные механизмы распределения событий и реализуя то поведение, которое вам необходимо. С другой стороны, за простотой и элегантностью системы обработки событий скрывается несколько нетривиальных механизмов, незнание которых может сделать ваш интерфейс неотзывчивым и, более того, привести всю программу к сложной ситуации конфликта нескольких потоков выполнения (тупику). В этой главе мы исследовали все нюансы процесса обработки событий в Swing, начав с самого простого и закончив исследованием «начинки» системы обработки событий.

# Глава 4. Рисование в Swing

В любой библиотеке, предназначенной для построения графических интерфейсов, важнейшим является процесс вывода содержимого непосредственно на экран. Любое добавление к внешнему виду приложения требует знания этого процесса в деталях. Система рисования Swing обосновалась в классе JComponent и его помощнике RepaintManager, и основана на уже имеющейся системе рисования AWT. Все это мы рассмотрим очень подробно, чтобы ни один пиксел на экране не был для нас загадкой, а также узнаем, какие инструменты для отслеживания и отладки сложных ситуация на экране нам доступны.

Если пока вы собираетесь довольствоваться стандартными компонентами Swing с минимальными изменениями, комбинируя их на экране, эта глава будет даже излишней, и вы можете смело переходить к следующим. Однако как только дело дойдет до первого собственного компонента или первым попыткам нарисовать что-то в Swing-приложении, непременно возвращайтесь сюда.

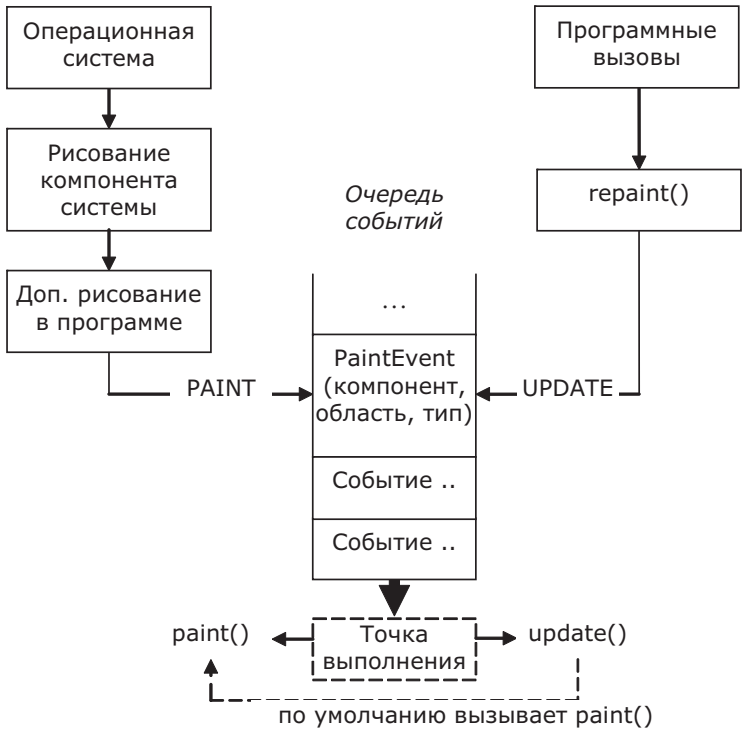
## *Система рисования*

Прежде чем говорить о деталях системы рисования, использованной в библиотеке Swing, имеет смысл увидеть, как рисование происходит в базовой библиотеке AWT. Именно она связывает абстрактные вызовы Java и реальные действия операционной системы на экране.

Во всех современных операционных системах процесс рисования происходит примерно одинаково. При этом ваша программа играет пассивную роль и терпеливо ожидает, когда настанет нужный момент. Момент этот определяют механизмы операционной системы, и настает он, когда необходимо перерисовать часть окна, принадлежащего вашему приложению, например, когда окно впервые появляется на экране или прежде скрытая его часть открывается глазам пользователя. Операционная система при этом определяет, какая часть окна нуждается в перерисовке, и вызывает специальную часть вашей программы, отвечающую за рисование, либо посылает вашему приложению сообщение о перерисовке, которое вы при необходимости обрабатываете. Частью программы в графических компонентах AWT, которая вызывается системой для прорисовки компонента, является метод paint().

Программная, или принудительная, перерисовка, также необходима — вам необходимо иметь возможность вручную указывать системе, что пора заново нарисовать тот или иной фрагмент вашего экрана. Этого требует анимация или любые динамические изменения в интерфейсе, не ждать же, в самом деле, пока операционная система соизволит перерисовать ваше окно. Перерисовку позволяет вызвать еще один доступный всем компонентам AWT метод repaint().

Проще всего оценить схему рисования, использованную по умолчанию в Java и AWT, на простой диаграмме, и мы сразу увидим всю ее подноготную:




Отталкиваться приходится от того, что графическая система хранит все свои события в очереди (мы выяснили это в подробностях в главе 3), чтобы избежать мусора на экране, вызванного его непоследовательным обновлением. Вызовы о прорисовке того или иного фрагмента экрана также необходимо разместить в очереди, и делается это с помощью специального события PaintEvent.

В первом варианте призыв нарисовать фрагмент экрана присылает операционная система, когда, по ее мнению, он был «поврежден», то есть свернут, закрыт другим окном и т.п. Если в этот фрагмент входит системный компонент (тот самый, что представлен компонентами AWT, такими как кнопки Button или списки List), он перерисовывает себя сам, и выглядит именно так, как ему положено в данной операционной системе. После этого помощник (peer) компонента или системная часть Java создаст событие PaintEvent с типом PAINT, укажет в нем, какую область экрана необходимо перерисовать и в каком компоненте и поместит его в очередь событий EventQueue.

Программная перерисовка много проще и никаких системных вызовов не касается. В методе repaint() просто создается событие PaintEvent с типом UPDATE, указывается компонент (тот самый для которого и был вызван repaint()) и область перерисовки (ее можно указать вручную или будет использован весь размер компонента) и также помещается в очередь событий.

Вспоминая архитектуру событий Swing, логично было бы подумать, что для получения сигналов о прорисовке можно присоединять слушателей типа PaintEventListener, которые затем оповещаются при рассылке событий из методов dispatchEvent() и processPaintEvent(). Однако мы не можем обрабатывать сигналы о прорисовке как обычные события. Вместо этого события PaintEvent обрабатываются самой системой (в помощниках), когда поток рассылки событий «вытаскивает» их из очереди и передает в метод dispatchEvent() компонента, к которому они принадлежат. Для событий типа PAINT вызывается метод

paint() компонента, в котором необходимо провести перерисовку. Для событий UPDATE вызывается метод update(), который по умолчанию все также вызывает метод paint(). Все эти методы определены в базовом классе любого компонента Component.

В качестве средства для рисования в каждый из этих «рисующих» методов передается графический объект Graphics (часто его называют графическим контекстом). Именно с его помощью графические примитивы выводятся на экран. Получить его можно не только в этих методах, но и создать самому, вызвав метод getGraphics() (доступный в любом компоненте). Это позволяет нарисовать что-то мгновенно, не дожидаясь вызова «рисующего» метода, однако, это практически бесполезно. Любой следующий вызов «рисующего» метода все равно нарисует на экране то, что определено в нем, так что лучше все сводить к методу paint(). Кстати, объект Graphics для рисующих методов системная часть Java также создает методом getGraphics().

Простейший пример покажет нам все в действии:

```
// AWTPainting.java
// Процесс рисования в AWT очень прост
import java.awt.*;
import java.awt.event.*;

public class AWTPainting extends Frame {

public AWTPainting() {
super("AWTPainting");
// выход при закрытии окна
addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e) {
System.exit(0);
}
});
setLayout(new FlowLayout());
// попробуем закрасить часть кнопки
add(new Button("Перерисуем кнопку!") {
public void paint(Graphics g) {
g.setColor(Color.BLUE);
g.fillRect(2, 2, getWidth() — 5, getHeight() — 5);
}
});
setSize(200, 200);
}

// в этом методе производится рисование
public void paint(Graphics g) {
// заполняем все красным цветом
g.setColor(Color.RED);
g.fillRect(0, 0, getWidth(), getHeight());
```

```
}

public static void main(String[] args) {
new AWTPainting().setVisible(true);
}
}
```

В примере создается окно с рамкой Frame (мы наследуем свой класс от него), в нем мы в качестве менеджера расположения используем последовательное расположение FlowLayout и добавляем кнопку Button с незамысловатым текстом. И в окне, и в кнопке процедура прорисовки компонента paint() заменена на нашу собственную. Окно полностью закрашивается красным цветом, а кнопка, не считая маленького ободка, синим (мы попросту стираем ее текст).


Запустив пример, мы ничего, кроме красочных цветов, не увидим, самое интересное происходит в движении, для чего размер окна надо увеличивать. Видно, что окно закрашивается полностью — каждый раз вызывается paint(), в котором методы getWidth() и getHeight() возвращают новые актуальные размеры. Если у вас не слишком производительный компьютер, вы легко увидите, как мерцает та область окна, которая прорисовывается, — такова скорость передачи операций из кода Java в операционную систему.

Что касается кнопки, то она на первый взгляд выглядит, как положено, но если присмотреться (все будет зависеть от вашей версии пакета JDK и вашей операционной системы), можно будет заметить, как время от времени системная кнопка пытается прорваться на экран через наш синий «заслон». Проблема все та же: AWT банально не успевает ее закрасить, и мы можем увидеть, что именно приложение пыталось от нас скрыть. Но если тут мы хоть что-то закрасили, то, например, список List на платформе Windows просто не даст себя закрасить. Так уж он устроен в операционной системе. Вывод прост — компоненты AWT просто не предназначены для модификации и смены их внешнего вида из кода Java.

## Метод paint() — резюме

Итак, в методе paint() размещается код, прорисовывающий компонент. Причем учитывать, что именно в компоненте поменялось, вовсе не обязательно, так как в метод передается графический контекст Graphics, которому уже задан прямоугольник отсечения (clip) (переданный или системой, или из метода repaint()). За преде-

лами этого прямоугольника прорисовка не производится и время не нее не затрачивается.

Добавлять к рисованию системных компонентов AWT свои детали не стоит — слишком неопределенна система взаимодействия системной процедуры прорисовки и метода paint(). Как правило, системный компонент будет «прорываться» через нарисованное вами, а то и вообще не даст ничего на себе нарисовать. Специально для рисования в AWT предусмотрен компонент-«холст» Canvas.

Если вам понадобится сменить какие-либо глобальные параметры графического объекта Graphics, например включить сглаживание, изменить прямоугольник отсечения, а ваш компонент может содержать другие компоненты, особенно легковесные, создавайте копию объекта, вызывая метод create() класса Graphics. В противном случае все ваши настройки перейдут по наследству всем компонентам, которые могут прорисовываться после вашего компонента. К тому же существует один нюанс: после завершения рисования для такого объекта придется явно вызвать метод dispose(), иначе ресурсы системы рисования могут быстро закончиться[1].

## Метод repaint() — пара дополнений

Как мы увидели, метод программной перерисовки repaint() для стандартных компонентов AWT вызывает сначала метод update(). Когда-то создатели AWT полагали, что это поможет реализовать эффективную технику инкрементальной прорисовки. Это значило, что каждый раз, когда вы вызывали repaint(), в методе update() к уже прорисованному компоненту можно было добавить какие-то детали, а потом перейти к основной картине в методе paint() (установив предварительно нужный прямоугольник отсечения (clip), чтобы не пропало то, что было только что нарисовано).

Однако такой подход бывает редко востребован, а эффективную прорисовку можно осуществить и путем ограничения области рисования в методе paint() (применяя тот самый прямоугольник отсечения). Так что задумка создателей AWT не удостоилась внимания масс.

Интерес также могут вызвать некоторые версии repaint(), которым можно указать время (в миллисекундах), за которое перерисовка должна бы произойти. Документация JDK всерьез утверждает, что этот параметр именно так и действует, однако на самом деле он пока не реализован.

Самой же главное рекомендацией остается вызов repaint() с максимально суженной областью перерисовки компонента. Это особенно верно для сложных, наполненных динамическим содержимым и анимацией компонентов, так как это приносит огромную экономию по времени прорисовки. Как мы сейчас увидим, Swing старается взять большую часть этой задачи на себя, однако где возможно, все равно стоит отслеживать минимальную область прорисовки самим.

## Рисование легковесных компонентов

Как обрабатывается рисование системных компонентов, мы только что увидели. Однако они настолько редко применяются (и мерцание при прорисовке еще раз доказывает, что не зря), что их можно расценивать лишь как приятное дополнение

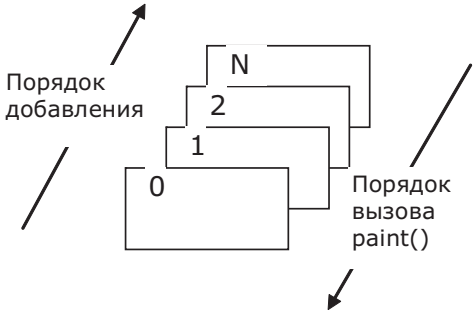
[1] Можно предложить применять интересный вариант кода, позволяющий всегда гарантировать удаление созданного объекта Graphics. Как мы все знаем, всегда вызывается секция finally. Так как исключений при рисовании практически не возникает, секцию catch можно не указывать:

```
try {
 создаем новый объект Graphics, рисуем…
} finally {
  объект.dispose()
}
```

к легковесным компонентам. Как мы знаем, легковесный компонент представляет собой область экрана тяжеловесного контейнера. Для того чтобы он мог правильно отображаться на экране, ему также необходимо получать от системы прорисовки вызовы своего метода paint(), однако операционная система, заведующая этим процессом в AWT, ничего не знает о существовании легковесных компонентов, она видит лишь «родные» тяжеловесные компоненты и контейнеры, которым и отправляет запросы на перерисовку. Решение здесь очевидно — нужно встроить поддержку легковесных компонентов в тяжеловесные контейнеры, и именно так поступили создатели AWT.

Все контейнеры в AWT (а значит и в Swing) унаследованы от своего базового класса Container. Именно там легковесные компоненты и становятся полноправными участниками процесса вывода на экран. Никаких хитростей здесь нет, нужно лишь четко очертить круг участников этого процесса. Итак — легковесный компонент — это нечто, унаследованное от класса Component, и не связанное с операционной системой помощником (peer). Как он выглядит на экране, определяет исключительно его рисующий метод paint(). Эти компоненты можно добавлять в контейнер (любой, лишь бы он был унаследован от класса Container).

В контейнерах поддерживается иерархия компонентов, выстроенная «по оси Z» (z-order). Они как бы нанизываются на ось Z, устремленную *от нас*. Первый компонент имеет индекс 0, второй 1, и так далее. Рисование идет в обратном порядке, так что первый добавленный компонент всегда закрывает остальные, если они перекрывают друг друга. Диаграмма окончательно все прояснит:




В итоге, когда операционная система запрашивает прорисовку принадлежащего ей тяжеловесного контейнера (в случае приложения Swing это будет как правило окно JFrame либо диалог JDialog), вызывается его метод paint() (мы только что выяснили это в предыдущем разделе). В нем, согласно порядку добавления компонентов, от последнего к первому, происходят следующие шаги:

- Устанавливается область отсечения для рисования (clip), соответствующая области, занимаемой легковесным компонентов в контейнере.
- Для прорисовываемого компонента в объекте Graphics выставляются текущие цвета фона и шрифта (берутся из контейнера, впрочем, компоненту совсем не обязательно их использовать).
- Устанавливается текущий шрифт (из контейнера)
- Настроенный объект Graphics передается в метод paint() легковесного компонента, который и рисует себя (а фактически, просто участвует в процессе прорисовки все того же тяжеловесного контейнера).

Идея проста — каждый легковесный компонент вносит свою лепту в процесс рисования контейнера, в то время как операционная система считает, что контейнер просто рисует себя. Так как установлена область отсечения, легковесный компонент никогда не сможет «залезть» в область, которая ему не принадлежит, и пририсовать там что-то лишнее. Поскольку рисование идет от последних добавленных компонентов к первым, первые всегда будут перекрывать последние в случае пересечения.

Отсюда же возникает чудесная способность легковесных компонентов быть прозрачными и принимать любые формы — их никто не заставляет хоть что-то рисовать и не обязывает закрашивать фоновым цветом занимаемый на экране прямоугольник, как это происходит с тяжеловесными компонентами. Таким образом, внутри тяжеловесного компонента руки у нас развязаны и компонентам можно придать любые формы, хотя по сути они, конечно же, остаются прямоугольными.

Долгое время из этой идиллии выпадали лишь тяжеловесные компоненты, если их добавляли в тот же контейнер. Вне зависимости от их позиции по оси Z, они *перекрывали* легковесные компоненты. Природа тяжеловесных и легковесных компонентов слишком различна, поэтому и рекомендовалось вообще *не совмещать* их в одном контейнере во избежание проблем с прорисовкой и расположением на экране. Однако с появлением обновления Java версии 6, малой версии 12 и выше, и, конечно же, в Java 7 проблема была устранена и тяжеловесные компоненты перестали бессовестно «тянуть на себя одеяло» в контейнерах. Как правило, при разработке Swing-приложений совмещения компонентов практически не требовалось, но в случае необходимости задействовать какой-либо системный графический ресурс, например панель с ускорением трехмерной графики, возможность использовать вместе с такими тяжеловесными компонентами все богатство Swing бывает весьма кстати.

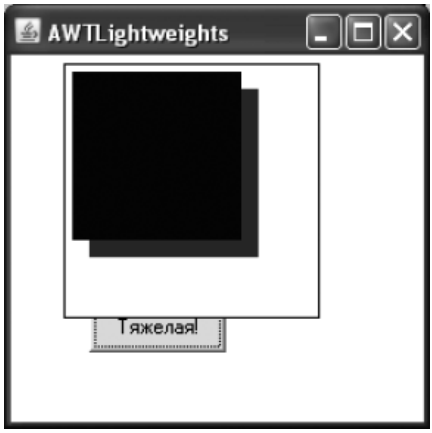
А теперь рассмотрим простой пример и убедимся в справедливости всего сказанного:

```
// AWTLightweights.java
// Использование легковесных компонентов в AWT
import java.awt.*;
import java.awt.event.*;

public class AWTLightweights extends Frame {

  public AWTLightweights() {
    super("AWTLightweights");
    // при закрытии окна приложение завершается
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });
    // добавляем пару легковесных компонентов
    LightweightRect rect1 =
        new LightweightRect(Color.BLUE, true);
    LightweightRect rect2 =
        new LightweightRect(Color.RED, true);
    LightweightRect transparentRect =
```

```
        new LightweightRect(Color.BLACK, false);
    // укажем координаты вручную, чтобы компоненты
    // перекрывались
    setLayout(null);
    rect1.setBounds(40, 40, 100, 100);
    rect2.setBounds(50, 50, 100, 100);
    transparentRect.setBounds(35, 35, 150, 150);
    add(transparentRect);
    add(rect1);
    add(rect2);
    // последним добавляем тяжеловесный компонент
    Button button = new Button("Тяжелая!");
    button.setBounds(50, 175, 80, 30);
    add(button);
    // выводим окно на экран
    setSize(250, 250);
    setVisible(true);
  }

  // легковесный компонент — цветной прямоугольник
  class LightweightRect extends Component {
    private Color color;
    private boolean fill;

    // параметы — цвет и нужно ли зарисовывать всю область
    public LightweightRect(Color color, boolean fill) {
      this.color = color;
      this.fill = fill;
    }
    public void paint(Graphics g) {
      g.setColor(color);
      if (fill)
        g.fillRect(0, 0, getWidth() — 1, getHeight() — 1);
      else
        g.drawRect(0, 0, getWidth() — 1, getHeight() — 1);
    }
  }
  public static void main(String[] args) {
    new AWTLightweights();
  }
}
```

В примере в качестве тяжеловесного контейнера используется окно Frame, от которого мы наследуем. Легковесным компонентом выступит класс LightweightRect, который унаследован от базового класса Component и переопределяет метод для рисования paint(). Это прямоугольник, которому через конструктор можно задать цвет и свойство заполнения цветом — будет он закрашен или нет(нарисуется просто рамка).

Чтобы заставить компоненты перекрываться, придется отказаться от менеджера расположения (применяя метод setLayout(null) — подробнее о расположении компонентов рассказывается в главе 7), так как они не рассчитаны на перекрывание компонентов. Мы просто задаем позицию компонентов и их размеры методом setBounds(). Порядок добавления компонентов определяет их место и «первенство» на экране. Сначала идет незакрашенный прямоугольник, затем два закрашенных, и в последнюю очередь тяжеловесная кнопка Button. Несмотря на то, что прозрачный прямоугольник должен закрыть оба закрашенных, последние прекрасно видны — все благодаря прозрачности легковесных компонентов. Закрашенные прямоугольники перекрываются согласно старшинству — наверху тот, что был добавлен первым. Тяжеловесная кнопка находится в самом низу, так как была добавлена последней, то есть имеет самую дальнюю позицию по оси Z[2]. Однако прозрачность легковесного компонента на нее не действует, и вы легко это увидите, запустив пример. Причины кроются во внутренних механизмах библиотеки AWT, ну а нам придется аккуратнее совмещать тяжеловесные и легковесные компоненты, если это все же понадобится.




Для полноты картины остается добавить, что для легковесных компонентов немного изменилось поведение метода repaint(). Суть его осталась прежней — он все также помещает в очередь событий сообщение о необходимости перерисовки, но происходит это не напрямую из метода repaint() легковесного компонента. Когда вы вызываете repaint() для легковесного компонента, он переадресует запрос контейнеру, в котором содержится, указав в качестве области перерисовки ту область, что он занимает в контейнере. Так продолжается до тех пор, пока запрос не достигнет тяже-

[2] Если у вас есть инструмент, выводящий список окон для приложения (такой, как Spy++ для Windows), вы можете провести интересный эксперимент. Запустите программу с нашим примером и выведите ее список окон. Вы увидите, что их всего два — это само окно Frame и кнопка Button. Как мы и говорили, легковесных компонентов «родная» система не видит. С точки зрения Java это, тем не менее, самые настоящие компоненты.

ловесного контейнера (легковесные компоненты могут содержаться в легковесных контейнерах, так что тяжеловесный контейнер можно искать долго). В итоге запрос на перерисовку в очередь событий помещает именно тяжеловесный контейнер, причем перерисовка «заказывается» лишь для области нужного легковесного компонента. Это позволяет сократить затраты, потому что легковесный компонент может занимать лишь малую часть контейнера[3].

## Легковесные компоненты: резюме

Легковесные компоненты не доставляют проблем, если помнить некоторые детали:

- Легковесные компоненты хранятся в контейнере в порядке добавления (по оси Z), и рисуются наоборот — значит, те компоненты, что добавлены сначала, имеют на экране приоритет.
- Легковесные компоненты могут быть прозрачны или рисовать себя в любой форме — через незатронутые области могут «просвечивать» другие компоненты или фон контейнера.
- Метод repaint() работает для легковесных компонентов — но за кулисами перерисовывается часть тяжеловесного контейнера. Метод update() не вызывается.
- Если вы переопределяете контейнер и его метод paint() и планируете затем добавлять в него другие компоненты, не забудьте вызвать базовый метод super.paint(), иначе о прорисовке легковесных компонентов придется позаботиться самостоятельно.

# *Система рисования Swing*

По большому счету, легковесные компоненты ничем не ограничены, и создать с их помощью, не мудрствуя лукаво, даже такую богатую библиотеку, как Swing, не представляет собой технических трудностей. Однако во времена зарождения Swing пользовательские интерфейсы Java страдали от проблем с производительностью. Богатые возможностями, иногда даже прозрачные компоненты Swing просто сделали бы из пользовательского интерфейса мало удобоваримого тихохода. Действительно, все легковесные компоненты рисуются один за другим, вне зависимости от того, есть ли среди них непрозрачные области, и каждая операция с графикой — это долгая операция по обращения к «родному» коду операционной системы. Оптимизация была необходима как воздух, и оптимизация серьезная.

Отличие системы рисования Swing от стандартной состоит в оптимизации и поддержке UI-представителей. Другими словами, библиотека Swing в своей системе рисования руководствуется всего тремя принципами, которые позволяют нам избавиться от лишней работы и дают уверенность в том, что рисование максимально оптимизировано:

- Кэшируй
- Разделяй и властвуй
- С глаз долой, из сердца вон

[3] Вспоминая, что при вызове repaint() сначала вызывается метод update(), мы видим, что и тут метод update() вызывается — но для тяжеловесного контейнера. А вот для легковесных компонентов update() не вызывается, так как рисующий их контейнер напрямую вызывает метод paint().

## Кэширование

Стратегия кэширования, применяемая в Swing, крайне проста и стара как мир. Если устройство выводит данные на экран слишком медленно (не то чтобы сейчас медленные видеокарты, но помните что в Java это вызовы «родного» кода), необходимо заранее подготовить все в быстром хранилище-буфере (в памяти), а затем одним махом (одной операцией) записать все в устройство (на экран). В качестве буфера применяется область в памяти в формате экрана, по возможности — область в памяти видеокарты. Буфер этот хранит вспомогательный класс для рисования в Swing под названием RepaintManager. Он же проверяет, что буфер корректен, при необходимости заново создает его и выполняет тому подобную техническую работу. Также этот класс позволяет компонентам запросить прорисовку в буфер с последующим выводом на экран.

Интересная и часто ведущая к проблемам при сложной прорисовке особенность компонентов Swing состоит в том, что они «общаются» друг с другом посредством системы флагов. Вся эта система встроена в базовый класс JComponent. Таким образом, рассчитывать на то, что компоненты Swing являются автономными сущностями, каждый со своей жизнью и процедурой прорисовки (как по большому счету это обстоит в AWT), не стоит. Посмотрим, как рисуется компонент Swing, добавленный в тяжеловесный контейнер:

1. Операционная система просит окно заново нарисовать содержимое
2. Окно проходит по списку содержавшихся в нем легковесных компонентов и находит там компонент Swing
3. Вызывается метод paint(). Чтобы встроить оптимизацию во все компоненты Swing, метод paint() в них не переопределяется, и таким образом, работу всегда выполняет базовый класс JComponent.
4. Если (согласно флагам), это первый вызванный для рисования компонент Swing, то он запрашивает рисование через буфер у класса RepaintManager. Тот настраивает буфер и вызывает метод paintToOffscreen() вызвавшего его компонента, передавая ему объект Graphics для рисования в памяти. Выставляется флаг, который отныне говорит, что рисование пошло в буфер.
5. Снова вызывается метод paint(), но так как флаг рисования в буфер уже выставлен, RepaintManager больше не применяется. Вызываются «рабочие» методы прорисовки компонента, рамки и компонентов-потомков (об этих методах чуть ниже).
6. Компоненты-потомки рисуют своих потомков (напрямую, согласно флагам), и так далее до полной прорисовки всей иерархии данного компонента Swing.
7. После окончания этой процедуры управление (согласно шагу 4) возвращается в RepaintManager, которые копирует изображение компонентов из буфера на реальный экран.

Данная процедура верна для любого компонента Swing. Как видно, наиболее удачным решением было бы наличие одного компонента, который занимал бы всю площадь тяжеловесного контейнера, а значит, именно этот компонент и был бы единственным участником, передающим управление от AWT в Swing, и копировать изображение из буфера на экран пришлось бы только один раз. Именно так и сделано, а роль такого компонента в контейнерах высшего уровня Swing играет корневая панель JRootPane.

Кэширование (буферизацию) графики в Swing можно отключить либо методом RepaintManager.setDoubleBufferingEnabled(), либо непосредственно на компоненте методом

setDoubleBuffered(). Правда, смысл выключения двойной буферизации для отдельных компонентов немного меняется. Мы уже видели, что компонент рисует с помощью буфера, если буфер используется его родительским компонентом, независимо от того, включена или выключена двойная буферизация для него самого. Если все компоненты находятся в корневой панели, выключать двойную буферизацию имеет смысл только для нее (это будет равносильно выключению двойной буферизации для всех компонентов, находящихся внутри этого контейнера высшего уровня). Учтите, что с появлением в AWT, начиная с версий Java 1.4, системной поддержки буферизации, система Swing может быть отключена во избежание дублирования. С другой стороны, отключать оптимизацию самостоятельно приходится редко, один случай мы вскоре рассмотрим, а вообще лучше довериться Swing.

## Разделение обязанностей

Как видно, метод paint() в компонентах Swing занят довольно-таки важным делом — он кэширует вывод графики компонентов на экран, пользуясь помощью класса RepaintManager. Переопределять этот метод в каждом компоненте чтобы нарисовать его больше нельзя — мы помним про систему флагов и что метод paint() действует по разному в зависимости от ситуации. Здесь создатели Swing применили лозунг «разделяй и властвуй» — рисование отныне производится в других методах, а метод paint() является носителем внутренней логики библиотеки и переопределять его без дела не следует. Это значительно упрощает обновление компонентов и изменение их внешнего вида, позволяя вам при создании нового компонента не думать о том, как реализовать для него эффективный вывод графики. Вы просто рисуете свой компонент, оставляя низкоуровневые детали механизмам базового класса.

Рисование компонента в Swing, в отличие от простой до невозможности процедуры в библиотеке AWT, разбито на три этапа, и за каждый отвечает свой метод в классе JComponent. Как раз эти методы вы будете переопределять, если вам захочется нарисовать по-своему что-либо на компоненте Swing.

## Метод paintComponent()

Метод paintComponent() вызывается при прорисовке компонента первым, и именно он рисует сам компонент. Разница между ним и классическим методом paint(), используемым в AWT, состоит в том, что вам не нужно заботиться ни об оптимизации рисования, ни о правильной прорисовке своих компонентов-потомков. Обо всем этом позаботятся механизмы класса JComponent. Все, что вам нужно сделать, — нарисовать в этом методе компонент и оставить всю черновую работу базовому классу.

Как вы помните, в Swing используется немного модифицированная архитектура MVC, в которой отображение компонента и его управление выполняются одним элементом, называемым UI-представителем. Оказывается, что прорисовка компонента с помощью UI-представителя осуществляется именно из метода paintComponent(), определенного в базовом классе JComponent. Действует метод очень просто: определяет, есть ли у компонента UI-представитель (не равен ли он пустой ссылке null) и, если представитель есть, вызывает его метод update(). Метод update() для всех UI-представителей работает одинаково: по свойству непрозрачности проверяет, нужно ли закрашивать всю свою область цветом фона, и вызывает метод paint(), определенный в базовом классе всех UI-представителей — классе ComponentUI. Последний метод и рисует компонент. Остается лишь один вопрос: что такое свойство непрозрачности?

Мы отмечали, что одним из самых впечатляющих свойств легковесных компонентов является их способность быть прозрачными. Однако при написании библиотеки

создатели Swing обнаружили, что набор из нескольких десятков легковесных компонентов, способных «просвечивать» друг сквозь друга, приводит к большой загрузке системы рисования и соответствующему замедлению работы программы. Действительно, перерисовка любого компонента оборачивалась настоящей каторгой: сквозь него просвечивали другие компоненты, которые задевали еще одни компоненты, и так могло продолжаться долго. В итоге, перерисовка даже небольшой части одного компонента приводила к перерисовке доброго десятка компонентов, среди которых могли оказаться и очень сложные. С другой стороны, компоненты вроде текстовых полей или кнопок редко бывают прозрачными, и лишняя работа для них совершенно не к чему. Так и появилось свойство непрозрачности (opaque), имеющееся у любого компонента Swing.

Если в AWT любой легковесный компонент автоматически считается прозрачным, то в Swing все сделано *наоборот*. Свойство непрозрачности определяет, обязуется ли компонент закрашивать всю свою область, чтобы избавить Swing от дополнительной работы по поиску и прорисовке всего того, что находится под компонентом. Если свойство непрозрачности равно true (а по умолчанию оно равно true), то компонент обязан закрашивать всю свою область, иначе на экране вместо него появится мусор. Дополнительной работы здесь немного: всего лишь необходимо зарисовать всю свою область, а облегчение для механизмов прорисовки получается значительное. Ну а если вы все-таки решите создать компонент произвольной формы или прозрачный, вызовите для него метод setOpaque(false), и к вам снова вернутся все чудесные возможности легковесных компонентов — система прорисовки будет предупреждена. Однако злоупотреблять этим не стоит: скорость прорисовки такого компонента значительно падает. Во многом из-за этого в Swing не так уж и много компонентов, имеющих прозрачные области.

Вернемся к методу paintComponent(). Теперь роль его вполне очевидна: он прорисовывает компонент, по умолчанию используя для этого ассоциированного с компонентом UI-представителя. Если вы собираетесь создать новый компонент с собственным UI-представителем, то он будет прекрасно вписываться в эту схему. Унаследуйте своего UI-представителя от базового класса ComponentUI и переопределите метод paint()[4], в котором и рисуйте компонент. Базовый класс позаботится о свойстве непрозрачности. Если же вам просто нужно что-либо нарисовать, унаследуйте свой компонент от любого подходящего вам класса (лучше всех для этого подходят непосредственно классы JComponent или JPanel, потому что сами они ничего не рисуют) и переопределите метод paintComponent(), в котором и рисуйте. Правда, при таком подходе нужно позаботиться о свойстве непрозрачности (если оно равно true) самостоятельно: потребуется закрашивать всю область прорисовки или вызывать перед рисованием базовую версию метода super.paintComponent()[5].

## Метод paintBorder()

Благодаря методу paintBorder() в Swing имеется такая замечательная вещь, как рамка (border). Для любого компонента Swing вы можете установить рамку, используя метод setBorder(). Оказывается, что поддержка рамок целиком и полностью обеспечи-

[4] С названиями рисующих методов в Swing сплошная путаница: все они называются одинаково. Здесь речь идет конечно о методе paint(), определенном в классе ComponentUI, а не об основном рисующем методе любого компонента. Видимо, создатели Swing хотели наглядно показать, что UI-представитель буквально забирает часть методов у компонента.

[5] Однако нужно помнить, что вызов базового метода правильно сработает только для панели (JPanel), у которой есть свой UI-представитель, способный обработать свойство непрозрачности. У класса JComponent такого представителя нет (он абстрактный, вы не сможете вывести его на экран, так что UI-представитель ему на самом деле не нужен), и если вы наследуете от него, то заботьтесь о свойстве непрозрачности самостоятельно.

вается методом paintBorder() класса JComponent. Он вызывается вторым, после метода paintComponent(), смотрит, установлена ли для компонента какая-либо рамка, и если рамка имеется, прорисовывает ее, вызывая определенный в интерфейсе Border метод paintBorder(). Единственный вопрос, который при этом возникает: где именно рисуется рамка? Прямо на пространстве компонента или для нее выделяется отдельное место? Ответ прост — никакого специального места для рамки нет. Она рисуется прямо поверх компонента после прорисовки последнего. Так что при рисовании компонента, если вы не хотите неожиданного наложения рамки на занятое место, учитывайте место, которое она занимает. Как это делается, мы узнаем в главе 8, часть которой полностью посвящена рамкам.

Переопределять метод paintBorder() вряд ли стоит. Работу он выполняет нехитрую, и как-либо улучшить ее или коренным образом изменить не представляется возможным. Если вам нужно создать для своего компонента фантасмагорическую рамку, лучше воспользоваться услугами интерфейса Border или совместить несколько стандартных рамок.

## Метод paintChildren()

Заключительную часть процесса рисования выполняет метод paintChildren(). Как вы помните, при обсуждении легковесных компонентов в AWT мы отмечали, что для их правильного отображения в контейнере, если вы переопределили их методы paint(), необходимо вызвать базовую версию paint() из класса Container, иначе легковесные компоненты на экране не появятся. Базовый класс JComponent библиотеки Swing унаследован от класса Container и вполне мог бы воспользоваться его услугами по прорисовке содержащихся в нем компонентов-потомков. Однако создатели Swing решили от услуг класса Container отказаться и реализовали собственный механизм прорисовки потомков. Причина проста — по сравнению с AWT компоненты Swing намного сложнее и требуют иного подхода. Улучшенный оптимизированный механизм и реализуется методом paintChildren(). Для придания ему максимальной скорости компоненты Swing используют два свойства: уже известное нам свойство непрозрачности opaque, а также свойство isOptimizedDrawingEnabled.

Метод paintChildren() действует по алгоритму, который был слегка вольнодумно назван нами «с глаз долой, из сердца вон». Он получает список содержащихся в компоненте потомков и начинает перебирать их, используя при этом текущий прямоугольник отсечения. Основной задачей его является нахождение «слепых зон», то есть зон, где компоненты закрываются друг другом. Если компонент закрыт или вообще не попадает в область отсечения, то зачем его рисовать — никто труда не заметит. Можно нарисовать только тот компонент, что виден сверху.

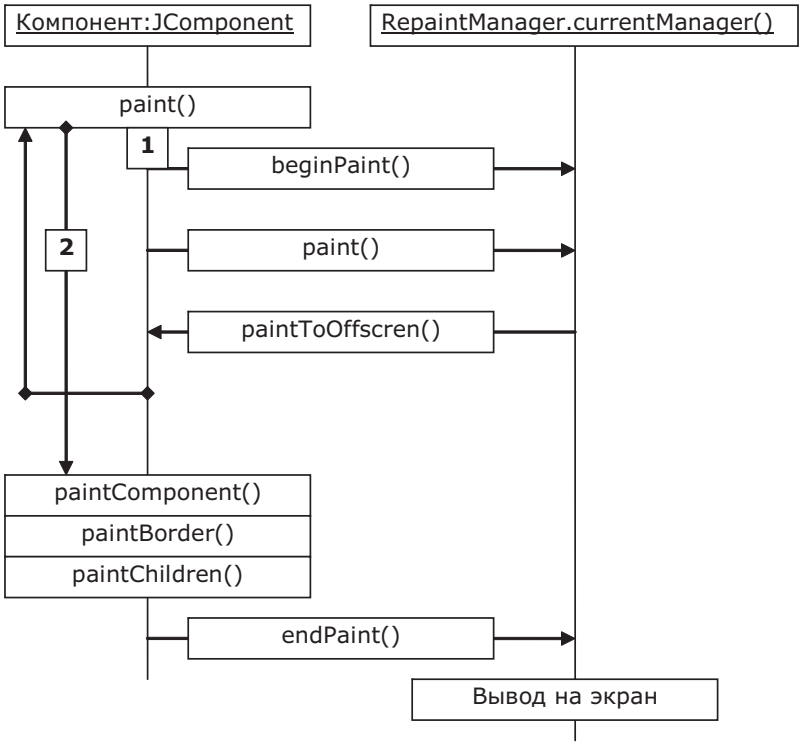
На этом этапе «вступают в бой» два вышеупомянутых свойства. С первым свойством все более или менее понятно: если свойство непрозрачности компонента равно true, это означает, что, сколько бы компонентов не находилось под ним и не пересекало бы его, он обязуется закрасить всю свою область, а значит продолжать поиск потомков в этой области не имеет смысла — их все равно не будет видно. Теперь понятно, почему так важно выполнять требование заполнения всей области экрана при использовании свойства непрозрачности: в противном случае на экране неизбежен мусор, который по договоренности должен убирать сам компонент, а не система прорисовки. Со свойством isOptimizedDrawingEnabled картина немного другая.

Данное свойство определено в классе JComponent как предназначенное только для чтения: вы не можете изменить его, кроме как унаследовав собственный компонент и переопределив метод isOptimizedDrawingEnabled(). По умолчанию для большинства компонентов свойство isOptimizedDrawingEnabled равно true. Это позволяет снизить загрузку системы прорисовки. Проще говоря, это свойство гарантирует, что из под одними по-

томками не «просвечивают» другие и не задевают их, при этом на них не наложен дополнительный прозрачный компонент и т. д. Когда речь идет о простых компонентах, это свойство приносит небольшие дивиденды, однако, если у вас есть сложные и медленно рисующиеся компоненты, оно значительно ускоряет процесс. Когда свойство isOptimizedDrawingEnabled равно true, метод paintChildren() просто перебирает потомков и перерисовывает их поврежденные части, не разбираясь, что они собой представляют и как друг с другом соотносятся. Данное свойство переопределяют лишь три компонента: это многослойная панель JLayeredPane, рабочий стол JDesktopPane и область просмотра JViewport. В них компоненты-потомки часто перекрываются и требуют особого внимания.

## Общая диаграмма рисования в Swing

Теперь, когда все детали оптимизации и разделения прорисовки в Swing нам известны, мы можем все объединить в несложную диаграмму, на которую можно смотреть если вдруг что-то на экране перестанет рисоваться как нужно:




Как мы выяснили, в методе paint() компонентов Swing есть два пути — первый работает в том случае, если компонент Swing рисуется кем-то, не являющимся компонентом Swing, — в этом случае включаются процессы RepaintManager и подготавливается буферкэш. Если же они включены, идет стандартный второй путь, рисующий все известными уже нам тремя методами. И тот же самый второй способ вызывается напрямую, если буферизация в компоненте отключена (вручную или если буферизация включена на уровне операционной системы).

## Программная перерисовка в Swing

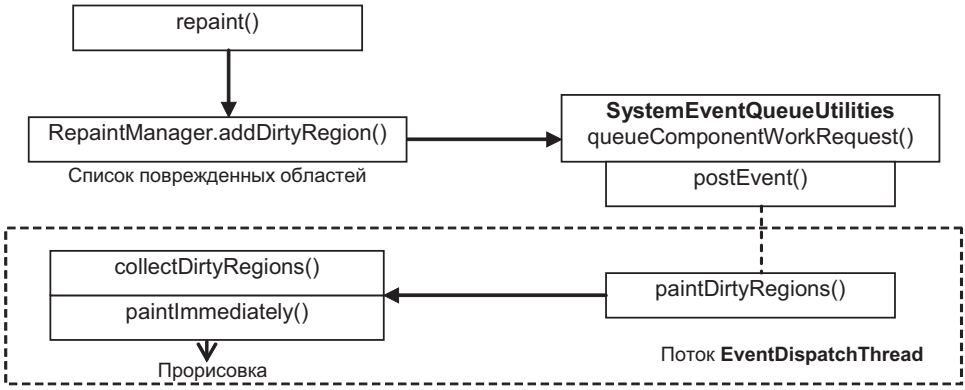
Если вспомнить как мы рассматривали программную перерисовку в AWT (метод repaint()), то легко видеть что это по сути постановка события на перерисовку (PaintEvent) в очередь событий интерфейса. Именно эта простая идея подтолкнула создателей Swing включить оптимизацию и здесь, на этот раз применив принцип пакетной обработки.

Глядя на очередь, очевидно, что каждое событие из очереди выполняется некоторое время, и пока дело дойдет до нашего события на перерисовку PaintEvent, в очереди могут появиться новые события такого же типа. Весьма вероятно, что они относятся не только к тому же окну, в котором была запрошена перерисовка в первый раз, но и даже к одному и тому же компоненту, и возможно, последующие события делают первые и вовсе ненужными, потому что полностью зарисуют то, что нарисуют первые. Вообразите активную прокрутку большой таблицы — подобный процесс будет генерировать огромное количество вызовов метода repaint(), и следующие вызовы уже будут перерисовывать области, которые должны бы были нарисованы первыми вызовами. Определенно здесь можно оптимизировать процесс и избавиться от лишнего рисования.

Метод repaint() переопределен в классе JComponent и вместо прямой постановки события PaintEvent в очередь событий просит нашего старого знакомого RepaintManager добавить область, которую нужно перерисовать, в список «загрязненных» (*dirty region).* «Загрязненные» области нужно перерисовать, как только до них дойдут руки потока рассылки событий.

RepaintManager объединяет все приходящие запросы на прорисовку «загрязненных» областей в один пакет, ставя в очередь событий особое событие, которое при срабатывании разом перерисует все накопившиеся (за время ожидания исполнения этого особого события, ведь очередь событий может быть занята) «грязные» области. Таким образом, мы избавляемся от избыточных событий. В дополнение к этому, в классе RepaintManager области для прорисовки оптимизируются — они объединяются в одну область для компонента, области, которые закрыты другими компонентами или более не видны на экране, выбрасываются из списка прорисовки и так далее.

Следующая схема прекрасно иллюстрирует, что происходит и какие классы участвуют в процессе:




Вспомогательный класс SystemEventQueueUtilities ставит то самое особое событие (его имя ComponentWorkRequest) в очередь, используя метод для постановки postEvent(). События PaintEvent в Swing вообще не применяются. Как только поток рассылки дохо-

дит до данного события, оно выполняется и вызывает метод все того же RepaintManager с названием paintDirtyRegions(). Опуская детали реализации, мы приходим к тому, что вызывается метод, определенный в базовом классе JComponent под названием paintImmediately(). Ну а в нем все уже совсем просто. В итоге методом getGraphics() создается графический объект и вызывается хорошо знакомый нам метод paint(). Работу этого метода в Swing мы подробно изучили чуть ранее, и все выполняется по той самой диаграмме, что мы составили. В результате компонент Swing перерисовывается, причем лишь один раз за определенный промежуток времени, и максимально оптимизировано.

## Рисование в Swing: резюме

Подводя итоги всей системы рисования Swing, можно определить следующие самые важные выводы:

- Метод paint() в Swing является частью внутренней системы и выполняет важную работу. Переопределять его для рисования компонента Swing не следует.
- Для рисования компонента или просто графики применяется метод paintComponent().
- При рисовании в Swing необходимо учитывать свойство непрозрачности opaque. Если компонент непрозрачен, необходимо закрашивать его фон или не оставлять непрорисованных областей. Делается это либо вручную, либо вызовом super.paintComponent(), чтобы фон закрасил UI-представитель. Однако, второй способ не работает для компонентов, унаследованных напрямую от JComponent. Если не выполнить данное условие, мусор на экране неизбежен.
- Перерисовка repaint() в Swing выполняется пакетами и оптимизируется. Если вас это не устраивает, придется переопределить или метод repaint(), или написать свой вариант класса RepaintManager. Впрочем, необходимости в этом практически не возникает.

## Рисование «готовых» компонентов

Разбирая тонкости систем рисования библиотеки Swing и ее основы AWT, мы рассматривали все возникающие вопросы с точки зрения рисования с нуля, в пределах одного компонента, как это обычно и бывает. Однако легко себе представить и другую ситуацию — стандартные компоненты Swing весьма функциональны и полезны, и у вас может возникнуть желание «прорисовать» какой-либо из них в своем собственном компоненте. К примеру, надписи JLabel умеют отображать многострочный текст формата HTML и изображения, представьте, что вам пришлось бы заниматься тем же самым самостоятельно при создании любого нового компонента. Интересно, что сложные компоненты Swing, такие как таблицы JTable, используют те же самые надписи для отображения своих ячеек или текста, так что идея неплоха.

При работе с обычными легковесными компонентами AWT представить себе процесс их прорисовки в рамках другого компонента легко —нужно всего лишь вызвать метод paint(), предварительно задав размер компонента и возможно поменяв какие-то графические параметры объекта Graphics. Компонент никогда не знает, кто вызывает его прорисовку, и должен выполнять ее независимо от этого. Однако в Swing картина немного другая.

Если вспомнить процесс прорисовки компонентов Swing, то легко увидеть, что все действия идут через объект RepaintManager, а сам компонент, если он находится в другом компоненте, «подневолен» и настроен рисовать себя через буфер своего

компонента-предка. Если мы создадим новый компонент Swing, не добавляя его на экран, он увидит, что предков у него нет, и решит, что является первым компонентом в иерархии компонентов, пытаясь нарисовать себя через объект RepaintManager и экранный буфер. Так как компонент на самом деле на экране не находится, то и содержимое буфера скопировать будет некуда и прорисовка не получится. В этом случае компоненты Swing приходится настраивать дополнительно: отключать для таких компонентов двойную буферизацию. Тогда они рисуют себя напрямую и появляются на экране.

Интересно, что в свежих версиях JDK отключение буферизации не требуется, так как она и так отключена и заменена на буферизацию системного уровня. Однако в том случае, если системная буферизация будет недоступна, стоит отключать буферизацию компонентов Swing, которые мы хотим рисовать на экране, но не добавлять в контейнеры.

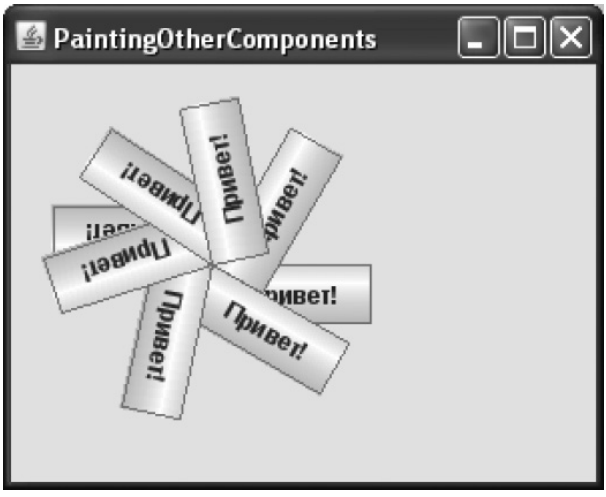
Рассмотрим небольшой пример с прорисовкой компонентов:

```
// PaintingOtherComponents.java
// Прорисовка других компонентов как изображений
import javax.swing.*;
import java.awt.*;

public class PaintingOtherComponents extends JFrame {
  public PaintingOtherComponents() {
    super("PaintingOtherComponents");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    add(new CustomPaintComponent());
    setSize(400, 300);
    setVisible(true);
  }
  class CustomPaintComponent extends JPanel {
    // кнопка для прорисовки
    private JButton button = new JButton("Привет!");
    // метод для рисования в Swing
    protected void paintComponent(Graphics g) {
      // необходимо вызвать для обработки свойства opaque
      super.paintComponent(g);
      // рисуем кнопки
      Graphics2D g2 = (Graphics2D) g;
      button.setSize(80, 30);
      // отключение двойной буферизации — не всегда нужно
      button.setDoubleBuffered(false);
      // переместим позицию рисования
      g2.translate(100, 100);
      for (int i=1; i<=8; i++) {
        // кручение кнопки по кругу
        g2.rotate(2*Math.PI / i);
```

```
            button.paint(g);
          }
        }
      }

      public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
          public void run() { new PaintingOtherComponents(); } });
      }
    }
```

В примере мы наследуем от окна с рамкой JFrame (подробности об окнах вы всегда сможете узнать в главе 6) и добавляем в его центр свой компонент Swing, унаследованный от панели JPanel. Если вспомнить все, что мы говорили о рисовании в Swing, то для рисования нам необходимо переопределить метод paintComponent(), вызвать метод родительского класса, чтобы тот позаботился о свойстве непрозрачности, а дальше делать с графическим объектом Graphics все, на что хватит фантазии.

В примере мы будем рисовать кнопку JButton с помощью стандартных средств AWT для рисования, поворачивая ее вокруг своей начальной точки на полный круг. Для этого предназначен метод rotate(), которому необходимо задать угол поворота в радианах. Кнопка создается одна, а нарисовать ее на экране можно бесчисленное количество раз. Нужно лишь не забыть задать ей корректный размер методом setSize() и, настроив по вкусу свойства рисования в объекте Graphics, вызвать метод paint(). Обратите внимание, что сама кнопка понятия не имеет о том, что ее «крутят», за все отвечает объект Graphics. Как правило, отключение двойной буферизации в свежих версиях JDK не требуется, вы можете убрать его и запустить пример заново в качестве простого домашнего упражнения.




Таким образом, все стандартные компоненты Swing к вашим услугам, если вы захотите нарисовать их в виде статичных изображений в своем компоненте. Интересно, что в Swing есть и другой, еще более эффективный способ рисовать компоненты. Это вспомогательный контейнер CellRendererPane, в который можно добавить компонент,

а нарисовать уже этот вспомогательный контейнер. Вдобавок к стандартному рисованию этот контейнер к тому же отключает всю проверку корректности и дополнительную перерисовку, которую могут производить сложные компоненты Swing. Если компонент рисуется часто, стоит предпочесть именно этот способ. Мы еще не раз вспомним про него, когда будем обсуждать *отображающие объекты* (renderer) для сложных компонентов Swing.

## *Рисование вне рамок*

Рассматривая систему рисования Swing, мы постоянно говорим об одном компоненте и рисовании в его пределах. Тем не менее, зачастую приходится задумываться о рисовании над многими компонентами сразу, а то и над всем интерфейсом приложения. Особенно это верно для насыщенных графикой и анимацией интерактивных пользовательских интерфейсов. Классический подход для таких задач не слишком подходит: мы увидели, что система рисования Swing тщательно оптимизирована, а переопределять рисующий метод главного окна приложения чревато нарушениями в стандартной системе и сложностями реализации.

Специального решения на этот случай в Swing не предусмотрено, однако мы можем воспользоваться побочными эффектами реализации библиотеки, применяя особые компоненты, такие как многослойные (layered pane) и прозрачные панели (glass pane), существующие в каждом окне Swing-приложений или апплете. В главе 6 мы рассмотрим их вместе со способами применения для создания эффектной графики .

## *RepaintManager как прикладной инструмент*

Подводя итоги рассмотрения механизмов рисования Swing, мы видим, какую серьезную роль играет в них класс RepaintManager. Весь процесс рисования компонентов Swing, благодаря особой реализации базового класса JComponent, проходит через его двойную буферизацию, он также полностью заведует программной перерисовкой компонентов, оптимизируя ее. С этой точки зрения он всего лишь внутренний «винтик» Swing, однако с учитом того, что все «нити» при рисовании сходятся в одно место, мы получаем уникальную возможность вмешаться в рабочий процесс рисования, так как можем заменить экземпляр класса RepaintManager на свой собственный, методом setCurrentManager().

Прежде чем задуматься о собственной реализации, важно узнать, какие дополнительные реализации этого класса уже существуют. Прежде всего, это инструменты, которые мы обсуждали в предыдущей главе, для слежения за тем, чтобы все рисование в Swing велось из потока рассылки событий. Обращение к экземпляру класса RepaintManager из другого потока сразу же обозначает ошибку программиста и необходимость вынести вызывающий код в поток рассылки событий. Настолько же интересно применяет свой экземпляр класса RepaintManager библиотека вспомогательных инструментов SwingX. В ней существует панель, позволяющая настроить уровень прозрачности, как свой, так и всех содержащихся в ней компонентов. Однако, как мы знаем, при необходимости компоненты сами перерисовывают друг друга, и со стороны никак нельзя «заставить» их рисовать себя полупрозрачными. Решением является особый объект RepaintManager, который при запросе на перерисовку компонентов, находящихся в прозрачной панели, направляет их самой полупрозрачной панели так, чтобы она смогла настроить все параметры рисования и только потом нарисовать потомков.

Таким образом, можно использовать свой объект RepaintManager для контроля каких-либо критичных параметров в процессе рисования или же определять, какой

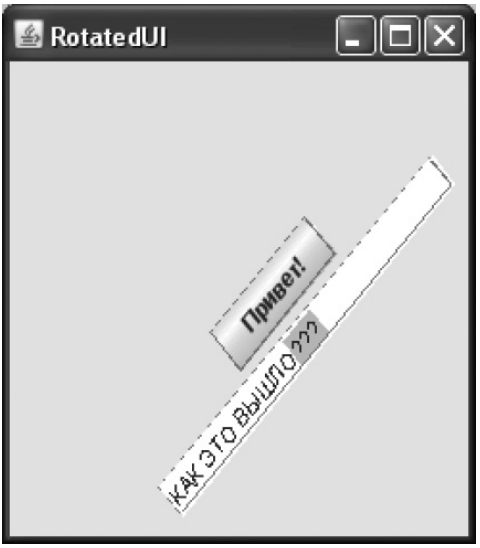
компонент будет перерисовываться, вне зависимости от того, кто запросил перерисовку. Чтобы убедиться в том, что все сказанное не пустые слова, напишем маленький, но интересный, пример. В нем мы попытаемся создать панель, которая будет крутить и растягивать все компоненты, попадающие в нее. Чтобы компоненты не могли рисовать себя сами нормальным образом, необходимо будет перехватывать их запросы на перерисовку и вместо этого направлять их нашей панели. Вот что получится:

```
// RotatedUI.java
// Кручение и верчение стандартных компонентов
import javax.swing.*;
import java.awt.*;

public class RotatedUI extends JFrame {
  public RotatedUI() {
    super("RotatedUI");
    // выход при закрытии окна
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // добавляем особую панель
    RotatingPanel rp = new RotatingPanel();
    add(rp);
    // добавляем в панель компоненты
    rp.add(new JButton("Привет!"));
    rp.add(new JTextField(20));
    // устанавливаем свой RepaintManager
    RepaintManager.setCurrentManager(
        new RotatingRepaintManager());
    // выводим окно на экран
    setSize(200, 300);
    setVisible(true);
  }
  // компонент, который поворачивает всех потомков
  class RotatingPanel extends JPanel {
    // отвечает за прорисовку потомков
    protected void paintChildren(Graphics g) {
      Graphics2D g2 = (Graphics2D) g;
      g2.translate(50, 200);
      // поворот на 45 градусов
      g2.rotate(-Math.PI/4);
      // небольшое растяжение
      g2.shear(-0.1, -0.1);
      // обычное рисование предков
      super.paintChildren(g);
```

```
    }
  }
  // особый тип RepaintManager
  class RotatingRepaintManager extends RepaintManager {
    // все запросы на перерисовку попадают сюда
    public void addDirtyRegion(JComponent c,
        int x, int y, int w, int h) {
      // ищем нужного предка
      Container parent = c;
      while (! (parent instanceof RotatingPanel)) {
        parent = parent.getParent();
        if ( parent == null ) {
          // мы не нашли нашего предка, сброс
          parent = c;
          break;
        }
      }
      // перерисовываем весь компонент полностью
      super.addDirtyRegion((JComponent) parent,
          0, 0, parent.getWidth(), parent.getHeight());
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() { new RotatedUI(); } });
  }
}
```

В примере мы привычно создаем окно с рамкой, и добавляем в его центр наш особый компонент, «крутящую» панель. Мы переопределили метод paintChildren(), который, как было выяснено в данной главе, отвечает за прорисовку всех предков, то есть за прорисовку всех компонентов, которые в него будут добавлены. В этом методе мы настраиваем графический объект Graphics на небольшое кручение и растяжение (все это относится к стандартным средствам Java2D), а дальше просто просим базовые механизмы Swing все за нас нарисовать. В особую панель мы добавляем кнопку и текстовое поле.

Если остановиться на этом и запустить пример, то поначалу интерфейс будет прокручен и растянут, однако любое обновление кнопки или поля приведет к тому, что они будут рисовать себя привычным образом. Замена стандартного объекта RepaintManager на собственный позволяет нам перехватывать желания кнопки и поля. Наш объект унаследован от стандартного и переопределяет метод addDirtyRegion(), который вызывается при *любом* запросе любого компонента Swing на перерисовку. Мы смотрим, не находится ли компонент в нашей «крутящей» панели, и, если да, просто полностью перерисовываем ее, а если нет, позволяем рисовать оригинальному «просителю». Производительность перерисовки при таком грубом подходе конечно упадет, но это просто пример. Запустив его, вы убедитесь, что интерфейс выглядит более чем авангардно.




Компоненты также будут реагировать на клавиатуру, печать клавиш, будет мерцание курсора — все это можно незамедлительно видеть растянутым и повернутым. Однако события от мыши не знают о наших манипуляциях с графикой и попрежнему будут приходить из оригинального местоположения компонентов в левом верхнем углу экрана. Впрочем, задача преобразования координат для мыши при поворотах интерфейса уже совсем другая.

Как мы видим, власть у RepaintManager имеется порядочная. Однако не стоит забывать, что на все Swing-приложение имеется лишь один объект RepaintManager, и он может быть уже заменен такой библиотекой, как SwingX или даже сторонним внешним видом. Использовать свой собственный объект RepaintManager стоит в случае крайней необходимости и при этом тщательно проверять, в полном ли объеме работают все дополнительные библиотеки, компоненты и инструменты. Стандартные же компоненты Swing без труда переносят замену объекта RepaintManager.

## *Отладка графики*

Иногда, при создании собственных компонентов или при рисовании анимации приходится тщательно следить, как и в каком порядке происходит прорисовка компонента. Это помогает найти трудно обнаружимые ошибки. Обычные методы отладки здесь не помогают, потому что результат графических операций мгновенно выводится на экран. Ставить задержки после каждой графической операции вручную или выводить диагностические сообщения может быть утомительно и не всегда возможно. Swing позволяет справиться с такими ситуациями с меньшими усилиями, предоставляя для отладки графических операций класс DebugGraphics. Он позволяет отлаживать графику не только в своих компонентах, но и в любых компонентах Swing.

Класс DebugGraphics — это тонкая оболочка вокруг стандартного класса рисования Graphics. Всю работу по рисованию, как и прежде, выполняет Graphics, а DebugGraphics по вашему желанию замедляет графические операции и выделяет их специальным цветом (который потом стирается) или выводит о них диагностические сообщения[6]. Можно использовать DebugGraphics и напрямую, создав объект этого класса и передав

[6] На языке шаблонов проектирования класс DebugGraphics представляет собой не что иное, как типичный *декоратор*.

в его конструктор ссылку на стандартный объект Graphics, после чего производить рисование полученным объектом. Указать при этом, какой способ отладки графики вы предпочитаете, позволяет метод setDebugGraphicsOptions(). Однако базовый класс JComponent имеет встроенную поддержку класса DebugGraphics, и, чтобы включить для компонента Swing режим отладки графики, надо всего лишь указать способ отладки, вызвав тот же метод setDebugGraphicsOptions(), но на этот раз уже для самого компонента. После этого компонент и все его потомки будут использовать для рисования объект DebugGraphics. В качестве параметра методу setDebugGraphicsOptions() передается целое число, представляющее собой одну из констант, определенных в классе DebugGraphics, или комбинацию этих констант, составленную с помощью операции логического «ИЛИ» (табл. 4.1).

**Таблица 4.1.** Параметры метода setDebugGraphicsOptions()

| Константа из класса DebugGraphics | Действие |
|---|---|
| NONE_OPTION | Отключает режим отладки графики для используемого объекта DebugGraphics |
| LOG_OPTION | Позволяет выводить диагностические сообщения обо всех производимых графических операциях (то есть обо всех вызываемых методах). По умолчанию сообщения выводятся в стандартный поток вывода System.out, изменить поток вывода сообщений позволяет статический метод класса DebugGraphics setLogStream() |
| FLASH_OPTION | Установка этого флага приводит к тому, что все появляющееся на экране рисуется в замедленном режиме и выделяется особым цветом, так что вы своими глазами можете проследить, как и где происходит рисование. Настроить данный режим позволяют три статических метода класса DebugGraphics: метод setFlashColor() устанавливает цвет, которым выделяются все операции рисования, метод setFlashCount() позволяет задать количество «мерцаний» при рисовании, наконец, метод setFlashTime() используется для того, чтобы указать задержку после каждой операции рисования. Имейте в виду, данный режим отладки доступен только при отключении двойной буферизации |
| BUFFERED_OPTION | В данном режиме операции рисования будут дополнительно выводиться в отдельное окно, создаваемое классом DebugGraphics. При этом учитываются остальные режимы отладки — они будут действовать в созданном окне |

А теперь рассмотрим небольшой пример, который наглядно продемонстрирует возможности отладки графики:

```
// DebugPainting.java
// Демонстрация возможностей отладки графики в Swing
import java.awt.*;
import javax.swing.*;

public class DebugPainting extends JFrame {
  public DebugPainting() {
    super("DebugPainting");
```

```
    // выход при закрытии окна
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // добавляем рисующий компонент
    PaintingComponent pc = new PaintingComponent();
    add(pc);
    // включаем для него отладку графики
    RepaintManager.currentManager(pc).
        setDoubleBufferingEnabled(false);
    pc.setDebugGraphicsOptions(DebugGraphics.LOG_OPTION
        | DebugGraphics.FLASH_OPTION);
    DebugGraphics.setFlashTime(50);
    DebugGraphics.setFlashCount(3);
    // выводим окно на экран
    setSize(200, 200);
    setVisible(true);
  }

  // компонент, который что-то рисует
  class PaintingComponent extends JPanel {
    public void paintComponent(Graphics g) {
      super.paintComponent(g);
      // три простые фигуры
      g.setColor(Color.orange);
      g.fillRect(10, 10, 100, 100);
      g.setColor(Color.green);
      g.drawOval(50, 50, 50, 50);
      g.setColor(Color.blue);
      g.fillOval(100, 20, 50, 50);
    }
  }

  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() { new DebugPainting(); } });
  }
}
```

В примере мы создаем небольшое окно, в которое добавляем собственный компонент, унаследованный от класса JPanel. Интересно, что если бы мы унаследовали компонент обычным образом, непосредственно от класса JComponent, система отладки графики работать бы с ним не стала. Связано это с тем, что у базового класса библиотеки нет собственного UI-представителя — ведь это абстрактный класс, его нельзя добавить в окно, и UI-представитель ему просто не нужен. В обычных программах это не причиняет неудобств, однако система отладки графики отказывается работать с компонентом, у которого нет

UI-представителя. Эта небольшая проблема имеет два решения: можно написать для своего компонента, даже для такого простого как наш, UI-представителя, который, используя ресурсы своего базового класса ComponentUI, реально ничего делать не будет, но мы еще не добрались до подробного обсуждения UI-представителей. Поэтому в примере применено второе решение: мы унаследовали компонент от класса JPanel, на котором тоже можно спокойно рисовать (сам он ничего не рисует), к тому же у него есть свой UI-представитель.

Далее мы, так как в примере будет использоваться отладка с условием FLASH_OPTION, полностью отключаем двойную буферизацию методом класса RepaintManager() (почти то же самое можно было бы сделать, вызвав метод setDoubleBuffered(false) для корневой панели нашего окна). Для нашего компонента мы включаем два режима отладки: вывод диагностических сообщений и выделение цветом, для этого константы класса DebugGraphics объединяются операцией логического «ИЛИ». Напоследок идет небольшая настройка режима выделения цветом (мы увеличиваем количество мерцаний и задержку, чтобы лучше видеть процесс рисования). Остается запустить приложение и посмотреть на результат.

Диагностические сообщения, выводимые в режиме отладки графики на консоль (хотя можно выводить их в любой поток вывода), имеют простой формат:

```
Graphics(N1-N2) Операция: Параметры
```

Здесь N1 — номер используемого в данный момент объекта DebugGraphics (каждый раз при создании объекта DebugGraphics он нумеруется), а N2 — код режима отладки. Значение N2 в нашем примере равно 3 — это именно то число, которое мы передаем в метод setDebugGraphicsOptions() (оно получается при логическом объединении констант). Оно дает возможность узнать, какие операции отладки используются в данный момент. Далее следует краткое текстовое описание графической операции (которое не слишком информативно и фактически повторяет название вызываемого метода) и список параметров, переданных вызванному методу.

Данный пример может быть полезен не только в качестве наглядного пособия по использованию встроенной системы отладки графики, но и как пример работы системы прорисовки Swing. После запуска закройте область нашего окна другим окном, потом откройте ее и посмотрите, как перерисовывается графика. Если была закрыта только часть окна приложения, то заново прорисуется только закрытая часть, хотя диагностические сообщения показывают, что вызываются все операции прорисовки компонента PaintingComponent. Механизмы Swing сами заботятся о прямоугольнике отсечения и максимально эффективном выводе на экран, а нам остается лишь нарисовать компонент и получить готовый результат. Отладку графики также удобно проводить для стандартных компонентов Swing, если с ними возникают затруднения, например, если они неправильно перекрываются или неверно располагаются в контейнере. В любом случае систему отладки графики, любезно предоставленную нам Swing, стоит «держать под рукой».

Отладка графики вдохновила создателей некоторых библиотек на встраивание в них на том или ином уровне подобных механизмов. К примеру, в популярном стороннем менеджере расположения MigLayout, о котором мы вкратце расскажем в главе 7, есть визуальная отладка, позволяющая отследить, где располагаются все компоненты и какое именно пространство им выдал менеджер расположения.

## *Резюме*

Внутренние механизмы системы рисования Swing, незаметно выполняющие для нас разнообразную «грязную» работу, довольно сложны и в основном скрыты в базовом классе библиотеки JComponent. Все компоненты библиотеки наследуют от него и поэтому эффективно рисовать в Swing можно лишь четко представляя, что при этом происходит внутри библиотеки. К тому же знание «внутренностей» системы открывает широкие возможности для отладки проблем и смелых экспериментов с графикой.

# Глава 5. Внутренние винтики Swing

Эта глава будет несколько отличаться от всех остальных глав книги, в которых мы рассматриваем конкретные вопросы, компоненты или способы работы с библиотекой. Здесь мы ближе взглянем на некоторые внутренние механизмы библиотеки Swing, которые обычно работают как часы, однако при возникновении более сложных ситуаций требуют нашей осведомленности в них. Во-первых, речь будет идти о проверке корректности («валидации») компонентов. Процесс это довольно простой, однако на практике иногда упорно «показывающий зубы», он имеет свою интерпретацию в классе JComponent. Мы рассмотрим, как проверка производится классически, и узнаем механику работы метода revalidate().

Далее мы разберемся с более тонкими способами работы с клавиатурой. Обработка событий от клавиатуры с помощью слушателей во многих ситуациях недостаточна, так как предназначена для конкретного компонента. Клавиатурные сокращения Swing, вкупе с системой передачи фокуса ввода, позволяют тоньше настроить работу с клавиатурой, и мы рассмотрим, как это делается и почему это так.

Эта глава не предназначена для новичков в Swing. Многое в Swing очень доступно и очень просто, и прежде чем переходить к деталям и мелким концепциям настолько мощной библиотеки, необходимо изучить ее основы, особенно способ рассылки событий и размещение компонентов в контейнерах, в том числе и высшего уровня. Если же вам будет недостаточно, можно прочитать и эту главу.

## *Проверка корректности компонентов*

Проверка корректности компонентов требуется для того, чтобы расположение компонентов в контейнере и их размеры соответствовали действительности. На самом деле, если вы прямо во время работы программы увеличите размер шрифта на кнопке, то прежнего размера кнопки (которого, как правило, как раз хватает для размещения надписи) уже будет недостаточно, и часть надписи окажется непонятно где.

Если компонент инициализируется перед выводом на экран, то последовательность действий четко определена — вы создаете его, задаете свойства компонента, в том числе те, что влияют на его внешний вид, к примеру размер шрифта и значков, а затем добавляете в некоторый контейнер. В контейнере работает менеджер расположения, который выясняет, какой размер желает иметь компонент, и выделяет ему пространство на экране, где его можно будет увидеть, как только приложение выведет контейнер высшего уровня на экран.

Однако, если контейнер уже на экране, а вы меняете свойство содержащегося в нем компонента, влияющее на его внешний вид, а самое главное, размеры, для обновления контейнера, его размеров и размеров всех соседей-компонентов необходимо провести проверку корректности («валидацию»), которая заново распределит пространство на экране и перерисует компоненты в их новом состоянии.

В AWT, в котором, как мы помним, компоненты лаконичны до крайности, каждый из них хранит булевский флаг valid, показывающий, находится ли сейчас компонент в корректном состоянии. Как только происходит нечто, нарушающее размеры компонента (к примеру, смена шрифта или текста), флаг устанавливается в false специальным методом invalidate(). Так как обычно компонент хранится в контейнере, а тот в другом контейнере, и так далее, изменение его размеров влияет на все содержащие его контейнеры.

Чтобы пометить этот факт, метод invalidate() вызывает точно такой же метод контейнера, в котором компонент хранится, и все идет по цепочке до самого верха.

Сам же процесс приведения к корректному виду осуществляется методом validate(). Для обычных компонентов AWT, таких как кнопки или надписи, он просто перерисовывает компонент. А вот для контейнеров все сложнее: метод заново вызовет менеджер расположения контейнера и перераспределит пространство между компонентами в контейнере и изменит их размеры согласно их новым желаниям, вызовет для всех компонентов в контейнере их метод validate(), а потом и перерисует сам контейнер. Так что, если вы хотели привести конкретный компонент к нужному размеру, вряд ли стоило вызывать для него validate(), если только вы уже не задали ему новый подходящий размер вручную — это будет равносильно его перерисовке. В общем же всегда вызывается метод validate() того контейнера, в котором находится измененный компонент. Стоит заметить, что validate() работает лишь тогда, когда компоненты уже выведены на экран.

Рассмотрим небольшой пример:

```
// Базовая валидация AWT — при изменении размеров
//  или других параметров остается вызвать validate()
import java.awt.*;

public class AWTValidateShow extends Frame {
  private static Button button;

  public AWTValidateShow() {
    setSize(400, 300);
    Panel contents = new Panel();
    button = new Button("Текст");
    Button button2 = new Button("Текст 2");
    contents.add(button);
    contents.add(button2);
    add(contents);
  }

  public static void main(String[] args)
      throws InterruptedException {
    new AWTValidateShow().setVisible(true);
    Thread.sleep(2000);
    button.setLabel("Очень длинный текст");
    // С этого момента размер поменялся — вызван invalidate()
    // можно вызывать validate() в контейнере
    Thread.sleep(2000);
    // будет заново расположен весь контейнер
    // и все его содержимое (кнопка)
    button.getParent().validate();
  }
}
```

В примере мы создаем окно и помещаем в него в панель с двумя кнопками. Обратите внимание, пример написан исключительно с использованием AWT, так что беспокоиться о многозадачности нам не придется — компоненты AWT синхронизированы, и мы можем напрямую вызывать из других потоков, отличных от потока рассылки событий. Маленький нюанс примера — окна AWT не умеют сами закрывать себя, а мы не написали слушателя окна, который заканчивал бы приложение при закрытии окна, так что приложение это придется заканчивать вручную, снимая задачу.

Когда у кнопки меняется надпись, она помечает себя и все содержащие ее контейнеры как некорректные. Нам же остается вызвать validate() для содержащего ее контейнера (мы получаем его методом getParent()), чтобы он получил данные о новом размере кнопки и заново расположил ее. В примере специально вставлены задержки — вы успеете увидеть, как выглядит некорректная кнопка. В качестве упражнения попробуйте вызвать validate() только для кнопки, чтобы убедиться, что это не поможет ей обрести корректный размер.

## Метод Swing: revalidate()

Одной из основных задач Swing является более быстрая и эффективная работа компонентов библиотеки, в сравнении с AWT. Проверка корректности не стала исключением. Прежде всего, компоненты Swing намного более сложны, и некоторые из них в любом случае «поглотят» любые изменения содержащихся в них компонентов. К примеру, это внутренние окна JInternalFrame (все, что меняется в окне, остается в его рамках) или панель прокрутки JScrollPane (изменение размеров содержимого лишь меняет полосы прокрутки, но не размер самой панели). Однако мы увидели, что при изменении компонентов некорректными становятся все компоненты, вплоть до контейнера высшего уровня — так устроен метод invalidate().

Чтобы каким-то образом помечать компоненты, на которых можно остановить проверку корректности, в базовом классе JComponent появился метод isValidateRoot(). Те компоненты, которые утверждают, что все изменения их содержимого не отразятся на их размерах (и значит, на размерах всего окружающего), вернут в этом методе true (к ним относится JScrollPane и корневая панель JRootPane, которая является основой любого окна в Swing). Такие компоненты мы называем корнем валидации. По умолчанию метод возвращает false.

Как мы уже убедились (в описании прорисовки), Swing старается всегда скомпоновать похожие события графической системы, что минимизировать издержки. Эта же схема применена и для проверки корректности, и делает это новый метод revalidate() из класса JComponent. Он не проводит мгновенной проверки корректности, вместо этого, он помечает компонент как некорректный в классе RepaintManager. Тот, в свою очередь, находит корень валидации для компонента, и помещает в очередь событий отложенное задание для окна, в котором этот корень валидации находится (все это происходит только в том случае, если компонент уже выведен на экран, до вывода на экран в проверке просто нет смысла). Когда очередь до этого события дойдет, в данном корне валидации и данном окне могут накопиться несколько компонентов, требующих проверки корректности, и выполнение проверки за один раз позволяет сократить издержки.

Само же задание выполняет проверку корректности уже знакомым нам методом validate() из AWT. Он вызывается для корня валидации, а это значит, что все находящиеся в нем компоненты будут заново расположены с учетом их новых пожеланий по размерам. Компоненты, которые находятся в контейнерах «выше» корня валидации, таким образом проверяться не будут, что сэкономит драгоценные секунды.

Теперь понятно, что эффект revalidate() коренным образом отличается от validate(). Если validate() нужно вызывать с умом, выбирая тот контейнер, проверка которого заново расположит все нужные нам компоненты, то revalidate() совершенно безразличен к тому,

откуда его вызывают. Он в любом случае найдет корень валидации и вызовет проверку именно для него, что гарантирует нам нужный результат. Более того, большая часть компонентов Swing вызывает revalidate() при смене свойств, меняющих внешний вид и размер, автоматически, что совсем избавляет нас от раздумий, свойственных системе проверки корректности в AWT.

Рассмотрим пример:

```
// Валидация Swing — большинство компонентов
// позаботятся о себе сами. В остальном метод revalidate()
// позволяет не задумываться о деталях
import javax.swing.*;

public class SwingValidateShow extends JFrame {
  private static JButton button, newButton;

  public SwingValidateShow() {
    setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    setSize(400, 300);
    JPanel contents = new JPanel();
    button = new JButton("Текст");
    JButton button2 = new JButton("Текст 2");
    contents.add(button);
    contents.add(button2);
    add(contents);
  }

  public static void main(String[] args)
      throws InterruptedException {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new SwingValidateShow().setVisible(true);
      }
    });
    Thread.sleep(2000);
    // Кнопка при смене параметра сама вызовет
    // revalidate() и мы сразу же увидим изменения
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        button.setText("Очень длинный текст");
      }
    });
    // при добавлении в контейнер revalidate()
    // автоматически не вызывается
```

```
      SwingUtilities.invokeLater(new Runnable() {
        public void run() {
          newButton = new JButton("Новичок");
          button.getParent().add(newButton);
        }
      });
      Thread.sleep(2000);
      // revalidate() может быть вызван из любого потока
      newButton.revalidate();
    }
  }
```

В этом примере мы снова используем Swing, а значит должны работать со всеми компонентами из потока рассылки событий. Как и в примере с AWT, создается окно с двумя кнопками. Однако, Swing тут же доказывает свое удобство, автоматически проверяя корректность кнопки при смене ее надписи методом setText(). Затем мы добавляем в панель с кнопками еще одну кнопку, что не приведет к автоматической проверке. Чтобы ее провести, мы вызываем revalidate() для самой кнопки. Это найдет корень валидации (корневую панель нашего окна) и заново расположит все компоненты в ней. Для какого компонента вызывается revalidate(), не так уж и важно.

Сам метод revalidate() может быть безопасно вызван из любого потока — он автоматически заботится о том, чтобы дальнейшие действия происходили в потоке рассылке событий.

На деле пример не так прост, как кажется. Если вам повезет, вы сможете увидеть как новая кнопка появляется в окне сразу же, еще до вызова revalidate(). Секрет тут прост: в таких случаях мы успеваем добавить ее еще до того, как выполнилась задача проверки корректности, запущенная методом setText(), и она «прихватывает» с собой и новый компонент в панели.

Интересно, что удобство Swing, благодаря которому компоненты сами вызывают revalidate() при смене своих размеров, является чуть ли не главной причиной того, что даже до вывода приложения на экран с компонентами можно работать только из потока рассылки событий. На самом деле, revalidate() помещает задание в очередь событий (это не так, только если компонент пока не добавлен в контейнер), а это автоматически запускает поток рассылки. Насколько это оправдывает себя, вопрос спорный. Иногда возможность заранее подготовить части сложного интерфейса в фоновом режиме бывает очень кстати, но Swing это сделать не позволяет.

## Проверка корректности в Swing: резюме

Соединяя вместе правила работы классического метода validate() и более изощренного revalidate(), мы получаем следующие выводы:

revalidate() эффективнее и пытается объединить похожие события в пакет. Однако вы не можете рассчитывать на мгновенный эффект, и должны быть уверены в том, что среди контейнеров, в которых находится ваш компонент, найдется корень валидации (isValidateRoot() возвращает true). В противном случае проверки не произойдет.

validate() проводит проверку немедленно, но только среди потомков контейнера, для которого вы его вызываете. Контейнер нужно выбрать правильно, чтобы привести к корректному виду все нужные вам компоненты. Для Swing этот метод можно вызывать только из потока рассылки событий.

При разработке собственных сложных компонентов, не меняющих свои размеры (к примеру, легковесных плавающих диалогов), вам может пригодится метод

isValidateRoot(). Правильный выбор корня валидации увеличит производительность, не «пуская» валидацию за пределы нужного контейнера. Это важно в Swing, если помнить что практически любое изменение моделей или свойств стандартных компонентов автоматически запускает revalidate().

Оба этих метода сработают, только если компонент, который вы хотите привести к корректному виду, уже находится на экране и виден. Действительно, для невидимых компонентов нужные размеры будут заданы в момент их вывода на экран.

## *Клавиатурные сокращения*

При создании Swing одной из приоритетных целей было создание продуманной и универсальной системы *клавиатурных сокращений* (keyboard bindings, или keyboard shortcuts). В большинстве современных оконных систем клавиатурные сокращения используются весьма широко, и не зря: возможность совершить какое-либо частое или сложное действие путем нажатия двух–трех клавиш очень удобно и ценится пользователями, потому что всегда намного быстрее работы мышью.

Поддержка клавиатурных сокращений обеспечивается базовым классом библиотеки JComponent. Начинается все с метода processKeyEvent(), основной целью которого является сортировка событий от клавиатуры и рассылка их соответствующим методам зарегистрированных слушателей событий от клавиатуры (мы обсуждали его в главе 2). Класс JComponent бесцеремонно вмешивается в этот процесс, переопределяя данный метод и заменяя стандартный механизм своей цепочкой обработки событий от клавиатуры. Но прежде чем выяснять, как это происходит, познакомимся с классом KeyStroke и картами входных событий и команд, принимающими в этом непосредственное участие.

### Класс KeyStroke

Класс KeyStroke инкапсулирует клавиатурное сокращение, позволяя указать, в случае какого события от клавиатуры оно возникает. Ничего особого этот класс не делает, он просто хранит код символа (или сам символ), клавиша которого нажимается для активизации клавиатурного сокращения, и набор модификаторов, уточняющих условие срабатывания клавиатурного сокращения, например, удержание управляющей клавиши Ctrl или Alt. Также можно указать, какой именно тип нажатия клавиши приведет к срабатыванию сокращения: это может быть простое нажатие клавиши (используемое чаще всего), печать символа (пригодное только для клавиш, отвечающих за печатные символы, а также для клавиш Enter и Esc) или отпускание клавиши, когда сокращение срабатывает только при отпускании заданного сочетания клавиш (которые предварительно были нажаты и могли удерживаться).

Создать экземпляр класса KeyStroke напрямую у вас не получится, потому что все его конструкторы объявлены закрытыми (private). Создатели этого класса решили таким образом встроить в него возможность кэширования создаваемых объектов KeyStroke, предвидя ситуацию, когда в приложении может быть задействовано значительное количество одинаковых клавиатурных сокращений (например, при использовании в приложении нескольких текстовых компонентов, у каждого из которых имеются одинаковые сокращения для частых действий, таких как выделение текста). Создать объект класса KeyStroke позволяют определенные в нем перегруженные версии статического метода getKeyStroke()[1]. Когда вы вызываете этот метод для создания объекта KeyStroke, он прежде всего проверяет, не содержится ли объект с такой же информацией в кэше, и при положительном ответе возвращает вам кэшированный объект. Так как объекты KeyStroke всего

[1] Одновременно с выполнением своих основных обязанностей класс KeyStroke еще и функционирует как фабрика по созданию своих экземпляров.

лишь хранят информацию о клавиатурных сокращениях и сами не выполняют никаких действий, использование одного и того же объекта в разных местах программы безопасно и, кроме того, позволяет реально экономить память и время на инициализацию.

## Карты входных событий и команд

Клавиатурные сокращения хранятся в специальных картах (таблицах, map), позволяющих строить иерархию клавиатурных сокращений от компонентов-родителей к компонентам-потомкам и проще манипулировать ими. Такие карты дают возможность легко изменить реакцию компонента на любые воздействия с клавиатуры и легко настраиваются.

Первая такая карта, хранящаяся в классе JComponent, называется *картой входных событий* (input map). Это экземпляр класса InputMap; он представляет собой отображение некоторых входных событий, которые воспринимает компонент, на объекты, отвечающие за действие компонента в случае возникновения этих событий. В идеале в такой карте должны храниться все входные события, воспринимаемые компонентом, но пока Swing использует ее лишь для хранения клавиатурных сокращений, поддерживаемых компонентом. Все клавиатурные сокращения, которые должен обрабатывать компонент, необходимо поместить в эту карту, для этого в классе InputMap определен метод put(KeyStroke, Object). В качестве второго параметра метода чаще всего указывают строку, идентифицирующую действие, которое должно вызывать помещаемое в карту клавиатурное сокращение (впрочем, вторым параметром может быть любой объект). Новое клавиатурное сокращение обычно добавляют в уже имеющуюся в компоненте карту входных событий, которую можно получить методом getInputMap(). Однако реакцию компонента на действия пользователя с клавиатуры можно и полностью изменить, установив свою карту методом setInputMap(). Именно так поступают UI-представители при настройке компонента, это позволяет им «одним махом» придать компоненту поведение, свойственное ему на той платформе, которую они представляют.

На самом деле у компонентов не одна карта входных событий, а целых три. Каждая их них хранит клавиатурные сокращения, поддерживаемые компонентом в одном из трех состояний (табл. 5.1).

**Таблица 5.1.** Состояния компонента при обработке клавиатурных сокращений

| Состояние | Описание |
|---|---|
| WHEN_FOCUSED | Компонент обладает фокусом ввода, то есть все события прежде всего идут к нему, а затем уже к родительским компонентам |
| WHEN_ANCESTOR_OF_FOCUSED_COMPONENT | Фокусом ввода обладает один из потомков компонента. Ни один из потомков событие от клавиатуры не обработал, и оно предлагается самому компоненту |
| WHEN_IN_FOCUSED_WINDOW | Событие пришло к какому-то компоненту, находящемуся в том же окне, что и наш компонент. Тот компонент и его предки событие не обработали, и оно по очереди предлагается всем находящимся в окне компонентам Swing |

Получить карту входных событий для определенного состояния позволяет все тот же метод getInputMap(), точнее, его перегруженная версия, которой необходимо

передать константу, идентифицирующую это состояние. Вызов этого метода без параметров возвращает карту для состояния WHEN_FOCUSED. Таким образом вы можете зарегистрировать клавиатурные сокращения для любого состояния вашего компонента. Чаще других используются состояния WHEN_FOCUSED (когда сокращение срабатывает лишь в том случае, если компонент обладает фокусом ввода: к примеру, выделение текста в текстовом поле имеет смысл только в этом состоянии) и WHEN_IN_FOCUSED_WINDOW (когда для инициирования какого-либо глобального действия в программе сокращение должно сработать, в каком бы месте вашего окна пользователь его не нажал).

Самым интересным свойством карты входных событий является ее свойство поддерживать предков. В классе InputMap определен метод setParent(), который позволяет задать для карты ее предка. Означает это следующее: если поиск в карте клавиатурного сокращения результатов не дал, карта просит своего предка (если он есть) провести поиск того же сокращения, у предка в свою очередь может быть свой предок, и поиск клавиатурного сокращения происходит по всей иерархии карт. Это свойство полезно, когда вы не хотите полностью заменять карту входных событий какого-либо компонента, а вместо этого собираетесь всего лишь немного изменить ее, возможно даже на небольшой промежуток времени. Тогда имеющуюся карту можно сделать предком вашей новой карты, а затем при необходимости легко восстановить ее. К примеру, сложным текстовым компонентам в качестве основы проще использовать карты простых текстовых компонентов, потому что простые действия с текстом для них не меняются.

Карта второго типа, хранящаяся в классе JComponent, называется *картой команд* (action map). Это отображение некоторых объектов-ключей на классы, реализующие интерфейс Action (унаследованный от интерфейса ActionListener). Интерфейс Action предназначен для объектов, которые выполняют какое-то действие в компоненте. Мы подробнее обсудим этот интерфейс в главе 9, сейчас важно знать лишь то, что в нем есть метод actionPerformed(), который и вызывается для выполнения команды. Карты команд, реализованные классом ActionMap, содержат все команды, поддерживаемые компонентом. Чаще всего в карте команд в качестве ключей хранятся те же строки, что и в карте входных событий, и таким образом карты входных событий и карты команд связывают сообщения от клавиатуры и действия, которые они совершают. Аналогично картам входных событий, карты команд могут иметь предков. Для получения карты команд используется пара методов get/set.

«Почему же клавиатурные сокращения не связаны с командами напрямую, в одной карте?» — спросите вы. Здесь несколько причин. Во-первых, клавиатурные сокращения различны для трех состояний компонента, в то время как в карте команд просто хранятся все поддерживаемые компонентом команды. Во-вторых, так проще изменять поведение компонента: вы можете заменить клавиатурное сокращение, не изменяя команды, которое оно выполняет, а можете изменить команду, оставив клавиатурное сокращение тем же самым. Ну и, в-третьих, как мы уже отмечали, карты InputMap и ActionMap позволяют использовать предков, что может быть очень полезным. Не стоит забывать и о том, что у раздельных карт есть «запас прочности»: в будущем Swing может использовать их не только для клавиатурных сокращений.

## Методы поддержки клавиатурных сокращений

Итак, как уже отмечалось, поддержка клавиатурных сокращений обеспечивается базовым классом библиотеки JComponent, который переопределяет метод processKeyEvent() и заменяет стандартный механизм своей цепочкой обработки событий от клавиатуры. Действие происходит следующим образом: приходит событие от клавиатуры. Система обработки событий AWT вызывает для текущего (обладающего фокусом ввода) компонента метод processKeyEvent().

Если дело происходит в компоненте библиотеки Swing, вызывается метод processKeyEvent() класса JComponent. Прежде всего он вызывает базовую версию метода super.processKeyEvent(), которая распределяет события по слушателям (если, конечно, таковые были зарегистрированы для компонента). После этого JComponent смотрит, не было ли событие полностью обработано (то есть не указал ли на это какой-либо из слушателей, вызвав метод consume()). Если событие обработано, работа этого события от клавиатуры завершается. Таким образом, слушатели имеют приоритет на обработку события. Кстати, все события от клавиатуры всегда добираются до метода processKeyEvent() класса JComponent, даже если в компоненте не были зарегистрированы слушатели, а значит, события соответствующего типа не добавлены в маску компонента (маскирование событий мы обсуждали в главе 3). Следит за этим конструктор класса JComponent, в котором в маску компонента добавляются события от клавиатуры (методом enableEvents()).

Далее, если ни слушатели, ни сам компонент так и не проявили интереса к событию, приходит пора проверить, не является ли это событие клавиатурным сокращением, зарегистрированным в компоненте. Для этого прежде всего запрашивается специальный внутренний класс KeyboardState, определенный в классе JComponent. Функция его проста — он отслеживает нажатие любых сочетаний клавиш и позволяет избежать срабатывания некоторого сочетания клавиш в том случае, если оно было нажато в другом окне или в другом приложении, а отпущено уже в нашем приложении. Обслуживаются лишь те сочетания клавиш, которые нажимаются пользователем в окнах нашего приложения. Если никаких препятствий для обслуживания сочетания клавиш не выявлено, управление передается методу processKeyBindings().

Метод processKeyBindings() приступает непосредственно к поиску зарегистрированных в компоненте клавиатурных сокращений и проверке того, не соответствует ли пришедшее от клавиатуры событие одному из этих сокращений. Помогают ему в этом класс KeyStroke, а также карты входных событий и команд.

Метод processKeyBinding() проверяет, не зарегистрировано ли в карте входных событий для состояния WHEN_FOCUSED пришедшее клавиатурное сокращение. Если оно там есть, метод получает из карты объект, соответствующий сокращению, и с его помощью пытается найти в карте команд соответствующую команду. Если такая команда есть, она выполняется, и работа завершается. (Кстати, вы можете переопределить метод processKeyBinding(), если захотите реализовать в своем компоненте особенный способ обработки сокращения.)

Если сокращение не было обработано, начинается опрос родительских компонентов. Для каждого из них также вызывается метод processKeyBinding(), но на этот раз ему указывают, что компонент находится в состоянии WHEN_ANCESTOR_OF_FOCUSED_COMPONENT, так что поиск производится в соответствующей карте входных событий.

Таким образом, базовый класс JComponent вместе с помощниками заботится о поддержке компонентами Swing клавиатурных сокращений. Механизм клавиатурных сокращений используется практически всеми компонентами, начиная от кнопок и заканчивая текстовыми элементами. Там, где к нему обращаются очень часто, нам предоставляют более удобные способы регистрации клавиатурных сокращений (например, мнемоники для кнопок или клавиши быстрого доступа для команд меню), но и для остальных компонентов использовать их несложно. Более того, это единственный способ надежно обработать событие от клавиатуры, не происходящее непосредственно в самом компоненте. Напоследок рассмотрим небольшой пример:

```
// KeyBindingTest.java
// Пример использования клавитурных сокращений

import javax.swing.*;
import java.awt.event.*;
```

```
public class KeyBindingTest extends JFrame {
  public KeyBindingTest() {
    super("KeyBindingTest");
    // выход при закрытии окна
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // настраиваем карты команд и входных событий для
    // корневой панели приложения
    InputMap im = getRootPane().getInputMap();
    ActionMap am = getRootPane().getActionMap();
    // срабатывает при отпускании сочетания Ctrl+A
    im.put(KeyStroke.getKeyStroke(
        KeyEvent.VK_A,
        KeyEvent.CTRL_MASK, true), "Action");
    // срабатывает при печати буквы 'Я'
    im.put(KeyStroke.getKeyStroke('Я'), "Action");
    am.put("Action", new AnAction());
    // выводим окно на экран
    setSize(200, 200);
    setVisible(true);
  }

  // класс нашей команды
  class AnAction extends AbstractAction {
    public void actionPerformed(ActionEvent e) {
      System.out.println("OK");
    }
  }

  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new KeyBindingTest(); } });
  }
}
```

В данном примере мы создаем маленькое окно и настраиваем клавиатурные сокращения для корневой панели этого окна (в любом контейнере высшего уровня Swing есть корневая панель). В нашем примере используются два сокращения: первое срабатывает при отпускании сочетания клавиш Ctrl+A (именно при отпускании, вы увидите это, запустив пример, такие сокращения позволяет создавать класс KeyStroke), а второе показывает возможность работы и с кириллицей. Правда, кириллические буквы будут распознаваться только для событий типа KEY_TYPED (печать символа), потому что в событиях KEY_PRESSED и KEY_RELEASED участвуют только символы с виртуальными кодами, определенными в классе KeyEvent, а кириллических символов там нет. Поэтому

при создании объектов KeyStroke для кириллических символов используйте те методы getKeyStroke(), которые создают клавиатурное сокращение для событий KEY_TYPED. Один из таких методов мы и задействовали в примере. Эта проблема очень неприятно сказывается на клавиатурных сокращениях, используемых в кнопках и меню Swing, когда клавиши быстрого доступа отказываются работать с русской раскладкой клавиатуры[2]. Мы обсудим это в главе 9, посвященной кнопкам.

Между делом, отметьте, какую гибкость способны придать приложению карты входных событий и команд. Мы создали два клавиатурных сокращения (поместив их в карту для состояния WHEN_FOCUSED, именно эта карта возвращается методом getInputMap() без параметров), но привязали их одному и тому же событию. Будь сокращения отображены непосредственно на команды, сделать этого не удалось бы. Остается лишь запустить приложение и убедиться в том, что механизмы поддержки клавиатурных сокращений Swing действуют.

Манипулировать картами входных событий и команд напрямую в обычных приложениях приходится редко: в них чаще всего клавиатурные сокращения создаются только для меню и кнопок, а там существуют более удобные способы их создания. Они полезнее в больших приложениях, буквально «напичканных» функциями, многие из которых срабатывают только при нажатии клавиатурного сокращения. В таких приложениях очень полезна возможность легко добавлять необходимые пользователю сокращения или полностью менять их, в зависимости от его пристрастий. Также не обойтись без работы с картами входных событий и команд при создании своих собственных UI-представителей, новых внешнего вида и поведения, новых компонентов. В таких случаях замена карт позволяет сразу же указать, как должен реагировать компонент на клавиатуру на определенной платформе.

## *Система передачи фокуса ввода*

Фокус ввода — довольно простое определение. Компонент, обладающий фокусом ввода, первым получает все события от клавиатуры. Эти события не обязательно остаются в самом компоненте (как мы уже видели, клавиатурные сокращения обрабатываются даже в тех случаях, когда событие от клавиатуры произошло в другом компоненте), но тем не менее у компонента, обладающего фокусом ввода, всегда есть шанс обработать нажатие клавиш первым. Фокус ввода передается от компонента к компоненту различными способами: когда пользователь щелкает на другом компоненте мышью (если тот обрабатывает щелчки мыши) или нажимает специальные клавиши (чаще всего клавишу Tab или ее сочетания с управляющими клавишами). Система передачи фокуса ввода, встроенная в любую схему пользовательского интерфейса, определяет, какие компоненты могут обладать фокусом ввода и порядок его передачи.

Вся власть и полномочия по передаче фокуса ввода находятся у класса библиотеки AWT с названием KeyboardFocusManager. Любые сочетания клавиш и события, подходящие для передачи фокуса ввода, приходят прежде всего к нему. Происходит это на самом низком уровне обработки событий базового компонента всех графических библиотек Java Component, что позволяет унифицировать процесс передачи фокуса для любых компонентов, независимо от того, тяжеловесные они или легковесные и к какой библиотеке принадлежат. Любой унаследованный от Component класс (а значит и все компоненты библиотеки Swing) находится под контролем новой системы фокуса ввода и класса KeyboardFocusManager.

Прежде чем перейти к обсуждению деталей реализации системы передачи фокуса ввода, посмотрим, как она функционирует с точки зрения ее пользователя, то есть программиста, который настраивает поведение своего компонента.

[2] Кстати, не только с русской, но и с любой другой, отличной от английской.

## Настройка системы передачи фокуса

Если не вдаваться в подробности реализации системы передачи фокуса ввода, можно сказать, что она исключительно прозрачна и проста. Для любого компонента, унаследованного от класса Component, вы можете указать, нужен ли этому компоненту фокус ввода, вызвав для этого метод setFocusable() (по умолчанию считается, что компонент нуждается в фокусе ввода, то есть обрабатывает события от клавиатуры), и настроить список клавиатурных сокращений, согласно которым фокус ввода будет передаваться к следующему или предыдущему компоненту. Клавиатурные сокращения, служащие для передачи фокуса ввода, хранятся во множествах Set, передать их в компонент можно методом setFocusTraversalKeys(). По умолчанию для передачи фокуса используются клавиши Tab и Shift+Tab, для передачи фокуса к следующему и предыдущему компоненту соответственно. После настройки ваш компонент окажется под надежной опекой класса KeyboardFocusManager, который позаботится о том, чтобы фокус ввода вовремя оказывался у вашего компонента и покидал его только при нажатии тех клавиш, что вы указали. Рассмотрим небольшой пример:

```
// FocusKeysTest.java
// Настройка клавиш перехода фокуса ввода
import javax.swing.*;
import java.awt.*;
import java.awt.event.KeyEvent;
import java.util.HashSet;

public class FocusKeysTest extends JFrame {
  public FocusKeysTest() {
    super("FocusKeysTest");
    // выход при закрытии окна
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // добавляем пару кнопок
    setLayout(new FlowLayout());
    // особая кнопка
    JButton button = new JButton("Особая");
    add(button);
    add(new JButton("Обычная"));
    // настроим клавиши перехода фокуса
    HashSet<AWTKeyStroke> set = new HashSet<AWTKeyStroke>();
    set.add(AWTKeyStroke.getAWTKeyStroke(
        'Q', KeyEvent.CTRL_MASK));
    button.setFocusTraversalKeys(
        KeyboardFocusManager.FORWARD_TRAVERSAL_KEYS, set);
    // выводим окно на экран
    setSize(200, 200);
    setVisible(true);
  }
```

```
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new FocusKeysTest(); } });
  }
}
```

В данном примере мы добавляем в окно две кнопки, для одной из которых специально настраиваем клавиши передачи фокуса ввода следующему компоненту. Для этого необходимо поместить в множество Set (класс множеств Set является абстрактным, у него есть несколько подклассов, мы использовали HashSet) клавиатурные сокращения, при нажатии которых будет происходить передача фокуса. Клавиатурные сокращения для системы передачи фокуса создаются статическими методами класса AWTKeyStroke[3]. В примере мы создаем сочетание Ctrl+Q, добавляем его в множество, и передаем множество компоненту, указывая константой FORWARD_TRAVERSAL_KEYS, что это сочетание будет использоваться для передачи фокуса следующему компоненту (клавиши для остальных ситуаций при этом не меняются). Запустив программу, вы убедитесь, что теперь фокус ввода передается от первой кнопки ко второй только новым сочетанием клавиш, в то время как вторая кнопка исправно реагирует на нажатие клавиши Tab.

Теперь, когда мы разобрались, как настраивать клавиши, используемые для передачи фокуса ввода от одного компонента к другому, интересно будет узнать, как система передачи фокуса ввода определяет, какому компоненту будет передан фокус. Этим заведуют подклассы абстрактного класса FocusTraversalPolicy. При срабатывании сочетания клавиш, отвечающего за передачу фокуса ввода, текущий подкласс FocusTraversalPolicy определяет, какой именно компонент получит фокус ввода следующим[4]. Создать собственный алгоритм нетрудно, однако AWT и Swing предоставляют несколько стандартных алгоритмов передачи фокуса ввода, которых вполне хватает для большинства ситуаций (табл. 5.2).

**Таблица 5.2.** Стандартные алгоритмы передачи фокуса

| Имя класса | Предназначение |
|---|---|
| ContainerOrderFocusTraversalPolicy | Данный алгоритм имитирует использовавшийся в прежних версиях JDK переход фокуса ввода согласно очередности добавления компонентов в контейнер. Первый добавленный в контейнер компонент получает фокус при выводе окна на экран, фокус переходит к компоненту, добавленному вторым и т. д. |
| DefaultFocusTraversalPolicy | Работает аналогично предыдущему алгоритму, и к тому же взаимодействует с помощниками (peer) компонентов, что расширяет его возможности по работе с тяжеловесными компонентами. По умолчанию используется в приложениях AWT. В данный момент плохо сочетается с легковесными компонентами Swing |

[3] Это полный аналог класса KeyStroke. С версии JDK 1.4 все функции класса KeyStroke перешли к AWTKeyStroke, а KeyStroke просто наследует от него.

[4] Класс FocusTraversalPolicy — это ни что иное, как стратегия, или подключаемый алгоритм.

**Таблица 5.2** *(продолжение)*

| Имя класса | Предназначение |
|---|---|
| SortingFocusTraversalPolicy | Предоставляется библиотекой Swing. Позволяет организовать передачу фокуса ввода согласно некоторому признаку, который определяет, какие компоненты получают фокус раньше, а какие позже. Для работы требует объект Comparator, обеспечивающий сравнение компонентов. Может использоваться для получения весьма интересных специализированных алгоритмов передачи фокуса, так как написать объект сортировки Comparator куда проще, чем создавать алгоритм передачи фокуса «с нуля» |
| LayoutFocusTraversalPolicy | Расширяет предыдущий алгоритм, передавая ему для сортировки компонентов объект LayoutComparator. Последний получает список компонентов контейнера и выясняет, как они располагаются на экране. Это позволяет передавать фокус ввода именно в том порядке, в котором расположены компоненты перед глазами пользователя. Учитывает особенности локализованных программ (в том числе письмо справа налево). По умолчанию в окнах Swing используется именно этот алгоритм |

Помимо того что описанные алгоритмы определяют очередность получения компонентами фокуса ввода, они еще выполняют много мелкой работы, точно выясняя, подходит ли компонент для получения фокуса, и учитывая при этом расположение, видимость компонента и его личное желание обладать фокусом. При написании собственного алгоритма обо всем этом придется думать самому, так что при такой необходимости лучше использовать класс SortingFocusTraversalPolicy, позволяющий сортировать компоненты в отдельном объекте. Напоследок давайте рассмотрим небольшой пример, иллюстрирующий различия описанных алгоритмов:

```
// FocusPolicyTest.java
// Различные алгоритмы передачи фокуса ввода
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class FocusPolicyTest extends JFrame {
  public FocusPolicyTest() {
    super("FocusPolicyTest");
    // при закрытии окна выход
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // добавляем три кнопки
    add(new JButton("Левая"), "West");
    // добавляем эту кнопку второй, но она будет ниже
    // двух других кнопок
```

```
    JButton button = new JButton("Сменить");
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        // при нажатии сменим алгоритм для окна
        setFocusTraversalPolicy(
            new ContainerOrderFocusTraversalPolicy());
      }
    });
    add(button, "South");
    add(new JButton("Правая"), "East");
    // выводим окно на экран
    setSize(200, 200);
    setVisible(true);
  }

  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new FocusPolicyTest(); } });
  }
}
```

Здесь мы создаем небольшое окно с тремя кнопками, причем добавляются они в окно таким образом, чтобы порядок расположения их в окне не совпадал с порядком добавления их в контейнер. По умолчанию в окне используется полярное расположение, его мы и учли для получения такого эффекта, добавив вторую по счету кнопку в нижнюю часть окна. К ней добавляется слушатель событий, обрабатывающий нажатие кнопки и меняющий при этом алгоритм передачи фокуса в окне.

Запустив программу с примером, вы сможете оценить работу двух различных алгоритмов передачи фокуса ввода. Сначала действует алгоритм LayoutFocusTraversalPolicy, и фокус ввода передается так, как располагаются кнопки в окне. Сменив алгоритм на ContainerOrderFocusTraversalPolicy, вы увидите, во-первых, что теперь фокус переходит так, как мы добавляли кнопки в окно (и с точки зрения пользователя это довольно неудобно), а, во-вторых, что на некоторое время фокус исчезает в каких-то невидимых компонентах. Это проблем, так как система считает, что *любой* компонент нуждается в фокусе ввода, хотя корневая панель в нем не нуждается. Именно в корневой панели и «застревает» ненадолго фокус ввода. Алгоритм LayoutFocusTraversalPolicy справляется с этой проблемой, потому что дополнительно проверяет старые свойства, связанные с фокусом ввода, определенные в классе JComponent еще до появления новой системы. Поэтому лучше не использовать в приложениях Swing алгоритмы передачи фокуса AWT.

## Расширенные возможности

В системе передачи фокуса ввода также есть несколько дополнительных возможностей. Прежде всего стоит отметить понятие *цикла передачи фокуса* (focus cycle) и его *корня* (focus cycle root). Фокус ввода передается только в пределах цикла передачи фокуса, который представляет собой набор из некоторых компонентов. Компоненты находятся

в контейнере, который называется корнем цикла. По умолчанию в качестве корня используются только контейнеры высшего уровня, и все добавляемые в них компоненты оказываются в одном цикле, однако корнем можно сделать любой контейнер, что может быть полезным для очень больших контейнеров, напичканных компонентами. Для перехода из одного цикла передачи фокуса в другой применяется специальное множество клавиатурных сокращений, его позволяет задать класс KeyboardFocusManager. Чтобы заставить контейнер вести себя как корень цикла, необходимо переопределить его метод isFocusCycleRoot() так, чтобы он возвращал значение true. Тем не менее, в обычных приложениях с обычными пользовательскими интерфейсами создание дополнительных циклов передачи фокуса требуется редко, поскольку лишь «добавляет пользователям головной боли».

Далее стоит сказать о возможностях окон Java, унаследованных от класса Window. Благодаря усилиям класса KeyboardFocusManager, у окон AWT имеется возможность отказываться от фокуса ввода. Для отключения возможности получения фокуса ввода в окне вы можете использовать метод setFocusableWindowState(false). После этого компоненты, расположенные в окне, не будут получать сообщений от клавиатуры. События (описанные в интерфейсе слушателя WindowFocusListener) позволяют различить активизацию окна (в общем случае это означает, что пользователь использовал окно последним) и получение окном фокуса ввода (что может произойти только в окне, которое соглашается обладать фокусом ввода). Интересно, что события, сообщающие о получении фокуса ввода, полностью создаются внутри Java классом KeyboardFocusManager, а не приходят из операционной системы.

Также благодаря тому, что система передачи фокуса ввода полностью написана на Java, появилась возможность программно управлять передачей фокуса, и вдобавок получать исчерпывающую информацию о состоянии системы и компонентах, обладающих фокусом. Все это также относится к функциям класса KeyboardFocusManager. С его помощью вы сможете легко передать фокус следующему, предыдущему или произвольному компоненту, а также узнать, какой компонент обладает в данный момент фокусом ввода и какое окно является активным. Более подробно познакомиться с возможностями системы передачи фокуса ввода и методами ее центрального класса KeyboardFocusManager можно в интерактивной документации Java.

## *Всплывающие подсказки и клиентские свойства*

Кроме тех важнейших базовых возможностей, которыми обеспечивает библиотеку Swing класс JComponent и которые мы только что рассмотрели, у него осталось еще несколько обязанностей. С ними он также справляется с блеском, и представить Swing без этого просто невозможно.

Очень полезным свойством всех без исключения компонентов Swing является возможность показывать *всплывающие подсказки* (tool tips) при небольшой задержке указателя мыши над компонентом. Поддержка Swing всплывающих подсказок также обеспечивается базовым классом JComponent. В нем определен метод setToolTipText(), в который вы передаете текст подсказки. При передаче в данный метод текста JComponent регистрирует компонент во вспомогательном классе ToolTipManager. Последний представляет собой одиночку и централизованно управляет процессом вывода подсказок в системе. При регистрации компонентов он присоединяется к ним в качестве слушателя событий от мыши, и при появлении указателя мыши на компоненте запускает таймер, следящий за временем, в течение которого указатель мыши остается на компоненте неподвижным, и таким образом определяет, когда следует вывести подсказку. Подробнее о всплывающих подсказках мы узнаем в главе 8.

Наконец, в арсенале класса JComponent есть еще одно оружие — *клиентские свойства* (client properties), весьма удачная находка создателей Swing. Эти свойства практически

аналогичны свойствам JavaBeans, только они не требуют наличия в классе компонента поля, хранящего значение самого свойства, и пары методов get/set. Вместо этого значение свойства хранится в специальном ассоциативном массиве, а для его изменения и получения используется пара методов putClientProperty() и getClientProperty(). Клиентские свойства, как и свойства JavaBeans, являются привязанными и позволяют настраивать и изменять компонент без его переписывания и наследования от него. Иногда это может быть очень полезно, например при появлении свойств, которые еще не совсем готовы или чересчур усложняют компонент. В Swing клиентские свойства используются очень широко, особенно для настройки внутренней работы компонентов и их взаимодействия, а также для наделения компонента новыми свойствами, которые предположительно (но не обязательно) станут затем обычными свойствами JavaBeans, а пока просто используются для изучения реакции разработчиков на нововведение (как это было, например, с плавающими панелями инструментов). Клиентские свойства весьма удобны при разработке новых компонентов, позволяя сократить количество вспомогательных методов и полей, их также удобно задействовать при добавлении к уже существующим компонентам новых возможностей, когда наследование от компонентов излишне громоздко. Впрочем, злоупотреблять ими не следует, и если у вас появляется с десяток свойств, реализованных в виде клиентских, стоит подумать о создании нового компонента с обычными свойствами JavaBeans.

## *Резюме*

Внутренние «винтики» Swing, незаметно для нас крутящиеся внутри всех компонентов, в основном скрыты в базовом классе библиотеки JComponent. Все компоненты библиотеки наследуют от него и поэтому поддерживают основные свойства библиотеки, как-то: всплывающие подсказки, улучшенные системы рисования и проверки корректности, клавиатурные сокращения и многое другое. Как правило, вдаваться в детали внутренних процессов Swing приходится редко, но если вам понадобилось особым образом настроить работу своих компонентов, знание о том, что происходит внутри, оказывается как нельзя кстати.

# Глава 6. Контейнеры высшего уровня

После создания пользовательского интерфейса необходимо вывести его на экран, чтобы пользователь увидел его и смог им воспользоваться. Для этого предназначены специальные компоненты библиотеки Swing, называемые *контейнерами высшего уровня* (top level containers). Они представляют собой окна операционной системы, в которых вы размещаете компоненты своего пользовательского интерфейса. К контейнерам высшего уровня относятся окна JFrame и JWindow, диалоговое окно JDialog, а также апплет JApplet (который не является окном, но тоже предназначен для вывода интерфейса в браузере, запускающем этот апплет). В этой главе мы также рассмотрим компонент-«рабочий стол» JDesktopPane и внутренние окна JInternalFrame, которые хоть формально и не являются контейнерами высшего уровня, но по сути выполняют такие же задачи.

Мы говорили, что все компоненты библиотеки Swing являются легковесными и не требуют поддержки операционной системы — это верно, но только не для контейнеров высшего уровня. Они представляют собой тяжеловесные компоненты (точнее, контейнеры) и являются исключением из общего правила. Ничего страшного в этом исключении нет: достаточно вспомнить, что легковесные компоненты (к ним относятся все компоненты Swing) являются просто областью экрана некоторого тяжеловесного контейнера, который любезно предоставляет им свое пространство. Как раз контейнеры высшего уровня и предоставляют остальным компонентам Swing пространство экрана для работы, а также следят, чтобы они вовремя получали сообщения операционной системы о перерисовке и возникновении событий (как мы знаем, операционная система понятия не имеет о существовании легковесных компонентов).

На самом деле оказывается, что контейнеры высшего уровня Swing — это самые незатейливые компоненты библиотеки. Они просто наследуют от соответствующих классов AWT (Frame, Window, Dialog и Applet), а также, чтобы смягчить разницу между контейнерами AWT и компонентами Swing, застилают свое пространство специальной «соломкой» в виде так называемой корневой панели, которая обеспечивает легковесные компоненты некоторыми возможностями, отсутствующими в контейнерах AWT (например, специальным местом для строки меню Swing, о которой AWT знать не знает). На этом работа контейнеров высшего уровня Swing заканчивается. Можно попробовать разместить компоненты Swing напрямую в окнах AWT, но в результате получится не слишком работоспособный интерфейс, так как корневая панель предоставляет компонентам Swing весьма важные услуги.

Поэтому, прежде чем перейти к рассмотрению контейнеров высшего уровня Swing (а они очень просты), мы поподробнее рассмотрим корневую панель и ее роль в создании интерфейса и функционировании некоторых компонентов.

## *Корневая панель JRootPane*

Каждый раз, когда вы создаете новый контейнер высшего уровня, будь то обычное окно, диалоговое окно или апплет, в конструкторе этого контейнера создается особый компонент — экземпляр класса JRootPane, который добавляется в контейнер, причем так, чтобы занять все доступное в контейнере место. Более того, все контейнеры высшего уровня Swing следят за тем, чтобы другие компоненты не смогли «пробраться» за пределы этого специального компонента: они переопределяют метод add(), предназначенный для добавления компонентов, и при вызове этого метода принудительно добав-

ляют их в специально предназначенное для этого место. Этот специальный компонент и называется *корневой панелью* (root pane). Все компоненты своего пользовательского интерфейса вы будете добавлять именно в корневую панель.

Корневая панель используется вовсе не для того, чтобы все запутать — у нее есть вполне определенное и очень важное назначение, что и позволяет компонентам Swing комфортно чувствовать себя в тяжеловесном контейнере. Прежде всего, она добавляет в контейнеры свойство «глубины» — возможность не только размещать компоненты одним над другим, но и при необходимости менять их местами, увеличивать или уменьшать глубину расположения компонентов. Такая возможность в первую очередь востребована многодокументными приложениями Swing (окна которых представляют собой обычные легковесные компоненты, которые необходимо размещать друг над другом), а также выпадающими меню и всплывающими подсказками. Далее, корневая панель позволяет разместить строку меню Swing и предлагает еще несколько весьма полезных возможностей, позволяя в том числе ускорить процесс прорисовки компонентов, потому что благодаря ей управление этим процессом сразу передается в библиотеку Swing и класс JComponent.

Вообще, корневая панель — это несколько определенным образом расположенных панелей, каждая из которых исполняет определенную роль[1]. Обсудив каждую часть корневой панели по отдельности, мы сможем увидеть, какие функции она выполняет в целом. Давайте начнем с самого начала.

## Многослойная панель JLayeredPane

В основании корневой панели лежит так называемая *многослойная панель* JLayeredPane. Она занимает все доступное пространство контейнера, и именно в этой панели затем располагаются все остальные части корневой панели, в том числе и все компоненты вашего пользовательского интерфейса. Вполне можно сказать, что это — самая важная часть корневой панели.

Многослойная панель используется для добавления в контейнер свойства *глубины* (depth), иногда еще говорят, что многослойная панель упорядочивает компоненты в соответствии с их *порядком в стопке* (Z-order). То есть многослойная панель позволяет организовать в контейнере новое, третье, измерение, вдоль которого располагаются *слои* (layers) компонента. В обычном контейнере расположение компонента определяется прямоугольником, который показывает, какую часть контейнера занимает компонент. При добавлении компонента в многослойную панель необходимо указать не только прямоугольник, занимаемый компонентом, но и слой, в котором он будет располагаться. Слой в многослойной панели определяется целым числом — слой находится тем выше, чем больше определяющее его число. Окончательно прояснит ситуацию небольшая иллюстрация (рис. 6.1).

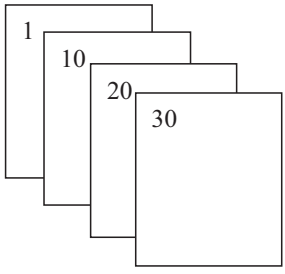
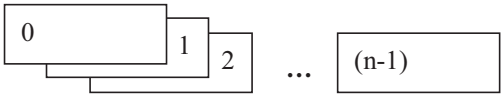



**Рис. 6.1.** Слои в многослойной панели

[1] Если выражаться языком шаблонов проектирования, можно сказать, что корневая панель — это *компоновщик* (composite).

Как видно, многослойную панель можно представить себе как своего рода «слоеный пирог» — она словно состоит из нескольких контейнеров, расположенных друг на друге. На самом деле контейнер всего один, просто внутренние механизмы многослойной панели следят за тем, чтобы компоненты располагались соответственно своим слоям. Компоненты, находящиеся в более высоком слое, всегда будут перекрывать компоненты из более низких слоев (впрочем, это верно только для ситуации, когда компоненты реально перекрывают друг друга, в противном случае то, что они находятся в разных слоях, внешне никак не проявляется). Слоев может быть столько, сколько целых чисел вы сможете придумать, главное здесь то, что слои с большими номерами располагаются в стопке выше слоев с меньшими номерами.

Кроме слоя, который определяет, как высоко будут находиться все компоненты, находящиеся в этом слое, многослойная панель также позволяет задать *позицию* компонента в слое. Позиция компонента позволяет указать, как будут располагаться компоненты из одного и того же слоя. Вполне может быть, что два компонента находящиеся в одном и том же слое, перекрывают друг друга. В таких ситуациях в дело и вступает позиция компонента — она определяет, какой из компонентов будет в стопке выше. Позиция задается несколько иначе, чем слой — она определяется целым числом от нуля и больше, но выше в стопке будет располагаться компонент с *меньшим* номером. Объяснить это очень просто: позиция соответствует очередности добавления компонентов в контейнер — первый добавленный компонент всегда имеет нулевую позицию, второй первую и т. д. На рис. 6.2 показано, как располагаются компоненты согласно своим позициям.




n – количество компонентов в слое

**Рис. 6.2.** Позиции компонентов в слое

Такое поведение (первый добавленный в контейнер компонент оказывается выше компонентов, добавленных позже) унаследовано многослойной панелью от обычных контейнеров, которые не имеют слоев, и при добавлении пересекающихся компонентов располагают выше те компоненты, которые были добавлены раньше. Чаще всего вам не придется иметь дело с позициями компонентов, при добавлении компонентов они меняются автоматически. Тем не менее многослойная панель позволяет менять позиции компонентов динамически, уже после их добавления в контейнер.

Возможности многослойной панели широко используются некоторыми компонентами Swing, особенно они важны для многодокументных приложений, всплывающих подсказок и меню. Многодокументные приложения задействуют специальный контейнер JDesktopPane (буквально — «рабочий стол»), унаследованный от многослойной панели JLayeredPane, в котором располагаются внутренние окна Swing. Самые важные функции многодокументного приложения — расположение «активного» окна над другими, сворачивание окон, их перетаскивание — обеспечиваются механизмами многослойной панели. Основное преимущество от использования многослойной панели для всплывающих подсказок и меню — это ускорение их работы. Вместо создания для каждой подсказки или меню нового тяжеловесного окна, располагающегося над компонентом, в котором возник запрос на вывод подсказки или меню, Swing создает быстрый легковесный компонент. Этот компонент размещается в достаточно высоком слое многослойной панели

(которая есть в любом контейнере высшего уровня) выше в стопке всех остальных компонентов, и используется для вывода подсказки или меню.

Чтобы лучше понять, как многослойная панель используется во внутренних механизмах Swing, а затем правильно применять ее в собственных целях, мы рассмотрим стандартные слои, характерные для всех компонентов Swing.

Как мы уже знаем, многослойная панель позволяет организовать столько слоев, сколько целых чисел вы сможете придумать (доступны даже отрицательные целые числа). Тем не менее, компоненты, использующие многослойную панель, должны быть уверены в том, что при добавлении компонента в определенный слой не возникнет путаницы, и компоненты, которые должны находиться выше других в стопке, не окажутся закрытыми из-за того, что кто-то по незнанию занял более высокий слой. Чтобы избежать путаницы, многослойная панель определяет несколько стандартных слоев, которые и используются всеми компонентами Swing, что позволяет обеспечить правильную работу всех механизмов многослойной панели. Посмотрим теперь, какие стандартные слои имеются в Swing (рис. 6.3).

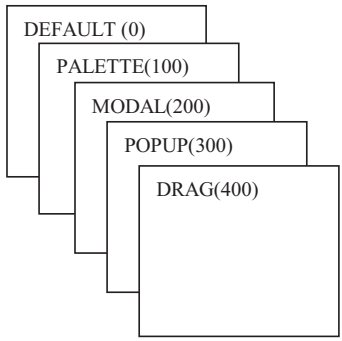



**Рис. 6.3.** Стандартные слои Swing

Как вы можете видеть, стандартных слоев не очень много, и у всех них достаточно выразительные имена, позволяющие догадаться, для чего используется тот или иной слой. Тем не менее, стоит познакомиться с этими слоями немного подробнее (табл. 6.1).

**Таблица 6.1.** Предназначение стандартных слоев многослойной панели

| Название слоя | Предназначение |
|---|---|
| Default | Этот слой используется для всех обычных компонентов, которые вы добавляете в контейнер. В нем же располагаются внутренние окна многодокументных приложений |
| Palette | Слой предназначен для размещения так называемых палитр, или окон с набором инструментов, которые обычно перекрывают остальные элементы интерфейса. Создавать такие окна позволяет панель JDesktopPane, которая размещает их как раз в этом слое |
| Modal | Судя по названию, разработчики планировали использовать этот слой для размещения легковесных модальных диалоговых окон. Однако такие диалоговые окна пока не реализованы, так что этот слой в Swing в настоящее время не используется |

**Таблица 6.1** *(продолжение)*

| Название слоя | Предназначение |
|---|---|
| Popup | Наиболее часто используемый слой, служащий для размещения легковесных всплывающих меню и подсказок |
| Drag | Самый верхний в стопке слой. Предназначен для обслуживания операций перетаскивания (drag and drop), которые должны быть хорошо видны пользователю |

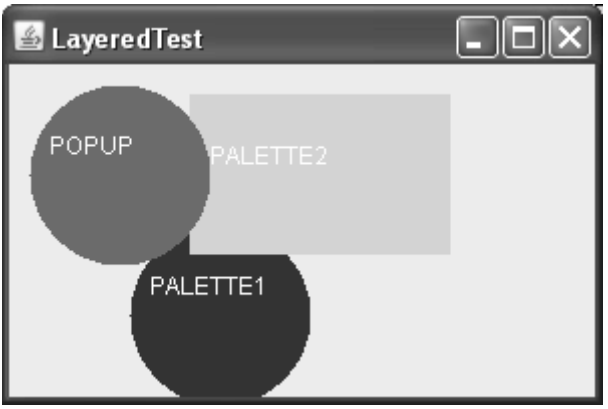
Чтобы окончательно познакомиться с многослойной панелью, рассмотрим небольшой пример. Он покажет, как добавлять компоненты в различные слои, и как слои располагаются друг над другом в стопке:

```
// LayeredTest.java
// Возможности многослойной панели
import javax.swing.*;
import java.awt.*;

public class LayeredTest extends JFrame {
  public LayeredTest() {
    super("LayeredTest");
    // выход при закрытии окна
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // получаем многослойную панель
    JLayeredPane lp = getLayeredPane();
    // три фигуры
    Figure fg1 = new Figure(Color.red, 0, "POPUP");
    Figure fg2 = new Figure(Color.blue, 0, "PALETTE1");
    Figure fg3 = new Figure(Color.green, 1, "PALETTE2");
    // расположение фигур в окне
    fg1.setBounds(10, 10, 120, 120);
    fg2.setBounds(60, 80, 160, 180);
    fg3.setBounds(90, 15, 250, 180);
    // добавляем в различные слои
    lp.add(fg1, JLayeredPane.POPUP_LAYER);
    lp.add(fg2, JLayeredPane.PALETTE_LAYER);
    lp.add(fg3, JLayeredPane.PALETTE_LAYER);
    // смена позиции одной из фигур
    lp.setPosition(fg3, 0);
    // выводим окно на экран
    setSize(300, 200);
    setVisible(true);
  }
  // класс, позволяющий рисовать два типа фигур с текстом
```

```
  class Figure extends JComponent {
    private Color color;
    private int type;
    private String text;
    // параметры: цвет и тип фигуры
    Figure(Color color, int type, String text) {
      this.color = color;
      this.type = type;
      this.text = text;
      setOpaque(false);
    }
    public void paintComponent(Graphics g) {
      // прорисовка фигуры
      g.setColor(color);
      switch (type) {
        case 0: g.fillOval(0, 0, 90, 90); break;
        case 1: g.fillRect(0, 0, 130, 80); break;
      }
      g.setColor(Color.white);
      g.drawString(text, 10, 35);
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new LayeredTest(); } });
  }
}
```

В данном примере мы создаем небольшое окно JFrame, указываем ему, что при закрытии окна необходимо завершить программу (с помощью метода setDefaultCloseOperation(), о котором мы вскоре узнаем подробнее), и добавляем в многослойную панель несколько компонентов Figure. Чтобы получить многослойную панель в любом контейнере высшего уровня Swing, достаточно вызвать метод getLayeredPane(). Вспомогательный класс Figure наследует от базового класса JComponent и позволяет различными цветами рисовать фигуры двух типов (круги и прямоугольники), вводя поверх фигур надписи. Параметры для прорисовки фигур задаются в конструкторе класса. Заметьте, что компоненты Figure не закрашивают цвет фона (свойство непрозрачности установлено в false, для этого мы в конструкторе воспользовались методом setOpaque()), это позволяет им лучше «просвечивать» друг сквозь друга. Здесь мы создаем три фигуры разного цвета (два круга и прямоугольник) и добавляем круг в слой POPUP_LAYER, а прямоугольники — в слой PALETTE_LAYER. Заметьте, что при добавлении компонентов приходится указывать их абсолютные экранные координаты, потому что в многослойной панели обычные менеджеры расположения не работают. В нашем примере мы для простоты сразу указали координаты фигур, однако в реальных приложениях лучше

тщательно рассчитать размеры компонентов, учитывая размеры шрифтов и другие графические параметры, потому что на разных платформах все может выглядеть поразному. Напоследок мы меняем позицию одного из прямоугольников, так чтобы он был первым в слое, хотя изначально добавлялся вторым. Запустив приложение, вы увидите, что многослойная панель работает и аккуратно располагает компоненты согласно их слоям и позициям.




В обычных приложениях многослойная панель редко используется напрямую, в них она выполняет свои функции незаметно. Тем не менее, иногда она помогает создать удивительные эффекты и необычные интерфейсы, позволяя, например, разместить поверх обычных компонентов анимацию или видео, не требуя для этого от разработчика нечеловеческих усилий и ухищрений. Чуть позже мы рассмотрим пример использования многослойной панели для создания необычного интерфейса.

## Панель содержимого

*Панель содержимого* (content pane) — следующая часть корневой панели, служащая для размещения компонентов пользовательского интерфейса вашей программы. Она занимает большую часть пространства многослойной панели (за исключением того места, что занимает строка меню). Чтобы панель содержимого не закрывала добавляемые впоследствии в окно компоненты, многослойная панель размещает ее в специальном очень низком слое с названием FRAME_CONTENT_LAYER, с номером –30000. Обычные компоненты размещать в этом слое не стоит, потому что тогда вы их вряд ли увидите, а для панели содержимого этот слой подходит в самый раз, потому что она служит всего лишь для размещения других компонентов (при добавлении в панель содержимого все компоненты оказываются в слое DEFAULT_LAYER).

Именно в панель содержимого добавляются все компоненты вашего пользовательского интерфейса, это основное отличие контейнеров высшего уровня Swing от их собратьев из AWT. Когда вы вызываете метод add(), добавляя что-то в контейнер высшего уровня, за кулисами он преобразуется в следующий вызов:

```
getContentPane().add(ваш_компонент)
```

Впрочем, панель содержимого — это всего лишь экземпляр обычной панели JPanel, в которой для совместимости с окнами AWT устанавливается полярное расположение BorderLayout, поскольку считается, что в окнах по умолчанию должно использоваться полярное расположение (мы увидим в главе 7, что это действительно удобно). Никто не

запрещает поместить все ваши компоненты в отдельную панель с удобным вам расположением, а затем сделать ее панелью содержимого вашего окна. Это дело вкуса. Рассмотрим небольшой пример:

```
// ContentPaneAdd.java
// Замена панели содержимого
import javax.swing.*;
import java.awt.*;

public class ContentPaneAdd extends JFrame {
  public ContentPaneAdd() {
    super("ContentPaneAdd");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создадим панель с двумя кнопками
    JPanel contents = new JPanel();
    contents.add(new JButton("Один"));
    contents.add(new JButton("Два"));
    // заменим панель содержимого
    setContentPane(contents);
    // выведем окно на экран
    setSize(200, 100);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new ContentPaneAdd(); } });
  }
}
```

В примере мы создаем небольшое окно и панель с двумя кнопками, которую затем методом setContentPane() делаем панелью содержимого нашего окна. Так мы вместо полярного расположения задействуем предлагаемый по умолчанию в панелях JPanel менеджер FlowLayout. Тем не менее, разница невелика и выбор того или иного подхода зависит от ваших пристрастий. В общем, панель содержимого не представляет собой ничего хитрого, надо лишь помнить, что компоненты добавляются именно в нее.

## Строка меню

Одной из самых важных причин использования корневой панели в Swing является необходимость размещения в окне *строки меню* (menu bar). Более или менее сложное приложение вряд ли сможет обойтись без системы меню, позволяющей пользователю легко получить доступ ко всем функциям приложения, которые к тому же логически организованы в группы. Библиотека Swing (подробнее мы узнаем об этом в главе 10) предоставляет прекрасные возможности для создания больших и удобных систем меню, и, само собой, строка меню JMenuBar также является легковесным компонентом.

Однако стандартные окна AWT, напрямую связанные с операционной системой, могут размещать только строки меню AWT и не знают о легковесных меню Swing. Здесь нам помогает корневая панель. Она выделяет под строку меню специальное место и предоставляет метод setJMenuBar(), позволяющий вам установить в контейнере новую строку меню[2] (строка меню может использоваться в окне JFrame, диалоговом окне JDialog и апплете JApplet).

Строка меню также размещается в многослойной панели, в том же специальном слое FRAME_CONTENT_LAYER, и занимает небольшое пространство в верхней части окна, в длину равное размеру окна, ширина используемого пространства при этом зависит от самой строки меню и содержащихся в ней элементов. Корневая панель следит за тем, чтобы панель содержимого и строка меню не перекрывались. Если строка меню в контейнере высшего уровня не требуется, корневая панель использует все пространство окна для размещения панели содержимого. В принципе, это все, что можно сказать о поддержке корневой панелью строки меню; здесь все очень просто.

## Прозрачная панель

*Прозрачная панель* (glass pane) — это последний составляющий элемент корневой панели. Прозрачная панель размещается корневой панелью выше всех элементов многослойной панели — как бы высоко не находился используемый вами слой, прозрачная панель всегда будет выше. Следит за выполнением этого правила корневая панель, которая размещает прозрачную панель выше многослойной панели, причем так, чтобы она полностью закрывала всю область окна, включая и область, занятую строкой меню.

Прозрачная панель используется в приложениях достаточно редко, поэтому по умолчанию корневая панель делает ее невидимой, что позволяет уменьшить нагрузку на систему рисования. Стоит иметь в виду, что если вы делаете прозрачную панель видимой, нужно быть уверенным в том, что она прозрачна (ее свойство непрозрачности opaque равно false), потому что в противном случае она закроет все остальные элементы корневой панели, и вы не увидите своего интерфейса. Для чего же может пригодиться прозрачная панель? Оказывается, с ее помощью можно реализовать очень полезные функции, для реализации которых «с нуля» понадобились бы приличные усилия. Благодаря размещению прозрачной панели над всеми компонентами интерфейса, она может быть использована для эффектной анимации, «плавающей» поверх всех компонентов, включая строку меню, или для перехвата событий, если некоторые из них необходимо обрабатывать и не отправлять в основную часть пользовательского интерфейса.

Давайте рассмотрим небольшой, но полезный пример — прототип системы справки для каждого компонента. Мы уже знаем, что у каждого компонента Swing может быть небольшая подсказка, всплывающая при наведении на него указателя мыши, но иногда пользователю этого недостаточно. В таком случае может быть полезна возможность получения для каждого компонента интерфейса помощи другого рода, действующей следующим образом: пользователь переходит в режим получения помощи, после чего компоненты интерфейса перестают реагировать на его действия. Вместо этого система помощи определяет, какой компонент выбирается пользователем, и выводит по нему подробную информацию. По завершении работы в режиме получения помощи интерфейс возвращается в свое обычное состояние, а получивший нужную информацию

[2] Конечно, можно было бы вручную пытаться разместить строку меню Swing в окне, используя подходящие менеджеры расположения. Однако реализованный подход позволяет сохранить общие черты окон AWT и Swing и его проще программировать: имеется специальное место для строк меню, а все остальное пространство находится в вашем распоряжении.

пользователь сможет эффективно им пользоваться. Попробуем реализовать такое поведение с помощью прозрачной панели:

```
// HelpSystemDemo.java
// Как прозрачная панель может помочь в создании
// системы помощи
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class HelpSystemDemo extends JFrame {
  // необходимые нам поля
  private JButton button1, button2, help;
  private HelpSystem hs = new HelpSystem();
  private InterceptPane ip = new InterceptPane();
  private ImageIcon helpIcon = new ImageIcon("Help.gif");

  public HelpSystemDemo() {
    super("HelpSystemDemo");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем нащ интерфейс
    button1 = new JButton("Что-то делает");
    button2 = new JButton("Тоже что-то делает");
    JPanel contents = new JPanel();
    contents.add(button1);
    contents.add(button2);
    // кнопка вызова помощи
    help = new JButton(helpIcon);
    help.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        // при нажатии включаем прозрачную панель
        ip.setVisible(true);
        // и специальный указатель мыши
        getRootPane().setCursor(getToolkit().
            createCustomCursor(helpIcon.getImage(),
            new Point(0, 0), ""));
      }
    });
    contents.add(help);
    // настраиваем наш интерфейс и прозрачную панель
    setContentPane(contents);
    setGlassPane(ip);
```

```
    // выводим окно на экран
    setSize(200, 200);
    setVisible(true);
  }
  // компонент, перехватывающий события
  class InterceptPane extends JComponent {
    InterceptPane() {
      // надо включить события от мыши
      enableEvents(MouseEvent.MOUSE_EVENT_MASK);
      enableEvents(KeyEvent.KEY_EVENT_MASK);
      // по умолчанию невидим и прозрачен
      setVisible(false);
      setOpaque(false);
    }
    // перехватываем события от мыши
    public void processMouseEvent(MouseEvent e) {
      // отслеживаем нажатия мыши
      if ( e.getID() == MouseEvent.MOUSE_PRESSED) {
        // определяем, какой компонент был выбран
        Component contentPane = getContentPane();
        MouseEvent ne =
            SwingUtilities.convertMouseEvent(
                this, e, contentPane);
        // видимый компонент в указанных координатах
        Component visibleComp =
            SwingUtilities.getDeepestComponentAt(
                contentPane, ne.getX(), ne.getY());
        // показываем справочную информацию
        JOptionPane.showMessageDialog(
            null, hs.getHelpFor(visibleComp));
        // отключаемся
        setVisible(false);
        // возвращаем на место обычный указатель мыши
        getRootPane().setCursor(
            Cursor.getDefaultCursor());
      }
    }
  }
  // прототип системы помощи
  class HelpSystem {
    // получает помощь для компонентов
```

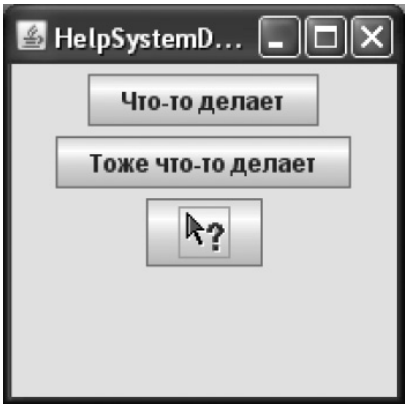
```
    public String getHelpFor(Component comp) {
      if ( comp == button1)
        return "Останавливает реактор. Лучше не жмите";
      else if ( comp == button2 )
        return "Хотите лимонада? Тогда жмите смело!";
      return "Даже и не знаю, что это такое";
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new HelpSystemDemo(); } });
  }
}
```

В данном примере мы создаем небольшое окно с тремя кнопками, две из которых представляют собой обычные элементы пользовательского интерфейса (здесь для простоты они ничего не делают), а третья, со специальным значком[3] вместо надписи, включает режим получения помощи, в котором элементы интерфейса перестают реагировать на нажатие кнопок мыши, вместо них манипуляции кнопками контролирует прозрачная панель, вызывая для выбранного компонента систему помощи. Чтобы реализовать подобное поведение, нам потребовалось несколько этапов. Во-первых, мы создали специальный компонент InterceptPane, унаследовав его от базового класса JComponent. Заметьте, что мы сразу же указываем в конструкторе, что наш компонент является прозрачным (невидимым) — свойство непрозрачности меняется на false (как было отмечено в главе 4, по умолчанию оно равно true). Интересно, что здесь же, в конструкторе нашего компонента, нам приходится вручную включать режим получения им сообщений от мыши (методом enableEvents()). Мы не добавляем слушателей, переопределяя вместо этого метод processMouseEvent(), что в нашем случае удобнее (это позволяет получить доступ сразу ко всем событиям от мыши), и поэтому события не включаются автоматически (маскирование событий мы обсуждали в главе 3). Компонент InterceptPane будет использоваться как прозрачная панель нашего приложения, и в его методе processMouseEvent() мы будем перехватывать нажатия кнопок мыши, вызывая при этом помощь для компонента, в области которого было зафиксировано нажатие кнопки. Роль справочной системы играет внутренний класс HelpSystem, который в нашем примере просто возвращает для каждого компонента небольшое сообщение, а в реальности может представлять собой полнофункциональную систему помощи.

В итоге мы получаем следующую картину: после запуска приложения прозрачная панель (представленная экземпляром класса InterceptPane) невидима и не перехватывает событий от мыши (вы можете свободно щелкать на первых двух кнопках). Если же вы щелкните на третьей кнопке, которая делает прозрачную панель видимой (а заодно заменяет указатель мыши более подходящим, напоминающим значок, который находится на кнопке), последняя начинает перехватывать нажатия кнопок мыши. После этого попытка щелчка мышью на любой из первых двух кнопок вызывает систему помощи, однако сами кнопки при этом и не подумают пошевелиться, потому что событие до них не доходит. Для этого в методе

[3] Этот значок взят из стандартного набора графики *Java Look And Feel Graphics Repository*, распространяемого сайтом java.sun.com и созданного специально для внешнего вида Metal.

processMouseEvent() мы делаем следующее: выясняем, что произошло нажатие кнопки мыши, а не что-либо еще (используя идентификатор типа события) и начинаем поиск компонента, на котором пользователь щелкнул мышью. Здесь нам помогает метод класса SwingUtilities getDeepestComponentAt(), позволяющий узнать, находится ли какой-либо видимый компонент в указанной точке. Он сделает всю работу, в том числе пройдет по всем вспомогательным панелям и контейнерам. Правда, напрямую передавать в него точку, хранящуюся в событии MouseEvent, нельзя, потому что эта точка находится в пространстве координат прозрачной панели, а метод getDeepestComponentAt () мы хотим применить в пространстве координат панели содержимого. Как раз для таких ситуаций в классе SwingUtilities имеется статический метод convertMouseEvent(), создающий новый объект MouseEvent в нужной системе координат. Далее мы вызываем помощь для компонента, на котором был выполнен щелчок мыши (если такой компонент обнаруживается). В заключение мы делаем прозрачную панель невидимой, чтобы интерфейс мог снова получать события, и меняем указатель мыши на обычный. Это поможет пользователю определить, что интерфейс снова действует в обычном режиме.




Наш пример — всего лишь прототип системы помощи, однако его легко приспособить под нужды реальных приложений. Достаточно заменить класс HelpSystem, использовав вместо него более мощную справочную систему, основанную, например, на языке разметки HTML, поддерживаемом в Swing, и придумать более удобный способ сопоставления компонента и справки для него. Удачным решением может быть обращение к клиентским свойствам компонентов Swing; с их помощью вы можете задать для каждого компонента (не наследуя от него) уникальный идентификатор, который и будет задействован в системе помощи для поиска нужной информации[4].

Так как пример намеренно оставлен простым для понимания и акцентируется именно на прозрачной панели, у него имеется несколько изъянов, свойственных таким «примитивным» применениям прозрачной панели. Прежде всего, не перехватываются события от клавиатуры, и даже после вызова справочной системы вы можете перемещаться и активировать кнопки с помощью клавиатуры, что, как правило, нежелательно. Более гибкое поведение предлагает новый компонент библиотеки Swing JXLayer, который мы вскоре рассмотрим.

[4] Если вы думаете о приложении с мощной справочной системой, попробуйте использовать пакет *JavaHelp*, созданный компанией Sun; вы сможете найти его на сайте java.sun.com. Он наделен всеми возможностями современных справочных систем и легко встраивается в приложение.

Вы видите, что прозрачная панель с легкостью позволяет вам работать поверх пользовательского интерфейса программы, как бы он ни был сложен. Мы уже придумали ей несколько применений, можно еще добавить, что упрощается создание обучающих систем: вы можете временно отрезать события от большинства компонентов, рисуя подробные комментарии поверх них и позволяя пользователю постепенно знакомиться с интерфейсом программы. Это может быть очень эффектно и способно значительно поднять привлекательность вашего приложения. Сделать это довольно просто – чтобы задать прозрачной панели форму, в пределах которой она будет перехватывать события от мыши, необходимо применить метод contains(), который мы обсуждали в главе 2. Рисовать же вполне можно в пределах всего окна, особенно полупрозрачными кистями.

Правда, прежде чем закончить разговор о прозрачной панели и всех ее прекрасных качествах, необходимо сказать, что это *общий ресурс* любого окна Swing, и в каждом окне она одна. Вы не можете быть уверены в том, что другие части приложения или даже сторонние библиотеки не захотят заменить прозрачную панель на свою собственную или поменять ее поведение. Более того, сама библиотека «тайно» пользуется прозрачной панелью для внутренних целей, прежде всего в многооконных приложениях. Так что, если у вас есть способ не пользоваться прозрачной панелью, лучше этого лишний раз не делать, а если и делать, то возвращать стандартную панель «на место». Альтернативы в виде многоуровневой панели и компонента JXLayer мы вскоре рассмотрим.

## Корневая панель — итог

Теперь, когда мы рассмотрели все составные части корневой панели и узнали их предназначение, можно наглядно представить себе, из чего она состоит (рис. 6.4)

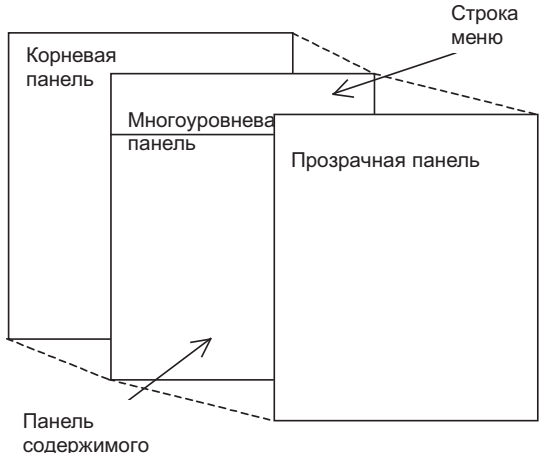



**Рис. 6.4.** Корневая панель

Сама корневая панель JRootPane представляет собой контейнер, унаследованный от базового класса Swing JComponent. В этом контейнере действует специальный алгоритм расположения компонентов, или *менеджер расположения* (см. главу 7), реализованный во внутреннем классе RootPaneLayout. Как раз этот менеджер расположения и отвечает за то, чтобы все составные части корневой панели располагались так, как им следует: многослойная панель занимает все пространство окна, в ее слое FRAME_CONTENT_LAYER

мирно уживаются строка меню и панель содержимого (которая в дальнейшем послужит для размещения компонентов пользовательского интерфейса программы), а над всем этим в стопке находится прозрачная панель.

Корневая панель позволяет получить или изменить все свои составляющие части, для этого у нее имеется набор методов get/set (например, уже использованные нами методы getContentPane() и setContentPane()). В любом контейнере высшего уровня Swing вы можете получить задействованную в нем корневую панель, вызвав метод getRootPane(). Для удобства в контейнерах высшего уровня также имеются методы get/set, позволяющие напрямую получать или изменять составные части корневой панели. Кроме контейнеров высшего уровня, которые мы уже упоминали, корневая панель также применяется во внутренних окнах JInternalFrame, создаваемых в многодокументных приложениях и располагающихся на «рабочем столе» JDesktopPane. Это позволяет забыть про то, что данные «окна» представляют собой обычные легковесные компоненты, и работать с ними как с настоящими контейнерами высшего уровня, а также обеспечивает их правильное поведение.

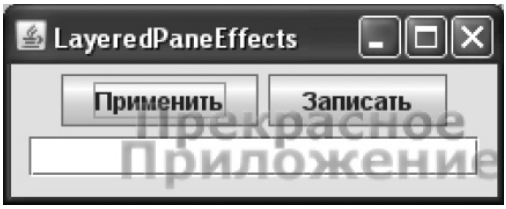
Напоследок стоит сказать, что корневая панель все же представляет собой вспомогательный компонент, и не так часто явно используется в приложениях. Тем не менее, мы уже видели примеры, когда составные части корневой панели помогают при реализации нестандартных решений, и сейчас рассмотрим еще один пример с участием многослойной панели. Мы узнаем, как можно эффектно разнообразить интерфейс несколькими строчками кода, не прибегая к созданию собственного внешнего вида и поведения программы, и придать своему приложению уникальный узнаваемый облик:

```
// LayeredPaneEffects.java
// Создание эффектов для интерфейса
// с помощью многослойной панели
import javax.swing.*;
import java.awt.*;

public class LayeredPaneEffects extends JFrame {
  public LayeredPaneEffects() {
    super("LayeredPaneEffects");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // несколько обычных кнопок и текстовое поле
    JPanel buttons = new JPanel();
    buttons.add(new JButton("Применить"));
    buttons.add(new JButton("Записать"));
    buttons.add(new JTextField(20));
    // добавляем в панель содержимого
    getContentPane().add(buttons);
    // добавляем компонент с анимацией в слой PALETTE
    Animation an = new Animation();
    an.setBounds(50, 10, anim.getWidth(this),
        anim.getHeight(this));
    getLayeredPane().add(an, JLayeredPane.PALETTE_LAYER);
    // выводим окно на экран
```

```
    setSize(250, 100);
    setVisible(true);
  }
  // изображение
  private Image anim =
      new ImageIcon("anim.gif").getImage();
  // компонент, рисующий анимированное изображение
  class Animation extends JComponent {
    public Animation() {
      setOpaque(false);
    }
    public void paintComponent(Graphics g) {
      Graphics2D g2 = (Graphics2D)g;
      // полупрозрачность
      g2.setComposite(AlphaComposite.getInstance(
          AlphaComposite.SRC_OVER, 0.3f));
      // рисуем изображение
      g2.drawImage(anim, 0, 0, this);
    }
    // мы никогда не получаем событий от мыши
    public boolean contains(int x, int y) {
      return false;
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
     new Runnable() {
      public void run() { new LayeredPaneEffects(); } });
  }
}
```

В примере мы создаем окно JFrame, в которое добавляем несколько обычных элементов пользовательского интерфейса: пару кнопок и текстовое поле. Затем в слое PALETTE многослойной панели размещается специальный компонент Animation, который является прозрачным (мы устанавливаем его свойство непрозрачности равным false) и занимается тем, что прорисовывает на экране анимированное изображение (в формате GIF, который полностью поддерживается Java). В методе paintComponent() этого компонента мы используем возможности библиотеки Java2D, чтобы сделать изображение полупрозрачным (иначе оно слишком сильно будет загораживать обычный интерфейс). Обратите внимание на метод contains(), который определяет, попадает ли указанная точка экрана в область компонента. Так как у нас прозрачный компонент, необходимо всегда говорить системе событий что ни одна точка в компонент не попадает, а иначе у нас будут проблемы, например, с курсором в текстовом поле, так как курсор будет использоваться от нашего декоративного компонента, а не текстовый.

Запустив приложение, вы увидите довольно интересную картину: пользовательский интерфейс программы работает как обычно, однако поверх него располагается простое анимированное изображение, которое придает приложению совершенно новый облик.




Может возникнуть справедливый вопрос, почему же мы не поместили изображение в прозрачную панель. Конечно, это возможно, однако многослойная панель может быть более гибкой. Мы поместили наш анимационный компонент в слой PALETTE, который находится ниже, чем слой POPUP, предназначенный для всплывающих меню. Если мы присоединим к текстовому полю такое меню, оно будет показано над нашей анимацией, что более логично. Тоже самое относится и к операциям перетаскивания, также отображаемых в многослойной панели. А используй мы прозрачную панель, которая «нависает» над всем окном, у нас не было бы простой возможности решить, какие компоненты и операции анимация перекрывает, а какие нет. Более того, мы уже знаем что прозрачная панель является общим ресурсом и представляет собой один единственный компонент, в то время как в многоуровневую панель можно добавить сколь угодно много компонентов подобных тем что мы создали в примере. Перефразируя, можно сказать, что если есть возможность не применять прозрачную панель, лучше этого не делать, и мы здесь последовали этому примеру.

Это всего лишь пример подобных эффектов, если включить в работу дизайнеров и фантазию, подобное применение корневой панели и ее составных элементов может придать вашему приложению невероятный облик. Впрочем, не стоит забывать о том, что пользовательский интерфейс, прежде всего, должен быть функциональным и простым. Лучше выбирать золотую середину между эффектностью и эффективностью.

## Расширение границ — J(X)Layer

Корневая панель, доступная в любом контейнере высшего уровня Swing, дает нам прекрасные возможности, в том числе способ перехватывать события от мыши, рисовать над остальными компонентами украшения и применять какие-либо художественные эффекты. Однако, при рассмотрении прозрачной панели мы сразу же увидели первый недостаток — практически невозможно «отключить» события от клавиатуры, и таким образом создать некоторый аналог легковесных модальных диалогов, перехватывающих все события от основного интерфейса. Также нет возможности «видеть» события не от конкретных компонентов, а от некоторой области на экране, в которой может находиться множество таких компонентов. Сама система событий Swing и AWT построена так, что все события приходят к конкретным компонентам, а не в какую-то ограниченную область.

Специально, чтобы избавиться от всех этих ограничений и наделить Swing-приложения расширенными возможностями, Александр Поточкин из команды разработчиков Swing создал инструмент под названием JXLayer. Поначалу этот инструмент развивался отдельно, а потом стал постепенно переводиться в библиотеку Swing в версии JDK 1.7 под названием JLayer. На момент написания книги отдельный

инструмент, который вы можете бесплатно скачать и присоединить к приложению в виде JAR-файла, обладает более широкими возможностями, так что рассмотрим мы его, ну а в дальнейшем мы вольны выбирать его или стандартную реализацию в JDK.

Суть инструмента JXLayer проста — часть уже имеющегося интерфейса, например отдельная панель, или весь он, это может быть панель содержимого вашего окна, помещается в JXLayer как в контейнер, а над ним размещается *слой* (layer, отсюда и название), в котором вы, во-первых, можете рисовать все, что захотите над всем интерфейсом, во-вторых, можете получать все события, которые над ним происходят. Разумеется, JXLayer необходимо разместить на экране, как правило вместо того интерфейса, который вы собираетесь дополнить слоем. JXLayer можно вкладывать друг в друга, получая несколько слоев. Дополнительные варианты слоев позволяют получить новые возможности, к примеру слой с буферизацией (BufferedLayerUI) сохраняет всю картину интерфейса в виде изображения в памяти, и вы можете применить к нему эффекты и преобразования (размыть, перевести в черно-белый цвет и т.п.). Интересным вариантом слоя является LockableUI, позволяющий не только отключить все события для компонента «под ним», но и сразу же применить какие-либо эффекты к нему как к изображению, что очень удобно при создании модальных эффектов и легковесных диалогов.

Рассмотрим пример, чтобы удостовериться, что все сказанное о JXLayer правда и работает на самом деле:

```
// JXLayerTest.java
// Пример использования слоев JXLayer
import org.jdesktop.jxlayer.JXLayer;
import org.jdesktop.jxlayer.plaf.BufferedLayerUI;
import javax.swing.*;
import java.awt.*;
import java.awt.event.MouseEvent;

public class JXLayerTest extends JFrame {
  public JXLayerTest() {
    super("JXLayerTest");
    // выход при закрытии окна
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создадим простое содержимое
    JPanel p = new JPanel();
    p.add(new JTextField(10));
    p.add(new JButton("OK"));
    // поместим содержимое в слой
    JXLayer<JPanel> layer = new JXLayer<JPanel>(p);
    // создадим слой реагирующий на события
    layer.setUI(new BufferedLayerUI<JPanel>() {
      private int lastX, lastY;
      // прорисовка слоя
      public void paint(Graphics g, JComponent c) {
```

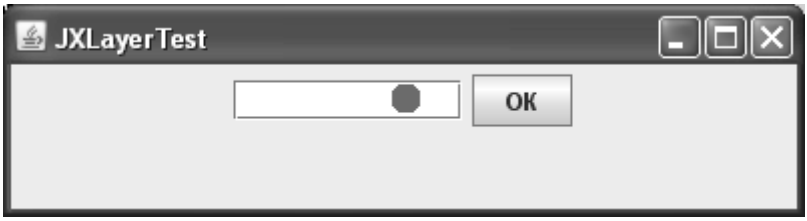
```
        super.paint(g, c);
        g.setColor(Color.RED);
        g.fillOval(lastX, lastY, 15, 15);
      }
      // получение событий в пределах слоя
      protected void processMouseMotionEvent(MouseEvent e,
                JXLayer<? extends JPanel> layer) {
        lastX = SwingUtilities.convertMouseEvent(
            (Component) e.getSource(), e, layer).getX();
        lastY = SwingUtilities.convertMouseEvent(
            (Component) e.getSource(), e, layer).getY();
        repaint();
      }
    });
    // JXLayer необходимо добавлять в контейнер
    add(layer);
    // выведем окно на экран
    setSize(400, 150);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new JXLayerTest(); } });
  }
}
```

Здесь мы создаем окно, в котором хотим разместить текстовое поле и кнопку, однако не напрямую, а вместе со слоем. При работе с инструментом JXLayer сразу стоит отметить, что он практически полностью параметризован, то есть требует указать при создании, какой конкретно тип компонента будет покрыт слоем, этот же тип передается и в сам слой. В нашем случае это будет панель JPanel, в которой будут располагаться компоненты. Ее необходимо передать в JXLayer как «основу», а метод setUI() задает слой в виде объекта LayerUI.

В нашем примере мы унаследовали от стандартного слоя BufferedLayerUI, хранящего изображение интерфейса в памяти, и переопределили метод для рисования paint(), и метод, получающий события о передвижениях мыши в пределах слоя. Обратите внимание, что у слоев компонента JXLayer нет слушателей, есть лишь знакомые нам по системе событий методы processXXXEvent(). Мы следим за перемещениями мыши в пределах слоя и рисуем рядом с курсором маленький красный кружок. Заметьте, что события, происходящие в слое, все равно распределены по компонентам, и если мы хотим получить координаты в слое, а не в текстовом поле или кнопке, нам всегда необходимо преобразовывать их с помощью класса SwingUtilities. Так устроена система событий Swing, и это надо учитывать при «глобальной» обработке событий от нескольких компонентов.

По окончанию настройки компонент JXLayer добавляется в центр окна и последнее выводится на экран. Запустив пример, вы увидите, что интерфейс функционирует как

обычно, получая все события, слой же получает их все, и способен изменять визуальное представление интерфейса на экране, рисуя на экране красный кружок у курсора вне зависимости от того, над каким компонентом находится мышь. Кстати, для запуска примера вам понадобится скачать библиотеку JXLayer из Интернета и добавить ее в свой системный путь CLASSPATH.




Как мы видим, компонент JXLayer прекрасно справляется с несовершенствами других способов работы поверх множества компонентов. Он не является разделяемым ресурсом системы и не требует осведомленности обычных компонентов о присутствии дополнительного слоя. Вы сможете применить к интерфейсу любые эффекты, в том числе эффект сворачивания и разворачивания страниц, размытие, постепенное проявление и любые другие. В простых же случаях можно обратить особое внимание на слой LockableUI, который парой методов позволяет отключить интерфейс от событий и применить к нему простые графические эффекты. Очень удобно при создании дополнительных эффектов внутри одного окна, контейнера или даже панели без лишних затрат.

# *Окна Swing*

Окна — это основа пользовательского интерфейса любой современной операционной системы. Они визуально разделяют выполняемые пользователем задачи, позволяя ему работать более эффективно. Мы уже отмечали, что окна, используемые в библиотеке Swing, мало чем отличаются от окон библиотеки AWT (а те в свою очередь представляют собой окна операционной системы), от которых наследуют, исключая корневую панель, в которой нам и приходится размещать все компоненты своего пользовательского интерфейса. Все окна библиотеки Swing — а к ним относятся окно без рамки JWindow, окно с рамкой JFrame и диалоговое окно JDialog — являются исключением из правила, согласно которому все компоненты Swing представляют собой легковесные компоненты и унаследованы от базового класса JComponent. Они наследуют напрямую от окон AWT и являются тяжеловесными контейнерами. Мы уже обсуждали, что это необходимо для легковесных компонентов — операционная система их не видит, так что размещать такие компоненты нужно в тяжеловесных контейнерах, иначе они не смогут прорисовываться и получать информацию о событиях. Для этого и предназначены окна Swing. Знакомиться с ними начнем мы с «родителя» всех окон Swing — окна без рамки JWindow.

## Окно без рамки JWindow

Класс JWindow представляет собой окно без рамки и без элементов управления (предназначенных, к примеру, для его закрытия или перемещения), оно дает пользователю минимальные возможности по своей настройке (в отличие от используемого чаще всего окна JFrame). Окно без рамки не так уж часто требуется в программах, однако иногда оно может быть полезно, особенно в тех случаях, когда вам необходимо ненадолго вывести на экран какую либо информацию, такую как заставку своей программы или подсказку для пользователя, и управлять окном с этой информацией не

нужно. К примеру, окна JWindow используются всплывающими меню JPopupMenu в тех ситуациях, когда в окне приложения не хватает места для размещения легковесного компонента в слое POPUP многослойной панели, где всплывающие меню располагаются по умолчанию. В такой ситуации вместо легковесного компонента создается небольшое окно без рамки JWindow, которое можно разместить в любом месте экрана, потому что оно принадлежит операционной системе. В этом окне и размещается всплывающее меню.

Мы рассмотрим один из вероятных способов применения окна без рамки JWindow — в качестве постоянно висящей на переднем плане рабочего стола «панели инструментов», которую нельзя закрыть или скрыть. Давайте попробуем:

```
// FloatingWindow.java
// Окно без рамки, всегда остающееся
// выше основного окна приложения
import javax.swing.*;


public class FloatingWindow extends JFrame {
  public FloatingWindow() {
    super("FloatingWindow");
    // выход при закрытии окна
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // выведем главное окно на экран
    setSize(400, 300);
    setVisible(true);
    // добавим плавающее окно
    JWindow window = new JWindow(this);
    // всегда над другими окнами
    window.setAlwaysOnTop(true);
    // выведем окно на экран
    window.setSize(100, 300);
    window.setVisible(true);

  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() { new FloatingWindow(); } });
  }
}
```

В этом примере мы наследуем от окна JFrame и выводим его на экран. Далее создается дочернее окно JWindow, которому мы передаем ссылку на родительское окно. Используя свойство alwaysOnTop, мы просим систему держать наше окно без рамки выше всех остальных окон и также выводим его на экран. Если вы уже пользуетесь выпуском Java 7, вы также сможете задействовать свойство type, чтобы поменять тип окна на вспомогательное (utility). Запустив пример, вы убедитесь в том, что непритязательный

серый прямоугольник действительно всегда перекрывает остальные окна вашего рабочего стола.

Обратите внимание на то, что, как правило, окна без рамки JWindow требуют при создании указывать своего «родителя»[5] — окно с рамкой JFrame или другое окно, унаследованное от класса Window, что иногда может быть неудобно. Специально для таких случаев в класс JWindow был добавлен конструктор без параметров, который создает вспомогательное невидимое окно JFrame и использует его как «родителя». После этого все окна без рамки, созданные таким образом, задействуют только это окно и экономят ресурсы. Стоит также сказать, что с помощью конструктора без параметров создается окно JWindow, неспособное получать фокус ввода. Чаще всего именно такое поведение необходимо (ведь панелям инструментов, всплывающим заставкам и меню фокус ввода не нужен), а если вы хотите исправить ситуацию и получать в свое окно без рамки фокус ввода, используйте метод setFocusableWindowState(true).

Рассказать что-либо очень интересное об окне без рамки JWindow и остальных окнах Swing очень трудно, потому что они являются лишь тонкой оболочкой для окон операционной системы, к которым у вас нет доступа (так сделано для лучшей переносимости — вы можете использовать только те свойства, которые гарантированно поддерживаются на всех платформах). Чаще всего вы просто создаете окна и добавляете в принадлежащую им корневую панель свои компоненты. Более того, лучше вообще свести к минимуму количество разнообразных плавающих окон в своем приложении, тем более что у нас нет достаточного контроля над ними, а любой дополнительный необязательный объект приложения на экране, да еще и «висящее» окно, крайне раздражает. Поэтому мы ограничимся списком методов, наиболее часто используемых для окон Swing (табл. 6.2).

**Таблица 6.2.** Наиболее полезные методы окон Swing

| Методы | Описание |
|---|---|
| setLocation(), setSize(), setBounds() | Эта группа методов позволяет задать позицию и размеры окна на экране (на рабочем столе пользователя). Первый метод задает позицию окна, второй позволяет указать его размеры, а с помощью третьего вы сможете сразу задать прямоугольник, который займет ваше окно на экране |
| pack() | Позволяет «упаковать» имеющиеся в окне компоненты, так чтобы они занимали столько места, сколько им необходимо, а не больше и не меньше. Подробнее об этом методе мы узнаем в главе 5. Интересно, что компоненты при вызове этого метода переходят в «видимое» состояние, то есть начинают использовать ресурсы системы, хотя и не появляются на экране до вызова следующего метода |
| setVisible() | Выводит ваше окно на экран или скрывает его с экрана (если параметром указано значение false) |
| dispose() | Убирает окно с экрана (если оно в момент вызова метода видимо) и освобождает все принадлежащие ему ресурсы. Если в вашем приложении много окон, во избежание нехватки памяти не забывайте вызывать этот метод для тех из них, что больше не используются |

[5] Здесь в библиотеке небольшая путаница: класс JFrame унаследован от JWindow, и в тоже время конструктор JWindow требует передачи параметра JFrame.

## Окно с рамкой JFrame

Окно с рамкой JFrame унаследовано от класса JWindow и представляет собой наиболее часто используемое в приложениях окно «общего назначения». В отличие от окон JWindow окна JFrame обладают рамкой (которая позволяет пользователям легко изменять их размер), заголовком с названием приложения (хотя можно оставлять этот заголовок пустым), иногда системным меню (позволяющим проводить манипуляции с этим окном) и кнопками для управления окном (чаще всего для его закрытия и свертывания). Именно класс JFrame применяется в подавляющем большинстве приложений для размещения компонентов пользовательского интерфейса. Прежде почти всегда при создании окна с рамкой приходилось заботиться о том, чтобы при щелчке пользователем на кнопке закрытия окна приложение заканчивало работу, а необходимость даже для самого скромного приложения писать слушателя событий от окна сильно раздражала. Создатели Swing услышали ропот недовольных программистов и добавили в класс JFrame специальный метод, позволяющий упростить эту операцию. Рассмотрим небольшой пример:

```
// FrameClosing.java
// Использование окна с рамкой
import javax.swing.*;

public class FrameClosing extends JFrame {
  public FrameClosing() {
    super("Заголовок Окна");
    // операция при закрытии окна
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // значок для окна
    setIconImage(getToolkit().getImage("icon.gif"));
    // вывод на экран
    setSize(300, 100);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
     new Runnable() {
      public void run() { new FrameClosing(); } });
  }
}
```

Это, наверное, самый простой пример в книге. Мы создаем в нем подкласс JFrame, указываем заголовок окна (в конструкторе базового класса, хотя можно было бы использовать и метод setTitle()) и, прежде чем задать размеры окна и вывести его на экран, вызываем метод setDefaultCloseOperation(). Он позволяет указать, какое действие будет произведено в методе предварительной обработки событий processWindowEvent() при закрытии окна. По умолчанию применяется константа HIDE_ON_CLOSE, убирающая с экрана окно при его закрытии. Мы использовали (и на протяжении книги почти всегда будем использовать) значение EXIT_ON_CLOSE, которое указывает, что при закрытии окна необходимо закончить работу приложения. Остальные варианты вы сможете найти в интерактивной документации. В книге имеется множество примеров размещения компонентов вашего пользовательского интерфейса в окне с рамкой, даже в этой главе

мы уже успели несколько раз его встретить, поэтому мы не станем рассматривать здесь еще один пример. Остается лишь сказать о довольно полезном методе setIconImage(), позволяющем задать значок для вашего окна (в зависимости от платформы он может находиться только на кнопке свернутого окна или в заголовке окна в нормальном состоянии). Собственный значок позволит вашему приложению «выделиться», так что пользователь легче его запомнит.

Из дополнительных возможностей окна с рамкой JFrame можно упомянуть о его способности «прятать» свои «украшения»: рамку и элементы управления окном. Делает это метод setUndecorated(true). После его вызова окно JFrame будет разительно похоже на уже знакомое нам окно без рамки. Пока непонятно, зачем нужно это делать, но вскоре мы найдем применение данному методу (правда, вызывать его нужно до того, как окно появится на экране). Также стоит упомянуть метод setExtendedState(). Он позволяет задать состояние окна, например, свернуть его, но к сожалению работает на разных платформах по-разному, так что совсем полагаться на него не стоит, но всегда можно узнать, какие операции с окном поддерживаются на данной платформе с помощью метода isFrameStateSupported() из класса Toolkit.

## События окон

Окна Swing (JWindow, JFrame, а также еще не рассмотренное нами диалоговое окно JDialog) поддерживают три типа событий: первое, представленное слушателем WindowListener, позволяет узнать об изменениях в состоянии окна, второе, со слушателем WindowFocusListener, генерируется системой передачи фокуса ввода и сообщает о получении или потере компонентами окна фокуса ввода, наконец, третье, со слушателем WindowStateListener, сообщает о изменении состояния окна, например сворачивании. Полный список методов данных слушателей вы сможете найти в интерактивной документации Java, в дополнение к этому можно лишь сказать, что от событий окон немного толку. Без сомнения, вы сможете узнать обо всех изменениях в состоянии окна, но при этом у вас почти нет рычагов управления окнами, потому что они полностью принадлежат операционной системе. В результате обычно вы работаете с компонентами Swing, а окно используете просто как площадку для их размещения.

В интерфейсе слушателя WindowListener довольно много методов, и они редко требуются все вместе. Чаще остальных применяют метод windowClosing(), вызываемый системой событий при попытке пользователя закрыть окно, так что вы можете провести какие-то действия перед закрытием окна (например, запомнить все документы и настройки приложения) или спросить пользователя, уверен ли он в том, что хочет закончить работу. Рассмотрим небольшой пример с этим методом:

```
// ConfirmClosing.java
// Подтверждение о выходе из приложения
import javax.swing.*;
import java.awt.event.*;

public class ConfirmClosing extends JFrame {
  public ConfirmClosing() {
    super("Приложение");
    // отключаем операцию закрытия
    setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
    // добавляем слушателя событий от окна
    addWindowListener(new WindowAdapter() {
```

```
      @Override
      public void windowClosing(WindowEvent e) {
        // потверждение выхода
        int res = JOptionPane.
         showConfirmDialog(null, "Действительно выйти?");
        if ( res == JOptionPane.YES_OPTION )
         System.exit(0);
      }
      });
    // выводим окно на экран
    setSize(200, 100);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
     new Runnable() {
      public void run() { new ConfirmClosing(); } });
  }
}
```

Мы создаем небольшое окно JFrame и добавляем к нему слушателя событий WindowEvent. Чтобы не реализовывать все определенные в интерфейсе WindowListener методы, мы наследуем от адаптера WindowAdapter и переопределяем нужный нам метод windowClosing(). При закрытии окна мы спрашиваем у пользователя, действительно ли он желает закончить работу с приложением, и завершаем работу, если ответ положительный. Кстати, обратите внимание, что нам приходится указывать в методе setDefaultCloseOperation(), что при закрытии окна ничего делать не надо (передавая в него константу DO_NOTHING_ON_CLOSE), иначе окно скроется с глаз пользователя при любой реакции последнего.

## Диалоговое окно JDialog

Диалоговые окна довольно часто используются в приложениях для получения информации, не имеющей прямого отношения к основному окну, например, для установки параметров, вывода важной вспомогательной информации, получения дополнительной информации от пользователя. Диалоговые окна могут быть *модальными* (modal) — они запрещают всем остальным окнам приложения получать информацию от пользователя, пока тот не закончит работу с модальным диалоговым окном. Модальные диалоговые окна полезны в тех ситуациях, когда информация, получаемая от пользователя, коренным образом меняет работу приложения и поэтому до ее получения продолжение работы невозможно. Внешний вид диалоговых окон мало отличается от окон с рамкой JFrame, но обычно у них меньше элементов управления (чаще всего, имеется только кнопка закрытия окна) и отсутствует системное меню. Как правило, диалоговые окна всегда располагаются поверх основного окна приложения, напоминая о том, что необходимо завершить ввод информации.

Диалоговые окна в Swing реализуются классом JDialog. Данный класс позволяет создавать обычные и модальные диалоговые окна, поддерживает (так же как и класс JFrame) закрытие окна, а в остальном сходен с другими окнами Swing, потому что унаследован от базового класса окон JWindow. При создании диалоговых окон Swing необходимо указы-

вать «родительское окно», которым может быть окно с рамкой JFrame или любое другое окно, унаследованное от базового класса окон Window (для удобства есть конструктор, не требующий «родительского» окна, но использующий вспомогательное невидимое окно, о котором мы говорили в разделе об окнах без рамки JWindow). Как уже отмечалось, диалоговое окно располагается на экране с учетом положения «родительского» окна. Рассмотрим небольшой пример, где создадим пару диалоговых окон:

```
// DialogWindows.java
// Диалоговые окна в Swing
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class DialogWindows extends JFrame {
  public DialogWindows() {
    super("DialogWindows");
    // выход при закрытии
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // пара кнопок, вызывающих создание диалоговых окон
    JButton button1 = new JButton("Обычное окно");
    button1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        JDialog dialog =
          createDialog("Немодальное",
              Dialog.ModalityType.MODELESS);
        dialog.setVisible(true);
      }
      });
    JButton button2 = new JButton("Модальное окно");
    button2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        JDialog dialog =
          createDialog("Модальное",
              Dialog.ModalityType.APPLICATION_MODAL);
        dialog.setVisible(true);
      }
      });
    // создаем панель содержимого и выводим окно на экран
    JPanel contents = new JPanel();
    contents.add(button1);
    contents.add(button2);
    setContentPane(contents);
    setSize(350, 100);
```

```
      setVisible(true);
    }
    // создает диалоговое окно
    private JDialog createDialog(String title, Dialog.ModalityType
  modal){
      JDialog dialog = new JDialog(this, title, modal);
      dialog.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
      dialog.setSize(200, 60);
      return dialog;
    }
    public static void main(String[] args) {
      SwingUtilities.invokeLater(
       new Runnable() {
        public void run() { new DialogWindows(); } });
    }
  }
```

В этом примере мы создаем окно с рамкой JFrame, в панель содержимого которого добавляем две кнопки JButton, щелчками на которых и будут создаваться диалоговые окна. Чтобы не дублировать в слушателях похожий код, мы создали отдельный метод createDialog(), создающий простое диалоговое окно с заданным заголовком и при необходимости модальностью. Заголовок диалогового окна, его «родитель» и признак модальности указываются в конструкторе[6]. При закрытии диалогового окна мы использовали операцию DISPOSE_ON_CLOSE, удаляющую окно после закрытия. Первая кнопка позволяет создать немодальное диалоговое окно, а вторая — его модальный вариант. Запустив приложение, вы увидите разницу: немодальные окна не запрещают работу с основным окном приложения, и вы можете снова и снова создавать их, не закрывая уже созданные окна. Такое поведение удобно там, где информация, находящаяся в диалоговом окне, не критична, или часто меняется, и каждый раз отвлекаться на модальное окно неудобно (хорошим примером может быть диалоговое окно смены шрифта в текстовом редакторе). Модальные окна запрещают вам работу со всеми остальными окнами приложения. Мы уже отмечали, что такое поведение требуется для получения важной информации, без которой невозможно продолжение работы, например, для получения пароля, позволяющего авторизовать пользователя, или запроса на выход из приложения.

Признак модальности задается с помощью перечисления (enum) из класса AWT Dialog, значения которого могут быть следующими:

**Таблица 6.3.** Виды модальности диалогов Swing

| Вид модальности | Описание |
|---|---|
| MODELESS | Диалоговое окно не препятствует работе остальных окон, в том числе и родительского, но всегда находится выше родительского окна |

[6] Нетрудно задать эти качества вашего диалогового окна и с помощью свойств title и modalityType. Смысл здесь в том, что при создании (вызове конструктора) диалогового окна вы чаще всего уже прекрасно знаете, каким оно должно быть.

**Таблица 6.3** *(продолжение)*

| Вид модальности | Описание |
|---|---|
| APPLICATION_MODAL | При создании и выводе такого диалогового окна все окна, уже созданные приложением (формально, тем же экземпляром виртуальной машины), будут заблокированы для пользователей. Дочерние диалоги и окна, создаваемые модальным диалогом, доступны пользователю |
| DOCUMENT_MODAL | Блокирует работу окон, принадлежащих к одному родительскому окну или иерархии окон, создавших друг друга, от самого первого родителя. Другие окна не из этой иерархии не блокируются. |
| TOOLKIT_MODAL | Предназначено для апплетов, блокирует все окна, созданные апплетами из одной страницы, даже если сами апплеты раздельные |

Кроме того, что вы можете создавать собственные диалоговые окна, используя класс JDialog, библиотека Swing предоставляет вам богатый набор стандартных диалоговых окон, используемых для получения и вывода несложной информации. Стандартные диалоговые окна хороши тем, что пользователи быстро привыкают к ним, их легко настраивать и применять в своих приложениях. Так что прежде чем создавать собственное диалоговое окно, всегда стоит подумать о возможности использовать стандартное. Стандартные диалоговые окна Swing мы будем обсуждать в главе 14.

## Специальное оформление окон

Окна Swing предоставляют вам возможность настраивать так называемое визуальное «оформление» окон: их рамку, элементы управления (такие как кнопки закрытия или свертывания), системное меню. Необходимость этого ощущалась с самых первых выпусков Swing после появления механизма подключаемых внешнего вида и поведения. Какой бы внешний вид и поведение для своей программы вы не использовали, уверенности в том, что она будет выглядеть и вести себя одинаково на любой платформе не было, потому что окна, в которых размещался интерфейс, находился вне вашей власти, и их внешний вид мог довольно сильно контрастировать с вашим интерфейсом. Особенно это было заметно при выборе внешнего вида, который не имитировал какую-либо известную платформу, такую как Windows или Macintosh, а применялся только для Swing (например, используемый по умолчанию внешний вид Metal). Внешний вид, имитирующий ту или иную платформу, чаще всего именно на этой платформе и использовался, поэтому окна у приложения были соответствующие. Но вот другие внешние виды не обеспечивали нужного оформления окон.

Имитация внешнего вида окон возможна, в первую очередь, благодаря UI-представителю корневой панели JRootPane. Если программист-клиент библиотеки Swing захочет особым образом оформить окна, UI-представитель корневой панели создаст специальные рамки, заголовок, системное меню и кнопки управления окном, подходящие для текущих внешнего вида и поведения, разместит их нужным способом в корневой панели, используя специализированный менеджер расположения, и сообщит частям корневой панели, что пространства в ней стало меньше, так что они не смогут занимать место, занятое рамками окна. Кроме того, при новом оформлении окон отключаются системные элементы окна (вот где пригодится метод setUndecorated()). Можно заставить корневую панель выглядеть как окно, и не отключая системные элементы и рамки, только вот смотреться это будет непривлекательно.

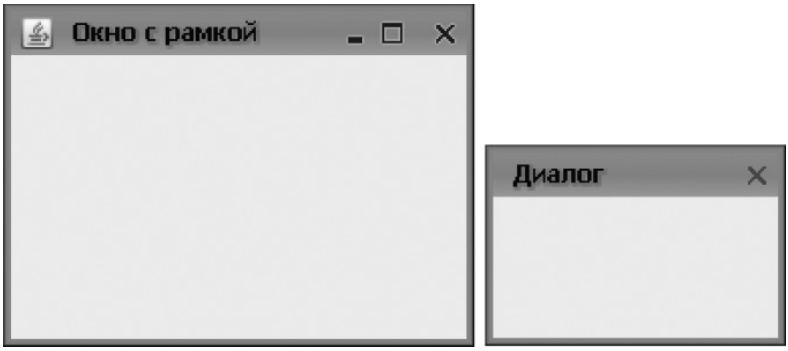
Чтобы не принуждать вас для оформления окон каждый раз настраивать корневую панель и убирать с окон имеющиеся «украшения», в классы JFrame и JDialog (окну JWindow оформление не нужно) был добавлен статический метод setDefaultLookAndFeelDecorated(), обеспечивающий возможность оформления всех создаваемых окон. Единственное, что

стоит отметить — оформление окон поддерживается не всеми менеджерами внешнего вида, это «необязательное» свойство. Однако почти все дополнительные подключаемые внешние виды для Swing поддерживают это свойство, так что использовать его имеет смысл, поскольку на любой платформе приложение приобретает законченный облик. Рассмотрим небольшой пример:

```
// WindowDecorations.java
// Специальное оформление окон Swing
import javax.swing.*;

public class WindowDecorations {
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() {
            // включим украшения для окон
            JFrame.setDefaultLookAndFeelDecorated(true);
            JDialog.setDefaultLookAndFeelDecorated(true);
            // окно с рамкой
            JFrame frame = new JFrame("Окно с рамкой");
            frame.setDefaultCloseOperation(
                JFrame.EXIT_ON_CLOSE);
            frame.setSize(200, 200);
            frame.setVisible(true);
            // диалоговое окно
            JDialog dialog = new JDialog(frame, "Диалог");
            dialog.setDefaultCloseOperation(
                JDialog.DISPOSE_ON_CLOSE);
            dialog.setSize(150, 100);
            // так можно задавать тип оформления окна
            dialog.getRootPane().setWindowDecorationStyle(
                JRootPane.INFORMATION_DIALOG);
            dialog.setVisible(true);
          }
        });
  }
}
```

В этом примере мы прямо в методе main(), ни от чего не наследуя, создаем простое окно с рамкой и диалоговое окно. Но еще до их создания вызывается метод setDefaultLookAndFeelDecorated(), означающий, что для создаваемых окон JFrame и JDialog потребуется специальное оформление. Далее мы просто настраиваем окна и выводим их на экран. Остается обратить внимание на метод корневой панели setWindowDecorationStyle(), позволяющий настроить тип оформления окна. Если окно с рамкой имеет только один тип оформления, то диалоговые окна в зависимости от их цели (вывод информации, сообщение об ошибке и т. д.) могут выглядеть по-разному. В нашем примере мы указали, что создаваемое диалоговое окно требу-

ется для вывода информации и должно выглядеть соответствующим образом. Запустив программу с примером, вы увидите, как оформляет свои окна внешний вид Java по умолчанию, а например, в известном стороннем внешнем виде Substance окна выглядят так:




Специальное оформление окон не только позволяет придать вашему приложению законченный облик, оно может быть полезно как средство, позволяющее полностью, вплоть до окон, управлять внешним видом вашего приложения. Создав собственного UI-представителя корневой панели, вы сможете придать своему приложению уникальный вид, легко узнаваемый пользователями.

## Прозрачность и произвольные формы

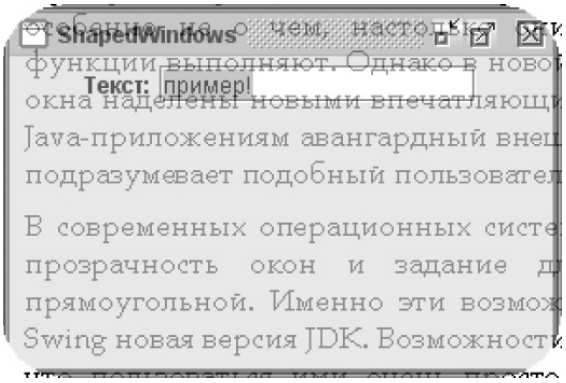
При разговоре об окнах все время закрадывается мысль, что и говорить особенно не о чем, настолько они просты и настолько вспомогательные функции выполняют. Однако в новой версии JDK 1.7 (или если угодно, Java 7), окна наделены новыми впечатляющими возможностями, способными придать Java-приложениям авангардный внешний вид, конечно, если само приложение подразумевает подобный пользовательский интерфейс.

В современных операционных системах уже довольно давно поддерживается прозрачность окон и задание для них произвольной формы вместо прямоугольной. Именно эти возможности и предоставляет для нас в Java и Swing новая версия JDK. Включаются они всего парой методов, так что пользоваться ими очень просто. Давайте сразу же посмотрим на все в действии:

```
// ShapedWindows.java
// Полупрозрачные окна произвольных форм
import javax.swing.*;
import java.awt.FlowLayout;
import java.awt.geom.RoundRectangle2D;

public class ShapedWindows extends JFrame {
  public ShapedWindows() {
    super("ShapedWindows");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // размер окна
    setSize(300, 200);
    // несколько компонентов в ряд
```

```
        setLayout(new FlowLayout());
        add(new JLabel("Текст:"));
        add(new JTextField(15));
        // задаем округлую форму
        RoundRectangle2D.Float roundedShape =
            new RoundRectangle2D.Float(0, 0, 300, 200, 70, 70);
        setShape(roundedShape);
        // задаем прозрачность
        setOpacity(0.7f);
        setVisible(true);
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
          public void run() {
            JFrame.setDefaultLookAndFeelDecorated(true);
            new ShapedWindows();
          }
        });
    }
}
```

В примере мы сразу же, еще до создания нашего окна с рамкой, указываем, что у всех окон должно быть специальное оформление, так мы лучше увидим особую форму нашего окна. В конструкторе в центр окна для наглядности добавляются надпись с небольшим текстовым полем, так что запустив пример, вы убедитесь что ввод текста работает несмотря ни на какие формы и прозрачность. Как мы видим, форма окна задается с помощью свойства shape, которое принимает объекты типа Shape из пакета java.awt.geom, описывающего разнообразные геометрические формы. Мы используем форму прямоугольника с закругленными краями. Прозрачность задается еще проще, нужно задать свойство opacity, в виде дробного числа от 0 (полностью прозрачное окно) до 1 (непрозрачное). У нас окно будет прозрачным примерно на 30%. Запустив пример, вы увидите какую форму принимает окно и что все его элементы функционируют как всегда. На иллюстрации окно находится над редактором с текстом книги.



Как вы видите, задать прозрачность и форму окна проще простого, однако стандартные внешние виды пока что еще не слишком то к этому приспособились (на иллюстрации с окном края окна обрезаны, кнопки обрезаны — внешний вид просто не принимает во внимание формы окна). Возможно, с окончательным выходом Java 7 некоторые сторонние внешние виды будут принимать во внимание форму окна и правильно рисовать его границы. Ну а пока создание окна произвольной формы подразумевает что вы будете рисовать его содержимое, или по крайней мере границы, вручную. Прозрачность же окон хороша и сама по себе, к примеру, вы можете делать вспомогательные окна более прозрачными, когда они теряют фокус ввода или становятся менее нужными, но убирать с экрана их нельзя.

И еще одно — полупрозрачные окна и окна произвольных форм могут не поддерживаться на некоторых платформах, и вызов рассмотренных нами методов приведет к исключениям. Если вы рассчитываете, что ваше приложение будет исполняться на самых разнообразных платформах, имеет смысл еще перед вызовом методов удостовериться в том, что они работают. Это можно сделать с помощью класса AWT GraphicsDevice, который позволяет узнать конфигурацию текущей графической системы.

## *Кратко об апплетах — класс JApplet*

В этой книге мы говорим о платформе Java и библиотеках Java Foundation Classes как о прекрасном средстве создания мощных и переносимых между платформами приложений. Однако в первых выпусках Java все средства разработки пользовательских интерфейсов в основном были направлены на создание *апплетов* (applets): небольших приложений, написанных в соответствии с простым программным интерфейсом и ориентированных на выполнение в браузере при просмотре HTML-страниц. Апплеты на самом деле были революционным шагом: они не зависели от браузеров и выполнялись на любой платформе с установленной виртуальной машиной Java, позволяя задействовать всю мощь Java внутри обычной HTML-страницы. Начиная с Java 2 (JDK 1.2), Sun предложила для выполнения апплетов концепцию *модуля расширения* (plug-in) Java. Этот модуль надо загрузить с официального сайта java.sun.com, что гарантирует получение самых свежих версий.

Однако нельзя сказать, что новая модель загрузки апплетов, предложенная Sun, повысила их популярность. Модуль расширения Java довольно объемен, и загружать его для выполнения некого апплета захочет не каждый пользователь. С другой стороны, возросли возможности и потенциал языков сценариев, встроенных в HTML-страницы, возможности самого языка HTML, чрезвычайно возросла популярность динамических страниц (AJAX), содержимое которых генерируется на сервере в зависимости от того, что хочет увидеть и получить пользователь и обновляется частями, без перезагрузки страницы. Все это получает название Web 2.0. Поэтому мы рассматриваем в книге создание отдельных Java-приложений, не ограниченных системой безопасности и браузером, к тому же приложения эти теперь можно легко распространять через Web (используя технологию Java Web Start) вместе с виртуальной машиной Java. Тем не менее, остаются ситуации, в которых использование апплетов с новыми возможностями оправдано, к примеру, встраивание в апплет уже готового мощного Swing-приложения для доступа к нему через Web. Рассмотрим вкратце процесс создания апплетов с использованием Swing.

Любой апплет (небольшое приложение, работающее в составе HTML-страницы), предполагающий задействовать компоненты Swing, должен быть унаследован от базового класса JApplet. Класс JApplet унаследован от класса Applet, описывающего инфраструктуру всех апплетов, и отличается от него лишь тем, что имеет в своем составе корневую панель, в которой вы размещаете свои компоненты. Класс JApplet, так же как и окна Swing, представляет собой тяжеловесный контейнер высшего уровня, именно он затем

размещается на HTML-странице. В апплетах нет метода main(), начинающего работу приложения, вместо этого вы переопределяете специальные методы, которые затем вызываются браузером при выводе вашего апплета на экран. Подробности об апплетах вы сможете найти в интерактивной документации Java, а сейчас мы рассмотрим небольшой пример, демонстрирующий процесс создания самого простого апплета:

```
// SwingApplet.java
// Простой апплет с использованием Swing
import javax.swing.*;
import java.awt.*;

public class SwingApplet extends JApplet {
  // этот метод вызывается при создании апплета
  @Override
  public void init() {
    // создание интерфейса
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() {
            JPanel contents = new JPanel();
            contents.add(new JTextField(10));
            contents.add(new JButton("Ввод"));
            setContentPane(contents);
        }});
  }
}
```

Как и полагается, мы унаследовали класс своего апплета от JApplet, и поместили код, создающий пользовательский интерфейс, в метод init(). Этот метод вызывается браузером при создании апплета, когда пользователь в первый раз попадает на страницу, где он размещен. Чаще всего именно в нем создается пользовательский интерфейс апплета. Обратите внимание, что данный метод вызывается *не* из потока рассылки событий, поэтому нам нужно вручную перейти в поток рассылки, как мы всегда делаем в методе main(). Создание интерфейса для апплета ничем не отличается от создания интерфейса для обычного приложения: те же компоненты, та же корневая панель. Заметьте, что благодаря корневой панели апплет может обладать строкой меню или быть многодокументным приложением, что для части HTML-страницы (а апплет на самом деле — всего лишь часть страницы) просто удивительно. Чтобы запустить апплет в браузере, необходимо поместить его на HTML-страницу, например, на такую:

```
<HTML>
<HEAD><TITLE>SwingApplet</TITLE></HEAD>
<BODY>
<APPLET code=SwingApplet height=100 width=200>
</APPLET>
</BODY>
</HTML>
```

Взглянув на страницу, вы увидите, что для запуска апплета необходимо указать его класс (в параметре code), а также размеры (аналогично тому, как мы задавали размеры окон).

Идея апплетов хороша, и если пользователь соглашается загрузить к себе на компьютер модуль расширения Java, апплеты могут принести много пользы: достаточно даже того, что они могут использовать для создания интерфейса все возможности библиотеки Swing. С другой стороны, модель безопасности апплетов (закрытая от системы «песочница») запрещает им работу с файлами и многими другими ресурсами системы, что предотвращает создание приложений, работающих с документами и данными пользователя. Во многих ситуациях проще создавать обычные приложения и распространять их с помощью технологии Java Web Start, свежую информацию, о которой вы сможете отыскать на официальном сайте Java по адресу java.sun.com.

## *Многооконное окружение*

Как мы уже выяснили в этой главе, окно в современных графических операционных системах используются для разделения задач, выполняемых пользователем, в одном из окон он может редактировать текст, в другом окне просматривать фотографии.

Как только концепция окон была изобретена и довольно успешно начала справляться с разделением задач пользователя на основную и второстепенные, возникло желание продвинуться немного дальше и применить ее уже внутри одного приложение, то есть разделить задачу внутри окна на несколько задач используя все те же окна, но на этот раз уже внутри другого, главного, окна. Подобные интерфейсы и называются многодокументными. Приложения, берущие на вооружение такие интерфейсы, как правило, способны выполнять задачи такого широкого спектра, что уместить все в одно окно становится затруднительно без ущемления удобства пользователя. На помощь приходят дополнительные окна, в которых можно разместить информацию, имеющую косвенное отношение к основной, выполняемой в данный момент, задаче.

Применение многодокументных интерфейсов на практике вызывает жаркие споры. Сам по себе интерфейс операционной системы является многодокументным приложением, и встраивание в него еще одного подобного интерфейса может запутывать пользователя. Окна операционной системы работают по своему, окна приложения по своему, и все это не прибавляет уверенности пользователю в том, что происходит и как этим следует управлять. Тем не менее, как показывает практика, полностью избавиться от дополнительных окон в сложных приложениях затруднительно (обычно пробуют размещать различные задачи приложения в большой панели с вкладками, которая, конечно же, есть и в Swing). Мы все-таки изучим возможности Swing в данном вопросе.

Для того чтобы организовать многодокументный интерфейс, библиотека Swing предлагает нам особый компонент, «рабочий стол» JDesktopPane, и так называемые «внутренние» окна JInternalFrame, которые на этом «столе» размещаются. И компонент JDesktopPane, и окна JInternalFrame являются легковесными, и мы не можем их назвать контейнерами высшего уровня, однако, по сути, внутренние окна абсолютно идентичны свои собратьям JFrame, поэтому мы и рассмотрим их здесь. Начнем с простого примера, который продемонстрирует нам, что «внутренние» окна ничем не отличаются от привычным нам окон операционной системы:

```
// SimpleMDI.java
// Демонстрация внутренних окон Swing
import javax.swing.*;
```

```
public class SimpleMDI extends JFrame {
  public SimpleMDI() {
    super("SimpleMDI");
    setSize(400, 300);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем рабочий стол Swing
    JDesktopPane desktopPane = new JDesktopPane();
    // добавляем его в центр окна
    add(desktopPane);
    // создаем несколько внутренних окон
    JInternalFrame frame1 =
        new JInternalFrame("Окно 1", true);
    JInternalFrame frame2 =
        new JInternalFrame("Окно 2", true, true, true, true);
    JInternalFrame frame3 =
        new JInternalFrame("Палитра", false, true);
    // смена типа окна на "палитру"
    frame3.putClientProperty("JInternalFrame.isPalette", true);
    // добавляем внутренние окна на рабочий стол
    desktopPane.add(frame1);
    desktopPane.add(frame2);
    desktopPane.add(frame3);
    // задаем размеры и расположения, делаем окна видимыми
    frame1.setSize(200, 100);
    frame1.setLocation(80, 100); frame1.setVisible(true);
    frame2.setSize(200, 60); frame2.setVisible(true);
    frame3.setSize(100, 200); frame3.setVisible(true);
    // выводим окно на экран
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new SimpleMDI(); } });
  }
}
```

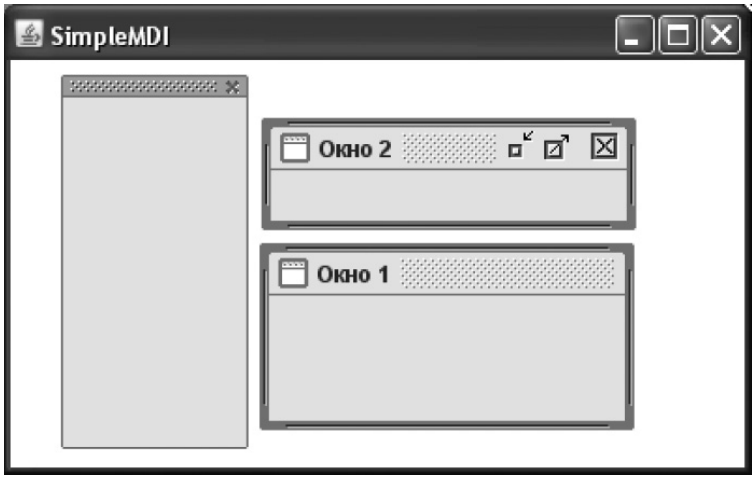
В обычном окне JFrame размещается рабочий стол JDesktopPane. Как видно, никаких параметров конструктору не требуется, так же как не требуется для нормальной работы и дополнительной настройки. О рабочем столе можно думать как о специализированной панели, специально предназначенной для размещения и управления внутренними окнами.

Далее создаются внутренние окна. Интересно, что класс JInternalFrame обладает целым букетом конструкторов с различным набором параметров, которые позволяют

задать все атрибуты внутренних окон. Обязательно присутствует лишь заголовок окна (как первый параметр всех конструкторов). Первое окно создается с заголовком и изменяемого размера, за что отвечает второй параметр конструктора. По умолчанию, если не указывать true, внутренние окна считаются неизменяемого размера, без возможности сворачивания, развертывания, более того, их даже невозможно закрыть. Все эти возможности включаются либо передачей параметров в конструктор, либо меняются через соответствующие свойства.

Второе окно создается вызовом наиболее развернутого конструктора. Здесь мы указываем заголовок окна, а также делаем его размер изменяемым, для самого окна включаем поддержку развертывания, свертывания и способность закрываться по кнопке закрытия окна. Достаточно своеобразное решение создателей Swing не включать все эти свойства по умолчанию, и передавать их в довольно длинный конструктор, но придется с этим мириться. Наконец третье окно создается без возможности смены размера, но с возможностью закрытия. Для него дополнительным, не описанным как поле класса JInternalFrame, свойством мы включаем внешний вид «палитры инструментов». Не все внешние виды поддерживают «палитры», и нужно проверить работу предпочитаемого вами внешнего вида, перед тем как использовать «палитры».

Внутренние окна стараются во всем походить на своих старших братьев — окна высокого уровня, унаследованные от базового класса Window, и поэтому они по умолчанию невидимы, имеют нулевой размер и располагаются в начале экрана. Работа с внутренними окнами здесь не отличается от работы с обычными. Мы меняем размер окон, устанавливаем их позиции и делаем их видимыми. Только после этого они появятся на рабочем столе JDesktopPane.




Запустив приложение, вы увидите рабочий стол и располагающиеся на нем внутренние окна. Вы можете таскать окна, сворачивать их и менять их размеры, если, конечно, они вам это позволяют (первое окно в нашем примере довольно ограниченно). В дополнение ко всему, рабочий стол позаботится о том, чтобы активное окно (выделенное в данный момент) перекрывало все остальные и было на переднем плане.

На этом можно считать многодокументное приложение Swing готовым. Вы можете конструировать свои интерфейсы и добавлять их во внутренние окна, располагая их, так как вам удобно. Принципиальной разницы между внутренними окнами и окнами тяжеловесными, которые мы уже рассмотрели в этой главе, нет.

## Внутренние окна JInternalFrame

Легко видеть, что внутренние окна по сути стараются далеко не отходить от обычных окон Swing. Основным качеством окон является присутствие в них корневой панели, которая, как мы знаем, состоит из нескольких составляющих, и позволяет окнам быть окнами, в том числе отображать полоску меню и контекстные меню. Внутренние окна содержат корневую панель в качестве основного компонента, и для нас это означает, что они являются обычными окнами, просто без связи с операционной системой.

Как и у обычных окон, мы найдем во внутренних окнах такие свойства, как rootPane, contentPane, defaultCloseOperation, title и соответствующие методы get и set. Все они имеют тоже значение, что и в обычных окнах: мы можем управлять корневой панелью, добавлять компоненты в панель содержимого и менять в ней менеджер расположения, устанавливать действие по закрытию окна и менять его заголовок. Как и в обычном окне, во внутреннем мы можем организовать систему меню любой сложности и добавить его, применив свойство jMenuBar. С другой стороны, внутренние окна полностью находятся во власти Swing, являясь легковесными компонентами, и поэтому настроить их можно более тонко, в отличие от окон операционной системы. Для этого служат несколько свойств (табл. 6.4):

**Таблица 6.4.** Основные свойства внутренних окон JInternalFrame

| Свойство | Описание |
|---|---|
| frameIcon | Задает значок для внутреннего окна, в том случае если UI-представитель окон поддерживает значки |
| iconifiable | Управляет тем, способно ли окно полностью сворачиваться и разворачиваться обратно, и появлением соответствующей кнопки на заголовке окна |
| resizable | Определяет, смогут ли пользователи изменять размеры окна. Если возможность отключена, на границах окна даже не появится специальный курсор, указывающий на возможность смены размеров. |
| maximizable | Определяет, можно ли внутреннее окно развертывать на весь размер рабочего стола и сворачивать обратно к оригинальному размеру, и появлением соответствующей кнопки на заголовке окна |
| closable | Определяет, может ли пользователь закрыть окно, соответственно этому свойству появляется или исчезает кнопка закрытия на заголовке окна |

Для корректной работы (перетаскивания, сворачивания и остальных операций с окнами) внутренние окна JInternalFrame всегда необходимо добавлять в рабочий стол JDesktopPane, иначе внутренние окна просто не поймут, что им нужно делать, а пользователь запутается. Как правило, рабочий стол занимает если не всю область главного окна приложения, то, как минимум, большую его часть, иначе работать с многодокументным интерфейсом крайне неудобно.

Управляет жизнью внутренних окон на рабочем столе специальный объект-«стратегия» DesktopManager, который доступен нам через соответствующее свойство класса JDesktopPane. Именно он решает, как ведет себя окно при перетаскивании, сворачивании и разворачивании, изменении размеров. Реализация по умолчанию просто поддерживает произвольное положение окон на экране, а написав свой вариант, вы, к примеру, можете организовать «причаливание» окон к одной из границ рабочего стола, если это будет необходимо.

На этом обсуждение многодокументных интерфейсов в Swing можно закончить. Суть, как мы увидели, состоит в том, что внутренние окна ничем не отличаются от обычных окон, просто организация рабочего пространства пользователя будет другая.

Но стоит всегда держать в уме, что неудачный многодокументный интерфейс с избытком маленьких окон запутывает пользователя и делает работу сложнее, так что перед его использованием все нужно тщательно продумывать.

## *Резюме*

Контейнеры высшего уровня Swing, предназначенные для окончательного вывода интерфейса на экран, обладают определенными особенностями. Сложности в основном кроются в работе корневой панели и составляющих ее элементов, которые по большей части используются внутренними процессами Swing. Но на самом деле здесь все просто, и корневая панель способна помочь вам в создании интерфейса с удивительными эффектами и необычным внешним видом. Окна Swing, единственные тяжеловесные контейнеры библиотеки, весьма незатейливы и предоставляют вам немного способов управления собой. Так что чаще всего они нужны как площадка для размещения компонентов, правда в новых версиях Java у них появились новые возможности. Помимо окон вы можете использовать для размещения своего интерфейса и апплеты, которые, хотя и потеряли значительную часть своей прежней привлекательности, все еще способны прямо на вебстранице предоставить пользователю удивительные возможности. Также к вашим услугам и средства создания многодокументных интерфейсов, которые хоть и не всегда хороши с точки зрения удобства пользователя, но в сложных ситуациях могут пригодиться.

# Глава 7. Искусство расположения

Как вы уже могли убедиться из предыдущих примеров, добавление компонентов на форму в Java происходит достаточно необычно. Мы использовали метод add(), и, как по волшебству, на форме появлялись наши компоненты — кнопки, надписи, текстовые поля. Это может показаться странным, потому что для создания интерфейса необходимо точно указывать размер компонента и его расположение на экране — причем часто для этого используются так называемые «абсолютные» координаты, полностью зависящие от разрешающей способности экрана и его размеров. Чтобы облегчить работу с такими интерфейсами, для их создания в подавляющем большинстве сред разработки используются визуальные средства, «строители GUI». Процесс происходит следующим образом. Вы создаете новую главную форму — обычно окно с рамкой или диалоговое окно — и располагаете на ней необходимые вам компоненты, задавая для них подходящие размеры и позиции. После этого ваша форма запоминается в файлах ресурсов, которые при компиляции программы присоединяются к программе и загружаются при ее запуске. Все очень быстро и удобно.

Однако оказывается, что использование такой же схемы в Java является не лучшей идеей. Ведь здесь дела обстоят несколько иначе — не стоит забывать, что программа на Java «пишется один раз, а выполняется везде». Если уж мы собираемся писать коммерческое приложение, опирающееся на все возможности и всю мощь этого языка и его библиотек, то, конечно, мы не упустим шанса написать программу, поддерживающую все широко распространенные операционные системы, при этом абсолютно ее не изменяя. Преимущества Java перед другими языками, средами и подходами здесь неоспоримы. Но в таком случае придется как-то учесть все различия между этими операционными системами и используемыми в них виртуальными машинами Java — представьте себе, что разрешение экрана в одной системе превосходит разрешение других систем в несколько раз, а значит, меняются размеры всех объектов, выводимых на экран. Практически во всех операционных системах компоненты имеют свои особенности, и кнопка, прекрасно подходящая по размерам для Windows, в другой операционной системе может оказаться слишком большой или слишком маленькой. Или, например, у некоторого пользователя со слабым зрением все шрифты специально увеличены в два раза — в программе, использующей абсолютные размеры и написанной для другой системы, он вообще ничего не увидит. Поэтому обычные подходы (явное указание координат компонента и его размеров) в Java не оправданы.

Язык Java недаром носит титул переносимого между платформами — со всеми различиями операционных систем прекрасно справится виртуальная машина Java. Однако остается вопрос правильного расположения компонентов на форме — здесь необходима гибкость и независимость от конкретных размеров. Все это обеспечивает *менеджер расположения* (layout manager), который играет очень важную роль в разработке пользовательского интерфейса. Менеджер расположения — это некая программная служба[1], ассоциированная с формой (в Java обычно говорят *с контейнером*) и определяющая, каким образом на ней будут располагаться компоненты. Независимо от платформы, виртуальной машины, разрешения и размеров экрана менеджер расположения гарантирует, что

[1] На языке шаблонов проектирования менеджер расположения можно описать как *стратегию* — алгоритм расположения компонентов, реализованный во внешнем классе, этот алгоритм можно легко и быстро подключить и при необходимости сменить.

компоненты будут иметь предпочтительный или близкий к нему размер и располагаться в том порядке, который был указан программистом при создании программы.

Надо сказать, что в Swing менеджер расположения играет еще большую роль, чем обычно. Он позволяет не только сгладить различия между операционными системами, но и дает возможность менять с легкостью внешний вид приложения, не заботясь о том, как при этом изменяются размеры и расположение компонентов. На самом деле, различные внешние виды приложений придают компонентам различные размеры, и если вы не будете использовать менеджер расположения для вашего интерфейса, переход от одного внешнего вида к другому может стать непосильной задачей.

В этой главе мы узнаем, как функционируют менеджеры расположения и что является самым важным при работе с ними. После этого мы познакомимся со стандартными менеджерами расположения, которые в большинстве своем очень просты. Иногда этого достаточно, но, как правило, сложные интерфейсы требуют применения универсальных менеджеров, и больше всего внимания мы уделим расположению BoxLayout, которое, хотя и является относительно простым, остается прекрасным «оружием» создателя пользовательского интерфейса, и гибкому табличному расположению GridBagLayout, а также его конкурентам.

## *Как работает менеджер расположения*

Поддержка менеджеров расположения встроена в базовый класс всех контейнеров java.awt.Container. Все компоненты библиотеки Swing унаследованы от базового класса этой библиотеки JComponent, который, в свою очередь унаследован от класса Container. Таким образом, для любого компонента Swing вы сможете установить требуемый менеджер расположения или узнать, какой менеджер им используется в данный момент. Для этого предназначены методы setLayout() и getLayout(). Конечно, изменять расположение вы будете только в контейнерах, которые предназначены для размещения в них компонентов пользовательского интерфейса, то есть в панелях JPanel и окнах (унаследованных от класса Window). Вряд ли стоит менять расположение в кнопках или флажках, хотя такая возможность имеется.

Обязанности менеджеров расположения описаны в интерфейсе LayoutManager из пакета java.awt (есть и расширенная версия этого интерфейса с именем LayoutManager2). Любой класс, претендующий на роль менеджера расположения, должен реализовать один из этих двух интерфейсов. Они не слишком сложны, и методов в них немного, так что написать собственный менеджер расположения — не такая уж непосильная задача. Впрочем, в стандартной библиотеке Java существует несколько готовых менеджеров расположения, и с их помощью можно реализовать *абсолютно любое* расположение. Поэтому основная часть данной главы будет посвящена изучению уже имеющихся менеджеров, однако и вопросы создания собственных менеджеров расположения мы также затронем.

Контейнер вызывает методы менеджера расположения каждый раз при изменении своих размеров или при первом появлении на экране. Кроме того, вы можете и программно запросить менеджер расположения заново расположить компоненты в контейнере: для этого служит так называемая *проверка корректности* (валидация) контейнера и содержащихся в нем компонентов. Проверка корректности очень полезна, если ваш интерфейс динамически меняется, к примеру, если динамически меняются размеры компонентов (вы можете прямо во время выполнения программы изменить текст надписи или количество столбцов таблицы — все это приведет к изменению размеров). После изменений компонентам может не хватать прежних размеров или наоборот, прежний размер будет для них слишком велик, так что расположение компонентов нужно будет провести заново, иначе интерфейс рискует стать неряшливым и неудобным. Для этого и предназначена проверка корректности. Выполнить проверку корректно-

сти для любого компонента Swing, будь это контейнер или отдельный компонент, позволяет метод revalidate(), определенный в базовом классе библиотеки JComponent[2] .

Менеджер расположения обязан расположить добавляемые в контейнер компоненты в некотором порядке, зависящем от реализованного в нем алгоритма, и придать им некоторый размер, при этом он обычно учитывает определенные свойства компонентов.

- *Предпочтительный размер.* Такой размер идеально подходит данному компоненту. По умолчанию все размеры компонентов устанавливаются UI-представителями текущего внешнего вида и поведения (look and feel), но вы можете изменить их. Предпочтительный размер можно изменить с помощью метода setPrefferedSize().
- *Минимальный размер.* Этот параметр показывает, до каких пор менеджер расположения может уменьшать размер компонента. После достижения минимального размера всех компонентов контейнер больше не сможет уменьшить свой размер. Минимальный размер также можно изменить, для этого предназначен метод setMinimumSize().
- *Максимальный размер.* Этот параметр говорит о том, насколько можно увеличивать компонент при увеличении размеров контейнера. Например, максимальный размер текстового поля JTextField не ограничен, что не всегда удобно (чаще всего оно должно сохранять свой предпочтительный размер). Эту оплошность мы сможем исправить с помощью метода setMaximumSize(), устанавливающего максимальный размер. Большая часть менеджеров расположения игнорируют максимальный размер, работая в основном с предпочтительным и минимальным размерами.
- *Выравнивание по осям X и Y.* Эти параметры нужны только менеджеру BoxLayout, причем для него они играют важнейшую роль. Поэтому их мы рассмотрим, когда дойдем до описания этого менеджера.
- *Границы контейнера.* Эти параметры контейнера, которые позволяет получить метод getInsets(), определяют размеры отступов от границ контейнера. Иногда менеджеру расположения приходится их учитывать. В Swing и для компонентов, и для контейнеров применяются рамки Border, которые находятся прямо в пространстве компонента, отдельных отступов нет, что делает работу компонентов Swing более простой и предсказуемой.
- *Базовая линия.* Типографский термин, обозначающий основание букв, по которому выравниваются строки текста. Для компонентов интерфейса это означает, что существует линия, по которой выравнивание компонента будет оптимальным, к примеру, базовая линия текста в текстовом поле и базовая линия текста надписи на кнопке. В противном случае менеджер расположения не знает, как выравнивать компоненты и сможет выровнять их только по верхней и нижней границе или по центру, что может привести к неудовлетворительным результатам и визуальному расхождению. Для определения базовой линии предназначен метод getBaseline(), который в своих компонентах можно переопределять.

Работа менеджера расположения происходит следующим образом: он ждет прихода сигнала от контейнера, требующего расположить в нем компоненты (этому соответствует вызов метода layoutContainer() из интерфейса LayoutManager). В методе layoutContainer() и происходит основная работа по расположению компонентов в контейнере. Менеджер расположения, принимая во внимание различные размеры и свойства компонентов

[2] Мы подробно обсуждали механизмы (к слову, довольно сложные) работы метода revalidate() в главе 5. Как правило, проверка корректности не вызывает проблем, но если вы вдруг оказались в сложной ситуации, ответ можно найти в упомянутой главе 5.

и контейнера, должен расположить компоненты на определенных позициях в контейнере, вызывая для каждого из них метод setBounds(), позволяющий указать (в пикселах, в системе координат контейнера) прямоугольник, который будет занимать компонент. Для сложных менеджеров расположения с гибким поведением это означает массу работы и сложные вычисления, однако простые алгоритмы расположения компонентов реализовать совсем несложно. Попробуем теперь написать свой первый менеджер расположения компонентов, он будет располагать компоненты вертикально с расстоянием между ними в 5 пикселов и использовать для всех компонентов предпочтительный размер. Вот что у нас получится:

```
// VerticalLayout.java
// Простой менеджер расположения, располагает
// компоненты в вертикальный ряд с отступами
import java.awt.*;
import javax.swing.*;

public class VerticalLayout implements LayoutManager {
  // отступ между компонентами
  public int GAP = 5;
  // сигнал расположить компоненты в контейнере
  public void layoutContainer(Container c) {
    Component comps[] = c.getComponents();
    int currentY = GAP;
    for (Component comp : comps) {
      // предпочтительный размер компонента
      Dimension pref = comp.getPreferredSize();
      // указываем положение компонента на экране
      comp.setBounds(GAP, currentY,
          pref.width, pref.height);
      // промежуток между компонентами
      currentY += GAP;
      currentY += pref.height;
    }
  }
  // эти два метода нам не понадобятся
  public void addLayoutComponent(
      String name, Component comp) {
  }
  public void removeLayoutComponent(
      Component comp) {
  }
  // минимальный размер для контейнера
  public Dimension minimumLayoutSize(Container c) {
    return calculateBestSize(c);
```

```
}
// предпочтительный размер для контейнера
public Dimension preferredLayoutSize(Container c) {
  return calculateBestSize(c);
}

private Dimension size = new Dimension();
// вычисляет оптимальный размер контейнера
private Dimension calculateBestSize(Container c) {
  // сначала вычислим длину контейнера
  Component[] comps = c.getComponents();
  int maxWidth = 0;
  for (Component comp : comps) {
    int width = comp.getWidth();
    // поиск компонента с максимальной длиной
    if (width > maxWidth) maxWidth = width;
  }
  // длина контейнера с учетом левого отступа
  size.width = maxWidth + GAP;
  // вычисляем высоту контейнера
  int height = 0;
  for (Component comp : comps) {
    height += GAP;
    height += comp.getHeight();
  }
  size.height = height;
  return size;
}

// проверим работу нового менеджера
public static void main(String[] args) {
  SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          JFrame frame = new JFrame("VerticalLayout");
          frame.setDefaultCloseOperation(
              JFrame.EXIT_ON_CLOSE);
          // панель будет использовать новое расположение
          JPanel contents = new JPanel(
              new VerticalLayout());
          // добавим пару кнопок и текстовое поле
```

```
            contents.add(new JButton("Один"));
            contents.add(new JButton("Два"));
            contents.add(new JTextField(30));
            frame.add(contents);
            frame.setVisible(true);
            frame.pack();  } });
    }
}
```

Наш первый менеджер расположения, получивший название VerticalLayout, как и положено любому менеджеру расположения, реализует интерфейс LayoutManager. Первый метод, который нам придется реализовать, одновременно является и самым важным — именно в методе layoutContainer() менеджер расположения должен расположить все компоненты, содержащиеся в контейнере, по своим местам. Поведение нашего менеджера расположения незамысловато: он размещает компоненты в вертикальный ряд, отделяя их расстоянием в 5 пикселов (расстояние хранится в переменной GAP), все компоненты имеют предпочтительный размер. Чтобы реализовать такое поведение, понадобились следующие действия: мы просмотрели массив содержащихся в контейнере компонентов (получить этот массив позволяет метод контейнера getComponents()), получили для каждого из компонентов предпочтительный размер и указали (методом setBounds()) позицию компонента на экране, постоянно отслеживая текущую координату по оси Y, увеличивая ее на высоту очередного компонента с учетом «декоративного» расстояния из переменной GAP. Так все компоненты будут располагаться друг над другом, отделенные отступами[3]. Обратите внимание, что мы отделяем компоненты и от левой границы контейнера на то же самое расстояние.

Далее следуют два метода, служащие для добавления и удаления компонентов из списка компонентов менеджера расположения. Заметьте, что метод addLayoutComponent() позволяет ассоциировать с компонентом строку, которая может быть использована менеджером расположения как рекомендация относительно того, где именно программист-клиент желает видеть данный компонент. Наш простой менеджер расположения не поддерживает подобных рекомендаций, просто располагая все компоненты контейнера в вертикальный ряд, однако более сложные расположения, к примеру, полярное расположение BorderLayout, которое мы вскоре изучим, используют эти строки для более тонкого управления местом компонента в контейнере.

Следующими являются два метода, сообщающие предпочтительный (preferredLayoutSize()) и минимальный (minimumLayoutSize()) размеры контейнера при использовании в нем данного менеджера расположения. Когда в контейнере работает менеджер расположения, именно он запрашивается о том, каков должен быть предпочтительный или минимальный размер контейнера, и это вполне объяснимо: менеджер расположения распределяет место на экране, так что он должен знать, сколько этого места в контейнере требуется для правильной работы. Для нашего простого менеджера расположения минимальный и предпочтительный размеры контейнера совпадают, их вычисляет метод calculateBestSize(). Вычислить оптимальный размер контейнера несложно. Для начала мы в цикле ищем компонент с самой большой длиной, прибавляем к найденной длине размер отступа от левой границы контейнера из поля GAP и получаем оптимальную длину контейнера. Для высоты вычисления чуть сложнее: приходится складывать высоты всех

[3] Обратите внимание, что наш просто менеджер расположения не принимает во внимание, видим ли компонент на экране, когда отсчитывает позиции. Компонент вполне может находиться в контейнере, но быть невидимым, и тогда его следует исключать из расчетов. Для простоты примера мы опустили эту проверку.

находящихся в контейнере компонентов, а также прибавлять к ним высоту отступа между всеми компонентами. Полученная сумма и является оптимальной высотой контейнера.

После реализации всех методов интерфейса LayoutManager мы сможем проверить новый менеджер расположения в работе. Для этого в методе main() мы создаем небольшое окно с рамкой JFrame и добавляем в это окно панель JPanel , устанавливая для нее наш новый менеджер расположения. В панель добавляется пара кнопок и довольно большое текстовое поле, после чего окно выводится на экран. Чтобы оценить правильность расчета нашим менеджером оптимальных размером, мы вызываем метод pack(), который придает окну предпочтительный размер. Запустив программу с примером, вы сможете оценить, как компоненты располагаются вертикально друг над другом, и верно ли был проведен расчет оптимальных размеров.


Знание основных принципов работы менеджера расположения и умение быстро написать новый алгоритм расположения компонентов, лучше всего подходящий в вашей ситуации, — мощное оружие, дающее вам неограниченную власть над расположением компонентов и способность быстро создать любой интерфейс. Однако увлекаться написанием собственных менеджеров расположения ни в коем случае не стоит. Прекрасно известно, что код читается гораздо чаще и дольше, чем пишется, а читать и поддерживать код, располагающий компоненты по собственным алгоритмам, совсем непросто. Гораздо проще использовать хорошо изученные стандартные менеджеры расположения, позволяющие с помощью различных приемов получить *любое* расположение. Этим мы сейчас и займемся.

## *Стандартные менеджеры расположения*

В пакете разработки JDK 1.7 имеется довольно много готовых стандартных менеджеров расположения. Условно их можно разделить на несколько групп: очень простые (к ним относятся FlowLayout, GridLayout и BorderLayout), универсальные (к таким без всяких сомнений стоит отнести расположения BoxLayout, GroupLayout и GridBagLayout) и специализированные (CardLayout и SpringLayout). Универсальные менеджеры мы рассмотрим в последнюю очередь и наиболее подробно, а пока быстро «пройдемся» по всем остальным менеджерам.

### Полярное расположение BorderLayout

Менеджер BorderLayout специально предназначен для обычных и диалоговых окон, потому что позволяет быстро и просто расположить наиболее часто используемые элементы любого окна: панель инструментов, строку состояния и основное содержимое. Для этого окно разбивается им на четыре области, или полюса (отсюда его название), а все оставшееся место заполняется компонентом, выполняющим основную функцию приложения (например, в редакторе это будет текстовое поле, в многооконном приложении — рабочий стол).

Надо сказать, что работает этот менеджер немного не так, как все остальные — чтобы добавить с его помощью компонент, в методе add() необходимо указать дополнительный

параметр, который показывает, в какую область контейнера следует поместить компонент. Ниже перечислены допустимые значения этого параметра.

- Значение BorderLayout.NORTH или строка «North» — компонент располагается вдоль верхней (северной) границы окна и растягивается на всю его ширину. Обычно так размещается панель инструментов.
- Значение BorderLayout.SOUTH или строка «South» — компонент располагается вдоль нижней (южной) границы и растягивается на всю ширину окна. Такое положение идеально для строки состояния.
- Значение BorderLayout.WEST или строка «West» — компонент располагается вдоль левой (западной) границе окна и растягивается на всю его высоту, однако при этом учитываются размеры северных и южных компонентов (они имеют приоритет).
- Значение BorderLayout.EAST или строка «East» — компонент располагается вдоль правой (восточной) границы окна. В остальном его расположение аналогично западному компоненту.
- Значение BorderLayout.CENTER или строка «Center» — компонент помещается в центр окна, занимая максимально возможное пространство.

## СОВЕТ

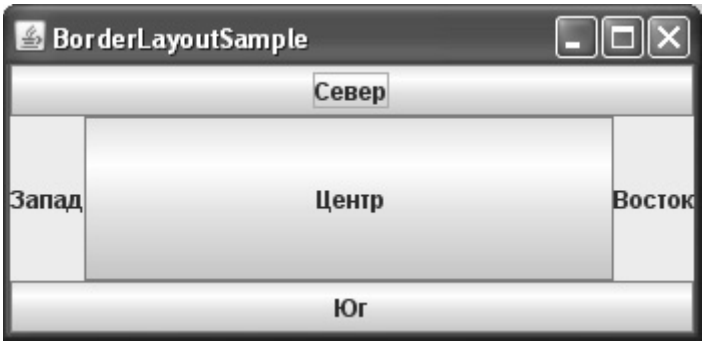
> На север помещайте панель инструментов вашего приложения. На юг помещайте строку состояния. Оставляйте западные и восточные зоны окна свободными — только в этом случае панель инструментов можно будет перетаскивать. Для главного окна вашего приложения всегда используйте расположение BorderLayout.

Рассмотрим простой пример. В нем создается окно JFrame, в котором менеджер BorderLayout используется по умолчанию. Во все доступные зоны добавляются компоненты:

```
// BorderLayoutSample.java
// Полярное расположение
import javax.swing.*;
import java.awt.*;

public class BorderLayoutSample extends JFrame {
  public BorderLayoutSample()  {
    super("BorderLayoutSample");
    setSize(400, 300);
    setDefaultCloseOperation( EXIT_ON_CLOSE );
    // добавляем компоненты
    // в качестве параметров можно использовать строки
    add(new JButton("Север"), "North");
    add(new JButton("Юг"), "South");
    // … или константы из класса BorderLayout
    add(new JLabel("Запад"), BorderLayout.WEST);
    add(new JLabel("Восток"), BorderLayout.EAST);
```

```
    // если параметр не указывать вовсе, компонент
    // автоматически добавится в центр
    add(new JButton("Центр"));
    // выводим окно на экран
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
     new Runnable() {
      public void run() { new BorderLayoutSample(); } });
  }
}
```

В примере есть несколько интересных мест. Во-первых, обратите внимание на то, что метод установки менеджера расположения setLayout() не вызывался, потому что в окнах JFrame (а также в окнах без рамки JWindow и диалоговых окнах JDialog) расположение BorderLayout применяется по умолчанию. Во-вторых, будьте осторожнее с использованием строк в качестве параметров метода add(). Так легко сделать трудно обнаруживаемую ошибку, в то время как ошибку при использовании констант сразу же обнаружит компилятор. Картина на экране в итоге получается следующая:




Как легко заметить, возможности полярного расположения довольно ограничены. Оно создано специально для окон, но, тем не менее, иногда может помочь и в более сложных случаях. Об этом мы поговорим немного позднее.

## Последовательное расположение FlowLayout

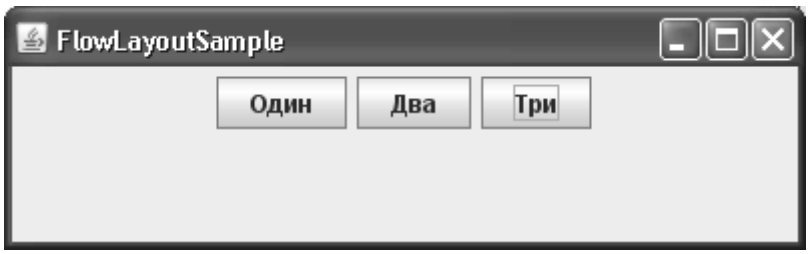
Последовательное расположение FlowLayout работает очень просто, но тем не менее эффективно. Оно выкладывает компоненты в контейнер, как пирожки на противень: слева направо, сверху вниз, начиная с верхнего левого угла контейнера. Это расположение устанавливается по умолчанию в панелях JPanel. Основным свойством его следует признать то, что компонентам всегда придается предпочтительный размер (например, ярлык с текстом JLabel соответствует по размерам тексту). Рассмотрим простой пример:

```
// FlowLayoutSample.java
// Последовательное расположение
```

```
import java.awt.*;
import javax.swing.*;

public class FlowLayoutSample extends JFrame {
 public FlowLayoutSample() {
     super("FlowLayoutSample");
     setSize(400, 200);
     setDefaultCloseOperation( EXIT_ON_CLOSE );
     // устанавливаем последовательное расположение с
     // выравниванием компонентов по центру
     setLayout( new FlowLayout( FlowLayout.CENTER ));
     // добавляем компоненты
     add( new JButton("Один"));
     add( new JButton("Два"));
     add( new JButton("Три"));
     // выводим окно на экран
     setVisible(true);
 }
 public static void main(String[] args) {
    SwingUtilities.invokeLater(
     new Runnable() {
      public void run() { new FlowLayoutSample(); } });
 }
}
```

Конструктор класса FlowLayout имеет три перегруженные версии: без параметров, с параметрами выравнивания компонентов, с параметрами выравнивания компонентов и расстояний между ними. В примере использована вторая версия, устанавливающая последовательное расположение по центру.

Запустите программу с примером, и вы увидите, как работает менеджер последовательного расположения.




Если места для всех компонентов достаточно, они размещаются горизонтально в одну строку. Уменьшите горизонтальный размер окна — как только места в одной строке всем компонентам станет мало, лишние перейдут на строку ниже (расстояние между строками и компонентами можно указать в конструкторе). Теперь уменьшайте размер

окна по вертикали — вы увидите, что компоненты, не помещающиеся в окно, просто исчезают из поля зрения! Это не какой-то недостаток последовательного расположения, как может показаться на первый взгляд, а только следствие его главного свойства: *при последовательном расположении всегда сохраняется предпочтительный размер компонентов.* Менеджер FlowLayout неизменно работает по такому правилу, и если места в контейнере становится мало, то он просто прячет «лишние» компоненты, а не уменьшает их размеры. Вторым по важности свойством менеджера расположения FlowLayout следует признать то, что при вызове метода preferredLayoutSize() или minimumLayoutSize(), позволяющего узнать предпочтительный и минимальный размеры контейнера, в котором этот менеджер действует, метод возвращает размер, соответствующей ситуации расположения всех компонентов *в одну строку*. Это особенно важно при совмещении последовательного расположения с другими вариантами расположения. Вследствие этого свойства менеджеры других вариантов расположения будут стараться найти достаточно места для размещения компонентов в одну строку. Вскоре мы увидим совмещение в действии.

Поэтому использовать последовательное расположение следует только в контейнере, где достаточно места, или там, где контейнеру некуда будет «прятать» свои компоненты (пример мы рассмотрим в следующем разделе). Тем не менее, этот замечательный по своей простоте менеджер расположения очень хорош при организации несложных вариантов расположения.

## Табличное расположение GridLayout

Как нетрудно догадаться по названию, менеджер расположения GridLayout разделяет контейнер на таблицу с ячейками одинакового размера Количество строк и столбцов можно указать в конструкторе, причем существует возможность задать произвольное количество либо строк, либо столбцов (но не одновременно). Все ячейки имеют *одинаковый размер*, маленькие компоненты растягиваются по всем направлениям, большие компоненты сжимаются. Табличное расположение идеально подходит для ситуаций, где нужно разбить контейнер на несколько одинаковых частей с примерно одинаковым по размеру содержимым. В качестве первого примера рассмотрим окно с кнопками различных размеров:

```
// GridLayoutSample.java
// Табличное расположение
import java.awt.*;
import javax.swing.*;

public class GridLayoutSample extends JFrame {
 public GridLayoutSample() {
     super("GridLayoutSample");
     setSize(300, 200);
     setLocation(100, 100);
     setDefaultCloseOperation( EXIT_ON_CLOSE );
     // вспомогательная панель
     JPanel grid = new JPanel();
     // первые два параметра конструктора GridLayout -
     // количество строк и столбцов в таблице
     // вторые два — расстояние между ячейками по X и Y
     GridLayout gl = new GridLayout(2, 0, 5, 12);
     grid.setLayout(gl);
```

```
        // создаем 8 кнопок
        for (int i=0; i<8; i++) {
            grid.add(new JButton("Кнопка " + i));
        }
        // помещаем нашу панель в центр окна
        add(grid);
        // устанавливаем оптимальный размер
        pack();
        // показываем окно
        setVisible(true);
    }
    public static void main(String[] args) {
       SwingUtilities.invokeLater(
        new Runnable() {
         public void run() { new GridLayoutSample(); } });
    }
}
```

Обратите внимание на то, как в примере задается произвольное количество столбцов — для этого в конструкторе вместо конкретного числа столбцов указывается ноль. Аналогично можно задать и произвольное количество строк в таблице. Главное, чтобы оба числа не были нулями. В нашем примере GridLayout разделит компоненты так, чтобы они помещались в два ряда, неважно, как много столбцов для этого придется создать.


Запустите программу с этим примером и попробуйте произвольно увеличить размер окна. Вы увидите, что менеджер расположения GridLayout очень «эгоистичен» — он занимает все место, доступное в данный момент в окне, да еще к тому же безобразно «раздувает» свои компоненты. Если же вы уменьшите окно, то компоненты никуда не исчезнут, как это было в случае с последовательным расположением FlowLayout, а станут очень маленькими (вплоть до того, что на них ничего не станет видно). Ниже перечислены основные свойства табличного расположения.

- Все компоненты имеют одинаковый размер. Доступное пространство контейнера с табличным расположением разбивается на одинаковое количество ячеек, в каждую из которых помещается компонент.
- Все компоненты всегда выводятся на экран, как бы ни было велико или малодоступно пространство.

Чаще всего мы не хотим, чтобы компоненты были слишком велики или слишком малы, поэтому почти всегда имеет смысл размещать панель с табличным расположением

GridLayout в дополнительной панели с последовательным расположением FlowLayout, которое наоборот, никогда не делает содержащиеся в нем компоненты больше их предпочтительного размера, или использовать другое расположение с такими же свойствами (GridBagLayout). Чтобы проиллюстрировать сказанное, создадим элемент пользовательского интерфейса, встречающийся практически в любом диалоговом окне, а именно строку кнопок (обычно это кнопки OK и Отмена). Табличное расположение придаст кнопкам одинаковый размер, а последовательное расположение не даст им «расплыться» и заодно выровняет их по правому краю:

```
// CommandButtons.java
// Создание панели командных кнопок
import javax.swing.*;
import java.awt.*;

public class CommandButtons extends JFrame {
 public CommandButtons() {
     super("CommandButtons");
     setSize(350, 250);
     setLocation(150, 100);
     setDefaultCloseOperation( EXIT_ON_CLOSE );
     // создаем панель с табличным расположением для
     // выравнивания размеров кнопок
     JPanel grid = new JPanel(
         new GridLayout(1, 2, 5, 0) );
     // добавляем компоненты
     grid.add( new JButton("OK"));
     grid.add( new JButton("Отмена"));
     // помещаем полученное в панель с последовательным
     // расположением, выравненным по правому краю
     JPanel flow = new JPanel(
         new FlowLayout( FlowLayout.RIGHT ));
     flow.add(grid);
     // помещаем строку кнопок вниз окна
     add(flow, BorderLayout.SOUTH );
     // выводим окно на экран
     setVisible(true);
 }
 public static void main(String[] args) {
    SwingUtilities.invokeLater(
     new Runnable() {
      public void run() { new CommandButtons(); } });
 }
}
```

В конструкторе табличного расположения мы указываем один ряд с двумя столбцами, с расстоянием между компонентами по вертикали в 5 пикселов. Теперь, как бы вы не изменяли размер окна, в какой бы операционной системе не оказалось ваше приложение, его пользовательский интерфейс окажется «на высоте»: полярное расположение заставит панель с кнопками разместиться вдоль нижней границы окна, табличное расположение приведет к выравниванию размеров кнопок, а последовательное расположение заставит их прижаться к левому краю.


В дальнейшем при создании различных диалоговых окон мы не раз будем использовать этот пример.

## Динамические вкладки и CardLayout

Хотя менеджер расположения CardLayout поддерживается в пакетах JDK с самых первых версий, при создании пользовательского интерфейса им практически не пользуются. Менеджер CardLayout до появления библиотеки Swing использовался для создания так называемых *вкладок* (tabs), выбирая которые, можно было поочередно открывать доступ к панелям, занимающие одно и то же место. В библиотеке Swing имеется специальный класс JTabbedPane, который берет все заботы по организации вкладок на себя, а вам остается лишь добавлять их. Мы его вскоре изучим.

Вариантом применения CardLayout остается динамический интерфейс, которому не нужны визуальные вкладки, которые предоставляет JTabbedPane, но нужна возможность динамически, во время работы программы, менять одни панели на другие. Но такая возможность легко реализуется и без CardLayout — необходимо убрать из контейнера старую панель методом remove(), добавить новую панель, и провести проверку корректности контейнера методом revalidate(). Таким образом, область применения менеджера CardLayout весьма ограничена и мы даже не станем рассматривать его на примере.

## Менеджер расположения SpringLayout

В пакете разработки JDK имеется универсальный менеджер расположения, способный помочь добиться любого, даже самого сложного расположения компонентов. Это менеджер расположения SpringLayout из пакета javax.swing. Однако несмотря на универсальность, действие его весьма специфично и не похоже на действие ни одного из уже знакомых нам менеджеров расположения. С каждым компонентом ассоциируется особый информационный объект Spring, который позволяет задать расстояние (в пикселах) между парой границ различных компонентов. Границ у компонента четыре — это его северная, восточная, западная и южная сторона. Можно задавать расстояние и между границами одного и того же компонента: например, задав расстояние между северной и южной сторонами одного компонента, вы укажете его высоту. По умолча-

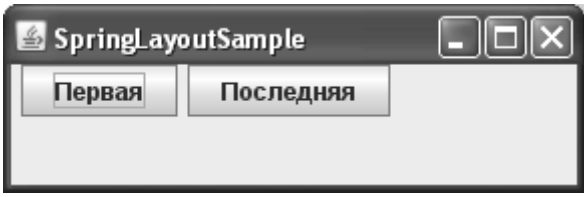
нию все компоненты имеют предпочтительный размер, однако менеджер SpringLayout тщательно учитывает и два остальных размера, не делая компоненты меньше минимального и больше максимального размеров.

На первом этапе работы менеджера SpringLayout все компоненты находятся в начале координат контейнера и имеют предпочтительный размер. Чтобы разместить их по нужным позициям, обычно проводят следующие действия: первый компонент отделяют некоторым расстоянием от границы контейнера, второй отделяют от первого расстоянием между нужными границами, далее размещают третий компонент и т. д. Рассмотрим небольшой пример и увидим все в действии:

```
// SpringLayoutSample.java
// Работа менеджера SpringLayout
import javax.swing.*;
import java.awt.*;

public class SpringLayoutSample extends JFrame {
 public SpringLayoutSample() {
     super("SpringLayoutSample");
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     // панель с использованием SpringLayout
     SpringLayout sl = new SpringLayout();
     JPanel contents = new JPanel(sl);
     // добавим пару компонентов
    JButton button1, button2;
     contents.add(button1 = new JButton("Первая"));
     contents.add(button2 = new JButton("Последняя"));
     // настроим распорки
     sl.putConstraint(SpringLayout.WEST, button1,
         5, SpringLayout.WEST, contents);
     sl.putConstraint(SpringLayout.WEST, button2, 5,
         SpringLayout.EAST, button1);
     // выведем окно на экран
     setContentPane(contents);
     setSize(300, 200);
     setVisible(true);
 }
 public static void main(String[] args) {
    SwingUtilities.invokeLater(
     new Runnable() {
      public void run() { new SpringLayoutSample(); } });
 }
}
```

Мы создаем небольшое окно JFrame, его панелью содержимого является панель JPanel, использующая менеджер расположения SpringLayout. В полученную панель добавляются две

кнопки. Для того чтобы они заняли свое место, необходимо задать для них нужные расстояния. Расстояние задается методом putConstraint() класса SpringLayout. В нем нужно указать границу компонента, сам компонент, расстояние, а также границу второго компонента и сам второй компонент (это тот компонент, что вы хотите отделить заданным расстоянием от первого компонента). Первый вызов putConstraint() отделяет западную (левую) границу первой кнопки от западной же границы контейнера. Второй вызов разделяет западную (левую) границу второй кнопки и восточную (правую) границу первой кнопки пятью пикселами. В итоге мы получаем две кнопки, расположенные горизонтально и отделенные пятью пикселами. Вы сможете легко в этом убедиться, запустив программу с примером.




Менеджер SpringLayout довольно мощен и гибок, но совершенно не приспособлен для создания интерфейса «вручную». Посмотрите: чтобы расположить компоненты, нужны все ссылки на них, приходится оперировать абсолютными значениями расстояний между ними, и создание не самого сложного расположения из десятка компонентов рискует превратиться в мешанину из вызовов putConstraint(). Поддерживать такой код очень сложно, и это не удивительно: менеджер SpringLayout предназначен, прежде всего, для автоматизированного построения интерфейса, а для ручной работы есть более удобные приемы.

## Абсолютное расположение

Никто не настаивает на обязательном использовании в контейнере менеджера расположения, вы можете удалить его, вызвав метод setLayout(null). В этом случае вся ответственность за правильное расположение компонентов на экране ложится на ваши плечи: размеры и позиции компонентов придется задавать прямо в программе, вызывая для каждого компонента метод setBounds() с подходящим прямоугольником.

*Никогда* не используйте абсолютное расположение, иначе рискуете навлечь на себя праведный гнев любого Java-программиста, который увидит ваш код. Все преимущества динамической смены внешнего вида приложения средствами библиотеки Swing, вся легкость перехода на другие платформы — все будет повергнуто в прах в случае абсолютного расположения. Забудьте о нем так, как будто его нет. Если ваше средство автоматизированного построения интерфейса предлагает проектировать интерфейс в абсолютных координатах, а затем безуспешно пытается привести то, что получилось, к расположению, соответствующему какому-либо менеджеру (в результате вам все равно приходится использовать абсолютное расположение), выбросите это средство.

Единственным случаем, в котором приходится применять абсолютное расположение, являются графические приложения, где мы рисуем или располагаем компоненты поверх остального интерфейса, например, логотип компании. Часто для этого применяется компонент JXLayer или прозрачная панель (glass pane), которые мы обсуждали в главе 6.

## Вложенные расположения

В предыдущем разделе, совмещая табличное и последовательное расположения, мы познакомились с примером так называемого *вложенного расположения* (nested layout), основная идея которого очень проста — вы создаете несколько контейнеров с различ-

ными менеджерами расположения, а затем добавляете их друг в друга для получения искомого результата. Подбирая соответствующим образом менеджеры расположения, можно добиться абсолютно любого расположения компонентов.

В принципе, обычный пользовательский интерфейс вполне можно создать на основе рассмотренных нами простейших расположений. Но, честно говоря, хотелось бы более гибкого подхода, особенно это касается расположения компонентов по вертикали. И потом, рассмотренные нами до этого расположения очень неаккуратно работают с расстоянием между компонентами — его можно задать один раз, причем сразу для всех входящих в расположение компонентов. Между тем, в качественном интерфейсе расстояние между компонентами играет важнейшую роль, и чаще всего между разными компонентами оно разное. Поэтому создавать большую часть интерфейса с помощью этой идеи мы, конечно, не станем, а рассмотрим универсальные расположения, применяемые при создании пользовательского интерфейса чаще других.

## Блочное расположение BoxLayout

Блочное расположение BoxLayout — прекрасная альтернатива всем остальным менеджерам расположения. Обладающее большими возможностями расположение BoxLayout не сложнее BorderLayout.

Менеджер блочного расположения выкладывает компоненты в контейнер блоками: столбиком (по оси Y) или полоской (по оси X), при этом каждый отдельный компонент можно выравнивать по центру, по левому или по правому краям, а также по верху или по низу. Расстояние между компонентами по умолчанию нулевое, но для его задания существуют специальные классы (об этом чуть позже). Как располагаются компоненты, хорошо видно на рис. 7.1

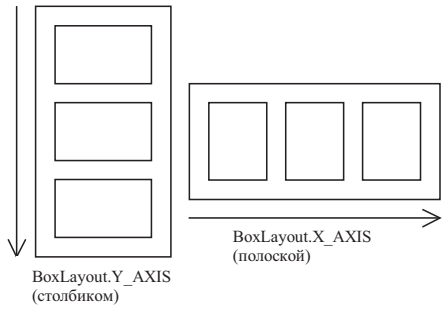



**Рис. 7.1.** Блочное расположение

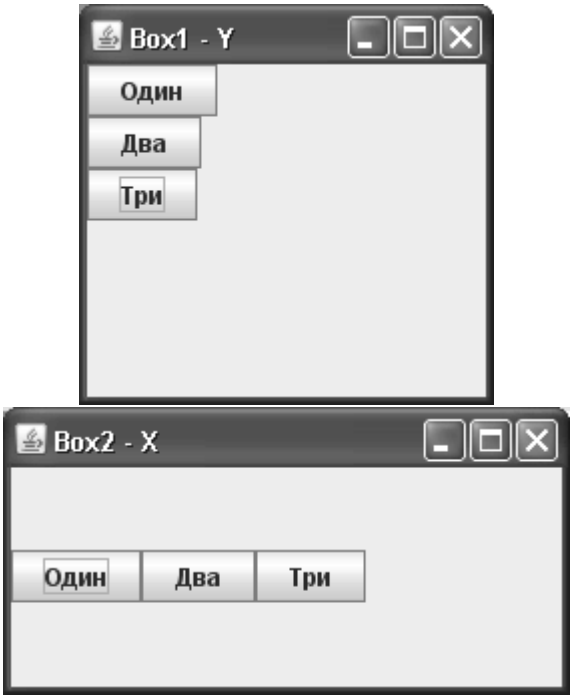
Стрелки показывают, в каком порядке происходит добавление компонентов в контейнер. Чтобы указать менеджеру, как нужно выкладывать компоненты — столбиком или полоской, используются константы из класса BoxLayout, также показанные на рисунке. Рассмотрим простой пример:

```
// Box1.java
// Блочное расположение
import javax.swing.*;
import java.awt.*;

public class Box1 extends JFrame {
```

```
public Box1() {
    super("Box1 — Y");
    setSize(400, 200);
    setDefaultCloseOperation( EXIT_ON_CLOSE );
    // получаем панель содержимого
    Container c = getContentPane();
    // устанавливаем блочное расположение по
    // оси Y (столбиком)
    BoxLayout boxy = new BoxLayout(c, BoxLayout.Y_AXIS);
    c.setLayout(boxy);
    // добавляем компоненты
    c.add( new JButton("Один"));
    c.add( new JButton("Два"));
    c.add( new JButton("Три"));
    // выводим окно на экран
    setVisible(true);
}
static class Box2 extends JFrame {
    public Box2() {
        super("Box2 — X");
        // устанавливаем размер и позицию окна
        setSize(400, 200);
        setLocation(100, 100);
        setDefaultCloseOperation( EXIT_ON_CLOSE  );
        // получаем  панель содержимого
        Container c  = getContentPane();
        // устанавливаем блочное расположение по
        // оси X (полоской)
        BoxLayout boxx =
            new BoxLayout(c, BoxLayout.X_AXIS);
        c.setLayout(boxx);
        // добавляем компоненты
        c.add( new JButton("Один"));
        c.add( new JButton("Два"));
        c.add( new JButton("Три"));
        // выводим окно на экран
        setVisible(true);
    }
}
public static void main(String[] args) {
   SwingUtilities.invokeLater(
```

```
        new Runnable() {
         public void run() { new Box1(); new Box2(); } });
    }
}
```

В этом примере создаются два окна. В одном из них реализовано блочное расположение по оси Y, в другом — блочное расположение по оси X. Как легко убедиться, при блочном расположении компоненты действительно размещаются вплотную друг к другу.




Вы можете видеть, что конструктор класса BoxLayout несколько необычен — ему необходимо указать контейнер, в котором он будет функционировать. Ни в одном из рассмотренных нами прежде менеджеров расположения такого не требовалось. Впрочем, можно не строить блочное расположение вручную, а использовать вспомогательный класс Box из пакета javax.swing. В нем определены два статических метода:

```
public static Box createHorizontalBox()
public static Box createVerticalBox()
```

Эти методы возвращают экземпляр класса Box, который создан специально для поддержки блочного расположения и унаследован от базового класса Swing JComponent. Поэтому объекты Box вы можете использовать как обычные контейнеры для компонентов, только с заранее установленным блочным расположением. Первый метод возвращает контейнер с горизонтальным блочным расположением, второй — с вертикальным.

Однако контейнер, который создается классом Box, не всегда так же полезен, как обычная панель JPanel. Дело в том, что внешний вид любого компонента Swing, как мы знаем, определяется UI-представителем этого компонента. Это верно и для панелей JPanel, и именно подобным способом некоторые внешние виды приложений добавляют

эффекты, такие как полупрозрачные фоновые изображения. У компонента Box своего UI-представителя нет, и вы рискуете создать контейнер, выдающийся из общего внешнего вида приложения.

Раз уж класс Box немного просчитался, напишем свой вспомогательный класс для работы с этим расположением. Тогда никто не останется в обиде:

```
// com/porty/swing/BoxLayoutUtils.java
// Класс для удобной работы с блочным расположением
package com.porty.swing;
import javax.swing.*;

public class BoxLayoutUtils {
  // возвращает панель с установленным вертикальным
  // блочным расположением
  public static JPanel createVerticalPanel() {
    JPanel p = new JPanel();
    p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));
    return p;
  }
  // возвращает панель с установленным горизонтальным
  // блочным расположением
  public static JPanel createHorizontalPanel() {
    JPanel p = new JPanel();
    p.setLayout(new BoxLayout(p, BoxLayout.X_AXIS));
    return p;
  }
}
```

Далее в примерах, когда нам понадобится блочное расположение, мы будем создавать его с помощью этого класса. Если вам захочется перейти на класс Box, замените в тексте программы название класса (вместо Panel пишите Box), но не забудьте, что это может иметь некоторые последствия для внешности вашего приложения.

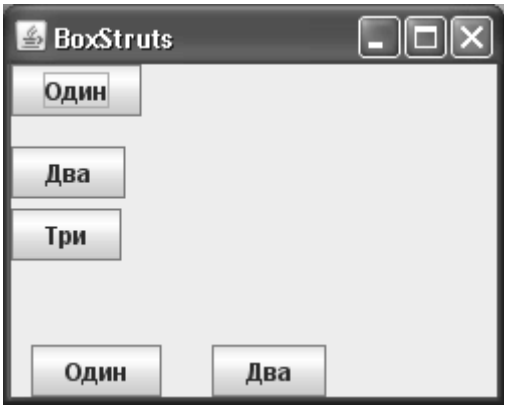
## Расстояние между компонентами

Нехорошо конечно, что при блочном расположении компоненты располагаются вплотную друг к другу. Получается неряшливо. Хотелось бы иметь возможность указывать расстояние между компонентами, причем между разными компонентами разное. Здесь нам на помощь вновь придет класс Box, неизменный помощник менеджера блочного расположения. В нем определены еще несколько статических методов, позволяющих создавать специальные невидимые компоненты определенного размера, которые помещаются между видимыми компонентами и разделяют их. Существует три типа таких компонентов — это *распорки* (struts), *заполнители* (glues) и *фиксированные области* (rigid areas).

Для начала рассмотрим распорки. Используют их как «палочки» (если хотите, «спички») определенного размера, которые ставят между компонентами и разделяют их. Так как при блочном расположении компоненты могут располагаться как по горизонтали, так и по вертикали, то и распорки бывают вертикальные и горизонтальные. Рассмотрим пример:

```
// BoxStruts.java
// Использование распорок при блочном расположении
import javax.swing.*;
// используем наш новый класс
import com.porty.swing.BoxLayoutUtils;

public class BoxStruts extends JFrame {
 public BoxStruts() {
     super("BoxStruts");
     setSize(250, 200);
     setDefaultCloseOperation( EXIT_ON_CLOSE );
     // панель с вертикальным блочным расположением
     JPanel p = BoxLayoutUtils.createVerticalPanel();
     p.add(new JButton("Один"));
     // создание вертикальной распорки
     p.add(Box.createVerticalStrut(15));
     // новый компонент и распорка другого размера
     p.add(new JButton("Два"));
     p.add(Box.createVerticalStrut(5));
     p.add(new JButton("Три"));
     // панель с горизонтальным блочным расположением
     JPanel p2 = BoxLayoutUtils.createHorizontalPanel();
     // распорки можно ставить и перед компонентами
     p2.add(Box.createHorizontalStrut(10));
     p2.add(new JButton("Один"));
     // создание горизонтальной распорки
     p2.add(Box.createHorizontalStrut(25));
     p2.add(new JButton("Два"));
     // добавляем панели на север и юг окна
     add(p, "North");
     add(p2, "South");
     // выводим окно на экран
     setVisible(true);
 }
 public static void main(String[] args) {
    SwingUtilities.invokeLater(
     new Runnable() {
      public void run() { new BoxStruts(); } });
 }
}
```

В примере создаются две панели с вертикальным и горизонтальным блочным расположением. Для этого используется класс BoxLayoutUtils, описанный в предыдущем разделе. Между компонентами помещены распорки различных размеров, созданные классом Box — для создания вертикальных распорок предназначен метод createVerticalStrut(int), в качестве параметра которого указывается размер распорки в пикселах. Аналогично работает метод createHorizontalStrut(int), создающий горизонтальную распорку. Также никто не запрещает использовать распорки перед всей группой компонентов или после нее, чтобы отодвинуть их от границ окна.



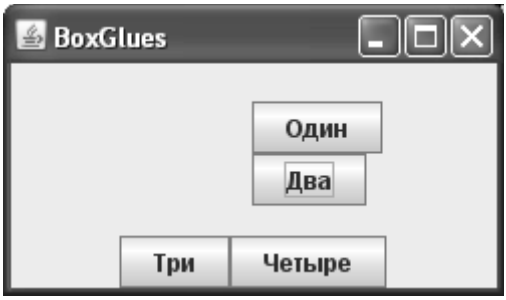

Размеры распорок задаются в пикселах — как же определить нужные расстояния, чтобы интерфейс выглядел достойно? Основная функция расстояния — эстетическая, призванная улучшить восприятие интерфейса человеком. К счастью, не нужно быть дизайнером или нанимать его, что справиться с задачей подбора расстояний между компонентами. Создатели практически любой системы пользовательского интерфейса публикуют рекомендации, предлагающие определенный дизайн компонентов и определенные расстояние между ними. Не является исключением и используемый по умолчанию в Swing внешний вид Metal. Фирма Sun выпустила набор рекомендаций по созданию пользовательского интерфейса в этой системе, и чуть позже мы подробнее узнаем об этом. Как правило, есть свои рекомендации и у внешних видов сторонних производителей.

Заполнители окончательно возводят блочное расположение в ранг универсального. Они работают как своего рода «пружины», помещаемые вертикально или горизонтально, соответственно текущему расположению, и раздвигают компоненты, разделяя все оставшееся в контейнере свободное место между собой. Именно с помощью заполнителей можно центрировать компоненты, размещать их не сверху, а снизу, равномерно распределять между ними все доступное пространство и т. д. Странно только, что создатели класса BoxLayout использовали название «glue» (заполнитель), а не оставили название «spring» (пружина), используемое в языке Smalltalk, откуда и пришла эта идея. Пояснит сказанное простой пример:

```
// BoxGlues.java
// Использование заполнителей
import javax.swing.*;
import com.porty.swing.BoxLayoutUtils;

public class BoxGlues extends JFrame {
```

```
    public BoxGlues() {
        super("BoxGlues");
        setSize(250, 200);
        setDefaultCloseOperation( EXIT_ON_CLOSE );
        // панель с вертикальным блочным расположением
        // в нее поместим все остальные панели
        JPanel main = BoxLayoutUtils.createVerticalPanel();
        // вертикальная панель
        JPanel pVert = BoxLayoutUtils.createVerticalPanel();
        // заполнитель перед компонентами отодвинет
        // их вниз
        pVert.add(Box.createVerticalGlue());
        pVert.add(new JButton("Один"));
        pVert.add(new JButton("Два"));
        // горизонтальная панель
        // теперь можно разместить компоненты по центру
        JPanel pHor = BoxLayoutUtils.createHorizontalPanel();
        pHor.add(Box.createHorizontalGlue());
        pHor.add(new JButton("Три"));
        pHor.add(new JButton("Четыре"));
        pHor.add(Box.createHorizontalGlue());
        // укладываем панели вертикально
        main.add(pVert);
        main.add(Box.createVerticalStrut(15));
        main.add(pHor);
        // добавляем панель в центр окна
        getContentPane().add(main);
        // выводим окно на экран
        setVisible(true);
    }
    public static void main(String[] args) {
        new BoxGlues();
    }
}
```

Запустив программу с этим примером, вы увидите, как работают заполнители, и убедитесь, что пользоваться ими совсем просто. В вертикальную панель сначала помещается заполнитель, а затем пара кнопок. Заполнитель занимает все свободное место, и кнопки отодвигаются к самому низу панели. В горизонтальную панель вводятся два заполнителя по краям группы кнопок, они делят свободное место поровну (когда вы используете не один, а несколько заполнителей, они всегда делят свободное место поровну — то есть работают как пружины), и кнопки оказываются в центре панели. Создаются заполнители двумя статическими методами класса Box — вертикальный заполнитель создается методом Box.createVerticalGlue(), а горизонтальный — методом Box.createHorizontalGlue().




Кстати, если перед вызовом метода setVisible() вы вызовете метод pack(), придающий окну и всем содержащимся в нем компонентам оптимальный размер (оптимальный размер компонентов, как вы помните, возвращает метод getPreferredSize()), то увидите, что заполнителей будто и след простыл. Так оно и должно быть — когда места в контейнере хватает ровным счетом для того, что разместить все видимые компоненты и распорки, то заполнитель ведет себя очень скромно — его минимальный и предпочтительный размеры становятся нулевыми.

## ВНИМАНИЕ

Заполнители блочного расположения работают, потому что имеют неограниченный максимальный размер. Когда в контейнере с блочным расположением имеется избыточное пространство, оно делится поровну между компонентами, которые еще могут «расти», то есть не достигли максимального размера. Некоторые компоненты, например панели прокрутки с таблицами, также могут расти, и надо учитывать, что свободное пространство поделится поровну между ними и «заполнителями». Когда в контейнере не остается компонентов, способных расти, все оставшееся пространство размещается после компонентов, снизу или справа, в зависимости от выбранной оси BoxLayout.

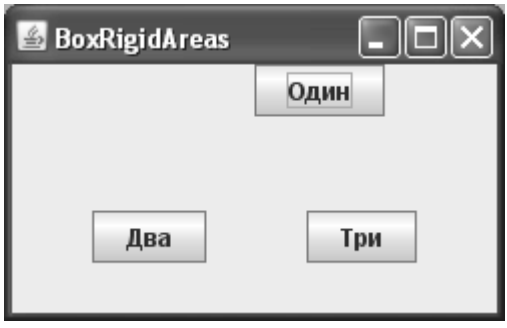
После запуска программы с примером у вас, возможно, возник еще один вопрос. Почему панели располагаются так необычно относительно друг друга? Вертикальный ряд кнопок находится не у левой границы окна, как в предыдущих примерах, а как-то не очень понятно, ближе к его середине. Дело в том, что для полного контроля за действиями менеджера расположения в классе Component предусмотрено еще два параметра, задающие выравнивание по осям X и Y, и менеджер BoxLayout их активно использует. Как направить эти параметры себе на пользу, мы узнаем в следующем разделе.

Наконец, в «шкатулке» у класса Box остался еще одно средство задания расстояний — фиксированные области. Назначение их нетрудно понять из названия — это обычные компоненты, только невидимые, и их размер вы задаете сами. Казалось бы, польза от них не так уж велика, в подавляющем большинстве случаев вполне достаточно распорок и заполнителей, однако вскоре мы увидим что все не так просто. А сейчас мы рассмотрим пример с фиксированными областями:

```
// BoxRigidAreas.java
// Пример использования фиксированных областей
import javax.swing.*;
import com.porty.swing.BoxLayoutUtils;
import java.awt.*;
```

```
public class BoxRigidAreas extends JFrame {
  public BoxRigidAreas() {
    super("BoxRigidAreas");
    setSize(250, 200);
    setDefaultCloseOperation( EXIT_ON_CLOSE );
    // вертикальная панель
    JPanel pVert = BoxLayoutUtils.createVerticalPanel();
    pVert.add(new JButton("Один"));
    // горизонтальная панель
    JPanel pHor = BoxLayoutUtils.createHorizontalPanel();
    pHor.add(new JButton("Два"));
    // размер пространства задается в виде объекта
    // Dimension из пакета java.awt
    pHor.add(Box.createRigidArea(new Dimension(50,120)));
    pHor.add(new JButton("Три"));
    pVert.add(pHor);
    // добавляем вертикальную панель в центр окна
    add(pVert);
    // выводим окно на экран
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new BoxRigidAreas(); } });
  }
}
```

В этом примере продемонстрирован наиболее вероятный вариант использования фиксированной области — в виде двух совмещенных распорок, горизонтальной и вертикальной.




Если вы знаете, что вам нужно разделить компоненты в обоих направлениях, можно ограничиться одной фиксированной областью. Обратите внимание, когда будете

запускать программу с примером — вертикальная панель снова уплывает куда-то к середине окна. Вскоре мы возьмем эту ситуацию под контроль, и любое расположение будет у нас в руках.

## Проблема с распорками

Распорки совершенно необходимы при создании интерфейсов с применением блочного расположения компонентов, и как мы посмотрели, работа их выглядит вполне прозрачно — мы просто вставляем своеобразные «спички» между компонентами. Однако после рассмотрения заполнителей оказывается, что возникает недоразумение с распорками, созданными классом Box. Дело в том, что максимальный размер по второй оси (к примеру, по вертикали для горизонтальной распорки) неограничен. Как мы уже выяснили, блочное расположение при наличии свободного пространства в контейнере распределяет его поровну между компонентами, которые могут «расти», еще не достигнув максимального размера. Так вот, если в панели с блочным расположением по горизонтали имеются стандартные распорки, общий максимальный размер панели по вертикали будет неограничен, и это, как правило, совсем не тот эффект, которого мы хотим добиться. Тоже самое случится и с вертикальными распорками в вертикальной панели.

Поэтому лучше создать распорки самим, не применяя стандартные. Тут нам помогут фиксированные области, у которых одно расстояние будет распоркой, а второе всегда нулевым. Именно такое поведение нам и требуется при разделении компонентов. Добавим в наш вспомогательный класс BoxLayoutUtils еще пару удобных методов:

```
// com/porty/swing/BoxLayoutUtils.java
// Класс для удобной работы с менеджером BoxLayout
package com.porty.swing;
import javax.swing.*;

public class BoxLayoutUtils
{
  // создает горизонтальную распорку
  // фиксированного размера
  public static Component createHorizontalStrut(int size) {
    return Box.createRigidArea(new Dimension(size, 0));
  }
  // создает вертикальную распорку
  // фиксированного размера
  public static Component createVerticalStrut(int size) {
    return Box.createRigidArea(new Dimension(0, size));
  }
// далее идут методы, описанные ранее
…
}
```

Наши распорки созданы на основе уже знакомых нам фиксированных областей и будут выполнять те же функции, что и стандартные распорки из класса Box, не устанавливая при этом неограниченным общий максимальный размер панели. А если нам понадобится увеличить максимальный размер панели, мы всегда можем сделать это вручную.

## Выравнивание компонентов по осям

Выравнивание по осям — достаточно каверзный аспект работы менеджера BoxLayout, и если не знать деталей, расположение вполне может получиться абсолютно противоположным тому, что изначально задумывалось. Вам может даже показаться, что поведение компонентов при блочном расположении вообще непредсказуемо. Впрочем, не все так страшно, и мы постараемся без особых трудностей справиться с выравниванием по осям.

Каждый компонент (унаследованный от класса java.awt.Component) имеет два параметра, служащие для его выравнивания по осям X и Y, то есть для его более точного размещения в контейнере. Что это за оси? Рассмотрим сначала вертикальную панель, где компоненты располагаются сверху вниз. Мы можем захотеть изменить горизонтальную позицию компонентов (то есть позицию по оси X) такой панели (вертикальные позиции жестко заданы положением компонентов, распорок и заполнителей) — поместить их слева, справа или по центру. Так вот, сделать это путем выравнивания по осям *нельзя*.

### ВНИМАНИЕ

> Выравнивание компонентов по осям не позволяет, как это может показаться на первый взгляд, изменить позицию компонентов относительно контейнера. Эти параметры задают положение компонентов относительно друг друга, то есть относительно остальных компонентов, размещенных в контейнере. То есть попытки использовать механизм выравнивания по осям так же, как и при последовательном расположении (FlowLayout), приведут к неудаче.

На самом деле расположение компонента с определенным выравниванием будет зависеть исключительно от выравнивания и размеров остальных компонентов. Для горизонтального выравнивания в классе Component определены следующие константы:

- Component.CENTER_ALIGNMENT — вертикальная ось будет проходить через середину компонента;
- Component.LEFT_ALIGNMENT — компонент будет прижат к вертикальной оси своей левой границей;
- Component.RIGHT_ALIGNMENT — компонент будет прижат к вертикальной оси своей правой границей.

При блочном расположении положение вертикальной оси определяется по компонентам с центральным выравниванием, а уже после этого компоненты с другими параметрами выравнивания размещаются относительно этой оси. Окончательно разъясняет ситуацию рис. 7.2.

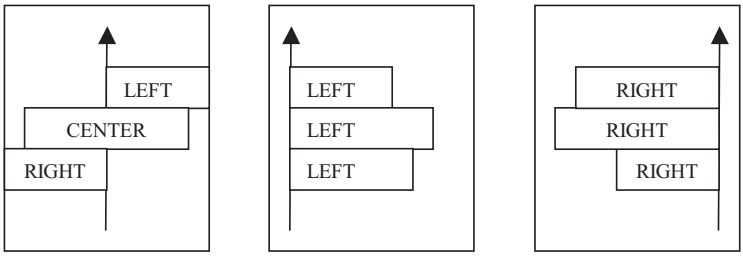


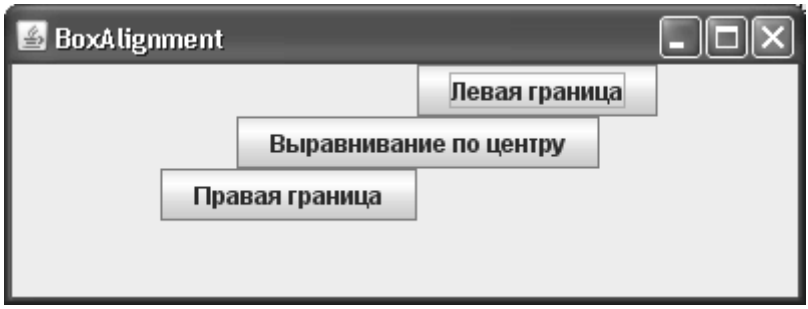

**Рис. 7.2.** Блочное расположение с горизонтальным выравниванием

Наиболее общий случай показан на рисунке слева. Здесь также можно безболезненно удалить один из трех компонентов — Когда в контейнере присутствуют два или три компонента с неодинаковым выравниванием, расположение всегда имеет такой вид. На двух других схемах показано, что происходит, если все компоненты имеют одинаковое выравнивание, причем компоненты с выравниванием по центру отсутствуют. Такое расположение действительно напоминает расположение FlowLayout (только не по горизонтали, а по вертикали), оно интуитивно понятно и предсказуемо. Блочное расположение становится одновременно мощным и достаточно простым именно тогда, когда все компоненты в контейнере выровнены согласованно, и в дальнейшем мы так и будем задавать параметры выравнивания. А пока, чтобы не быть голословными, убедимся, что все действительно так и работает:

```
// BoxAlignment.java
// Как блочное расположение обеспечивает выравнивание
// компонентов по осям
import javax.swing.*;
import com.porty.swing.BoxLayoutUtils;

public class BoxAlignment extends JFrame {
 public BoxAlignment() {
     super("BoxAlignment");
     setSize(400, 150);
     setDefaultCloseOperation( EXIT_ON_CLOSE );
     // вертикальная панель
     JPanel pv = BoxLayoutUtils.createVerticalPanel();
     // кнопка с выравниванием по левой границе
     JButton jb = new JButton("Левая граница");
     jb.setAlignmentX(LEFT_ALIGNMENT);
     pv.add(jb);
     // копка с центральным выравниванием
     jb = new JButton("Выравнивание по центру");
     jb.setAlignmentX(CENTER_ALIGNMENT);
     pv.add(jb);
     // наконец, кнопка с выравниванием по правому краю
     jb = new JButton("Правая граница");
     jb.setAlignmentX(RIGHT_ALIGNMENT);
     pv.add(jb);
     // добавляем панель в центр окна
     add(pv);
     // выводим окно на экран
     setVisible(true);
 }
 public static void main(String[] args) {
    SwingUtilities.invokeLater(
```

```
        new Runnable() {
         public void run() { new BoxAlignment(); } });
    }
}
```

Запустите программу с примером и посмотрите на результат. Чтобы окончательно освоиться с выравниванием по осям при блочном расположении, вы можете поэкспериментировать с этим примером, изменяя выравнивание компонентов по своему вкусу. Если у вас есть под рукой хорошая среда визуального построения пользовательского интерфейса с поддержкой блочного расположения (например, NetBeans), проведите опыты с ее помощью.




Аналогично обстоит дело с вертикальным выравниванием (по оси Y). Требуется такое выравнивание в панелях с горизонтальным расположением компонентов, и позволяет изменить вертикальные позиции компонентов относительно друг друга (горизонтальные позиции задаются самим расположением). В классе Component для вертикального выравнивания определены следующие константы:

- Component.CENTER_ALIGNMENT — горизонтальная ось будет проходить через середину компонента;
- Component.TOP_ALIGNMENT — компонент будет прижат к горизонтальной оси своей верхней границей;
- Component.BOTTOM_ALIGNMENT — компонент будет прижат к горизонтальной оси своей нижней границей.

Как и ранее, для компонента с центральным выравниванием ось будет проходить через его середину (рис. 7.3, слева).

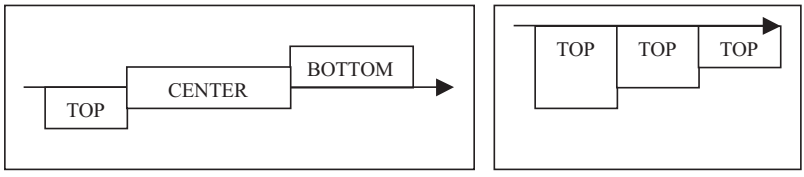



**Рис. 7.3.** Блочное расположение с вертикальным выравниванием

Если все компоненты имеют выравнивание TOP_ALIGNMENT, то ось проходит сверху контейнера, а все компоненты прижимаются к ней верхней границей, как показано справа на рис. 7.3. Схема для выравнивания BOTTOM_ALIGNMENT выглядит аналогично,

только ось проходит снизу контейнера, а все компоненты прижимаются к ней нижней границей. Так же как для горизонтального, для вертикального выравнивания компонентов можно сказать, что использовать его просто, когда все компоненты в контейнере имеют одинаковое выравнивание.

Кстати, если вы просмотрите документацию по классу Component, то увидите, что для задания параметров выравнивания компонентов используются числа с плавающей запятой (float) от 0 до 1. Можно сказать, что они определяют, какая часть компонента находится за осью. Так, например, константы RIGHT_ALIGNMENT и TOP_ALIGNMENT равны нулю. Это значит, что весь компонент находится по одну сторону от оси (с правой или с нижней стороны, соответственно). Если увеличивать значения выравнивания, то компонент будет потихоньку «переползать» за ось и, в конце концов, окажется по другую ее сторону. Константы LEFT_ALIGNMENT и BOTTOM_ALIGNMENT как раз равны единице (то есть компонент находится целиком за осью). Так что это гораздо более тонкий инструмент, чем кажется поначалу, — значения констант соответствуют наиболее вероятным вариантам выравнивания компонентов, но вы также можете и напрямую указывать свои значения, что позволяет очень точно выстраивать компоненты относительно друг друга.

Хотя все это кажется интересным, реальная польза от таких изощренных возможностей блочного расположения практически нулевая. Невероятно сложно, а иногда вообще невозможно предугадать, как выстроятся компоненты в контейнере, чтобы правильно использовать для них разные варианты выравнивания.

С другой стороны, нет ничего проще совмещения нескольких панелей с блочным расположением, содержащих по несколько компонентов с одинаковым выравниванием. Любой пользовательский интерфейс можно разбить на горизонтальные и вертикальные полосы с компонентами, применяя для них подобные панели. Именно такой способ создания интерфейса дает наибольшие преимущества разработчику, и мы всячески рекомендуем его вам. А чтобы задание параметров выравнивания компонентов не отнимало у вас слишком много времени и сил, добавим в наш вспомогательный класс BoxLayoutUtils еще пару методов:

```
// com/porty/swing/BoxLayoutUtils.java
// Класс для удобной работы с менеджером BoxLayout
package com.porty.swing;

import javax.swing.*;

public class BoxLayoutUtils
{
 // задает единое выравнивание по оси X для
 // группы компонентов
 public static void setGroupAlignmentX(float alignment,
           JComponent... cs) {
    for (JComponent c : cs) {
      c.setAlignmentX(alignment);
    }
 }
 // задает единое выравнивание по оси Y для
 // группы компонентов
 public static void setGroupAlignmentY(float alignment,
```

```
            JComponent... cs) {
        for (JComponent c : cs) {
          c.setAlignmentY(alignment);
        }
    }
// далее идут методы, описанные ранее
…
}
```

Теперь, создавая интерфейс с помощью менеджера блочного расположения, смело добавляйте любые нужные вам компоненты, а потом просто передайте массив ссылок на них в один из наших методов. После этого все будет работать именно так, как нужно, и даст отличный результат. Остается заметить, что более или менее сложный интерфейс не обойдется одной панелью с блочным расположением, и в таком случае для всех вложенных панелей также следует задать единое выравнивание.

Если вы помните, в предыдущем разделе был показан пример, который почему-то работал не так, как нам хотелось. Теперь понятно, почему — у панели было центральное выравнивание, а у кнопки — выравнивание по левой границе. Используя новые методы класса BoxLayoutUtils, можно легко согласовать параметры выравнивания всех компонентов, и они займут положенные им места, задуманные нами с самого начала.

## Расположение по группам: GroupLayout

Расположение GroupLayout было специально создано в поддержку визуального редактора для расположения компонентов Swing в редакторе NetBeans (редактор назывался Matisse), а затем перекочевало в JDK. Суть его состоит в том, что компоненты располагаются в вертикальных и горизонтальных группах. Похоже на совмещение нескольких блочных расположений в одном, однако особенностью является то, что один и тот же компонент вполне может участвовать в нескольких группах одновременно. К примеру, текстовые поля могут участвовать в горизонтальных группах вместе с другими компонентами, например надписями и кнопками, и одновременно с этим в вертикальных группах, что позволяет сделать их размер одинаковым. В группе можно указывать расстояние между компонентами. Окончательный вариант расположения создается двумя главными группами, вертикальной и горизонтальной.

Идея расположения по группам и их перекрытия неплоха, но скорее подходит для визуального редактора, собственно, для него это расположение и было разработано. При ручной разработке становится трудно читать код, так как компоненты располагаются на экране не совсем явно, и нужно выяснять, в каких группах они принимают участие. Так что создавая интерфейс в коде, лучше пользоваться другими альтернативами.

## Гибкая сетка: GridBagLayout

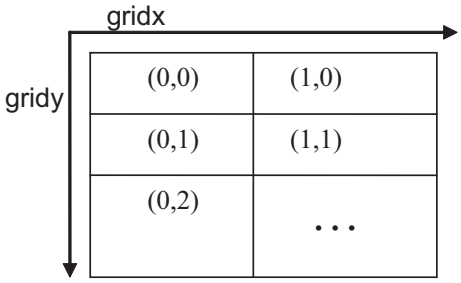
Как правило, расположение многих созданных нами объектов в современных интерфейсах практически всегда выровнено по горизонтали и вертикали, то есть компоненты находятся в «сетке» с ячейками разных размеров. Мы группируем компоненты по назначению, выравнивая их по осям (за исключением авангардных или неряшливых интерфейсов, но их мы не станем рассматривать).

Создатели GridBagLayout приняли во внимание эти факты, и создали менеджер расположения, позволяющий нам разместить компоненты в сетке с ячейками гибких размеров, от нулевых до максимальных, позволенных контейнером. Формально, не считая совсем уже экстремальных вариантов, всегда можно представить любое расположение

компонентов в контейнере в виде такой сетки. Если, при взгляде на интерфейс, вы видите компоненты, выровненные по горизонтали или вертикали, одинаковых размеров, размеров, кратных размерам других компонентов, расположение в виде гибкой сетки может быть весьма кстати.

Управление расположением компонентов в GridBagLayout осуществляется отдельным объектом под названием GridBagConstraints, который полностью описывает расположение компонента в сетке и его особенности. Настроив его, вы передаете его в метод add() вместе с добавляемым в контейнер компонентом. К сожалению, именно этот объект делает GridBagLayout такой удобной целью для нападок, а код с его применением напоминает прекрасно перемешанное спагетти. Дело в том, что все настройки хранятся в нем в виде полей (что противоречит концепциям объектно-ориентированного программирования), имена этих полей иногда не соответствуют производимому им эффекту, а количество «волшебных цифр» и избыточной информации поражает воображение и лишает код читаемости.

Давайте начнем с положения компонента в таблице. Его позиция определяется номером ячейки по вертикали и горизонтали:



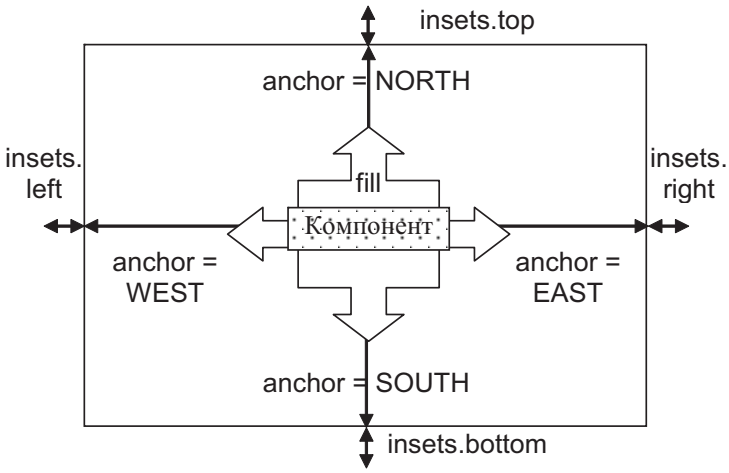

Координаты в таблице, таким образом, хранятся в полях gridx и gridy. Так как указывать каждый раз новые координаты несколько утомительно, GridBagConstraints любезно предоставляет нам константу RELATIVE, согласно которой компонент будет добавлен в тот же ряд и следующий столбец за последним добавленным компонентов. Такое поведение напомнит нам последовательное расположение FlowLayout, тем более что по умолчанию компоненту придается предпочтительный размер. По умолчанию, если вы не указываете значение полей gridx и gridy, как раз RELATIVE и применяется, располагая компоненты в один ряд.

Так как сетка гибкая, мы можем указать, какое количество ячеек занимает компонент по вертикали или горизонтали.

- gridwidth — количество ячеек, занимаемое компонентов по горизонтали, «длина» компонента в ячейках
- gridheight — количество ячеек, занимаемое компонентов по вертикали, «высота» компонента в ячейках

Очень часто компонент должен занять весь ряд или столбец, независимо от того, сколько там ячеек. Для этих случаев предусмотрено значение GridBagConstraints.REMINDER, которое и говорит, что компонент займет все оставшиеся ячейки по одной из осей или даже по обеим. Это тем более важно, что компонент в этом случае не зависит от появления большего количества ячеек в каком-то из рядом или столбцов, но все равно займет все ячейки.

Ячейки практически никогда не бывают одинакового размера, так как разнятся и размеры компонентов, и количество ячеек, ими занимаемое. Положение компонента в самой ячейке таблицы GridBagLayout определяется еще несколькими полями класса GridBagConstraints, и лучше всего увидеть, как они работают, на простой иллюстрации:




Итак, согласно иллюстрации мы получаем следующее:

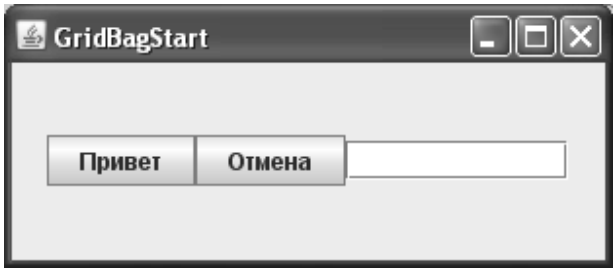
- fill — определяет, будет ли компонент заполнять все пространство ячейки, и если будет, то как. Можно применить константы HORIZONTAL, VERTICAL и BOTH, что будет соответственно обозначать заполнение всего размера ячейки по горизонтали, вертикали, или всего пространства ячейки (по обоим направлениям). По умолчанию значение этого поля в объекте GridBagConstraints равняется NONE, что означает, что компонент остается предпочтительного размера и не желает изменяться вместе с размерами ячейки.
- anchor — поле, значение которого указывает, к какому краю ячейки компоненту следует «прилепиться». На рисунке показаны наиболее вероятные варианты, которые, как мы видим, названы по сторонам света, к примеру, выравнивание по левому краю называется WEST (выравнивание на запад). Есть также возможность «прилеплять» компоненты к углам ячейки.
- insets — важные для любого прилично выглядящего интерфейса отступы между компонентами. Отступы задаются в виде объекта Insets, и определяют пространство между ячейками сверху, снизу, слева и справа, в пикселях.

Итак, основная идея GridBagLayout кажется понятной, и ячейки различных размеров, с полным контролем над их содержимым должны позволять создавать любые интерфейсы. Рассмотрим простой пример:

```
// GridBagStart.java
// Первые опыты с расположением GridBagLayout
import java.awt.*;
import javax.swing.*;

public class GridBagStart extends JFrame {
  public GridBagStart() {
    super("GridBagStart");
    // выход при закрытии окна
    setDefaultCloseOperation(EXIT_ON_CLOSE);
```

```
    // устанавливаем расположение компонентов
    setLayout(new GridBagLayout());
    // добавляем две кнопки, ячейки по умолчанию
    add(new JButton("Привет"));
    add(new JButton("Отмена"));
    // настройка ячейки для текстового поля
    GridBagConstraints textFieldConstraints =
        new GridBagConstraints();
    // заполнение ячейки по горизонтали
    textFieldConstraints.fill = GridBagConstraints.HORIZONTAL;
    // просим занять все оставшиеся ячейки
    textFieldConstraints.gridwidth =
        GridBagConstraints.REMAINDER;
    textFieldConstraints.weightx = 1.0f;
    add(new JTextField(10), textFieldConstraints);
    // выведем окно на экран
    setSize(400, 200);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new GridBagStart(); } });
  }
}
```

Здесь мы устанавливаем в качестве менеджера расположения GridBagLayout, добавляем две кнопки, причем, не указывая им явно объекта GridBagConstraints. Как мы помним, это означает, что они будут добавляться в ряд, друг за другом, и иметь предпочтительный размер (именно так по умолчанию настроены поля gridx, gridy и fill). Далее мы добавим текстовое поле, и хотим, чтобы оно заняло все оставшееся место контейнера. Для этого мы указываем в объекте GridBagConstraints что текстовое поле занимает все оставшиеся в ряду ячейки и заполняет ячейку по горизонтали. На первый взгляд, все просто и должно дать нужный результат. Запустив пример, мы увидим следующее:



Кнопки расположены именно так, как мы и ожидали, а вот с текстовым полем возникла некоторая проблема. Оно не растянуто на всю оставшуюся ширину окна, несмотря на то, что мы подходящим образом задали поля gridwidth и fill, и все равно осталось предпочтительного размера (длиной в 10 символов ввода, что мы задали в конструкторе для поля JTextField). Более того, если вы сделаете окно меньше, вы увидите, что при нехватке пространства поле сжимается до минимального размера «прыжком», не уменьшая свои размеры плавно. Пример показывает нам, что по умолчанию менеджер GridBagLayout ведет себя следующим образом:

- компоненты имеют предпочтительный размер и не более
- компоненты располагаются в центре контейнера
- при нехватке места компонент уменьшается до минимального размера (у кнопок он совпадает с обычным, и «прыжка» нет, а у текстового поля зависит и от текста, в нем содержащегося, попробуйте набрать в нем текст и снова поменять размер окна).

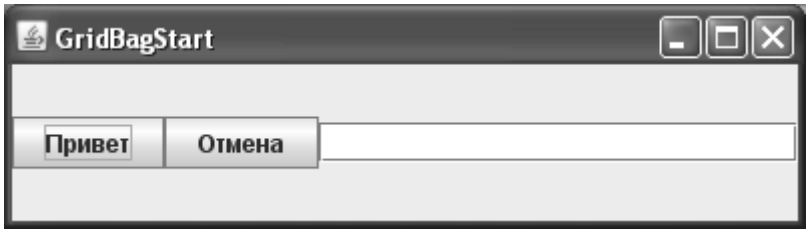
Как же заставить GridBagLayout делать компоненты больше их предпочтительного размера? Для этого есть пара несколько загадочных полей, определяющих так называемый «вес» ячейки.

- weightx, weighty — дробные числа, которые указывают менеджеру, сколько свободного места из контейнера следует отдавать ячейке, если таковое имеется. Числа эти нормированные — менеджер GridBagLayout собирает все значения из всех ячеек и вычисляет процент от максимального значения. Как правило, в качестве максимального значения используется единица — это 100% отдача всего свободного пространства ячейке. Числа от нуля до единицы будут означать проценты от свободного пространства, которые отдаются ячейке. Ну а по умолчанию значения полей weightx и weighty равны нулям, а это значит, что ячейка никогда не станет больше предпочтительного размера своего компонента и отступов. Интересным свойством полей weightx и weighty является и то, что если в контейнере, управляемом GridBagLayout, содержится хотя бы одна ячейка, в которой значения этих полей отличны от нуля, GridBagLayout займет все место контейнера. В противном случае будет занято лишь то место, которого достаточно для предпочтительных размеров компонентов, причем все компоненты будут располагаться по центру контейнера, что мы уже имели возможность наблюдать в нашем первом примере.

Теперь, кажется, понятно, что же мы упустили. Если мы хотим чтобы текстовое поле занимало весь остаток места по горизонтали, нам нужно установить поле weightx в 100%, в зависимости от того, какое число вы используете в качестве максимума. Если принять за 100% единицу, то дополнительная строчка кода будет выглядеть так:

```
textFieldConstraints.weightx = 1.0f;
```

А результат будет таков:



Изменяя размеры окна, вы сможете убедиться в том, что текстовое поле занимает всю его ширину, и размер его больше не «прыгает» и не зависит от количества набранного в нем текста.

Идея GridBagLayout не так и сложна, и большинство расположений довольно легко поддаются ему. Однако основной проблемой является объект для описания ячеек GridBagConstraints. Названия его полей не всегда ясны (к примеру, gridwidth и weight довольно двусмысленны), а значения для них подстегивают к добавлению к коду цифр (уже упомянутые поля, плюс поле insets), и мы сталкиваемся с проблемой «волшебных чисел», когда цифры в коде не несут очевидной смысловой нагрузки. Огромное количество объектов GridBagConstraints и кода для их настройки, щедро перемешанные с инициализацией компонентов и деловой логикой программы, становятся настоящей головной болью. Поэтому, если есть возможность, всегда стоит отказываться от чистого, «без примесей» расположения GridBagLayout, в пользу описанного нами выше подхода из нескольких вложенных панелей с блочным расположением BoxLayout. Кода будет не больше, а его смысл и простота его чтения намного лучше.

Впрочем, менеджер GridBagLayout является стандартной частью JDK и навсегда там останется. Если вы пишете интерфейс не с нуля и вынуждены поддерживать существующий код с применением GridBagLayout, или не можете переключиться на подход с сложенными воедино «полосками», по крайней мере стоит упростить задачу и сделать чтение и поддержку кода проще, все же не отказываясь от GridBagLayout.

Основная проблема состоит в том, что уровень объекта GridBagConstraints слишком низок, и вместо очевидных операций (растянуть компонент, добавить промежуток между компонентами, выровнять его по левому краю), мы манипулируем полями, и эффект от этих действий не «прочитать» напрямую. Но можно создать вспомогательный объект для работы с GridBagConstraints, и вызывая его методы, намного улучшить и упростить нашу работу с менеджером GridBagLayout. Давайте попробуем:

```
// com/porty/swing/GridBagHelper.java
// Вспомогательный класс, позволяющий писать
// качественный код для расположения GridBagLayout
package com.porty.swing;

import javax.swing.*;
import java.awt.*;

public class GridBagHelper {
  // координаты текущей ячейки
  private int gridx, gridy;
  // настраиваемый объект GridBagConstraints
  private GridBagConstraints constraints;

  // возвращает настроенный объект GridBagConstraints
  public GridBagConstraints get() {
    return constraints;
  }
  // двигается на следующую ячейку
  public GridBagHelper nextCell() {
    constraints = new GridBagConstraints();
```

```
    constraints.gridx = gridx++;
    constraints.gridy = gridy;
    // для удобства возвращаем себя
    return this;
  }
  // двигается на следующий ряд
  public GridBagHelper nextRow() {
    gridy++;
    gridx = 0;
    constraints.gridx = 0;
    constraints.gridy = gridy;
    return this;
  }
  // раздвигает ячейку до конца строки
  public GridBagHelper span() {
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    return this;
  }
  // заполняет ячейку по горизонтали
  public GridBagHelper fillHorizontally() {
    constraints.fill = GridBagConstraints.HORIZONTAL;
    return this;
  }
  // вставляет распорку справа
  public GridBagHelper gap(int size) {
    constraints.insets.right = size;
    return this;
  }
  … остальные вспомогательные методы
}
```

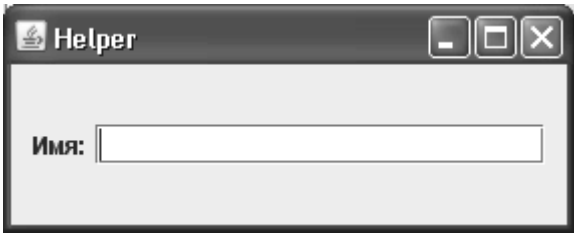
Класс GridBagHelper весьма прост. Он работает с объектом GridBagConstraints, настраивая его поля и скрывая некоторые не очень очевидные цифры и манипуляции с полями, вместо этого предоставляя нам весьма понятные методы, к примеру, nextRow() переходит на первую ячейку следующей строки таблицы, а gap() вставляет распорку справа от компонента. Для работы с этим объектом нужно создать его экземпляр, для каждой новой ячейки вызывать метод nextCell() (именно там создается новый объект GridBagConstraints), а в конце, когда настройка компонента проведена, получить результат методом get(). Для того чтобы вызывать методы было удобнее, каждый из них возвращает ссылку на объект GridBagHelper, поэтому методы можно вызывать «в ряд».

В качестве маленького примера можно привести надпись и текстовое поле, с распоркой в 5 пикселей между ними:

```
GridBagHelper helper = new GridBagHelper();
helper.nextCell().gap(5);
frame.add(new JLabel("Имя:"), helper.get());
```

```
helper.nextCell().span();
frame.add(new JTextField(20));
```

При запуске мы получим такой результат:




Мы получим и распорку, не работая со всеми четырьмя размерностями объекта Insets, и скажем полю занять все оставшиеся ячейки более понятным методом span(), и прозрачный способ перехода по ячейкам. В полной версии класса GridBagHelper, которую вы можете найти в исходных текстах программ книги, есть и много других полезных методов с простыми названиями. Такой код не только во много раз быстрее писать, но и легче читать и обновлять. В будущем, если вы решите применить на практике GridBagLayout, класс GridBagHelper или его эквиваленты будут весьма кстати, а применять напрямую запутанный класс GridBagConstraints вряд ли можно рекомендовать. Чуть позже в этой главе, когда мы разработаем диалоговое окно для входа в систему, мы еще раз применим новый класс.

## *Быстрый язык MigLayout*

Сама идея расположения большинства объектов интерфейса по сетке, придавая им разное выравнивание, размеры, и, возможно, количество ячеек сетки, занимаемое ими, позволяет быстро набросать интерфейс, даже не имея никаких визуальных инструментов. Некогда, до достаточного развития таблиц стилей CSS, именно так разрабатывалось расположение содержимого многих Web-страниц.

Идея эта была реализована в Java с помощью GridBagLayout, однако способ указания всех атрибутов расположения компонентов в сетке неудачна, количество объектов огромно, и код уже через пять минут напоминает неудавшееся спагетти. Поддерживать подобный код крайне тяжело.

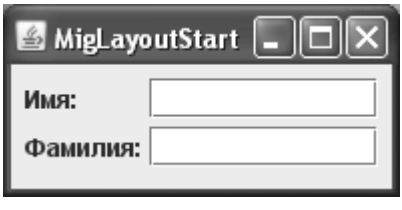
Однако сама идея неплоха, остается лишь более быстрым и удобным языком указывать менеджеру расположения, как и где должны располагаться компоненты в сетке. Мы уже создали вспомогательный класс, предлагающий удобными методами руководить действиями GridBagLayout. Также одним из удачных примеров реализации такого менеджера является бесплатно распространяемый менеджер MigLayout.

По сути, вспоминая наше рассмотрение табличного менеджера GridBagLayout и его основных концепций, можно сказать, что в целом мы представляем себе, как работает MigLayout. Другим способом осуществляется лишь описание ячеек таблицы — в виде строчек очень компактного, насыщенного смыслом текста, что без сомнения ускоряет процесс создания интерфейса. Вы добавляете компоненты примерно таким методом — add(компонент, "описание ячейки"). Более того, MigLayout обладает несравненно более широким набором функций, в сравнении со стандартными менеджерами Java. Дальнейшее знакомство мы проведем с помощью примера, тем более что язык MigLayout достаточно легко понять.

```
// MigLayoutStart.java
// Знакомство с MigLayout
```

```
import net.miginfocom.swing.MigLayout;
import javax.swing.*;

public class MigLayoutStart extends JFrame {
  public MigLayoutStart() {
    super("MigLayoutStart");
    // выход при закрытии окна
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // устанавливаем менеджер расположения
    setLayout(new MigLayout());
    // добавляем компоненты с описанием ячеек
    add(new JLabel("Имя:"), "gap, sg 1");
    add(new JTextField(10), "wrap");
    add(new JLabel("Фамилия:"), "gap, sg 1");
    add(new JTextField(10), "wrap");
    // выведем окно на экран
    pack();
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new MigLayoutStart(); } });
  }
}
```

Мы создаем окно и устанавливаем для него расположение MigLayout. Далее в окно добавляется две надписи JLabel, описывающие текстовые поля для ввода данных, и сами текстовые поля — типичный пример простого интерфейса. Для гармонии надписи обычно делают одинаковых размеров. Обратите внимание, как описываются ячейки при добавлении компонентов: в виде простой строки, команды разделяются запятыми, параметры команд следуют через пробел. Запустив пример, мы без особых усилий получим желаемый результат. Конечно же, для запуска вам понадобится скачать библиотеки MigLayout и добавить их в свой путь CLASSPATH, их легко найти на официальном сайте продукта в Интернете.




Мало того что мы в мановение ока сделали размеры надписей одинаковыми, выровняли надписи и текстовые поля, так еще в качестве приятного сюрприза MigLayout на

основе каких-то своих внутренних алгоритмов задал подходящие размеры между компонентами, как по горизонтали, так и по вертикали. Ни один стандартный менеджер расположения таким простым способом этого сделать не может. Откуда менеджер MigLayout берет размеры между компонентами, мы вскоре узнаем, а пока составим краткую таблицу наиболее полезных команд при описании ячеек интерфейса:

**Таблица 7.1.** Основные команды менеджера MigLayout

| Команда и параметры | Описание |
|---|---|
| gap [размеры] | Указывает, что после компонента необходимо вставить пустое пространство указанного размера. Не указывая размер, мы предлагаем расположению MigLayout самому определить этот размер наилучшим образом (вскоре мы увидим, откуда он берется). |
| wrap | Указание перейти после данного компонента на следующую строку сетки. По умолчанию вставляется пустой промежуток между строками. |
| sg [имя] | Указывает что компонент принадлежит группе компонентов с одинаковыми размерами, равными размеру самого большого компонента. У группы может быть свое имя. |
| span [количество ячеек] | Указывает, что компонент должен занять больше чем одну ячейку сетки |
| skip [количество ячеек] | Инструкция пропустить указанное количество ячеек (оставить их пустыми) и поместить компонент в следующую за ними ячейку |
| grow[x\|y] [вес] | Помечает, что компонент при наличии свободного места в контейнере желает «расти» в размерах, по оси X или Y, или по обоим. У «желания роста» есть весовой коэффициент, который работает аналогично тому, как действует поле weight в стандартном менеджере GridBagLayout. Интересно, что по умолчанию контейнер с менеджером MigLayout не умеет «расти», для этого необходимо задать специальный флаг в конструкторе MigLayout. |

Посмотрев на таблицу и сравнив ее с нашим примером, мы увидим, что создали сетку из двух столбцов и двух строк, а надписи находятся в группе с одинаковым размером. Расстояния между строками и столбцами были заданы автоматически, хотя всегда есть возможность задать их вручную, и не только в пикселах, но даже и в миллиметрах и других единицах.

Как мы видим, MigLayout может быть очень быстрым и невероятно продуктивным. Он обладает огромным количеством команд, которые достаточно очевидны, если принимать во внимание то, что расположение идет по гибкой сетке. Найдя сайт MigLayout в Интернете, вы сможете найти там удобную краткую сводку всех его команд и с успехом применить их на практике. Конечно, бывают случаи, когда расположение по сетке накладывает ограничения, но в таком случае всегда можно применить технику вложенных расположений.

## *Общий подход*

Теперь, когда мы узнали, какие менеджеры расположения предоставляет язык Java и его библиотеки, и как ими пользоваться, осталось выработать общий подход, который позволил бы «расколоть» любой интерфейс, с блеском проведя его разработку.

Для этого рассмотрим пример, достаточно объемный, чтобы можно было пройти все этапы создания интерфейса полностью. Попытаемся создать диалоговое окно входа в систему — довольно часто используемый элемент графического интерфейса. Обычно в нем присутствует два текстовых поля для ввода имени пользователя и его пароля,

а также пара кнопок, чтобы пользователь мог указать, когда он будет готов к продолжению работы.

Прежде всего, необходимо набросать примерный внешний вид будущего интерфейса. Конечно, можно обойтись и без этого, но тогда легко увлечься и в процессе разработки обнаружить, что вы полностью потеряли контроль над тем, что делаете, особенно когда дело касается сложных вариантов расположения. По поводу способа дизайна пользовательских интерфейсов сломано немало копий, защищены докторские диссертации и написаны тысячи статей, но область эта по прежнему сродни искусству. На сайте книги ipsoftware.ru вы найдете ссылки на некоторые интересные статьи. Существуют даже отдельные компании, занимающиеся так называемым «usability» (с трудом переводимое слово, что-то вроде «удобство в использовании»), означающим создание интерфейса, максимально удобного и понятного пользователю. Компании дизайнеров предлагают различные варианты внешнего вида интерфейсов. Одна часто программисту самому приходится проектировать интерфейс.

Если вы точно не уверены, что вам нужно, хотя уже знаете, из каких элементов будет состоять диалоговое окно, то можете руководствоваться базовыми принципами разработки пользовательских интерфейсов, а именно их простотой, стандартными компонентами, к которыми привыкли все пользователи, и очевидностью действий, которые интерфейс позволяет (можно назвать все это принципом *«не изумлять пользователя»*). Идеально также опросить будущих пользователей приложения, если это возможно на этапе проектирования. Это не совет на все случаи жизни, но очень часто самый простенький интерфейс оказывается на удивление симпатичным и удобным в работе. Попытайтесь, встав на место пользователя, воспроизвести его цепочку размышлений (это, в частности, поможет выяснить, в каком порядке компоненты должны передавать друг другу фокус ввода), например, при входе в систему так и «тянет» ввести свое имя, затем вспомнить пароль, а после этого поискать глазами что-то для продолжения работы. Руководствуясь такими нехитрыми размышлениями, получаем набросок диалогового окна, представленный на рис. 7.4.

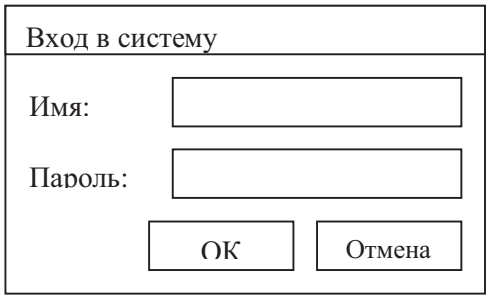



**Рис. 7.4.** Набросок диалогового окна для входа в систему

При создании пользовательского интерфейса можно пойти двумя путями. Можно самому быть «законодателем мод» в своем приложении, определяя, как и что будет в нем выглядеть и функционировать. Такой подход особенно хорош, если вы специально разработаете для своих приложений собственный внешний вид, однако это и долго, и дорого. А можно принять стандарт создателя Java — фирмы Sun (теперь уже Oracle), предложившей ряд рекомендаций относительно внешнего вида приложений. Команда Sun провела приличную исследовательскую работу, привлекла художников и дизайнеров, и нужно признать, что приложение, созданное в полном соответствии с этими рекомендациями, выглядит действительно гармонично. К тому же вы можете бесплатно за-

грузить достаточно обширный набор стандартной графики для меню, кнопок и панелей инструментов, что значительно повысит привлекательность вашего приложения, не увеличивая его стоимости и сроков разработки. Если вы решились, рассмотрим основные положения этих рекомендаций.

## Рекомендации от Sun

Рекомендации по созданию интерфейса Java-приложений (Java Look & Feel Design Guidelines) — это довольно объемный труд, выпущенный компанией Sun/Oracle специально для разработчиков Java-приложений. Там содержится действительно много полезной информации, необходимой при создании промышленных приложений. Рекомендации от Sun, в том числе, касаются способов расположения компонентов на форме, и особенно интересной, на взгляд автора, является информация о пространстве между компонентами. Следуя этим рекомендациям, можно в результате получить действительно привлекательный внешний вид. Однако учтите, что все рекомендации касаются только внешнего вида Metal или производного от него, используемого в Swing по умолчанию, а в случае другого внешнего вида стоит обратить внимание на рекомендации его производителя.

Рисуя набросок будущего диалогового окна для нашего приложения, мы, сознательно или подсознательно, оставили некоторое пространство между компонентами и отступили от границ окна. Это неудивительно — такой подход диктуется логикой, и нам кажется вполне естественным более акцентировано отделять логически малосвязанные компоненты, чем тесно связанные. Другой вопрос — насколько именно нужно раздвинуть эти компоненты, чтобы получить наилучший результат? Можно проводить долгие эксперименты или нанять дизайнеров. Можно этого не и не делать. Почему? Это уже проделано командой Sun. Никто не утверждает, что это — идеальный вариант и ничего лучше уже придумать невозможно. Но, тем не менее, он дает более чем приемлемый результат. Итак.

- *Расстояние между логически тесно связанными компонентами.* Чаще всего такими компонентами являются кнопки JButton, флажки JCheckBox и переключатели JRadioButton в группах, отвечающие за выбор пользователем одного из возможных вариантов (вряд ли вообще можно придумать более тесно связанные компоненты). Их следует отделять друг от друга на *5 пикселов* (в некоторых случаях рекомендуется 6 пикселов, однако это уже чрезмерные тонкости, а на современных дисплеях с высоким разрешением увидеть разницу в один пиксел можно, наверное, только с лупой). Наглядный пример в нашем случае — пара кнопок OK и Отмена.

- *Расстояние между группами компонентов.* Здесь имеются в виду те же самые группы из флажков и переключателей. Обычно, если они присутствуют в интерфейсе, то не поодиночке (каждая группа отвечает за выбор определенного условия). Так вот, расстояние между ними должно быть *12 пикселов* (здесь также есть вариант — 11 пикселов, связанный с неоднозначным восприятием человеком белых теней компонентов, но к нему вполне применим комментарий из предыдущего пункта). В нашем интерфейсе таких групп нет, но откройте любое диалоговое окно, предназначенное для настройки программы, и вы их увидите.

- *Пространство между границами окна и другими компонентами.* Задавая такое расстояние, мы акцентируем внимание на интерфейсе и отделяем его от границ окна, служащих для других целей. Те же соображения относятся и к панелям с рамками, набор которых в Swing просто потрясает воображение. Такие панели помогают четко структурировать интерфейс, хотя ни в коем случае не следует ими злоупотреблять. От границ этих панелей компоненты также нужно отделять. Для таких ситуаций используйте значение в *12 пикселов.*

- *Расстояние между «обычными» компонентами.* Под обычными компонентами подразумеваются те, которые, как правило, не встретишь в логически связанных группах. К ним относятся текстовые поля JTextField, надписи к компонентам JLabel, индикаторы процесса JProgressBar, ползунки JSlider и т. д. И здесь следует использовать уже встретившееся нам значение в *12 пикселов*.
- *Пространство между компонентами, выполняющими абсолютно разные функции.* Такие компоненты нечасто можно встретить в интерфейсе. Но нам повезло — в нашем диалоговом окне они есть. Текстовые поля служат для сбора информации от пользователя, а кнопки — для получения от него команд. Их целесообразно разделить более явно, и для этого используется немаленькое расстояние в *17 пикселов*. Обычно это — единственный вариант применения такого расстояния (отделение кнопок управления от «собирательной» части или информационной части интерфейса).
- *Внешний вид кнопок JButton.* Вообще-то это уже совершенно особый случай, и о нем следовало бы упомянуть в главе, посвященной элементам управления вообще и кнопкам в частности, но раз в нашем диалоговом окне есть кнопки, скажем об этом сейчас. Необходимо отделять смысловое наполнение кнопок (обычно текст, иногда значки) от их границ слева и справа на четкое расстояние в те самые *12 пикселов*. Когда вы просто создаете кнопку, в соответствие с внешним видом Metal расстояния получаются чуть большими, порядка 14 пикселов. Видимо, такое расстояние необходимо, чтобы выдержать общий «стиль» в 12 пикселов.

Это наиболее важные положения рекомендаций Sun, относящиеся к расположению компонентов (для интерфейса, который мы сейчас создаем, этого вполне достаточно). Но этими положениями рекомендации Sun не исчерпываются, в них вы можете найти множество полезной информации, касающейся того, как следует использовать прописные и строчные буквы в надписях компонентов, какой стиль письма подходит для пользовательских интерфейсов, какие должны быть значки для приложений с внешним видом Metal, как выглядит набор стандартных диалоговых окон и т. п. Также неплохо освещена сама философия пользовательского интерфейса. К сожалению, это не является темой данной книги (хотя кое-что их этих рекомендаций мы используем в главах, посвященных компонентам Swing, а за остальным можете обращаться на сайт java.sun.com). Аналогичные рекомендации вы сможете найти и для других известных внешних видов: Windows, Apple и остальных, если захотите создать свое Swing-приложение в похожем стиле.

Вернемся к нашему диалоговому окну. Как видно, выдержать стиль Java-приложения не так уж и трудно, по крайней мере, в отношении расстояния между компонентами и их расположения. Основным расстоянием везде служат 12 пикселов, ну а в особых случаях применяются 5 и 17 пикселов.

## Рекомендации по расположению и класс LayoutStyle

Как мы уже сказали, расположение компонентов и сам пользовательский интерфейс часто создаются программистами, и времени на визуальный дизайн нет, а зачастую нет и навыков такого дизайна. Рекомендации, подобные тем, что мы только что рассмотрели для внешнего вида Metal, являются настоящим спасением. Чтобы еще более упростить процесс следования подобным рекомендациям при создании интерфейса, в JDK версии 1.6 в пакете javax.swing появился класс под названием LayoutStyle, который любезно сообщает, какие расстояния между компонентами следует использовать при создании интерфейса. Применяя его, вы можете отказаться от следования конкретным цифрам и таким образом сделать стиль интерфейса совершенно универсальным, не зависящим даже от его внешнего вида.

Класс LayoutStyle является некоторого рода «одиночкой» — всегда имеется только один его экземпляр, доступный всем желающим через статический метод getInstance(). Экземпляр можно поменять на свой собственный (обычно это делает новый внешний вид приложения, когда вы его устанавливаете). А теперь с помощью простейшего примера проверим, какие расстояния нам предлагает LayoutStyle для внешнего вида по умолчанию (Metal):

```
// LayoutStyleTest.java
// Автоматическое определение стиля интерфейса
import javax.swing.*;
import static javax.swing.LayoutStyle.ComponentPlacement.*;

public class LayoutStyleTest {
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() {
            // компоненты
            JPanel panel = new JPanel();
            JTextField text = new JTextField();
            JLabel label = new JLabel("Тест");
            // отступ от границы контейнера
            LayoutStyle style = LayoutStyle.getInstance();
            System.out.println("" + style.getContainerGap(
                text, SwingConstants.WEST, panel));
            // расстояние между связанными компонентами
            System.out.println("" + style.getPreferredGap(
                  label, text, RELATED, SwingConstants.EAST, panel));
          } });
  }
}
```

Запустив пример, мы получим две цифры 12, что совпадает с рекомендациями, которые мы недавно рассмотрели для внешнего вида Metal. Кстати, вспоминая автоматическое разделение компонентов менеджером MigLayout, становится ясно, что расстояния он берет именно из класса LayoutStyle. Таким же образом может работать и групповое расположение GroupLayout, что делает его очень удобным при визуальном расположении компонентов в редакторе NetBeans.

В итоге, при создании интерфейса вполне можно рекомендовать пользоваться услугами класса, если конечно у ваших дизайнеров нет особого мнения. Подобное решение позволяет приложению быть полностью готовым к «выходу в свет» в любом внешнем виде.

## Реализация в коде: вложенные блоки

Попробуем реализовать наш набросок с помощью блоков BoxLayout. Прежде всего, нам необходимо перейти от чернового рисунка к конкретному плану действий, то есть определить, где, как и какие мы будем использовать менеджеры расположения.

Для этого мы расчертим наш интерфейс линиями так, чтобы компоненты, имеющие одинаковые позиции по вертикали или горизонтали, выстраивались вдоль этих линий (рис. 7.5). Тогда нам не составит труда определить, как построить интерфейс в программе.

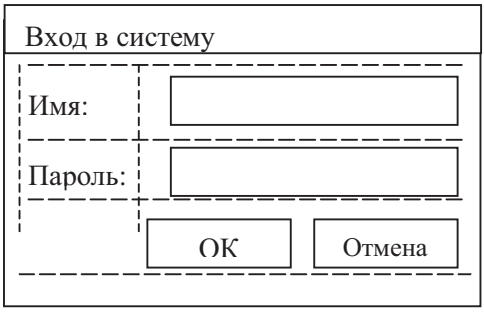



**Рис. 7.5.** Схема диалогового окна для входа в систему

«Легко сказать — расчертите линиями!» — заявите вы: «Чем при этом следует руководствоваться в первую очередь?» Это верно, но рецептов в таких случаях, как правило, не существует. В случае использования блочного подхода можно принять во внимание следующее:

- Выбирайте расположение так, чтобы при увеличении размеров окна интерфейс становился по возможности более функциональным, не теряя при этом во внешней привлекательности. В нашем случае этого легко добиться: нужно строить интерфейс так, чтобы текстовые поля при увеличении размеров позволяли бы увидеть больше информации, а связанные с ними надписи не уплывали от них в сторону. Подобные соображения подсказывают, что нужно использовать горизонтальные «полосы», а не вертикальные.
- Так как менеджер BoxLayout разделяет свободное пространство поровну между всеми компонентами, которые могут «расти» (мы обсудили это в разделе о заполнителях), нужно тщательно продумать, куда будет «уходить» свободное место.
- Необходимо помнить о выравнивании компонентов по осям.

Очень часто достаточно просто взглянуть на набросок интерфейса, чтобы увидеть возможность применения того или иного менеджера расположения, особенно это касается групп компонентов одинаковых размеров (сразу появляется работа для табличного расположения GridLayout). Однако помните, как неаккуратно работают простые менеджеры с расстоянием между компонентами, и если с ними начнутся проблемы, обычно можно получить тот же результат, прибегнув к блочному расположению. А трудности с блочным расположением практически всегда сводятся к несогласованному выравниванию компонентов по осям, но для нас это не проблема.

Что же, с помощью рисунка 7.5 легко увидеть, какие менеджеры расположения нам могут помочь. Учитывая соображения о расширении текстовых полей в ширину, интерфейс построим, основываясь на горизонтальных «полосах». Для них выберем блочное расположение по оси X.

Поклонники последовательного расположения FlowLayout могут сказать, что оно позволит создать эти полосы еще проще, чем блочное. Сперва кажется, что это действительно так, но стоит вспомнить о некоторых хитростях в расположении FlowLayout: при

уменьшении окна текстовые поля могут оказаться под надписями, так как FlowLayout переносит компоненты на другие строки, если им не хватает места. Кроме того, если впоследствии понадобится дополнить интерфейс и использовать разные расстояния между компонентами, с расположением FlowLayout возникнут трудности, в то время как блочное расположение позволит свободно наращивать и изменять интерфейс.

Итак, расстояние зададим распорками, а размеры надписей сделаем одинаковыми вручную. Как видно из рисунка 7.5, выравнивание должно быть центральным. А вот группа кнопок нами уже создавалась с помощью пары из табличного и последовательного расположения с выравниванием по правому краю. Эту идею используем и здесь. (Кстати, в качестве нехитрого упражнения можете реализовать эту же «полосу» с помощью одной только панели с блочным расположением.) Для получения окончательного результата все созданные горизонтальные панели уложим друг на друга с помощью менеджера вертикального блочного расположения. В таком случае эти панели необходимо согласованно выровнять по левую сторону оси.

Осталось только разработать набор инструментов, который позволил бы нам при создании пользовательского интерфейса не переписывать раз за разом одни и те же фрагменты кода, придающие компонентам одинаковые размеры и располагающие их в соответствие с некоторыми рекомендациями Sun. Это будет класс с набором статических методов:

```
// com/porty/swing/GUITools.java
// Набор инструментов для окончательной
// шлифовки и придания блеска интерфейсу
package com.porty.swing;

import javax.swing.*;
import java.awt.*;

public class GUITools {
  // инструмент для придания группе компонентов
  // одинаковых размеров (минимальных,
  // предпочтительных и максимальных).
  public static void makeSameSize(JComponent... cs) {
    // определение максимального размера
    Dimension maxSize = cs[0].getPreferredSize();
    for (JComponent c: cs) {
      if ( c.getPreferredSize().width > maxSize.width ) {
        maxSize = c.getPreferredSize();
      }
    }
    // придание одинаковых размеров
    for (JComponent c : cs) {
      c.setPreferredSize(maxSize);
      c.setMinimumSize(maxSize);
      c.setMaximumSize(maxSize);
    }
```

```
    }
    // позволяет исправить оплошность в
    // размерах текстового поля JTextField
    public static void fixTextFieldSize(JTextField field) {
      Dimension size = field.getPreferredSize();
      // чтобы текстовое поле по-прежнему могло
      // увеличивать свой размер в длину
      size.width = field.getMaximumSize().width;
      // теперь текстовое поле не станет выше
      // своей оптимальной высоты
      field.setMaximumSize(size);
    }
}
```

Первый метод самый полезный: в любом интерфейсе найдутся компоненты, которые нужно сделать одинаковыми. Заметьте, что makeSameSize() берет за основу ширину компонентов — определяет наиболее широкий компонент и выравнивает остальные по нему. Надо сказать, что в большинстве ситуаций выравнивание размеров компонентов происходит именно по ширине. Если вдруг вам понадобится метод для выравнивания по высоте, надо будет просто немного модифицировать этот метод и определять максимальный компонент как компонент с наибольшей высотой. Выравниваются все три размера компонентов. Обычно для компонентов, размеры которых специально делаются одинаковыми (кнопки, надписи, флажки, ползунки), это является наилучшим вариантом. Более сложные компоненты, которые при изменении окна должны определенным образом менять свои размеры (таблицы, текстовые компоненты, и т. п.) хорошо управляются с помощью подобранных нужным образом менеджеров расположения.

О назначении метода fixTextFieldSize() мы еще не говорили, но без него наше диалоговое окно может стать просто безобразным. Дело в том, что при появлении дополнительного пространства в окне текстовое поле JTextField увеличивается не только в длину, как ему и положено, но и в ширину, что совсем не вяжется с представлением об однострочном поле ввода. Это поведение не подходит для блочного расположения BoxLayout. Видимо, это связано с тем, что все текстовые компоненты в Swing унаследованы от базового класса JTextComponent, который и задает такие размеры, более подходящие для многострочных полей ввода и мини-редакторов. Чтобы исправить такую досадную оплошность, в классе GUITools и появился этот метод. Он оставляет длину текстового поля максимально возможной, а высоту заставляет оставаться на оптимальном уровне (который соответствует высоте текста).

Теперь, с классами BoxLayoutUtils и GUITools, все готово для того, чтобы в коде реализовать пользовательский интерфейс нашего диалогового окна:

```
// LoginDialog.java
// Этапы создание первоклассного
// пользовательского интерфейса на примере
// диалогового окна входа в систему
import javax.swing.*;
import java.awt.*;
import com.porty.swing.BoxLayoutUtils;
import com.porty.swing.GUITools;
```

```
public class LoginDialog extends JDialog {
  public LoginDialog(JFrame parent) {
    super(parent, "Вход в систему");
    // удаление окна при закрытии
    setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    // добавляем расположение в центр окна
    add(createGUI());
    // задаем предпочтительный размер
    pack();
    // выводим окно на экран
    setVisible(true);
  }

  // этот метод будет возвращать панель с
  // созданным расположением
  private JPanel createGUI() {
    // 1. Создается панель, которая будет содержать
    // все остальные элементы и панели расположения
    JPanel main =
        BoxLayoutUtils.createVerticalPanel();
    // Чтобы интерфейс отвечал требованиям Java,
    // необходимо отделить его содержимое от
    // границ окна на 12 пикселов.
    // Для этого используем пустую рамку
    main.setBorder(
        BorderFactory.createEmptyBorder(12,12,12,12));
    // 2. Поочередно создаются "полосы", на которые
    // был разбит интерфейс на этапе анализа
    // а) первое текстовое поле и надпись к нему
    JPanel name =
        BoxLayoutUtils.createHorizontalPanel();
    JLabel nameLabel = new JLabel("Имя:");
    name.add(nameLabel);
    name.add(BoxLayoutUtils.createHorizontalStrut(12));
    JTextField nameField = new JTextField(15);
    name.add(nameField);
    // б) второе текстовое поле и надпись к нему
    JPanel password =
        BoxLayoutUtils.createHorizontalPanel();
    JLabel passwrdLabel = new JLabel("Пароль:");
    password.add(passwrdLabel);
```

```
    password.add(BoxLayoutUtils.createHorizontalStrut(12));
    JTextField passwrdField = new JTextField(15);
    password.add(passwrdField);
    // в) ряд кнопок
    JPanel flow = new JPanel( new FlowLayout(
        FlowLayout.RIGHT, 0, 0) );
    JPanel grid = new JPanel( new GridLayout(
        1,2,5,0) );
    JButton ok = new JButton("OK");
    JButton cancel = new JButton("Отмена");
    grid.add(ok);
    grid.add(cancel);
    flow.add(grid);
    // 3. Проводятся необходимые действия по
    // выравниванию компонентов, уточнению их
    // размеров, приданию одинаковых размеров
    // а) согласованное выравнивание
    // вложенных панелей
    BoxLayoutUtils.setGroupAlignmentX(
        Component.LEFT_ALIGNMENT,
        name, password, main, flow);
    // б) центральное выравнивание надписей
    // и текстовых полей
    BoxLayoutUtils.setGroupAlignmentY(
        Component.CENTER_ALIGNMENT,
        nameField, passwrdField, nameLabel, passwrdLabel);
    // в) одинаковые размеры надписей к текстовым полям
    GUITools.makeSameSize(nameLabel, passwrdLabel);
    // д) устранение "бесконечной" высоты текстовых полей
    GUITools.fixTextFieldSize(nameField);
    GUITools.fixTextFieldSize(passwrdField);
    // 4. Окончательный "сбор" полос в интерфейс
    main.add(name);
    main.add(BoxLayoutUtils.createVerticalStrut(12));
    main.add(password);
    main.add(BoxLayoutUtils.createVerticalStrut(17));
    main.add(flow);
    // готово
    return main;
  }
  // тестовый метод для проверки диалогового окна
```

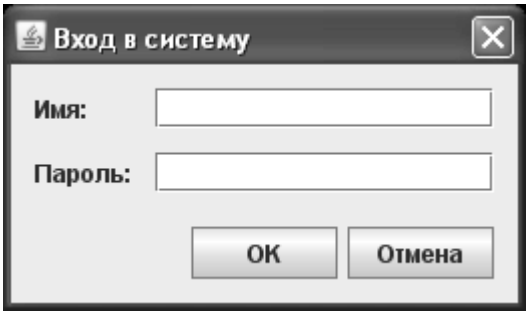
```
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new LoginDialog(new JFrame());
      }
    });
  }
}
```

Прочитав комментарии к программе, вы увидите, что и процесс реализации интерфейса можно условно разбить на несколько этапов. Сначала вы проводите обычные операции по созданию и настройке окна: вызываете конструктор базового класса, задаете название окна, его размеры, действия при выходе из него и добавляете в окно панель с интерфейсом, который в данном случае создается в отдельном методе с названием createGUI(). Можно создавать интерфейс и в конструкторе, это не имеет большого значения, особенно если в конструкторе не производится никаких критически важных действий, которые могут сорвать создание объекта и сделать последующие операции по размещению компонентов ненужными.

Все начинается с создания основной панели расположения, в которую затем будут добавлены все созданные компоненты или вспомогательные панели с компонентами. Такой подход к тому же позволяет легко реализовать требование об отступе от границ окна — основная панель просто будет иметь пустую рамку нужного размера. Затем формируются блоки («полосы»), из которых и будет состоять интерфейс. После прочтения настоящей главы этот этап не должен вызывать у вас вопросов. Для простоты и сокращения размера кода мы не стали определять расстояния между компонентами с помощью класса LayoutStyle, а прописали их вручную, однако лучше полагаться на способности LayoutStyle.

Далее проводятся действия по окончательному выравниванию компонентов (с помощью специально созданных нами классов BoxLayoutUtils и GUITools). Это важный момент, особенно при блочном расположении, и без него интерфейс выглядел бы просто нелепо. Размеры надписей к текстовым полям делаются одинаковыми.

В самом конце созданные «полосы» укладываются друг на друга с необходимыми интервалами. После этого остается только вывести диалоговое окно на экран и протестировать его поведение.




Созданный интерфейс еще не идеален — не хватает, например клавиатурных сокращений для быстрого доступа к текстовым полям, кнопки, нажимаемой по умолчанию, всплывающих подсказок к компонентам, локализации текста интерфейса. Все это мы рассмотрим в соответствующих главах.

Класс LoginDialog хорош еще и тем, что его чрезвычайно легко использовать повторно, а это — одно из основных требований к качественному коду. Перенося его в новую програм-

му, вы просто привязываете к компонентам принадлежащие новой программе слушатели событий и получаете уже спроектированный и проверенный в работе элемент пользовательского интерфейса программы. Как предоставлять доступ к компонентам диалогового окна — дело вкуса. Можно вообще не показывать компоненты, предоставляя лишь текст из текстовых полей и оповещая о нажатии кнопок. Действуя таким образом, можно создать целую библиотеку своих интерфейсов, и во много раз ускорить разработку приложения.

## Реализация: гибкая сетка

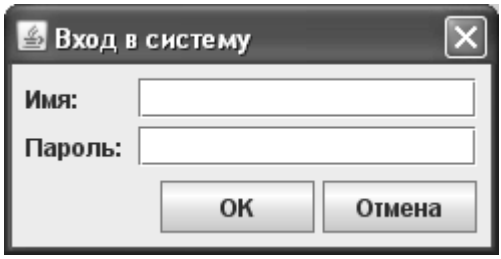
Предыдущий параграф показал реализацию интерфейса с помощью вложенных блоков и менеджера BoxLayout. Этот подход чрезвычайно гибок, однако требует достаточно много дополнительных действий, таких как выравнивание по осям и придание одинаковых размеров, а также ручное указание расстояний. С другой стороны, у нас есть менеджеры, работающие по принципу «гибкой сетки». Если расположение хорошо поддается разбиению на сетку, применение MigLayout или GridBagLayout (с помощью разработанного нами удобного инструмента) может быть быстрее, так как гибкая сетка уже подразумевает выравнивание компонентов и их размеров.

Попробуем теперь реализовать тот же самый набросок интерфейса диалога входа в систему с помощью менеджера MigLayout. Как видно, у нас будет три ряда и два столбца, причем кнопки придется разместить с помощью дополнительной панели, или же разбив ячейку на две части. Второй столбец должен расти в размерах по горизонтали. Вот что получится:

```
  …
  // этот метод будет возвращать панель с
  // созданным расположением
  private JPanel createGUI() {
    // 1. Основная панель
    // В конструкторе задаем "рост" второго столбца
    JPanel main = new JPanel(new MigLayout("", "[][grow]"));
    // первый ряд сетки
    main.add(new JLabel("Имя:"));
    main.add(new JTextField(15), "gap rel, wrap rel, growx");
    // второй ряд сетки
    main.add(new JLabel("Пароль:"));
    main.add(new JTextField(15), "gap rel, wrap unrel, growx");
    // третий ряд сетки — кнопки одинакового размера
    // пропускаем первую ячейку, разбиваем вторую на две
    main.add(new JButton("OK"), "skip 1, split, sg buttons, align
right");
    main.add(new JButton("Отмена"), "sg buttons");
    // готово
    return main;
  }
  …
```

Для краткости мы опустили конструктор и метод main(), так как они абсолютно не отличаются от варианта с блочным расположением. Одно отличие — класс примера будет называться LoginDialog2. В том же самом методе createGUI() создается интерфейс приложе-

ния. В случае с нашим диалогом входа в систему гибкая сетка, особенно реализованная с помощью менеджера MigLayout, дает на выходе гораздо более компактный результат, в том числе и благодаря тому, что последний автоматически вставляет расстояния между компонентами и между границами контейнеров и окна. Если вы вспомните раздел, посвященный менеджеру MigLayout, вы вполне сможете понять код примера, единственная хитрость была применена при создании двух кнопок — ячейку пришлось разбить на две — по одной на каждую кнопку, и вручную указать, что кнопки должны быть выровнены по правому краю ключевым словом align. Запустив пример, мы получим практически идентичное диалоговое окно.




Однако разница с блочным расположением все же есть — менеджер MigLayout по умолчанию использует собственные расстояния между компонентами, а не те, что рекомендуются для внешнего вида Metal. В принципе, ничего страшного в этом нет, расстояния можно указывать вручную, либо согласиться с тем, что предлагает MigLayout. Аналогичный подход можно реализовать и с помощью стандартного менеджера GridBagLayout, однако код в этом случае получится намного сложнее и пространнее.

Не стоит думать, что вариант гибкой сетки всегда выигрывает у блочного подхода, как это получилось в случае с диалогом входа в систему. Если компонентов много, и они не всегда выровнены по сетке, попытки разместить их все в единой сетке могут быть слишком сложны и образовывать в результате код, который трудно прочитать и поддерживать. В таком случае к вашим услугам блочная техника расположения, или несколько более простых вложенных расположений. В принципе, это аналогия с расположением содержимого для HTML-страниц: у вас есть вариант пользоваться для расположения таблицами (<table>), или применять блоки ( или ).

## *Резюме*

Расположение компонентов в контейнере сродни искусству, хотя используемые для этого менеджеры расположения просты для понимания и применения. Все дело в том, что вложенные расположения зачастую образуют непредсказуемые комбинации, поведение которых может быть нетривиально и совсем не похоже на то, чего вы могли бы ожидать от простого менеджера расположения. Тем не менее, если помнить несложные правила, разумно комбинировать блочные и сеточные стратегии и не забывать о фантазии и способностях стандартных менеджеров, можно быстро и просто добиться любого расположения компонентов.

# Глава 8. Вывод вспомогательной информации

Компоненты пользовательского интерфейса программы можно условно разделить на две части: одни компоненты применяются для получения информации от пользователя и для вывода данных программы, а другие служат для облегчения работы пользователя, обеспечивая его вспомогательной информацией об интерфейсе программы и задачах, которые он решает. В этой главе мы будем говорить именно о компонентах, позволяющих пользователю получать вспомогательную информацию. Они настолько удобны и так упрощают ознакомление пользователя с интерфейсом, что уже почти невозможно представить себе программы без них.

В Swing к компонентам, обеспечивающим пользователя информацией об интерфейсе и его задачах, прежде всего относится надпись JLabel, поддерживающая язык разметки HTML и позволяющая вывести любое сочетание значков и текста. С помощью компонента JLabel можно показать пользователю все что угодно, написав лишь несколько строк кода. Далее, для вывода краткой информации о любом элементе интерфейса в нужный момент времени используются всплывающие подсказки JToolTip, обладающие в Swing уникальными возможностями, благодаря все той же поддержке HTML. Немногие библиотеки пользовательских интерфейсов способны предоставить такие возможности без специальных усилий с вашей стороны. Наконец, для визуальной организации интерфейса и придания ему законченного вида удобно применять рамки. Все эти компоненты мы и рассмотрим в данной главе.

## *Надписи JLabel*

Компонент-надпись JLabel используется в Swing для вывода разнообразной информации, чаще всего состоящей из сочетания текста и значков, позволяя сообщить пользователю все, что вы захотите. Надо сказать, что надпись JLabel встречается в Swing буквально на каждом шагу: ее применяют для вывода вспомогательного текста в диалоговых окнах, для рисования значков и рисунков, и все сложные компоненты Swing, такие как деревья и таблицы, используют экземпляр надписи для прорисовки своих элементов: листьев деревьев и ячеек таблиц. Это настоящая «рабочая лошадка» библиотеки. Так что хорошее знание возможностей надписи JLabel в дальнейшем еще не раз позволит вам эффективно прорисовывать разнообразную информацию. Очень часто вместо создания собственного компонента со своей процедурой прорисовки проще использовать правильно настроенную надпись JLabel.

Начнем мы с изучения возможностей надписи JLabel по расположению своего содержимого (текста и значков) относительно друг друга. Здесь у вашей фантазии нет никаких ограничений, вы сможете поместить текст и значки совершенно произвольным образом. Рассмотрим пример:

```
// Labels.java
// Настройка содержимого надписей
import java.awt.*;
import javax.swing.*;
```

```
public class Labels extends JFrame
 implements SwingConstants {
 public Labels() {
     super("Labels");
     // при закрытии окна заканчиваем работу
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     // самая простая надпись
     JPanel contents = new JPanel();
     JLabel l1 = new JLabel("Ваше имя:");
     JTextField name = new JTextField(20);
     contents.add(l1);
     contents.add(name);
     // надпись со значком
     JLabel l2 = new JLabel(new ImageIcon("monkey.gif"));
     adjustLabel(l2);
     l2.setHorizontalAlignment(LEFT);
     contents.add(l2);
     // надпись с нестандартным выравниванием
     JLabel l3 = new JLabel("Текст и значок",
         new ImageIcon("bulb.gif"), RIGHT);
     adjustLabel(l3);
     l3.setVerticalTextPosition(BOTTOM);
     l3.setHorizontalTextPosition(LEFT);
     contents.add(l3);
     // вывод окна на экран
     setContentPane(contents);
     setSize(320, 300);
     setVisible(true);
 }
 // метод производит специальную настройку надписи
 private void adjustLabel(JLabel l) {
     l.setOpaque(true);
     l.setBackground(Color.white);
     l.setPreferredSize(new Dimension(250, 100));
 }
 public static void main(String[] args) {
        SwingUtilities.invokeLater(
         new Runnable() {
          public void run() { new Labels(); } });
 }
}
```

Надпись JLabel позволяет вывести на экран текст, значок, или оба элемента вместе. Текст и/или значок чаще всего задаются в конструкторе надписи, у которого имеются пять перегруженных версий на все случаи жизни, но можно также задать или изменить текст или значок потом, после создания объекта, манипулируя свойствами icon и text с помощью соответствующих методов get/set (это свойства JavaBeans). Вы также можете изменить шрифт вывода текста, используя метод setFont() (в примере мы этого не делаем, а по умолчанию применяется шрифт, предоставленный менеджером внешнего вида). В нашем примере мы создаем окно, в панель содержимого которого добавляем несколько надписей. Первая надпись — самый простой и наиболее распространенный вариант: просто некий необходимый пользователю текст выводится на экран. Рядом с ней было специально добавлено текстовое поле, и картина становится полной: надпись описывает, для чего предназначен другой компонент. Чаще всего так она и используется. Далее мы добавляем более сложные варианты надписей JLabel, одну со значком и вторую и со значком, и с текстом.

Вы видите, что для этих надписей вызывается вспомогательный метод adjustLabel(), позволяющий им, во-первых, быть непрозрачными (свойство opaque устанавливается в true), что помогает определить, какое пространство занимает надпись на экране и использовать нестандартный цвет фона, а, во-вторых, этот метод меняет их размер на заведомо больший, чем необходимо (только так мы сможем оценить изменения в выравнивании). Для того чтобы было видно, какую часть экрана занимает надпись, ее фон сделан белым (стандартная надпись обычно не выделяется в приложении цветом, ее роль сводится к выводу информации). Для загрузки значков используется вспомогательный класс ImageIcon, который мы вскоре обсудим подробнее, пока же достаточно того, что ему можно указать название файла с изображением. Кроме того, что мы с помощью надписей выводим текст и значки на экран, мы меняем их выравнивание относительно границ надписи и относительно друг друга. Для этого используются свойства, перечисленные в табл. 8.1.

**Таблица 8.1.** Свойства для выравнивания содержимого надписей

| Свойства | Предназначение |
|---|---|
| verticalAlignment, horizontalAlignment (и соответствующие методы get/set) | Позволяют задать расположение содержимого надписи (и текста, и значка) относительно ее границ. Возможные положения описаны константами из интерфейса SwingContants, у вас есть три варианта расположения по вертикали (центр, верх, низ) и столько же по горизонтали (центр, справа, слева). Всего получается девять возможных позиций |
| verticalTextPosition, horizontalTextPosition (и соответствующие методы get/set) | Дают возможность указать надписи, как ей следует располагать текст относительно значка. Вы можете задать положение текста по вертикали (по центру значка, по его верхней или нижней границе) и по горизонтали (по центру значка, слева или справа от него). В итоге также получается девять вариантов (можно даже разместить текст поверх значка) |
| iconTextGap | В нашем примере мы не использовали это свойство, но оно очень просто и позволяет указать расстояние между текстом и значком в пикселах |

Для того чтобы было проще использовать константы выравнивания, мы в нашем классе реализовали (implements) интерфейс SwingConstants, в котором они описаны, его подробное описание легко найти в интерактивной документации. Надпись со значком меняет расположение своего значка, а надпись со значком и текстом меняет не только расположение всего содержимого, но и положение текста относительно значка. Запустив программу с примером, вы увидите результаты наших действий и сможете убедиться в том, что у вас действительно очень широкие возможности по настройке содержи-

мого надписей. Однако стоит помнить, что если положение текста относительно значка всегда актуально, то нестандартное расположение содержимого будет заметно, только когда надпись займет больше места, чем ей нужно, что в случае использования менеджера расположения случается редко.


Стоит также отметить (хоть мы и не создали такой надписи в примере), что надписи JLabel могут быть «отключенными», так же как и большая часть других компонентов, таких как кнопки, надо лишь вызвать метод setEnabled(false). Это позволяет придать интерфейсу большую выразительность: если какие-то функции приложения недоступны, отключаются все элементы интерфейса, даже надписи. Если в надписи используется значок, то в отключенном состоянии он с помощью специального фильтра изображений[1] переводится в черно-белые цвета. Впрочем, вы можете задать собственный значок для отключенного состояния, вызвав метод setDisabledIcon().

## Значки Icon

Только что при рассмотрении примера с надписями JLabel мы видели, что для загрузки значков использовался класс ImageIcon. На самом деле в библиотеке Swing для вывода значков во всех компонентах библиотеке применяется интерфейс Icon, а класс ImageIcon — его самая популярная реализация с изображением в качестве значка[2]. «Поче-

[1] Этот фильтр изображений называется GrayFilter, находится в пакете javax.swing и используется всеми компонентами Swing с поддержкой значков Icon, чтобы создавать «отключенные» версии значков (в том случае если вы не предоставляете отключенные значки самостоятельно). Задействовать данный фильтр можно и напрямую: удобный статический метод getDisabledImage() позволяет мгновенно получить черно-белый вариант любого изображения.

[2] В данный момент JDK поддерживает форматы изображений JPEG, GIF (в том числе и анимированный) и PNG, то есть все самые популярные форматы Web. Особое внимание стоит обратить на формат PNG, который не имеет проблем с лицензированием и полностью поддерживает альфа-канал (различные степени прозрачности).

му же в библиотеке сразу не используются изображения Image?» — спросите вы. Все дело в том, что интерфейс Icon можно легко реализовать самому. Это позволяет использовать в качестве значков не только готовые изображения из файлов (которые легко украсть и трудно защитить), но и нарисованные прямо в программе (например, с помощью библиотеки Java2D). Такие рисунки заимствовать гораздо тяжелее, а создать иногда проще. К примеру, практически все стандартные значки и текстуры, используемые в поставляемых вместе с JDK внешних видах Swing, рисуются внутри классов в виде значков Icon, а не поставляются в виде изображений.

Создать свой значок Icon, используя только графические свойства Java, нетрудно:

```
// RedBullet.java
// Создание собственного значка
import javax.swing.*;
import java.awt.*;

public class RedBullet implements Icon {
    public int getIconWidth() {
        return 16;
    }
    public int getIconHeight() {
        return 16;
    }
    public void paintIcon(
        Component c, Graphics g, int w, int h) {
        g.setColor(Color.red);
        g.fillRect(0, 0, 16, 16);
    }
}
```

Здесь мы создаем значок в виде красного квадрата. Первые два метода позволяют задать размеры значка, в третьем методе вы рисуете свой значок. Иногда такой подход может быть быстрее и эффективнее создания изображения из файла. Использовать такие значки можно в любых компонентах Swing с их поддержкой (то есть почти во всех). Тем не менее, чаще все-таки используется класс ImageIcon. Он позволяет загрузить изображение поддерживаемого формата из файла, сетевого ресурса, заданного URL-адресом (Uniform Resource Locator — унифицированный указатель ресурса), и способен создавать изображение даже из массива байтов[3]. В отличие от стандартного метода загрузки изображений getImage() из класса java.awt.Toolkit, класс ImageIcon всегда загружает изображение полностью (метод getImage() сразу возвращает управление, даже если изображение загружено частично, в результате на экране может появиться совсем не то, что вы ожидаете). Для этого задействуется специальный класс MediaTracker, который в противном случае пришлось бы использовать самостоятельно. Остается лишь сказать, что в пути к изображению указывается прямой слеш, который автоматически заменяется разделителем пути текущей платформы:

```
new ImageIcon("data/resources/menu");
```

[3] Возможность создать изображение на основе массива байтов значительно облегчает распространение приложения в виде единого исполняемого JAR-файла, ресурсы из которого можно получить только в виде подобного массива.

На какой бы платформе ваше приложение ни оказалось, такая форма записи обеспечит правильный доступ к файлам и избавит вас от головной боли. В дальнейшем мы еще не раз будем использовать значки для своих компонентов.

## Использование HTML

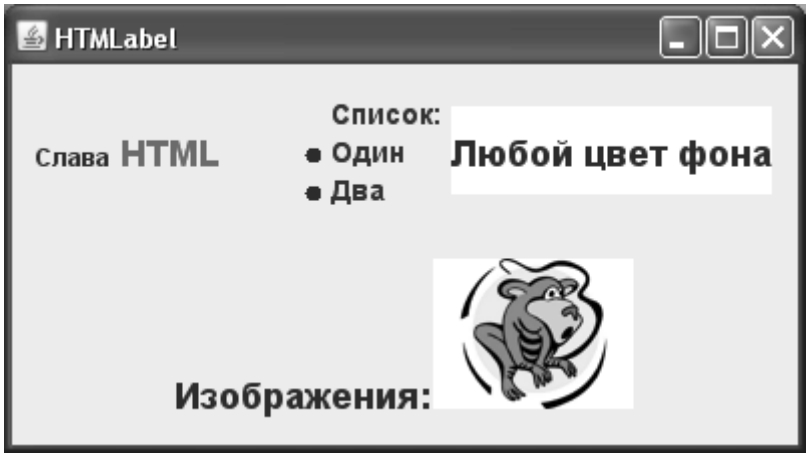
Возможность с помощью надписи JLabel выводить произвольные сочетания значков и текста, при желании менять цвет текста, шрифт и цвет фона надписи позволяет вам легко отображать на экране самую разнообразную информацию. Впрочем, бывают ситуации, когда относительно простые вещи нельзя вывести с помощью обычных надписей, к примеру, определенные трудности вызывает необходимость использования в одной надписи нескольких шрифтов или нескольких строк.

В состав Swing входит пакет javax.swing.text.html, поддерживающий для текстовых компонентов язык разметки HTML версии 3.2. Окончательный вывод текста на экран выполняет объект View (мы рассмотрим его в главе 16, посвященной текстовым возможностям Swing), который может использовать для вывода текста любой компонент. В любом объекте JLabel в виде клиентского свойства находится объект View, который и выполняет вывод текста на экран (что интересно, он занимается этим, даже если вы выводите простой текст, а не HTML). Чтобы вывести на надписи HTML-текст, достаточно указать в начале строки его отличительный признак — символы <html>. Рассмотрим пример:

```
// HTMLabel.java
// Использование в надписях языка HTML
import javax.swing.*;

public class HTMLabel extends JFrame {
 private String html1 =
     "<html><b>Слава</b><font size=5 color=red> HTML";
 private String html2 =
     "<html><font size=4 color=blue>" +
      "<ul>Список:<li>Один<li>Два";
 private String html3 =
     "<html><body bgcolor=white><h2>Любой цвет фона";
 private String html4 =
     "<html><h2>Изображения:<img src=\"file:monkey.gif\">";
 public HTMLabel() {
     super("HTMLabel");
     // при закрытии окна выход
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     // добавляем надписи
     JPanel contents = new JPanel();
     contents.add(new JLabel(html1));
     contents.add(new JLabel(html2));
     contents.add(new JLabel(html3));
     contents.add(new JLabel(html4));
     // выводим окно на экран
```

```
        add(contents);
        setSize(400, 300);
        setVisible(true);
    }
    public static void main(String[] args) {
           SwingUtilities.invokeLater(
            new Runnable() {
             public void run() { new HTMLabel(); } });
    }
}
```

В примере мы добавляем в окно с рамкой четыре надписи, поддерживающие возможности встроенного языка HTML. Главное при использовании HTML — помнить, что строка с текстом должна начинаться с «волшебных» символов <html>, иначе все теги будут выведены в виде обычного текста. Мы задействовали разнообразные возможности HTML: различные стили и размеры текста, разные цвета для фона страницы и ее текста, и даже вывели изображение. Вы можете видеть, что даже список некоторых элементов в надписи JLabel благодаря HTML организовать не сложно. Как все это выглядит, можно посмотреть, запустив программу с примером.




Все возможности, описанные в спецификации языка HTML версии 3.2, поддерживаются встроенным в надписи элементом View. Как видно, поддерживаются и изображения, однако при работе с ними надо помнить, что встроенный язык HTML в качестве источника изображений распознает только правильные URL-адреса. Это могут быть адреса файлов на локальной машине (причем задавать файлы вы можете и относительно текущего каталога пользователя, как мы сделали это в примере, и с помощью абсолютных путей, таких как file:C:/images/pub/img.jpg) или полные сетевые адреса, например http://somehost/dir/image.gif. Впрочем, изображения с помощью встроенного языка HTML выводят все же редко, хотя возможность эта захватывающая. Дело в том, что приложения обычно распространяют в виде единственного архивного JAR-файла, в котором в том числе хранятся и изображения. Загрузка изображений из архива происходит уже во время работы программы, так что для их вывода чаще используют возможности надписей JLabel, позволяющих выводить значки и выстраивать их относительно текста.

Поддержка надписями JLabel языка HTML возводит их в ранг универсального информационного элемента, позволяющего вывести любой тип информации. Многострочные надписи, различные шрифты и выравнивания, любые цвета, нумерованные и маркированные списки, изображения — все у вас в руках. Однако чересчур увлекаться возможностями HTML не стоит, потому что надписи должны сообщать пользователю полезную информацию, а не отвлекать его внимание буйством красок. К тому же HTML-текст выводится на надписи в своеобразной форме, несколько отличающейся от того, что показало бы большинство браузеров, и все это может привести к визуальному конфликту между стилем используемого внешнего вида и некоторыми компонентами, поэтому необходимо точно знать, что вы хотите получить.

## Надписи и события

Надпись JLabel — такой же компонент, как и остальные, унаследованный от базового класса JComponent, поэтому она может обрабатывать любые виды общих для всех компонентов событий: надо лишь добавить к ней соответствующих слушателей. Но чаще всего надпись просто сообщает что-то пользователю, а не обрабатывает событий, чтобы не вводить его в заблуждение (помните принцип простого интерфейса с очевидным предназначением). Пользователь привык, что надписи выводят информацию, и не отвечают за что-то большее. Если вам нужно выполнять в приложении какую-то функцию, инициируйте ее выполнение с помощью кнопки или клавиатурного сокращения, в качестве альтернативы можно предложить компонент-гиперссылку JXHyperlink из пакета дополнений SwingX.

Надписи также не принимают фокус ввода, их свойство focusable по умолчанию установлено в false. Если только вы не хотите огорошить пользователей новаторским интерфейсом, в котором надписи живут своей жизнью, стандартное поведение вряд ли стоит менять.

## Надписи и мнемоники

Создатели Swing добавили в надписи JLabel еще одну возможность, позволяющую отточить функциональность вашего пользовательского интерфейса. Надписи могут поддерживать *мнемоники* (mnemonics) для других компонентов, неспособных поддерживать их самостоятельно. Мнемоника — это специальное клавиатурное сокращение (на большинстве платформ сочетание клавиши Alt или Ctrl с кодом клавиши), позволяющая быстро активировать некоторый компонент интерфейса, например, кнопку или текстовое поле. В сложных интерфейсах, состоящих из множества компонентов, находить нужный компонент (который, возможно, является самым важным) может быть утомительно, и в таких ситуациях мнемоники очень полезны. Более того, они незаменимы для пользователей с ограниченными возможностями, которые не могут пользоваться мышью. Приложение высокого класса обязано предоставлять доступ с клавиатуры ко всем своим компонентам.

Некоторые компоненты, например кнопки, сами поддерживают мнемоники, другие, такие как текстовые поля или раскрывающиеся списки, могут поддерживать клавиатурные сокращения, но удобно информировать о них пользователя не в состоянии. Здесь пригодится надпись с мнемоникой около компонента: пользователь быстро увидит, как активируется данный компонент, потому что символ мнемоники в тексте надписи подчеркивается.

Поддержка мнемоник реализована в UI-представителе надписи JLabel, причем сделано все довольно интересно: когда вы сообщаете надписи, что хотели бы использовать ее для вывода мнемоники некоторого компонента, UI-представитель, во-первых, подчеркивает символ мнемоники при прорисовке надписи, во-вторых, регистрирует в кар-

те входных событий нужное клавиатурное сокращение. При срабатывании сокращения надпись на мгновение получает фокус ввода (фокус сохраняется на надписи, пока вы удерживаете клавиши), при отпускании клавиш фокус переходит к компоненту, для которого была задана мнемоника, а затем надпись снова возвращается в обычное состояние (без фокуса ввода). Давайте увидим все в действии:

```
// LabelMnemonic.java
// Использование надписей для вывода мнемоник
import javax.swing.*;
import java.awt.*;

public class LabelMnemonic extends JFrame {
 public LabelMnemonic() {
     super("LabelMnemonic");
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     // добавим пару текстовых полей
     JPanel contents = new JPanel(new GridLayout(2,2));
     JTextField tf = new JTextField(10);
     JLabel label = new JLabel("Ваше имя:");
     // настройка мнемоники
     label.setLabelFor(tf);
     label.setDisplayedMnemonic('И');
     // добавляем компоненты в таблицу
     contents.add(new JLabel("Ваша фамилия:"));
     contents.add(new JTextField(10));
     contents.add(label);
     contents.add(tf);
     // выведем окно на экран
     setContentPane(contents);
     pack();
     setVisible(true);
 }
 public static void main(String args[]) {
        SwingUtilities.invokeLater(
         new Runnable() {
          public void run() { new LabelMnemonic(); } });
 }
}
```

В примере мы создаем небольшое окно, в качестве панели содержимого которого используется панель с табличным расположением (2 строки и 2 столбца). В панель мы добавляем два текстовых поля и надписи к ним (весьма часто встречающаяся в пользовательском интерфейсе комбинация). Текстовое поле не может отобразить мнемонику, и здесь нам пригодится надпись. Мы настраиваем мнемонику для одной из созданных нами надписей: указываем компонент, активизируемый при срабатывании мнемоники

(setLabelFor()), а также символ мнемоники, клавишу которого вместе с управляющей клавишей и нужно будет нажать (setDisplayedMnemonic()). Запустив пример, вы увидите, что надпись подсказывает пользователю, каким символом можно активизировать текстовое поле при нажатии нужного сочетания (чаще всего Alt+символ, в нашем случае Alt+И). В текущих выпусках JDK 1.6 – 1.7 кириллические символы поддерживаются[4], что замечательно, так как в первом издании книги мы сломали немало копий по этому поводу.


Общая рекомендация для интерфейса говорит, что *все* компоненты должны быть снабжены мнемониками, поэтому рассмотренная нами возможность несомненно полезна.

## Всплывающие подсказки

*Всплывающие подсказки* (tool tips) — неизменный атрибут современного пользовательского интерфейса, и не зря: они намного упрощают ознакомление и последующую работу с вашим приложением. Это небольшие (чаще всего, хотя и не обязательно) текстовые описания компонентов вашего интерфейса, появляющиеся рядом с компонентом в том случае, если пользователь ненадолго задерживает на нем указатель мыши. Ценность всплывающих подсказок трудно переоценить — они намного ускоряют работу с любым, особенно сложным приложением. Вместо знакомства с документацией пользователь может прямо «на лету» узнать, для чего предназначен тот или иной элемент интерфейса, и быстро начать работу. Работу кнопками, на которых есть только значки, вообще невозможно представить без подсказок[5]. В Swing это особенно верно, потому что подсказки здесь обладают весьма широкими возможностями.

Любой компонент Swing может обладать всплывающей подсказкой, потому что поддержка их встроена в базовый класс библиотеки JComponent. Когда вы вызываете метод setToolTipText(), компонент регистрирует себя в классе ToolTipManager, который отвечает за правильный вывод подсказок. Класс ToolTipManager следит за перемещениями мыши на всех зарегистрированных в нем компонентах и при наступлении нужного момента (наведении указателя мыши) выводит подсказку на экран. Рассмотрим пример с подсказками:

```
// ToolTips.java
// Подсказки в Swing
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ToolTips extends JFrame {
```

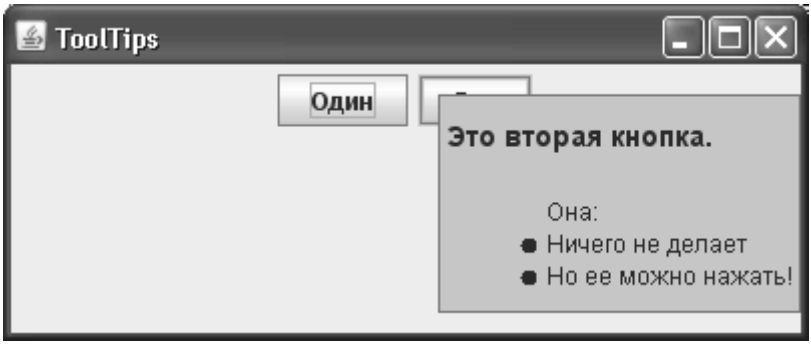
[4] К сожалению, только для надписей, но не для кнопок и меню.

[5] Интересно, что кнопки с мелкими значками, так популярные в современных приложениях, от Web до мобильных устройств, часто подвергаются критике специалистов по удобству интерфейсов. Действительно, и попасть трудно, и почитать надо перед тем как нажимать, универсальных значков мало. Автор согласен с тем, что если есть место для текста, лучше сделать элемент с текстом, чем с красочным значком.

```
    public ToolTips() {
        super("ToolTips");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        // добавим несколько кнопок с подсказками
        JButton b1 = new JButton("Один");
        b1.setToolTipText("Это первая кнопка");
        JButton b2 = new JButton() {
            public Point getToolTipLocation(MouseEvent e) {
                return new Point(10, 10);
            }
            public String getToolTipText(MouseEvent e) {
                if ( e.getY() > 10 ) {
                    return "Нижняя часть кнопки!";
                }
                return super.getToolTipText(e);
            }
        };
        b2.setText("Два");
        b2.setToolTipText("<html><h3>Это вторая кнопка.<ul>" +
          "Она:<li>Ничего не делает<li>Но ее можно нажать!");
        JPanel contents = new JPanel();
        contents.add(b1);
        contents.add(b2);
        // выводим окно на экран
        add(contents);
        setSize(400, 150);
        setVisible(true);
    }
    public static void main(String[] args) {
           SwingUtilities.invokeLater(
            new Runnable() {
             public void run() { new ToolTips(); } });
    }
}
```

Мы добавляем в окно две кнопки JButton, и задаем для каждой из них текст подсказки, что автоматически приводит к регистрации кнопок в классе ToolTipManager. Для первой кнопки мы используем простой текст, который появится в виде однострочной подсказки, шрифт и цвет которой зависят от текущих внешнего вида и расположения. Но вот для второй кнопки мы используем уже знакомый нам HTML-текст (помните, что начинать HTML-строку следует с символов <html>), и здесь у вас, так же как и в случае с надписями, просто безграничные возможности. Вы можете задействовать любые возможности HTML версии 3.2 для создания краткой, но мощной системы помощи, действующей прямо «на лету». В примере мы применили стиль заголовка и маркированный список — эти

элементы могут использоваться для перечисления возможностей компонента. С другой стороны, стоит помнить о том, что большие и красочные подсказки, неожиданно загораживающие экран при остановке указателя мыши на любом компоненте, могут раздражать пользователя — подробные описания функций приложения стоит оставлять для общей системы помощи.




Помимо метода setToolTipText(), позволяющего задать для вашего компонента подсказку, у вас есть еще пара методов, способных повлиять на ее вывод, — это методы getToolTipLocation() и getToolTipText(MouseEvent), определенные в базовом классе JComponent. Правда, пользоваться ими довольно неудобно, потому что соответствующего метода set нет, и для получения нужного эффекта их приходиться переопределять, что мы и сделали в нашем примере, создав анонимный подкласс кнопки JButton. Метод getToolTipLocation() позволяет указать классу ToolTipManager, в каком месте экрана следует выводить подсказку относительно компонента, для которого подсказка выводится. Можно указывать различные места для подсказки в зависимости от события MouseEvent, по которому подсказка выводится. В примере мы указали, что подсказка должна выводиться на расстоянии 10 пикселов по осям X и Y от верхнего левого угла нашей кнопки (можно указывать и отрицательные расстояния, это будет означать, что вы хотите вывести подсказку не ниже компонента, как обычно, а выше). Правда, действует описанный механизм, только если подсказка выводится как легковесный компонент, то есть когда ей хватает места в пределах окна нашего приложения. В противном случае класс ToolTipManager сам решает, где удобнее ее разместить. Полученный эффект вы сможете наблюдать, запустив программу с примером: подсказка первой кнопки будет выводиться в разных местах, в зависимости от положения указателя мыши, а для второй кнопки она всегда будет появляться на указанном нами расстоянии (если ей хватит места). Метод getToolTipText() позволяет выводить для одного и того же компонента различные подсказки в зависимости от того, где пользователь задержал указатель мыши (координаты этого места позволяет узнать объект MouseEvent). Данный метод часто задействуется сложными компонентами Swing, такими как списки или таблицы, состоящими из многих элементов. С его помощью они обеспечивают вывод собственных подсказок для каждого элемента. По умолчанию он возвращает для любой точки текст, заданный методом setToolTipText(), но переопределив этот метод, вы сможете изменить подобное поведение. В примере мы используем подсказку по умолчанию только для верхней части кнопки (высотой в 10 пикселов), а для нижней части возвращаем собственный текст.

Механизм работы всплывающих подсказок в Swing довольно прост. Как мы уже отмечали, при вызове метода setToolTipText() компонент регистрирует себя в классе ToolTipManager. Последний начинает следить за перемещениями мыши в компоненте (причем для всех компонентов используется один и тот же слушатель событий от мыши, поскольку для каждого события можно узнать его источник — компонент, к которому оно относится). Когда указатель мыши входит в область компонента (вызывается метод

слушателя mouseEntered()), класс ToolTipManager запускает таймер, при срабатывании которого (если при этом не происходит движение мыши, иначе таймер запускается заново) на экран выводится подсказка. Подсказка создается в два этапа. Собственно компонент-подсказка JToolTip создается методом createToolTip(), имеющимся в любом компоненте Swing. Другой компонент (тот, что будет «всплывать» над остальными компонентами, в нем размещается подсказка JToolTip) создается с помощью вспомогательного класса PopupFactory. Этот класс определяет тип компонента (легковесный или тяжеловесный) и размещает его в нужном месте экрана (в слое POPUP_LAYER многослойной панели или в новом окне без рамки JWindow). Общую картину иллюстрирует рис. 8.1.

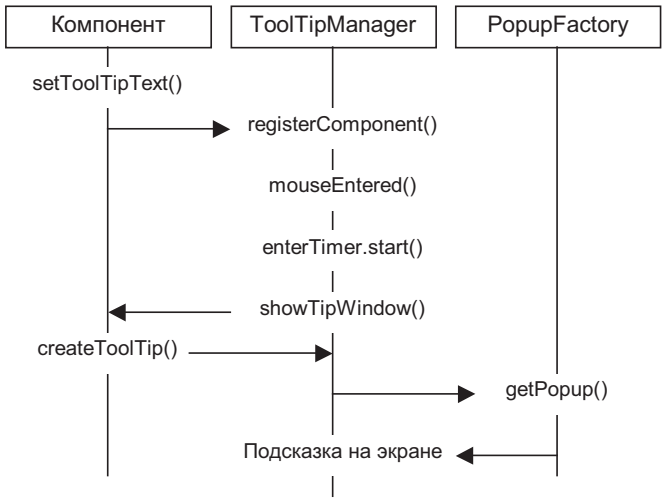



**Рис. 8.1.** Схема вывода всплывающей подсказки

Когда подсказка уже находится на экране, класс ToolTipManager запускает таймер exitTimer, при срабатывании которого она скрывается. Также во время нахождения подсказки на экране ToolTipManager следит за фокусом компонента (с помощью слушателя FocusListener) и щелчками мыши. В случае если пользователь щелкает мышью, когда ее указатель находится на компоненте или подсказке, или компонент теряет фокус ввода, подсказка скрывается.

У вас есть возможность управлять некоторыми аспектами работы класса ToolTipManager. Прежде всего, вы можете переопределить метод createToolTip() в вашем компоненте, если вам понадобится предоставить для него какую-то особенную подсказку, но делать это слишком часто не рекомендуется. Лучше, если у приложения имеется единый продуманный облик, и подсказки в него органично вписываются. Если вас не устраивают подсказки, предоставляемые используемым вашим приложением внешним видом, лучше всего написать для них новый UI-представитель. С другой стороны, имеются стандартные варианты настройки подсказок, и мы их сейчас рассмотрим.

## Настройка подсказок

Класс ToolTipManager позволяет настраивать параметры вывода подсказок на экран. В первую очередь это относится ко времени, которое проходит перед появлением подсказки, времени ее нахождения на экране и еще к некоторым параметрам, позволяющим тонко управлять всплывающими подсказками в вашем приложении. Параметры,

заданные по умолчанию, удачно выбраны, и менять их придется редко, но знать их все же полезно. Кроме того, вы можете «намекнуть» о том, какой элемент вы бы хотели видеть в качестве «хозяина» подсказки, легковесный или тяжеловесный. Рассмотрим теперь пример настройки подсказок, а затем обсудим их параметры в деталях:

```
// ToolTipsTuning.java
// Настройка подсказок
import javax.swing.*;

public class ToolTipsTuning extends JFrame {
 public ToolTipsTuning() {
     super("ToolTipsTuning");
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     // добавим кнопки с подсказками
     JPanel contents = new JPanel();
     JButton b1 = new JButton("Первая");
     b1.setToolTipText("Подсказка для первой");
     JButton b2 = new JButton("Вторая");
     b2.setToolTipText("Подсказка для второй");
     contents.add(b1);
     contents.add(b2);
     // настройка подсказок
     ToolTipManager ttm =
         ToolTipManager.sharedInstance();
     ttm.setLightWeightPopupEnabled(false);
     ttm.setInitialDelay(1000);
     ttm.setDismissDelay(500);
     ttm.setReshowDelay(1000);
     // выводим окно на экран
     add(contents);
     setSize(200, 100);
     setVisible(true);
 }
 public static void main(String[] args) {
        SwingUtilities.invokeLater(
         new Runnable() {
          public void run() { new ToolTipsTuning(); } });
 }
}
```

Как и в предыдущем примере, мы добавляем в окно две кнопки, для которых включаем вывод подсказок. Далее с помощью экземпляра класса ToolTipManager (нетрудно видеть, что он представляет собой классического одиночку, экземпляр которого возвращает метод sharedInstance()) мы производим настройку подсказок нашего приложения. Доступные свойства перечислены в табл. 8.2.

**Таблица 8.2.** Свойства класса ToolTipManager

| Свойство | Описание |
|---|---|
| lightWeightPopupEnabled | Рекомендует классу ToolTipManager использовать для подсказок легковесные компоненты, которые затем появятся в слое POPUP_LAYER многослойной панели. Если значение равно false, подсказки размещаются либо в тяжеловесных компонентах (если места в окне достаточно), либо в собственных окнах без рамки (когда места не хватает). Следит за этим фабрика классов PopupFactory, предназначенная для всплывающих окон. Это свойство может быть полезно, если вы совмещаете в одном окне тяжеловесные и легковесные компоненты, и велика вероятность того, что легковесные подсказки могут быть скрыты тяжеловесными компонентами |
| initialDelay | Задержка (в миллисекундах) между остановкой указателя мыши на компоненте и появлением подсказки |
| dismissDelay | Время, в течение которого подсказка остается на экране (если только не происходит каких-то других событий, например, потери окном фокуса) |
| reshowDelay | Время, в течение которого действует «режим подсказок». Если пользователь, прочитав подсказку для одного компонента, тут же переводит указатель мыши на другой компонент, подсказка для второго компонента выводится незамедлительно |

В примере мы использовали довольно необычные значения описанных свойств, и это легко увидеть, запустив программу с примером: подсказки будут появляться после значительной задержки, а исчезать быстро, но зато «режим подсказок» действует довольно долго (надо только успеть переместить мышь, пока подсказка находится на экране). Пользоваться такими подсказками неудобно, так что свободно манипулировать описанными выше свойствами стоит хорошо подумав. Также для примера мы отказались от легковесных компонентов, но делать так не рекомендуется: легковесные компоненты значительно быстрее и позволяют добиваться большего с меньшими усилиями.

# *Рамки*

*Рамки* (borders) библиотеки Swing позволяют визуально упорядочить пользовательский интерфейс, разделяя компоненты по их функциям и назначению. Также довольно часто они используются в качестве «украшения» для того или иного компонента библиотеки, и нередко именно рамка определяет особый колорит компонента. Если вы вспомните внешний вид кнопок JButton, специальные границы текстовых полей или панелей прокрутки, то убедитесь, что основу внешнего вида таких компонентов иногда составляет необычная рамка.

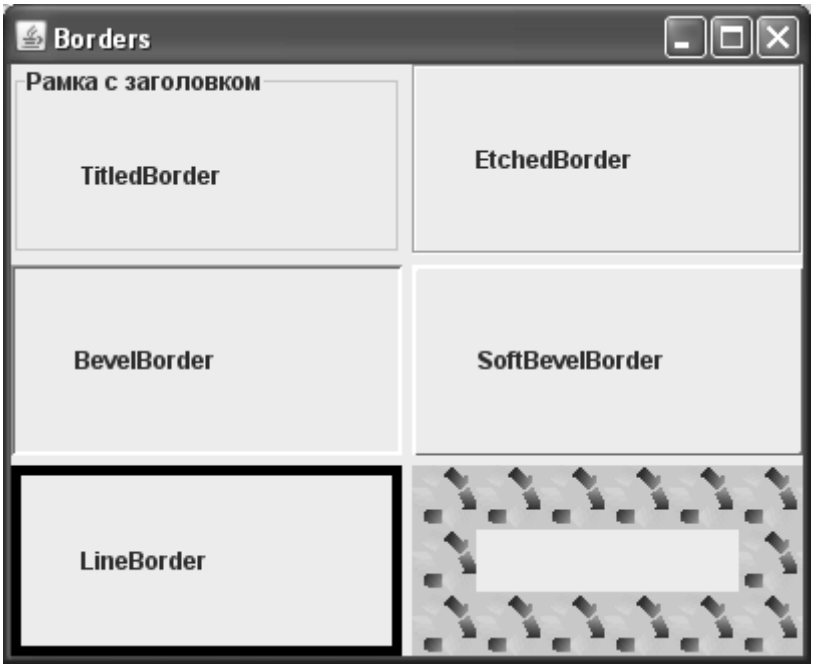
В качестве визуально организующего элемента рамки чаще всего используются для панелей JPanel, основных строительных кирпичиков, в которых вы размещаете свои компоненты с помощью того или иного менеджера расположения. В конечном итоге интерфейс складывается из множества определенным образом расположенных панелей JPanel, и панели, содержащие часть элементов интерфейса, отвечающих за определенные функции, удобно «помещать в рамку». Благодаря таким рамкам пользователю проще освоиться с приложением, потому что он быстро определяет, какая часть интерфейса за что отвечает, и без дополнительных усилий находит то, что ему нужно.

Поддержка рамок обеспечивается рисующими механизмами базового класса JComponent (детали мы исследовали в главе 4), так что для любого компонента Swing вы можете использовать нужную вам рамку. Рамка хранится в качестве свойства border, сменить или получить ее позволяют методы get/set. Обязанности рамок в Swing описаны в интерфейсе Border из пакета javax.swing.border, в этом же пакете находится впечатляющее количество стандартных рамок. Давайте рассмотрим пример и ознакомимся с ними:

```
// Borders.java
// Рамки Swing
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;

public class Borders extends JFrame {
 public Borders() {
     super("Borders");
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     // создаем панели с всевозможными рамками
     JPanel contents = new JPanel(
         new GridLayout(3, 2, 5, 5));
     contents.add(createPanel(new TitledBorder(
         "Рамка с заголовком"), "TitledBorder"));
     contents.add(createPanel(new EtchedBorder(),
         "EtchedBorder"));
     contents.add(createPanel(new BevelBorder(
         BevelBorder.LOWERED), "BevelBorder"));
     contents.add(createPanel(new SoftBevelBorder(
         BevelBorder.RAISED), "SoftBevelBorder"));
     contents.add(createPanel(new LineBorder(
         Color.BLACK, 5), "LineBorder"));
     contents.add(createPanel(new MatteBorder(
         new ImageIcon("matte.gif")), "MatteBorder"));
     // выводим окно на экран
     add(contents);
     pack();
     setVisible(true);
 }
 // метод создает панель с рамкой и надписью
 private JPanel createPanel(Border b, String text) {
     JPanel panel = new JPanel(new BorderLayout());
     panel.add(new JLabel(text));
     panel.setBorder(new CompoundBorder(
```

```
            b, new EmptyBorder(30, 30, 30, 30)));
        return panel;
    }
    public static void main(String[] args) {
            SwingUtilities.invokeLater(
             new Runnable() {
              public void run() { new Borders(); } });
    }
}
```

В примере мы создаем окно с панелью в центре, для которой выбираем табличное расположение с шестью ячейками; именно шесть наиболее употребительных рамок содержит пакет javax.swing.border. В ячейки табличного расположения мы добавляем панели с установленными рамками и надписью JLabel, которая помогает разобраться, какой класс позволяет создать ту или иную рамку. При этом используется вспомогательный метод createPanel(), который создает новую панель с полярным расположением, в центр ее добавляет надпись и устанавливает нужную рамку Border. Заметьте, что метод createPanel() «украшает» панели с помощью сразу трех рамок: собственно панель получает в качестве рамки объект CompoundBorder, позволяющий совместить две рамки в одну. Внешней рамкой является та, которая передается в метод в качестве параметра, а внутренняя рамка для всех панелей одна — это рамка EmptyBorder, оставляющая между границей панели и ее содержимым пустое пространство, чтобы надпись с названием рамки было проще читать. Запустив программу с примером, вы сможете насладиться разнообразием рамок Swing, и это только начало: каждая из рассмотренных нами шести рамок имеет дополнительные варианты оформления.



Когда вы будете выбирать рамку для своего приложения, вам сможет помочь табл. 8.3 с их краткими характеристиками.

**Таблица 8.3.** Стандартные рамки Swing

| Название рамки | Назначение |
|---|---|
| TitledBorder | Позволяет совместить любую рамку и текст, описывающий то, что в этой рамке находится. При этом текст может находиться на любой стороне рамки и иметь различные варианты выравнивания. Очень часто используется в интерфейсах |
| LineBorder | Одна из самых простых, но наиболее часто используемых рамок. Рисует рамку линией определенного цвета, толщину линии можно выбрать по вкусу, позволяет скруглять углы |
| EmptyBorder | Отличный помощник в создании выверенных пользовательских интерфейсов, позволяет окружить компонент пустыми полями со всех четырех сторон. Мы уже использовали эту рамку в главе 5 |
| BevelBorder | Позволяет создать объемную рамку, выпуклую или вогнутую, по желанию можно настроить цвета, требуемые для получения объемных эффектов. Имеет «сильный» визуальный эффект, поэтому использовать ее надо осторожно |
| SoftBevelBorder | Эта рамка аналогична предыдущей (унаследована от нее), обладает теми же свойствами, но дополнительно позволяет скруглить углы рамки |
| EtchedBorder | Рамка с тиснением (в стиле чеканки по металлу) выглядит скромно, но эффектно, поэтому используется часто. Может быть вогнутой или выпуклой |
| CompoundBorder | Позволяет совместить две рамки и избавляет от утомительного совмещения панелей. Вкладывая такие рамки друг в друга, можно совместить их произвольное количество |
| MatteBorder | Очень полезная и мощная рамка, позволяющая использовать узор из значков Icon. С ее помощью легко создавать очень необычные рамки. Она часто избавляет от необходимости создавать свои процедуры прорисовки и новые рамки |

С помощью стандартных рамок Swing можно добиться практически любого эффекта и в плане организации пользовательского интерфейса, и в плане оформления компонентов. Однако с рамками следует быть осторожным: слишком большое их количество утомляет и раздражает пользователя, интерфейс кажется пестрым и тяжелым для восприятия. Лучше ограничиться одной–двумя рамками для четкого разделения компонентов, выполняющих разные функции или собирающих данные разного предназначения, и *никогда* не следует вкладывать рамки друг в друга, если только это не относится к рамке EmptyBorder. Для организации компонентов интерфейса лучше всего использовать рамку TitledBorder, размещающую на других рамках текст, для оформления компонентов хорошо подходят неброские рамки LineBorder и EtchedBorder. Если вам понадобится особенно красочная рамка, вы можете обратиться к MatteBorder.

Всегда стоит держать под рукой рамку с пустым пространством EmptyBorder. Она не вносит в интерфейс излишней пестроты и позволяет хорошо организовать его. В главе 7 мы вкратце рассмотрели рекомендации по созданию пользовательских интерфейсов для Java -приложений, которые проще всего выполнить именно с помощью рамок EmptyBorder, особенно это касается отступов от границ окон и панелей.

## Фабрика BorderFactory

Мы уже узнали, как напрямую создавать рамки из пакета javax.swing.border. Оказывается, есть еще один способ создания рамки – использовать *фабрику* javax.swing.BorderFactory. Это класс с набором статических методов, создающих тот или иной тип рамки. Он позволяет отказаться от импорта пакета javax.swing.border, а также по мере возможности кэширует создаваемые объекты Border. На самом деле, компоненты Swing прорисовываются поочередно, один за другим (как вы помните, запрос на прорисовку приходит из очереди событий), и это позволяет использовать один и тот же объект Border для разных компонентов. Правда, успешно кэшировать можно лишь рамки без состояния, к которым относятся объемные рамки BevelBorder и SoftBevelBorder, а также рамка EtchedBorder. Остальные рамки отличаются друг от друга цветами, надписями и другими параметрами и создаются по одной на компонент. А теперь убедимся, что использовать класс BorderFactory не сложно:

```
// UsingBorderFactory.java
// Фабрика рамок BorderFactory
import javax.swing.*;

public class UsingBorderFactory extends JFrame {
 public UsingBorderFactory() {
     super("UsingBorderFactory");
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     // рамка для панели содержимого
     JPanel cp = (JPanel)getContentPane();
     cp.setBorder(BorderFactory.createTitledBorder(
         BorderFactory.createRaisedBevelBorder(),
             "Сделано на фабрике рамок"));
     // выводим окно на экран
     setSize(300, 200);
     setVisible(true);
 }
 public static void main(String[] args) {
        SwingUtilities.invokeLater(
         new Runnable() {
          public void run() { new UsingBorderFactory(); } });
 }
}
```

Фабрика BorderFactory обеспечивает создание рамки с заголовком для панели содержимого нашего окна. Как видите, рамку любого типа можно создать соответствующим статическим методом, а рамку с заголовком – на основе рамки другого типа.

Создание рамок напрямую или с помощью фабрики классов BorderFactory практически одинаково, что выбрать – дело вкуса. Класс BorderFactory имеет преимущество, лишь когда в вашем приложении создается много рамок, которые могут кэшироваться, таких как BevelBorder или EtchedBorder. В этом случае следует предпочесть именно его.

## Создание собственных рамок

В Swing множество стандартных рамок, мы их только что рассмотрели, и надо сказать, что рамок этих хватает для большинства ситуаций (особенно если учитывать тот факт, что практически каждую из стандартных рамок можно настраивать и вдобавок совмещать с другими рамками). Тем не менее процесс создания собственной рамки совсем несложен и позволяет использовать для украшения рамки любой орнамент, на который у вас хватит фантазии.

Для создания новой рамки необходимо реализовать интерфейс javax.swing.border.Border, в котором совсем немного методов[6]. Во-первых, вам нужно определиться в том, будет ли ваша рамка непрозрачна (opaque). Смысл свойства непрозрачности для рамок соответствует его смыслу для других компонентов Swing: если вы утверждаете, что ваша рамка непрозрачна, то обязуетесь закрашивать всю область, занимаемую рамкой. Это упрощает работу механизмов рисования. Во-вторых, надо определить размеры рамки (как вы помните из главы 4, рамка в Swing рисуется прямо поверх компонента, и последнему надо знать ее размеры, чтобы его части не оказались закрытыми рамкой). Ну и последним этапом является собственно прорисовка рамки в специальном методе paintBorder(). А теперь займемся непосредственно процессом создания рамки: попробуем создать рамку, составленную из эффектных кривых Безье:

```
// BezierBorder.java
// Рамка, составленная из кривых Безье
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.awt.geom.*;

public class BezierBorder implements Border {
  // свойства рамки
  private Color color, shadow;
  private float thickness;

  // значения свойств передаются в конструкторе
  public BezierBorder(Color color, Color shadow,
                      float thickness) {
    this.color = color;
    this.shadow = shadow;
    this.thickness = thickness;
  }

  // место, занимаемое рамкой
```

[6] Есть еще один способ создать собственную рамку — унаследовать ее от абстрактного класса AbstractBorder. Но такой способ вряд ли значительно снизит объем работ по созданию рамки, особенно если рамка весьма необычна, так что мы выбираем реализацию интерфейса Border.

```
public Insets getBorderInsets(Component comp) {
  return new Insets(9, 9, 9, 9);
}
// наша рамка частями прозрачна
public boolean isBorderOpaque() {
  return false;
}
// метод прорисовки рамки
public void paintBorder(Component c, Graphics g,
                        int x, int y, int width, int height) {
  // используем новый объект Graphics
  Graphics2D g2 = (Graphics2D)g.create();
  // настройка пера, координат и цвета
  g2.setStroke(new BasicStroke(thickness));
  g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
      RenderingHints.VALUE_ANTIALIAS_ON);
  x += 5; y += 5; width -= 10; height -= 10;
  g2.setColor(shadow);
  // прорисовываем тень и рамку
  for (int i=0; i<2; i++) {
    CubicCurve2D left = new CubicCurve2D.Double( x, y,
        x-5, y+height*1/3, x+5, y+height*1/3,
        x, y+height);
    CubicCurve2D right = new CubicCurve2D.Double(
        x+width, y, x+width-5, y+height*1/3, x+width+5,
        y+height*2/3, x+width, y+height);
    CubicCurve2D top = new CubicCurve2D.Double( x, y,
        x+width*1/3, y+5, x+width*2/3, y-5, x+width, y);
    CubicCurve2D bottom = new CubicCurve2D.Double( x,
        y+height, x+width*1/3, y+height+5, x+width*2/3,
        y+height+5, x+width, y+height);
    g2.draw(left);
    g2.draw(right);
    g2.draw(top);
    g2.draw(bottom);
    // на втором шаге рисуем саму рамку
    x--; y--; width--; height--;
    g2.setColor(color);
  }
  g2.dispose();
}
```

```
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() {
            JFrame frame = new JFrame("BezierBorder");
            // создаем панель с нашей рамкой
            JPanel p = new JPanel(new BorderLayout());
            Border b = new TitledBorder(new BezierBorder(
                Color.GREEN, Color.DARK_GRAY, 3f), "Bezier");
            p.setBorder(b);
            p.add(new JTextArea());
            // выводим окно на экран
            frame.setDefaultCloseOperation(
                JFrame.EXIT_ON_CLOSE);
            frame.add(p);
            frame.setSize(200, 200);
            frame.setVisible(true);
          } });
  }

}
```

Класс BezierBorder реализует интерфейс Border и рисует рамку с помощью кривых Безье и средств Java2D. Прежде всего, отметьте конструктор класса: он позволяет программисту-клиенту задавать толщину кривых (значение может быть дробным, это входит в возможности Java2D) и цвета прорисовки кривых и тени. Если вам не нужна тень, достаточно передать в конструктор два одинаковых цвета. Далее мы указываем (в методе getBorderInsets()), что наша рамка будет занимать примерно по 9 пикселов с каждой стороны компонента, для которого она предназначена (это грубый подсчет, на самом деле занимаемое рамкой место зависит от ее толщины, но если не делать толщину кривых огромной, то приближенный подсчет вполне сгодится). Затем мы методом isBorderOpaque() указываем системе прорисовки, что наша рамка не является непрозрачной, то есть не закрашивает всю занимаемую ею область компонента. Это на самом деле так: мы рисуем кривые Безье и тени, но не закрашиваем всю область рамки, так что сквозь кривые вполне могут просвечивать другие компоненты (те, что находятся в стопке ниже компонента с рамкой BezierBorder).

Основная работа происходит в методе paintBorder(). Именно в нем необходимо прорисовать с помощью переданного объекта Graphics нашу рамку. Заметьте, что мы сразу же создаем копию рисующего объекта Graphics, вызывая его метод create(). Эту технику мы описывали еще в главе 4, и так нужно делать *всегда*. Все дело в том, что в метод paintBorder() передается тот же самый объект Graphics, который используется для прорисовки собственно компонента, его потомков, рамок этих потомков и т. д. Если мы так или иначе поменяем настройку объекта Graphics и не вернем его параметры в исходное состояние (подобным образом при прорисовке рамки мы меняем цвет и включаем сглаживание графики), то наши новые параметры повлияют на прорисовку остальных компонентов и приведут к отталкивающим результатам. Создав копию объекта Graphics, мы преобразуем его к рисующему объекту Graphics2D библиотеки Java2D (это безопасно, поскольку библиотека Java2D используется во

всех рисующих операциях Swing) и настраиваем нужные нам параметры: создаем перо BasicStroke подходящей толщины, включаем сглаживание графики и для удобства немного изменяем переданные нам в метод paintBorder() координаты. Затем мы дважды рисуем четыре кривых Безье: один раз цветом тени, другой раз с небольшим смещением координат основным цветом. Чтобы не повторять идентичный код, мы использовали цикл из двух итераций. Кривые Безье рисуются с помощью класса CubicCurve2D из пакета java.awt.geom, для их прорисовки необходимо указать координаты четырех точек: начала и конца кривой и двух контрольных точек, которые будут отвечать за «искривление» кривой. Контрольные точки мы размещаем на расстоянии одной трети и двух третей от начала кривой, в пяти пикселах влево для первой точки и в пяти пикселах вправо для второй. Это позволяет получить небольшое эффектное искривление. Всего создаются четыре кривые Безье — по одной для каждой из сторон рамки. После завершения рисования необходимо удалить созданный нами объект Graphics методом dispose(), сборщик мусора здесь не поможет, потому что в объекте Graphics используются ресурсы операционной системы, время жизни которых определяем мы сами[7].

Для проверки работы нашей новой рамки мы добавили в класс BezierBorder метод main(). В нем (конечно, в потоке рассылки событий) мы создаем окно JFrame, в панель содержимого которого добавляем еще одну панель и устанавливаем для нее рамку с заголовком в качестве основы для нашей новой рамки (заметьте, что новая рамка легко совмещается со стандартными рамками Swing, в том числе и с использованной в примере рамкой с заголовком). В качестве параметров для BezierBorder мы указали зеленый цвет для рамки, темно-серый для тени, а толщину кривых установили равной трем пикселам. Чтобы удостовериться в том, что компоненты Swing легко совмещаются с нашей новой рамкой, мы добавили внутрь панели текстовую область JTextArea. Запустив программу с примером, вы увидите результаты наших трудов.


[7] Можно предложить применять интересный вариант кода, позволяющий всегда гарантировать удаление созданного объекта Graphics. Как мы все знаем, всегда вызывается секция finally. Так как исключений при рисовании практически не возникает, секцию catch можно не указывать:

```
try {
 создаем новый объект Graphics, рисуем…
} finally {
  объект.dispose()
}
```

Согласитесь, что новая рамка весьма эффектна, и вряд ли ее можно было бы создать с помощью стандартных рамок Swing. Вы можете добавить ее в свой арсенал и использовать, если вам понадобится особенно пышно выделить некий компонент или их группу.

## Рамки и разработка собственных компонентов

Как мы уже обсудили в главе 4, в компонентах Swing нет специального места для рамок, рисующие механизмы библиотеки, скрытые в классе JComponent, прорисовывают рамку прямо на компоненте, вызывая метод paintBorder() после метода paintComponent(). Поэтому, если вы создаете компонент с собственной процедурой прорисовки и не прочь использовать для него рамки Swing, придется учитывать то место, которое занимает рамка (если она есть) и не рисовать на нем. Определить место, занимаемое рамкой, можно и вручную — мы уже обсудили интерфейс Border и знаем, что размеры рамки возвращает метод getBorderInsets(). Получив эти размеры и вычислив, где находится рамка, вы сможете не рисовать на занятом ею месте. Есть и еще один, более простой, способ определить прямоугольник, в котором можно рисовать без риска повредить рамку. Подобный прямоугольник возвращает статический метод getInteriorRectangle() класса AbstractBorder. Рассмотрим небольшой пример и увидим, на что способен этот метод:

```
// PaintingWithBorders.java
// Рисование компонента с учетом рамки
import javax.swing.*;
import javax.swing.border.AbstractBorder;
import java.awt.*;

public class PaintingWithBorders extends JFrame {
 public PaintingWithBorders() {
     super("PaintingWithBorders");
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     // добавим к нашему компоненту рамку
     CustomComponent cc = new CustomComponent();
     cc.setBorder(BorderFactory.
         createTitledBorder("Рамка!"));
     // добавим компонент в окно
     getContentPane().add(cc);
     setSize(400, 300);
     setVisible(true);
 }
 // компонент со своей собственной процедурой прорисовки
 class CustomComponent extends JComponent {
     public void paintComponent(Graphics g) {
         // получаем подходящий прямоугольник
         Rectangle rect = AbstractBorder.
             getInteriorRectangle(this, getBorder(),
```

```
                0, 0, getWidth(), getHeight());
            // рисуем в нем
            g.setColor(Color.white);
            g.fillRect(
                rect.x, rect.y, rect.width, rect.height);
        }
    }
    public static void main(String[] args) {
       SwingUtilities.invokeLater(
        new Runnable() {
         public void run() { new PaintingWithBorders(); } });
    }
}
```

В примере мы создаем небольшое окно, в котором размещается специально созданный компонент CustomComponent со своей процедурой прорисовки. Процедура прорисовки, размещенная согласно правилам рисующих механизмов Swing в методе paintComponent(), не делает ничего экстраординарного: она просто закрашивает белым цветом область, занимаемую компонентом. Интересней увидеть, как учитывается место для рамки, которая, как мы знаем, может быть у каждого компонента, унаследованного от базового класса JComponent. Нам помогает метод getInteriorRectangle() класса AbstractBorder: он возвращает прямоугольник Rectangle, в котором можно рисовать, не опасаясь того, что вызванный после paintComponent() метод paintBorder() начнет рисовать рамку прямо на только что прорисованном нами компоненте. Метод getInteriorRectangle() вычисляет, какое пространство занимает рамка (вызывая ее метод getBorderInsets()) и на основе данных, передаваемых ему в качестве параметров (компонент, его рамка, координаты начала области, в которой вы намереваетесь рисовать, а также длина и высота этой области), определяет прямоугольник, прорисовка в котором не затронет рамку. Если рамки у компонента нет, то метод getInteriorRectangle() возвращает размеры компонента без изменений.

Мы устанавливаем для нашего компонента рамку с заголовком TitledBorder, и, запустив программу с примером, вы увидите, что рамка не затрагивает область, залитую белым цветом.


Если бы мы не учли место, занимаемое рамкой, то она появилась бы прямо на компоненте, и внешний вид его был бы безнадежно испорчен. Все стандартные компоненты Swing и их UI-представители учитывают при прорисовке возможность появления рамки, и вам при разработке собственных компонентов или UI-представителей

следует делать то же самое. Это не слишком элегантно и вносит в процесс прорисовки компонента определенные сложности, но другого пути нет — так уже реализована поддержка рамок в Swing.

## *Резюме*

Вы ничем не ограничены в выводе для пользователя вспомогательной информации, разве что собственной фантазией. Интересы пользователя программы должны быть на первом месте, и, чтобы передать ему нужную информацию в нужное время и в нужном месте, к вашим услугам впечатляющие возможности надписей, всплывающих подсказок и рамок, а также весь арсенал встроенного языка HTML.

# Глава 9. Элементы управления

Взгляните на современные приложения с графическим интерфейсом и скажите, какие компоненты интерфейса неизменно присутствуют в каждом из них. Конечно, это элементы управления. В разных обличиях и для разных целей, упакованные в меню, выстроенные в строгие ряды панелей инструментов, они вездесущи. Элементы управления позволяют пользователю общаться с программой и управлять ее ходом.

Количество и внешний вид элементов управления, предоставляемых библиотекой Swing, поражает уже при первом знакомстве. Swing содержит полный арсенал существующих сегодня элементов управления: это кнопки, флажки, переключатели, меню и его элементы и многое другое. Все эти элементы в библиотеке связаны, потому что все они унаследованы от абстрактного класса AbstractButton, определяющего поведение любого компонента, претендующего на звание элемента управления. Таким образом, изучив на примере какого-либо элемента управления важнейшие свойства этого базового класса, мы затем сможем с легкостью применять полученные знания и ко всем остальным элементам управления, разом освоив приличную часть библиотеки.

Основные свойства, присущие любому элементу управления Swing, проще всего изучать на примере компонента, который уже не раз встречался нам в примерах и выручал своей простотой, — это кнопка JButton.

## *Кнопки JButton*

В простейшем виде кнопки — это самые обычные прямоугольники с текстом; пользователь щелкает на них мышью, чтобы выполнить какое-либо действие или о чем-либо просигнализировать. Кнопки JButton хороши еще и тем, что кроме собственного внешнего вида, в них практически нет ничего уникального — все в них унаследовано от класса AbstractButton, и поэтому все, что верно для них, будет верно и для остальных элементов управления.

Чаще всего использование этого компонента сводится к его созданию, размещению в подходящем месте контейнера и привязыванию слушателя событий. Выглядит все обычно так:

```
JButton button = new JButton("Кнопка");
button.addActionListener(new ButtonAction());
```

Таким образом, время занимает не столько создание и настройка кнопки, сколько размещение ее в контейнере и написание обработчика событий. Впрочем, так оно и должно быть — кнопка призвана всего лишь дать сигнал о начале некоторого действия. Начнем знакомство с кнопками с того, что узнаем, как можно менять внешний вид кнопок, используя значки, цвета и различное выравнивание их содержимого.

### Внешний вид кнопок

Пожалуй, в библиотеке Swing нет больше других компонентов, облик которых можно так же легко изменить, как облик кнопок, не меняя менеджера внешнего вида и поведения и не переопределяя рисующих методов. С кнопками можно делать практически все — сопоставлять каждому движению пользователя свой значок, убирать рамку, закрашивать в любой цвет, перемещать содержимое по разным углам, не рисовать фокус.

И это выполняется с такой фантастической простотой (достаточно вызвать один-два метода), что в творческом порыве можно забыть буквально обо всем. Рассмотрим пример, в котором будут созданы кнопки самых необычных форм и размеров, а затем обсудим, как это делается:

```
// ButtonStyles.java
// Изменение внешнего вида кнопок JButton
// с помощью значков, цветов, рамок и т. п.
import javax.swing.*;
import java.awt.*;

public class ButtonStyles extends JFrame {
  public ButtonStyles() {
    super("ButtonStyles");
    setDefaultCloseOperation( EXIT_ON_CLOSE );
    // используем последовательное расположение
    setLayout(
        new FlowLayout( FlowLayout.LEFT, 10, 10));
    // самая простая кнопка
    JButton button = new JButton("Обычная кнопка");
    add(button);
    // кнопка со значками на все случаи жизни
    button = new JButton();
    button.setIcon(new ImageIcon("images/b1.gif"));
    button.setRolloverIcon(
        new ImageIcon("images/b1r.gif"));
    button.setPressedIcon(
        new ImageIcon("images/b1p.gif"));
    button.setDisabledIcon(
        new ImageIcon("images/b1d.gif"));
    // для такой кнопки лучше убрать
    // все ненужные рамки и закраску
    button.setBorderPainted(false);
    button.setFocusPainted(false);
    button.setContentAreaFilled(false);
    add(button);
    // кнопка с измененным цветом и HTML-текстом
    button = new JButton(
        "<html><h2><font color=\"yellow\">Зеленая кнопка");
    button.setBackground(Color.green);
    add(button);
    // изменение выравнивания текста и изображения
    button = new JButton("Изменение выравнивания",
```

```
            new ImageIcon("images/button.gif"));
        button.setMargin(new Insets(10, 10, 10, 10));
        button.setVerticalAlignment(SwingConstants.TOP);
        button.setHorizontalAlignment(SwingConstants.RIGHT);
        button.setHorizontalTextPosition(SwingConstants.LEFT);
        button.setVerticalTextPosition(SwingConstants.BOTTOM);
        button.setIconTextGap(10);
        // сделаем кнопку большой, чтобы увидеть выравнивание
        button.setPreferredSize(new Dimension(300, 100));
        add(button);
        // отключенная кнопка
        button = new JButton("Выключено");
        button.setEnabled(false);
        add(button);
        // выводим окно на экран
        setSize(400, 350);
        setVisible(true);
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(
            new Runnable() {
              public void run() { new ButtonStyles(); } });
    }
}
```

В этом примере создается небольшое окно, в котором используется менеджер последовательного расположения FlowLayout. Такое расположение гарантированно придаст кнопкам предпочтительный размер. Далее в окно помещается несколько кнопок, каждая из которых демонстрирует свой способ изменения внешнего вида.

Первой в контейнер помещается самая обычная кнопка, которая уже встречалась нам в примерах, просто для того, чтобы можно было сравнить ее с остальными кнопками. Затем следует кнопка, на которой вместо текста располагаются значки, причем на «все случаи жизни». Для задания значков применяются свойства, перечисленные в табл. 9.1.

**Таблица 9.1.** Свойства, позволяющие задавать значки для кнопок

| Свойства (и методы get/set) | Описание |
|---|---|
| icon | Используется для установки «повседневного» значка, который будет виден всегда, когда кнопка доступна |
| rolloverIcon | С помощью этого свойства можно установить значок для получения эффекта «наведения мыши» — когда указатель мыши оказывается на кнопке, появляется этот значок. Обычно в этом случае используется тот же значок, только к нему добавляется эффект объема или изменяется цвет части значка (как в нашем примере) |

**Таблица 9.1** *(продолжение)*

| Свойства (и методы get/set) | Описание |
|---|---|
| pressedIcon | Используется для установки значка, который выводится на экран при щелчке пользователем на кнопке (значок остается на экране до тех пор, пока кнопка мыши не отпущена и указатель находится в области кнопки) |
| disabledIcon | Если вы задаете значок посредством данного свойства, то, когда кнопка будет отключена с помощью метода setEnabled(false), появится специальный значок. Если специального значка нет, а обычный значок есть, то в качестве значка для отключенной кнопки будет использована черно-белая копия обычного значка (аналогично поведению надписи JLabel, которую мы подробно рассмотрели в главе 6) |

Если вывести такую кнопку на экран «как есть», она будет выглядеть, мягко говоря, не очень хорошо, потому что у нее останутся все свойства, присущие обычной кнопке с текстом: будет рисоваться рамка, при наличии фокуса появится контур, а при нажатии она будет закрашиваться темным цветом. Именно для того чтобы убрать эти эффекты, в примере задействуются методы setBorderPainted(), setFocusPainted() и setContentAreaFilled(). Первый из перечисленных методов позволяет отключить прорисовку рамки (то же самое можно сделать и с помощью вызова setBorder(null), но тогда вернуть кнопке ее рамку не удастся), второй отключает прорисовку специального контура, проявляющегося, если кнопка обладает фокусом ввода, а третий дает возможность отключить закраску кнопки в нажатом состоянии. Однако работать с такой кнопкой гораздо сложнее: будет непонятно, выбрана кнопка или нет, где она начинается и т. п. Поэтому мы рекомендуем быть осторожнее и применять эти методы только в тех приложениях, в которых весь интерфейс основан на разнообразных изображениях (например, в играх).

После этого в контейнер добавляется еще одна простая кнопка с текстом, для которой методом setBackground() устанавливается необычный цвет заполнения. В некоторых ситуациях бывает полезно просто изменить цвет кнопки, и вы видите, что сделать это действительно несложно. Помните только, что цвет заполнения изменится, только если у кнопки включено свойство непрозрачности (opaque). Все стандартные внешние виды, поставляемые со Swing, делают кнопки непрозрачными, но внешние виды от стороннего производителя могут иметь на это свою «точку зрения», так что иногда при использовании стороннего внешнего вида приходится вручную устанавливать для свойства непрозрачности значение true, если вы хотите закрасить кнопку в особенный цвет.

После прочтения главы 8 мы в полной мере смогли оценить мощь и гибкость встроенного в надписи и подсказки языка HTML, кнопки в этом отношении нас также не разочаровывают. Как и все остальные компоненты библиотеки, выводящие на экран некоторый текст, они поддерживают HTML, что вместе с широчайшими возможностями по настройке расположения содержимого и по управлению всеми аспектами внешнего вида кнопок дает вам неограниченную власть над видом вашего приложения. В примере мы парой строк кода смогли без особого напряжения создать кнопку, буквально «пылающую» яркими цветами и выделяющуюся нестандартными размерами. Не обладай кнопки поддержкой встроенного языка HTML, такой эффект дался бы нам гораздо большими усилиями, а про остальные возможности HTML (сразу же можно вспомнить незаменимый тег <br>, позволяющий легко организовать в компонентах Swing многострочные надписи) и вовсе пришлось бы забыть.

Далее показано, как и какими методами можно изменить положение содержимого кнопки (значка или надписи). Правда, чтобы увидеть это, пришлось придать кнопке

неестественно большой размер, потому что если оставить тот размер, который задан по умолчанию, менеджер FlowLayout придаст кнопке оптимальный размер, и как ни меняй после этого расположение содержимого, мы все равно ничего бы не увидели. Если же кнопка занимает больше пространства, чем ей надо, вопросы положения содержимого становятся актуальными.

Для демонстрации вариантов выравнивания была создана кнопка и с текстом, и со значком[1] (как вы можете видеть из кода программы, для этого случая у класса JButton даже имеется специальный конструктор). В табл. 9.2 описаны свойства, которые помогли нам добиться желаемого визуального эффекта.

**Таблица 9.2.** Свойства, позволяющие настроить содержимое кнопок JButton

| Свойства (и методы get/set) | Описание |
|---|---|
| margin | Это свойство позволяет управлять размером полей, которые отделяют содержимое кнопки (текст и значок) от ее границ. Как мы уже отмечали в главе 5, при использовании внешнего вида Metal рекомендуются поля в 12 пикселов |
| verticalAlignment<br>horizontalAlignment | Данная пара свойств используется для изменения позиции всего содержимого кнопки (и значка, и текста) относительно границ кнопки. С их помощью поместить это содержимое можно в любой из углов кнопки или в ее центр (по умолчанию оно там и находится). В нашем примере текст и значок помещаются в верхний правый угол кнопки. Полный список констант, управляющих положением содержимого, можно найти в интерактивной документации для класса SwingConstants, доступной на сайте java.sun.com |
| horizontalTextPosition<br>verticalTextPosition | С помощью этих свойств вы можете управлять положением текста кнопки относительно ее значка (если он есть, конечно) по горизонтали и вертикали. Использование в примере первого метода позволило разместить надпись не справа от значка, а слева[2]. Второй метод позволил поместить надпись ниже значка |
| iconTextGap | Это свойство долго отсутствовало в классе AbstractButton, хотя необходимость в нем была (например, в классе JLabel, имеющем аналогичный набор свойств для выравнивания содержимого, оно появилось вместе с появления библиотеки Swing). Свойство позволяет изменить расстояние между значком и текстом. |

Нетрудно увидеть, что содержимое кнопок и расположение в них значка относительно текста задается набором свойств, который полностью аналогичен набору свойств для выравнивания содержимого надписи JLabel. Мы обсуждали надписи в главе 8, поэтому большая часть возможностей кнопок должна быть вам прекрасно известна.

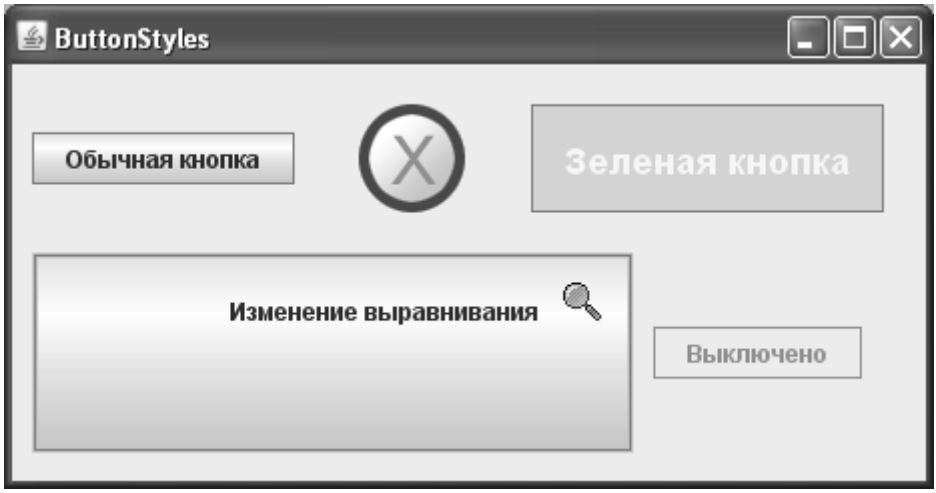
Наконец, самой последней в контейнер добавляется еще одна кнопка без всяких изысков, она показывает, как кнопку можно отключить. Делается это методом setEnabled(), этим же методом кнопка включается.

---

[1] Этот симпатичный значок автор взял из специальной коллекции стандартной графики для Java-приложений, выполненных во внешнем виде Metal. Эта коллекция бесплатно распространяется фирмой Sun/Oracle, найти ее можно на сайте java.sun.com.

[2] Кстати, использовать константы RIGHT (справа) и LEFT (слева) для выравнивания содержимого в приложениях, которые затем может понадобиться локализовать в странах с нестандартным письмом, не рекомендуется.

Запустив программу с примером, вы сможете убедиться, что внешний вид кнопок действительно меняется.




Конечно, в рассмотренном примере нет никаких хитростей, и все очень просто. Но так и должно быть — с каждой строкой кода вы создаете что-то действительно стоящее, а не решаете в очередной раз проблему рисования значка на кнопке. Библиотеки созданы для того, чтобы упростить работу программиста, вобрав в себя наиболее востребованные и сложные операции, и в этом библиотека Swing несомненно преуспела. И не забывайте о том, что все только что рассмотренные способы изменения внешнего вида действуют на все элементы управления библиотеки Swing, которые мы будем рассматривать в этой главе.

## У кнопок есть модель

Удивительно, но у таких простых компонентов как кнопки есть собственная, самая настоящая, *модель*, хранящая состояние кнопки (поддержка модели описана в базовом классе AbstractButton, так что модель имеется у всех обсуждаемых в этой главе элементов управления). В главе 1 при обсуждении архитектуры MVC, используемой в Swing, мы выяснили, что модель хранит данные компонента, а отображает эти данные его UI-представитель. Именно так все работает для кнопок: в модели ButtonModel хранится информация о состоянии кнопки (нажата ли она, находится ли над ней указатель мыши, в выбранном ли она положении; даже символ мнемоники и группа, к которой может принадлежать кнопка, хранятся в модели), а ее UI-представитель получает эту информацию из модели и соответствующим образом прорисовывает кнопку. Более того, все события, которые поддерживают кнопки и другие элементы управления (мы их вскоре обсудим подробнее и рассмотрим пример), на самом деле генерируются в модели кнопок ButtonModel (к примеру, если кнопка нажата, и на ней отпускается кнопка мыши, модель генерирует событие ActionEvent). С другой стороны, внешний вид кнопки (текст, значок, выравнивание) в модели не хранится, все это находится уже в конкретных классах, таких как JButton, так что модель отвечает лишь за состояние кнопки, но не за ее внешний вид.

«Зачем же кнопкам нужна отдельная модель?» — спросите вы. На самом деле, кнопки настолько просты и незатейливы, что отдельная модель кажется для них излишней роскошью. Вы в большинстве своих программ вряд ли используете эту модель отдельно от кнопок — все будет сводиться к простому созданию компонента и размещению его в контейнере. Тем не менее, у модели кнопок есть свои преимущества. Представим, что нам нужно

создать большое приложение с множеством функций и большую часть этих функций мы реализовали в виде разнообразных элементов управления: простых кнопок JButton, пунктов меню, всплывающих меню и т. д. Некоторые из этих элементов управления могут выполнять одну и ту же работу, но размещаться в разных местах: в диалоговых окнах, в меню, на панелях инструментов. Создав все нужные вам элементы управления, вы можете получить от одного из них модель (методом getModel()), и *разделить* ее между всеми элементами, выполняющими одно и то же действие (хотя внешний вид и расположение этих элементов управления могут быть разными). Разделение модели не таит в себе черной магии: вы просто передаете ссылку на одну и ту же модель разным элементам, используя метод setModel(). После этого, все элементы управления, имеющие одну и ту же модель, *всегда* будут находиться в одном и том же состоянии. К примеру, если некоторое действие становится недоступным, вы можете отключить одну из кнопок или сделать то же самое непосредственно с моделью, и все элементы управления, использующие эту модель, отключатся автоматически. Это сделает ваш код удивительно элегантным и чистым, легким в чтении и поддержке — в этом сама суть MVC. Чуть дальше мы узнаем об еще одном средстве удобной поддержки разных элементов управления с одинаковым предназначением — интерфейсе Action.

Модель кнопок описана в интерфейсе ButtonModel, а в качестве реализации Swing использует стандартный класс DefaultButtonModel. Писать собственную модель кнопок вам вряд ли стоит: данные в ней хранятся незамысловатые, стандартная модель работает хорошо, и никаких дивидендов при написании собственной модели не предвидится.

А теперь рассмотрим, о каких событиях может сообщать кнопка JButton и как в своей программе получать извещения об этих событиях.

## Обработка событий от кнопок

После того как вы создали кнопку, настроили ее внешний вид и разместили в контейнере, остается соединить ее с деловой частью вашей программы, то есть определить, какие действия будут выполняться при определенном действии пользователя над кнопкой. С помощью модели событий библиотеки Swing можно получить сообщение о событии в свою программу многими способами, так, как вам удобно, и писать действительно элегантные программы. Убедимся в этом еще раз на примере кнопок:

```
// ButtonEvents.java
// Обработка событий от кнопок JButton
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

public class ButtonEvents extends JFrame {
  private JTextArea info;
  public ButtonEvents() {
    super("ButtonEvents");
    setDefaultCloseOperation( EXIT_ON_CLOSE );
    // создаем кнопку и помещаем ее на север окна
    JButton button = new JButton("Нажмите меня!");
    add(button, "North");
    // поле для вывода сообщений о событиях
    info = new JTextArea("Пока событий не было\n");
```

```
    add(new JScrollPane(info));
    // привязываем к нашей кнопке слушателей событий
    // слушатели описаны как внутренние классы
    button.addActionListener(new ActionL());
    button.addChangeListener(new ChangeL());
    // присоединение слушателя прямо на месте
    button.addItemListener(new ItemListener() {
      public void itemStateChanged(ItemEvent e) {
        info.append("Это вы все равно не увидите");
      }
    });
    // выводим окно на экран
    setSize(400, 300);
    setVisible(true);
  }
  class ActionL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      info.append(
          "Получено сообщение о нажатии кнопки! От — "
              + e.getActionCommand() + "\n");
    }
  }
  class ChangeL implements ChangeListener {
    public void stateChanged(ChangeEvent e) {
      info.append(
          "Получено сообщение о смене состояния кнопки!\n");
      // это источник события
      Object src = e.getSource();
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new ButtonEvents(); } });
  }
}
```

Пример сам по себе очень прост — создается окно, на север которого помещается кнопка JButton, а в центр — многострочное текстовое поле JTextArea, вложенное в панель прокрутки JScrollPane. В это поле мы будем помещать сообщения о событиях, принятых кнопкой от пользователя.

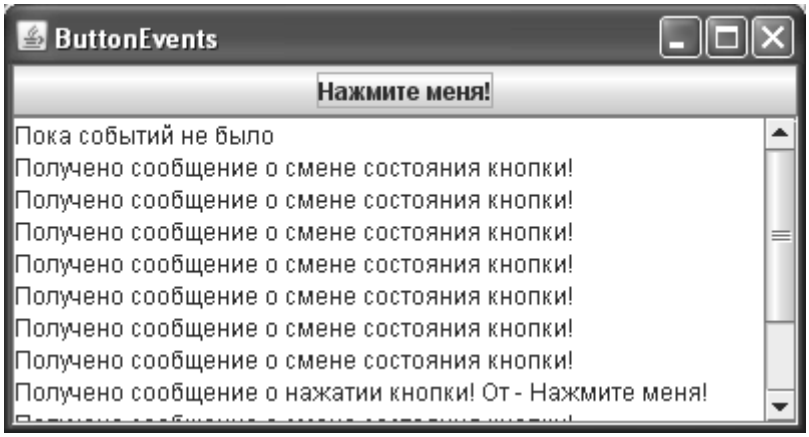
Элементы управления (в том числе кнопки), унаследованные от класса AbstractButton, могут посылать сообщения о трех типах событий (за исключением стандартных событий, общих для всех компонентов Swing). Эти события перечислены в табл. 9.3.

**Таблица 9.3.** События элементов управления, унаследованных от класса AbstractButton

| Событие | Описание |
|---|---|
| ActionEvent (слушатель ActionListener) | Самое понятное событие для любой кнопки и, пожалуй, самое простое и наиболее часто используемое событие вообще в любом графическом Java-приложении. Посылается кнопкой, когда пользователь щелкает на ней (нажимает и отпускает), чтобы произвести действие |
| ChangeEvent (слушатель ChangeListener) | Это событие мало что означает собственно для кнопок; с его помощью модель кнопок ButtonModel взаимодействует со своим UI-представителем. Модель при изменении хранящегося в ней состояния кнопки (это может быть изменение включенного состояния на выключенное, обычного на нажатое и т. п.) запускает событие ChangeEvent. UI-представитель обрабатывает это событие и соответствующим образом перерисовывает кнопку. Впрочем, вы можете обрабатывать это событие и сами, в том случае если вас интересуют малейшие изменения в состоянии кнопки |
| ItemEvent (слушатель ItemListener) | Это событие (мы рассмотрим его чуть позже) посылают компоненты, которые имеют несколько равноценных состояний (например, флажки и переключатели), чтобы сообщить о смене состояния |

После создания кнопки к ней присоединяются слушатели событий, по одному на каждое возможное событие. При этом также демонстрируются возможные способы присоединения слушателей. Для событий ActionEvent и ChangeEvent слушатели помещаются в отдельные внутренние классы, а слушатель события ItemEvent создается прямо на месте, как анонимный внутренний класс. Слушателей событий у кнопок может быть произвольное количество, поскольку кнопки относятся к компонентам с *групповой* (multicast) обработкой событий[3].

Запустив программу, вы увидите, как появляются события, читая сообщения в текстовом поле.




[3] На самом деле все компоненты в Swing поддерживают произвольное количество слушателей, благодаря использованию простого и эффективного средства для хранения слушателей — класса EventListenerList. Мы подробно обсуждали его в главе 2, когда создавали свои собственные типы событий.

Заметьте, что при щелчке на кнопке слушатель события определяет имя нажатой кнопки, используя метод getActionCommand() класса ActionEvent. Этот метод применяется для того, чтобы при обработке событий от нескольких кнопок в одном слушателе можно было их отличить. В качестве имени можно использовать произвольную строку символов, устанавливая ее методом setActionCommand() класса AbstractButton. Иногда этот механизм позволяет элегантно решать некоторые задачи. Например, если таким образом передавать имя класса, объект которого необходимо создать, то один слушатель будет способен обработать неограниченное количество кнопок.

События ActionEvent и ChangeEvent вообще несут не слишком много полезной информации, так как предполагается, что в приложении используется наиболее элегантный подход — каждому компоненту сопоставляется свой слушатель событий. Единственное, что можно себе позволить — это получить ссылку на источник события методом getSource(). Однако будьте осторожнее с вызовом этого метода, потому что это может привести к сильной связи между интерфейсом и деловой логикой, а это всегда нежелательно.

Запустив программу, вы также увидите, что событие ItemEvent не возникает вовсе. Это совсем не удивительно, потому что оно «работает» только с флажками, переключателями и другими компонентами, имеющими состояние, а кнопки его просто игнорируют.

Это все о событиях от кнопок JButton. У других элементов управления имеются дополнительные события, но в большинстве своем вам будет достаточно тех событий, которые мы только что обсудили. Говорить больше не о чем — остается лишь оценить простоту и мощь, которую способен привнести в программы продуманный объектно-ориентированный подход.

До сих пор вы, скорее всего, для выбора кнопок и других элементов управления в своих приложениях применяли мышь. В крайнем случае, вы могли воспользоваться встроенной в Swing системой перемещения фокуса ввода по компонентам путем нажатия клавиши Tab. Однако для настоящего приложения, претендующего на роль классного, этого мало.

## Мнемоники

Хорошее приложение не должно полностью полагаться на то, что у пользователя есть мышь или подобное ей устройство. Пусть сейчас уже трудно себе представить компьютер без такого приспособления, все равно эту ситуацию нужно учитывать. Еще более важно сделать приложение доступным для людей с ограниченными возможностями, которые не могут использовать манипуляторы. Именно для этого в приложениях и появились упоминавшиеся в предыдущей главе *мнемоники*, которые позволяют получить доступ к компоненту, нажимая специальную клавишу (обычно Alt) вместе с клавишей символа, идентифицирующего этот компонент.

Разработчики Swing учли это и наделили библиотеку поддержкой мнемоник для всех элементов управления, а также для надписей JLabel и вкладок JTabbedPane, так что вам не придется вручную прослушивать клавиатуру и заниматься обработкой клавиш. Рассмотрим сначала, как можно включить мнемоники в свою программу, а затем обсудим проблемы, которые при этом возникают:

```
// ButtonMnemonics.java
// Поддержка кнопками клавиатурных мнемоник
import javax.swing.*;
import java.awt.*;

public class ButtonMnemonics extends JFrame {
```

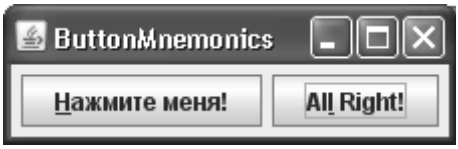
```
  public ButtonMnemonics() {
    super("ButtonMnemonics");
    setDefaultCloseOperation( EXIT_ON_CLOSE );
    // используем последовательное расположение
    setLayout(new FlowLayout());
    // создаем кнопку
    JButton button = new JButton("Нажмите меня!");
    // мнемоника (русский символ)
    button.setMnemonic('Н');
    add(button);
    // еще одна кнопка, только надпись на английском
    button = new JButton("All Right!");
    button.setMnemonic('L');
    button.setToolTipText("Жмите смело");
    button.setDisplayedMnemonicIndex(2);
    add(button);
    // выводим окно на экран
    pack();
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new ButtonMnemonics(); } });
  }
}
```

В этом очень простом примере в контейнер помещается две кнопки — надпись на первой сделана по-русски, а на второй — по-английски. Для того чтобы позволить пользователю получить доступ к кнопке с клавиатуры, были использованы перечисленные ниже методы класса AbstractButton.

- setMnemonic(). Позволяет указать мнемонику, то есть то, клавиша какого символа в сочетании с управляющей клавишей (Alt) будет вызывать нажатие кнопки. Можно просто указать символ (в одинарных кавычках, регистр не учитывается), а можно использовать константы из класса java.awt.event.KeyEvent, но первый подход проще и понятнее.
- setDisplayedMnemonicIndex(). Этот метод дает нам возможность управлять тем, какой из символов надписи кнопки будет подчеркиваться, то есть символизировать наличие мнемоники (если в надписи есть несколько одинаковых символов). В нашем примере в слове «All» этим методом можно выделить не первую букву «l», а вторую[4].

Запустив программу с примером, вы убедитесь, что символ мнемоники подчеркивается, и она работает... только для кнопки на английском языке.

[4] Пользуясь этим методом, не забывайте, что отсчет символов строки ведется с нуля.




Это очень неприятный недостаток текущей версии Swing, потому что он резко ограничивает возможности локализации приложений. Создатели библиотеки знают об этом, но проблема пока ими не решена. Для обычных компонентов можно использовать клавишу Tab, однако эта ошибка сказывается на меню особенно болезненно, и вы в этом вскоре убедитесь.

Есть один наиболее хитроумный способ справиться с проблемой мнемоник в Swing. Если внимательно изучить исходный текст базового UI-представителя всех элементов управления Swing BasicButtonUI, то выясняется, что поддержка мнемоник обеспечивается двумя событиями в карте входных событий (данные карты мы подробно обсуждали в главе 5) — это события с названиями «pressed» и «released». Данным событиям сопоставлены соответствующие действия из карты команд: первое действие переводит кнопку в нажатое состояние (когда пользователь нажимает и удерживает мнемонику), второе «отпускает» кнопку (когда пользователь отпускает клавиши). Можно получить карту входных команд любого элемента управления (используя метод getInputMap()), и беззастенчиво заменить входные события, соответствующие нажатиям мнемоник, своими, используя в качестве событий клавиатурные сокращения с кириллическими символами. Но есть одно препятствие: еще в главе 5 мы отметили, что система событий Java и класс событий от клавиатуры KeyEvent не поддерживают символы кириллицы для событий типы «нажать» и «отпустить» — кириллические символы можно только «напечатать», то есть они нажимаются и отпускаются одновременно. Это приводит к тому, что кириллические символы применять для мнемоник не имеет смысла — они не смогут работать как раздельное «нажатие» и «отпускание», а просто станут еще одним клавиатурным сокращением. Ситуация может быть исправлена только переписыванием события KeyEvent, а на это вряд ли стоит рассчитывать в ближайшее время.

Ну а пока, чтобы не обрекать приложение на беспомощность в случае отсутствия мыши, можно использовать такую «заплатку»:

```
JButton button = new JButton("Файл (F)");
button.setMnemonic('F');
```

Опытным пользователям может быть знакома такая форма записи с тех достопамятных времен, когда кириллическая раскладка клавиатуры плохо поддерживалась операционными системами, и для мнемоник всегда приходилось использовать латинские символы, даже если надпись была на другом языке. Выглядит это не очень элегантно, но здесь есть и свои плюсы — при переходе с одного языка на другой мнемоника не меняется (впрочем, утешение это слабое).

Заметьте, что реализованы мнемоники достаточно хорошо — когда вы нажимаете «волшебное» сочетание клавиш, кнопка переходит в «нажатое» состояние и остается в нем до тех пор, пока вы не отпустите клавишу с символом мнемоники (имитация щелчка мыши). Иногда «знает» о мнемонике и всплывающая подсказка кнопки — рядом с текстом подсказки она помещает сообщение о том, какое сочетание клавиш можно использовать для выбора, впрочем, это зависит от используемого внешнего вида и его версии.

В принципе, рассмотренных нами возможностей кнопок вполне достаточно для того, чтобы использовать их в настоящих приложениях. Однако создатели библиотеки на этом не ограничились и добавили в кнопки дополнительные возможности, которые способны в некоторых ситуациях значительно облегчить жизнь программиста.

## Интерфейс Action

Хороший стиль программирования пользовательских интерфейсов предусматривает четкое разделение интерфейса и деловой части программы. При таком подходе получаются более надежные программы, которые проще поддерживать и расширять. Как вы уже могли убедиться, Swing полностью поддерживает такое разделение с помощью своей модели событий. Однако при этом возникают и проблемы. При создании графического Java-приложения слушатели (обработчики) событий чаще всего располагаются во внутренних классах. Присоединить их не составляет труда — вы просто создаете экземпляр класса и передаете его компоненту. Однако ситуация в корне меняется, когда в приложении есть несколько компонентов, к которым необходимо присоединить одного и того же слушателя (чаще всего такими компонентами являются элементы управления, отвечающие за одно и то же действие, например кнопка на панели инструментов и команда меню). Решение кажется очевидным — создать для каждого компонента отдельный экземпляр слушателя событий, но это расточительно и более того, затрудняет управление программой (например, если действие недоступно, придется вручную выключать все связанные с этим действием компоненты).

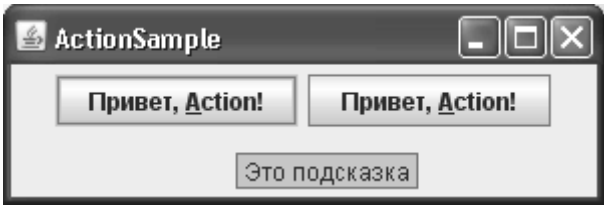
Для решения этих проблем в библиотеку Swing специально для элементов управления был включен новый интерфейс Action, расширяющий интерфейс ActionListener и позволяющий сосредоточить всю информацию о команде в одном месте. Теперь, создав всего один экземпляр класса, отвечающего за команду[5], вы сможете передать его всем нужным элементам управления, которые сами настроят свой внешний вид. Более того, в интерфейс Action встроена поддержка извещений об изменениях в команде — если вы что-то измените, все элементы управления автоматически узнают об этом. Рассмотрим небольшой пример:

```
// ActionSample.java
// Использование архитектуры Action
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ActionSample extends JFrame {
  public ActionSample() {
    super("ActionSample");
    setDefaultCloseOperation( EXIT_ON_CLOSE );
    // используем последовательное расположение
    setLayout(new FlowLayout());
    // создадим пару кнопок, выполняющих
    // одно действие
    Action action = new SimpleAction();
    JButton button1 = new JButton(action);
    JButton button2 = new JButton(action);
    add(button1);
    add(button2);
```

[5] Мы не случайно перешли от термина «слушатель» к термину «команда» — если вы знакомы с шаблонами проектирования, то поймете, что интерфейс Action представляет собой именно команду.

```
    // выводим окно на экран
    setSize(300, 100);
    setVisible(true);
  }
  // этот внутренний класс инкапсулирует нашу команду
  class SimpleAction extends AbstractAction {
    SimpleAction() {
      // установим параметры команды
      putValue(NAME, "Привет, Action!");
      putValue(SHORT_DESCRIPTION, "Это подсказка");
      putValue(MNEMONIC_KEY, new Integer('A'));
    }
    // в этом методе обрабатывается событие, как
    // и в прежнем методе ActionListener
    public void actionPerformed(ActionEvent e) {
      // можно выключить команду, не зная, к
      // каким компонентам она присоединена
      setEnabled(false);
      // изменим надпись
      putValue(NAME, "Прощай, Action!");
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new ActionSample(); } });
  }
}
```

В примере создаются две кнопки, которые будут выполнять одно и тоже действие, и соответственно обладать одним и тем же набором параметров. Вместо того чтобы дублировать код для настройки этих кнопок, создается один экземпляр команды SimpleAction, который и передается кнопкам. Остальное сделает за нас библиотека.




Гораздо интереснее, как устроен класс SimpleAction. Сам он не реализует довольно громоздкий интерфейс Action (так бы пришлось все писать «с нуля»), а задействует уже готовую заготовку, расширяя абстрактный класс AbstractAction. В данном абстрактном классе уже имеется поддержка слушателей PropertyChangeListener, которые оповещаются об изменениях в параме-

трах команды (кнопки выступают в роли таких слушателей, что и позволяет им быть в курсе всех изменений и вовремя отображать их на экране). Все сводится к настройке параметров команды (обычно это производится в конструкторе, как у нас в примере) и определению уже знакомого нам метода actionPerformed(), отвечающего за обработку событий.

Параметры команды хранятся в виде пар ключ-значение, где ключ — одна из строк, определенных в интерфейсе Action. Эта строка показывает, какой именно параметр команды хранится в паре. Для того чтобы изменить параметр, используется метод putValue(). Параметры, которые мы изменили в нашем примере, перечислены в табл. 9.4.

**Таблица 9.4.** Параметры для интерфейса Action

| Параметр | Описание |
|---|---|
| NAME | Этому ключу отвечает название команды, то есть надпись, которая будет выведена на кнопке или в меню |
| SHORT_DESCRIPTION | Ключ отвечает за краткое описание команды, появляющееся в виде всплывающей подсказки |
| MNEMONIC_KEY | Данный ключ позволяет указать мнемонику команды (заметьте, что для этого приходится создавать отдельный объект Integer для хранения кода клавиши) |

Конечно, это не полный список ключей — можно установить и несколько других параметров, в том числе значок и клавиатурное сокращение (что особенно полезно для меню), их полный список вы сможете найти в интерактивной документации Java.

Запустив программу, вы увидите две полностью настроенные и готовые к работе кнопки. Мы для этого не вызвали ни одного метода класса JButton, а просто указали в конструкторе, какое действие должна выполнять кнопка.

## *Элементы управления с двумя состояниями*

Кроме кнопок, рассмотренных нами в предыдущем разделе, в приложениях очень часто приходится использовать другие элементы управления, для которых характерно наличие двух устойчивых состояний. К ним относятся *флажки* (check boxes), *переключатели* (radio buttons) и *выключатели* (toggle buttons). Поддержка их также встроена в класс AbstractButton, и единственное отличие их от кнопок JButton состоит в том, что они могут находиться в одном из двух состояний и при смене состояний генерируют событие ItemEvent (которое кнопки игнорируют). Все остальные свойства, рассмотренные нами выше для кнопок JButton, верны и для них.

Поддержка двух состояний (выбрано — не выбрано), прежде всего, встроена в класс выключателя JToggleButton, от которого в свою очередь унаследованы классы флажков и переключателей. Так что начнем знакомство с этими компонентами именно с класса JToggleButton.

### Выключатели JToggleButton

Выключатель JToggleButton — довольно необычный элемент управления, и встретить его в простом пользовательском интерфейсе не так-то просто. Гораздо чаще он гостит на панелях инструментов, где с успехом заменяет флажки, которые из-за своего «непрезентабельного» вида иначе бы «портили» стройные ряды графических кнопок. Фактически, по виду это та же самая кнопка, только ее можно нажать, и она останется нажатой, а не «выпрыгнет» обратно. Можно использовать этот элемент управления и в обычном интерфейсе, например, когда нужно выбрать что-то из двух альтернатив, а применять

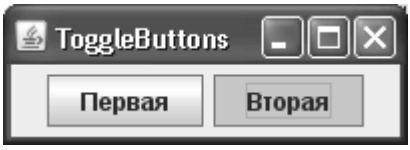
флажки или переключатели не совсем удобно (они могут занимать чересчур много места, особенно вместе с надписями).

Рассмотрим на примере, как работает выключатель, а потом обсудим остальные детали его поведения:

```
// ToggleButtons.java
// Использование выключателей JToggleButton
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ToggleButtons extends JFrame {
  public ToggleButtons() {
    super("ToggleButtons");
    setDefaultCloseOperation( EXIT_ON_CLOSE );
    // используем последовательное расположение
    setLayout(new FlowLayout());
    // создадим пару кнопок JToggleButton
    button1 = new JToggleButton("Первая", true);
    button2 = new JToggleButton("Вторая", false);
    // добавим слушатель события о смене состояния
    button2.addItemListener(new ItemListener() {
      public void itemStateChanged(ItemEvent e) {
        button1.setSelected(
            ! button2.isSelected());
      }
    });
    add(button1);
    add(button2);
    // выводим окно на экран
    pack();
    setVisible(true);
  }
  // ссылки на используемые кнопки
  private JToggleButton button1, button2;
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new ToggleButtons(); } });
  }
}
```

Здесь мы создаем два выключателя, один в нажатом состоянии, а другой в обычном (заметьте, что специальный конструктор сразу позволяет указать, нажат выключатель

или нет). Чтобы продемонстрировать, что выключатели относятся к событию ItemEvent с большим вниманием, чем кнопки, ко второму выключателю был привязан слушатель этого события, который при смене состояния второго выключателя должен изменять состояние первого на противоположное. Впрочем, с таким же успехом можно было использовать для наблюдения за сменой состояний слушатель ActionListener — нельзя ведь сменить состояние кнопки, не нажав ее. Однако для ясности и лучшей читабельности кода все же лучше использовать здесь слушатель ItemListener, который послужит еще и индикатором того, что нас интересует именно состояние кнопки, а не факт ее нажатия.




Запустив программу с примером, вы увидите, как все работает. Согласитесь, что чаще всего (если не всегда) эти элементы вы встречали на панелях инструментов (вспомните любой текстовый редактор и кнопки, которые делают символы полужирными, курсивными или подчеркнутыми). После запуска примера попробуйте сделать так, чтобы обе кнопки одновременно были отжаты. Если не знать, что это выключатели, можно легко спутать их с обычными кнопками, а это для хорошего интерфейса неприемлемо. Поэтому выключатели JToggleButton если и стоит использовать вне панелей инструментов, то только в группах из нескольких выключателей, где может быть выбрана только одна кнопка (аналог группы переключателей). Специально для таких групп в Swing был включен класс ButtonGroup, который мы сейчас и рассмотрим.

## Группы элементов управления ButtonGroup

Довольно часто возникают ситуации, когда пользователя необходимо поставить перед фактом: если и можно что-то выбрать, то только один вариант из множества. Непосредственное использование нескольких выключателей не даст желаемого эффекта — они будут менять свое состояние независимо друг от друга. Именно для таких ситуаций в библиотеку Swing и был включен класс ButtonGroup — он связывает несколько элементов управления в логическую группу, в которой выбранным может быть только один из них. При выборе пользователем другого элемента управления класс ButtonGroup позаботится о том, чтобы выбранный прежде элемент вернулся в исходное состояние. Проиллюстрирует сказанное пример:

```
// ButtonGroupUse.java
// Класс ButtonGroup помогает обеспечить
// эксклюзивный выбор
import javax.swing.*;
import com.porty.swing.BoxLayoutUtils;
import java.awt.*;

public class ButtonGroupUse extends JFrame {
  public ButtonGroupUse() {
    super("ButtonGroupUse");
    setDefaultCloseOperation( EXIT_ON_CLOSE );
    // создадим горизонтальную панель
```

```
        // с блочным расположением
        JPanel bh = BoxLayoutUtils.createHorizontalPanel();
        // надпись и отступ
        bh.add(new JLabel("Что Вы предпочитаете:"));
        bh.add(Box.createHorizontalStrut(12));
        // несколько выключателей JToggleButton
        JToggleButton b1 = new JToggleButton("Чай", true);
        JToggleButton b2 = new JToggleButton("Кофе");
        JToggleButton b3 = new JToggleButton("Лимонад");
        // добавим все кнопки в группу ButtonGroup
        ButtonGroup bg = new ButtonGroup();
        bg.add(b1);
        bg.add(b2);
        bg.add(b3);
        // добавим все кнопки в контейнер, учтем при
        // этом рекомендации интерфейса "Metal"
        bh.add(b1);
        bh.add(Box.createHorizontalStrut(2));
        bh.add(b2);
        bh.add(Box.createHorizontalStrut(2));
        bh.add(b3);
        getContentPane().add(bh);
        // выводим окно на экран
        pack();
        setVisible(true);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(
            new Runnable() {
              public void run() { new ButtonGroupUse(); } });
    }
}
```

В этом примере создаются несколько выключателей JToggleButton, среди которых нужно делать эксклюзивный выбор. С этим справляется класс ButtonGroup, и все, что для этого требуется, — включить каждый из выключателей в логическую группу методом add(). Если при добавлении класс ButtonGroup обнаружит, что выбрано несколько выключателей, то он оставит выбранным тот, который был включен в группу позже всех. Справедливости ради можно сказать, что название метода add() не совсем к месту — уж очень похожи фрагменты кода, создающие логическую группу и добавляющие элементы управления в контейнер. Постарайтесь не запутаться. Более того, сам код, который требуется написать для добавления кнопок в группу, весьма похож на спагетти, нет и намека на один метод с переменным количеством параметров, который позволил бы сходу добавить в группу все кноп-

ки, не говоря уже о такой роскоши, как возможность указать группу прямо в конструкторе кнопки. Похоже, что у разработчиков JDK иногда не хватает времени на маленькие радости жизни программистов, или они излишне надеются на визуальные инструменты. Новые версии JDK так и оставляют такие маленькие неприятности без внимания.


Отметьте, как созданные выключатели размещаются в контейнере: для создания панели с блочным расположением использовался вспомогательный класс (подробности см. в главе 7). Во-первых, они располагаются по горизонтали, потому что по вертикали в таких ситуациях всегда располагают описываемые далее переключатели (и, по совести говоря, они предпочтительнее, а созданный нами код может пригодиться, разве что, в условиях жесткой экономии места в контейнере). Во-вторых, в соответствие с рекомендациями Sun для внешнего вида Metal в таких ситуациях требуется разделять элементы управления пространством в 2 пиксела, что, согласитесь, довольно необычно. Заодно мы попрактиковались в написании интерфейса, соответствующего стандартам.

Остальное просто — запустите программу с примером, и вы в этом убедитесь. Все, что мы узнали о выключателях JToggleButton, нам еще пригодится при рассмотрении панелей инструментов JToolBar, а класс ButtonGroup практически неразлучен с переключателями JRadioButton, с которыми мы сейчас познакомимся поближе.

## Переключатели JRadioButton

Как мы уже отметили, объединение выключателей в группу — явление довольно редкое, и использовать такой подход следует там, где просто катастрофически не хватает места в контейнере. В подавляющем же большинстве ситуаций для реализации выбора «один из многих» применяются переключатели. Поодиночке в интерфейсе они практически не встречаются, обычно на эту роль выбирают флажки. В Swing переключатель реализован в классе JRadioButton, который напрямую унаследован от выключателя JToggleButton и отличается от него только внешним видом. Рассмотрим небольшой пример:

```
// RadioButtons.java
// Использование переключателей
import javax.swing.*;
import java.awt.*;

public class RadioButtons extends JFrame {
  public RadioButtons() {
    super("RadioButtons");
    setDefaultCloseOperation( EXIT_ON_CLOSE );
    // используем последовательное расположение
    setLayout(new FlowLayout());
    // отдельный переключатель
    JRadioButton r = new JRadioButton("Сам по себе");
    // группа связанных переключателей в своей
```

```
    // собственной панели
    JPanel panel = new JPanel(new GridLayout(0, 1, 0, 5));
    panel.setBorder(
        BorderFactory.createTitledBorder("Внешний вид"));
    ButtonGroup bg = new ButtonGroup();
    String[] names = { "Внешний вид Java",
        "MS Windows", "Aqua (Mac)" };
    for (String name : names) {
      JRadioButton radio = new JRadioButton(name);
      panel.add(radio);
      bg.add(radio);
    }
    // добавляем все в контейнер
    add(r);
    add(panel);
    // выводим окно на экран
    pack();
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new RadioButtons(); } });
  }
}
```

В этом примере в контейнер сначала добавляется ни с чем не связанный переключатель, просто для того чтобы показать, что и такое тоже возможно. Такой «одинокий» переключатель по поведению совсем не отличается от флажка, но для пользователя выглядит несколько непривычно, так что лучше в реальных интерфейсах подобные переключатели не использовать. После этого демонстрируется наиболее характерный способ использования переключателей: создается отдельная панель с заголовком, туда добавляется (по вертикали, разумеется) несколько переключателей, которые объединяются в логическую группу с помощью класса ButtonGroup, и все это выводится на экран. Встретить такую конструкцию можно практически в любом современном приложении.

Для создания панели в приложении использовался менеджер табличного расположения GridLayout. Табличное расположение позволило одновременно выровнять переключатели по размеру, поместить их в вертикальный столбик и задать рекомендуемое для связанных переключателей расстояние в 5 пикселов. Чтобы не дублировать при создании переключателей схожие фрагменты кода, переключатели были созданы в цикле из массива строк с их названиями. Для простоты никаких событий не обрабатывалось, но само собой, класс JRadioButton в этом плане ничем не отличается от своего родителя JToggleButton.

Таким образом, выключатели JToggleButton и переключатели JRadioButton фактически выполняют в интерфейсе одни и те же функции, выбор первого или второго варианта определяется положением компонентов. По горизонтали зачастую удобнее выключатели JToggleButton, потому что они немного компактнее, ну а по вертикали всегда используют переключатели JRadioButton. Нам осталось познакомиться с флажком JCheckBox — последним элементом управления, имеющим два состояния.

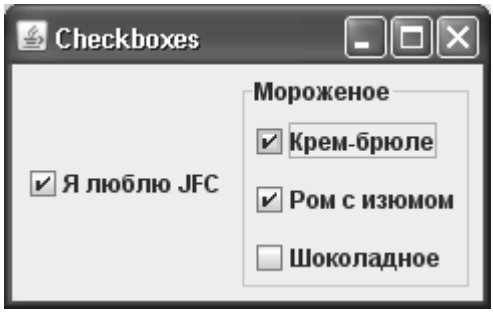
## Флажки JCheckBox

Флажки реализуются в библиотеке Swing классом JCheckBox. Они, так же как и переключатели, отличаются от выключателей только оригинальным внешним видом, навеянным всяческого рода анкетами и тестами, где после вопроса нужно отмечать галочкой несколько удовлетворяющих вас вариантов. Флажки используются там, где нужно предоставить пользователю возможность что-то включить или выключить. Они также группируются, но в отличие от двух предыдущих элементов управления, никогда не реализуют выбор «один из нескольких», а всегда позволяют выбрать несколько равноценных вариантов. Конечно, технически можно заставить флажки реализовать выбор «один из нескольких», однако это будет открытым вызовом устоявшимся стандартам. Рассмотрим пример, который является почти точной копией предыдущего примера:

```
// Checkboxes.java
// Использование флажков JCheckBox
import javax.swing.*;
import java.awt.*;

public class Checkboxes extends JFrame {
  public Checkboxes() {
    super("Checkboxes");
    setDefaultCloseOperation( EXIT_ON_CLOSE );
    // используем последовательное расположение
    setLayout(new FlowLayout());
    // отдельный флажок
    JCheckBox ch = new JCheckBox("Я люблю JFC", true);
    // группа связанных флажков в своей
    // собственной панели
    JPanel panel = new JPanel(new GridLayout(0, 1, 0, 5));
    panel.setBorder(
        BorderFactory.createTitledBorder("Мороженое"));
    String[] names = { "Крем-брюле",
        "Ром с изюмом", "Шоколадное" };
```

```
    for (String name : names) {
      JCheckBox check = new JCheckBox(name);
      panel.add(check);
    }
    // добавляем все в контейнер
    add(ch);
    add(panel);
    // выводим окно на экран
    pack();
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new Checkboxes(); } });
  }
}
```

Как видно, пример практически совпадает с примером переключателей, единственное отличие — мы не прибегли к логической группировке элементов управления. Все остальное, в том числе и расстояние между флажками, такое же. Применяйте флажки там, где пользователю нужно выбрать несколько возможных вариантов или что-то включить.




## Дополнительные значки

Как вы помните, кнопки Swing позволяют практически полностью поменять свой внешний вид и настроить его под свой вкус. Мы подробно рассмотрели все свойства, связанные с этим, еще в самом первом примере этой главы. Однако, теперь, когда мы изучили и кнопки с двумя состояниями, возникает вопрос, как кнопки, если мы меняем их внешний вид и применяем значки, будут показывать выбранное состояние, к примеру, те же флажки? Есть ли возможность повлиять и на это? Конечно, есть, и управляется это еще одним свойством с названием selectedIcon. Используя его, вы можете задать значок, который будет рисоваться на экране в том случае, если элемент выбран, вместо флажка или переключателя. Ну а когда элемент не выбран, рисуется стандартный значок, который задается уже знакомым нам свойством icon.

Возможность на лету и с легкостью перестроить внешний вид компонентов Swing иногда захватывает, но все же следует пользоваться ей с осторожностью. В противном случае есть риск сделать код программы «грязноватым», слишком сильно перемешав настройку внешнего вида с деловой логикой. К тому же не стоит забывать о подключаемом внешнем виде и UI-представителях. Написать их не так уж и трудно, а результатом будет полное отделение логики программы от внешнего вида и поведения. Случай, когда внешний вид ваших компонентов основан на изображениях и значках, особенно хорошо сочетается с настраиваемым внешним видом Synth, стандартным для Swing и JDK.

## *Резюме*

Благодаря элементам управления пользователь может немедленно изменить ход выполнения программы или что-либо выбрать. Позвольте ему это сделать: возможностей элементов управления Swing вам должно хватить на все случаи жизни.

# Глава 10. Меню и панели инструментов

Система меню — важнейший элемент пользовательского интерфейса современных приложений. Именно в меню находится полный перечень команд приложения, который к тому же организован и разбит на группы. Благодаря этому ознакомление с возможностями приложения проходит быстрее, и начать работу, имея под рукой хорошо организованную и интуитивно понятную систему команд, гораздо проще. Так что можно смело утверждать, что процесс создания меню является одним из главных этапов на пути разработки хорошего приложения. Меню — это «лицо вашего приложения», и именно по нему пользователь составит первое впечатление о возможностях и качестве приложения.

К «авангарду» пользовательского интерфейса вашего приложения относятся и панели инструментов. Панель инструментов представляет собой компактный набор компонентов, чаще всего кнопок (хотя в ней могут содержаться любые компоненты), которые отвечают за наиболее важные и часто используемые функции вашего приложения. Панель инструментов позволяет пользователю совершать нужные действия гораздо быстрее, так что эффективность его работы с вашим приложением повышается, а это первейший фактор при создании пользовательских интерфейсов. Продуманные удобные панели инструментов наряду с удобной системой меню способны мгновенно сказать пользователю все о высоком качестве вашего приложения.

## *Меню*

Хотя материал, посвященный меню, находится в отдельной главе, он вполне мог бы располагаться в главе 9, описывающей элементы управления. И это совсем не случайно. Ведь меню — это не что иное, как несколько кнопок, собранных в список и при необходимости вызываемых на экран. Данный факт не укрылся от разработчиков библиотеки Swing, и в результате элементы меню действительно оказались кнопками, унаследованными от хорошо знакомого нам класса AbstractButton, они отличаются от кнопок, флажков и переключателей только внешним видом да небольшими изменениями, позволяющими им находиться в меню. Поэтому, несмотря на то, что классы элементов меню имеют другие названия, никто не отнимет у вас знаний, полученных в предыдущих главах, — вы уже знаете, как изменять внешний вид элементов меню, назначать им мнемоники и обрабатывать события (хотя мы еще не видели ни одного примера). Чтобы ускорить процесс знакомства с системой меню библиотеки Swing, рассмотрим своеобразную таблицу соответствия уже рассмотренных нами элементов управления и элементов меню (табл. 10.1).

**Таблица 10.1.** Соответствие элементов управления и элементов меню

| Элементы управления | Элементы меню |
|---|---|
| Кнопка JButton | «Обычный» элемент меню JMenuItem |
| Переключатель JRadioButton | Элемент-переключатель JRadioButtonMenuItem |
| Флажок JCheckBox | Элемент-флажок JCheckBoxMenuItem |

Все, что рассказывалось относительно элементов управления, справедливо и для элементов меню. Вы можете заметить, что в системе меню нет эквивалента выключателя JToggleButton, и это не удивительно. При навигации по системе меню выбранный элемент меню становится «вжатым», а для выключателя такое поведение приведет к неоднозначному восприятию его состояния (в принципе, выключатель в меню и не нужен — с его функциями справляются флажки и переключатели).

## Создание системы меню

Давайте рассмотрим пример, который разъяснит, как создается простейшая система меню, как она появляется на экране, и что для этого нужно сделать:

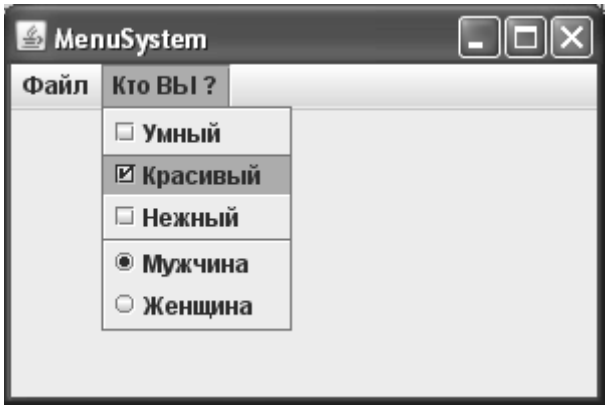
```
// MenuSystem.java
// Создание системы меню в Swing
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.*;

public class MenuSystem extends JFrame {
  public MenuSystem() {
    super("MenuSystem");
    setDefaultCloseOperation( EXIT_ON_CLOSE );
    // создаем строку главного меню
    JMenuBar menuBar = new JMenuBar();
    // добавляем в нее выпадающие меню
    menuBar.add(createFileMenu());
    menuBar.add(createWhoMenu());
    // и устанавливаем ее в качестве
    // меню нашего окна
    setJMenuBar(menuBar);
    // выводим окно на экран
    setSize(300, 200);
    setVisible(true);
  }
  // создает меню "Файл"
  private JMenu createFileMenu() {
    // создадим выпадающее меню, которое будет
    // содержать обычные пункты меню
    JMenu file = new JMenu("Файл");
    // пункт меню (со значком)
    JMenuItem open =
        new JMenuItem("Открыть",
            new ImageIcon("images/open16.gif"));
    // пункт меню из команды
```

```
    JMenuItem exit = new JMenuItem(new ExitAction());
    // добавим все в меню
    file.add(open);
    // разделитель
    file.addSeparator();
    file.add(exit);
    return file;
  }
  // создадим забавное меню
  private JMenu createWhoMenu() {
    // создадим выпадающее меню
    JMenu who = new JMenu("Кто ВЫ ?");
    // меню-флажки
    JCheckBoxMenuItem clever =
        new JCheckBoxMenuItem("Умный");
    JCheckBoxMenuItem smart =
        new JCheckBoxMenuItem("Красивый");
    JCheckBoxMenuItem tender =
        new JCheckBoxMenuItem("Нежный");
    // меню-переключатели
    JRadioButtonMenuItem male =
        new JRadioButtonMenuItem("Мужчина");
    JRadioButtonMenuItem female =
        new JRadioButtonMenuItem("Женщина");
    // организуем переключатели в логическую группу
    ButtonGroup bg = new ButtonGroup();
    bg.add(male); bg.add(female);
    // добавим все в меню
    who.add(clever);
    who.add(smart);
    who.add(tender);
    // разделитель можно создать и явно
    who.add( new JSeparator());
    who.add(male);
    who.add(female);
    return who;
  }
  // команда выхода из приложения
  class ExitAction extends AbstractAction {
    ExitAction() {
      putValue(NAME, "Выход");
```

```
    }
    public void actionPerformed(ActionEvent e) {
      System.exit(0);
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
     new Runnable() {
      public void run() { new MenuSystem(); } });
  }
}
```

Итак, в этом примере создается относительно простая система меню. Как мы уже отмечали, элементы меню — это фактически кнопки, собранные в список. Списки элементов меню, или правильнее *выпадающие меню* (drop-down menus), реализованы в Swing классом JMenu. Именно они создаются методами createFileMenu() и createWhoMenu(), внутри которых в выпадающие меню методом add() добавляются разнообразные элементы меню, в том числе элементы-флажки и элементы-переключатели. После этого остается вывести созданные выпадающие меню на экран, для чего служит так называемая *строка меню* (menu bar), создать которую позволяет класс JMenuBar. В нем также есть метод add(), только не для элементов меню, а для выпадающих меню JMenu. Разместив все выпадающие меню своего приложения в строке меню, вы можете поместить ее в окно, вызвав предназначенный для этого метод setJMenuBar().

Что касается элементов меню, то это те же кнопки, флажки и переключатели. Вы можете указывать для них названия, значки и мнемоники, создавать их на основе интерфейса Action, и все это проиллюстрирует наш пример. Единственное новшество в примере — это *разделитель* (separator), который позволяет организовать смысловые группы в выпадающих меню. В примере показано, как его создать, а подробнее мы поговорим о нем чуть позже.




Что же, система меню создается действительно просто, и ничего экстраординарного в этом процессе нет. Однако меню Swing способны и на многое другое, поэтому давайте рассмотрим их подробнее.

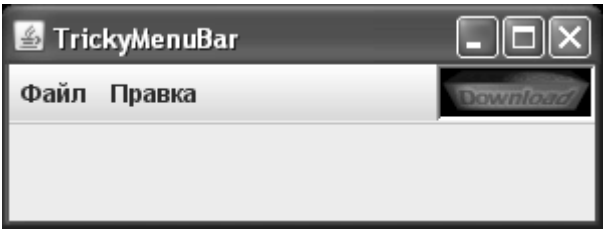
## Строка меню JMenuBar

Главное, что нужно помнить при создании строки меню JMenuBar в своей программе, — это самый обыкновенный контейнер, ничем не отличающийся от панели JPanel и обладающий теми же самыми свойствами. Не нужно думать, что строка меню рождена только для работы с выпадающими меню. Вы можете смело добавлять в нее всевозможные компоненты, например надписи со значками или раскрывающиеся списки. Беззастенчиво заглянув внутрь класса JMenuBar, можно увидеть, что даже менеджер расположения у строки меню не какой-то экзотический, а прекрасно знакомый нам менеджер BoxLayout с расположением компонентов по горизонтали. Так что создание с виду совершенно новаторского меню оказывается весьма простым делом. Например, можно сделать со строкой меню следующее:

```
// TrickyMenuBar.java
// Полоска меню JMenuBar может многое
import javax.swing.*;
import java.awt.*;

public class TrickyMenuBar extends JFrame {
  public TrickyMenuBar() {
    super("TrickyMenuBar");
    setDefaultCloseOperation( EXIT_ON_CLOSE );
    // создаем главную полоску меню
    JMenuBar menuBar = new JMenuBar();
    // добавляем в нее выпадающие меню
    menuBar.add(new JMenu("Файл"));
    menuBar.add(new JMenu("Правка"));
    // мы знаем, что используется блочное
    // расположение, так что заполнитель
    // вполне уместен
    menuBar.add(Box.createHorizontalGlue());
    // теперь поместим в полоску меню
    // не выпадающее меню, а надпись со значком
    JLabel icon = new JLabel(
        new ImageIcon("images/download.gif"));
    icon.setBorder(
        BorderFactory.createLoweredBevelBorder());
    menuBar.add(icon);
    // помещаем меню в наше окно
    setJMenuBar(menuBar);
    // выводим окно на экран
    setSize(300, 200);
    setVisible(true);
  }
  public static void main(String[] args) {
```

```
        SwingUtilities.invokeLater(
         new Runnable() {
          public void run() { new TrickyMenuBar(); } });
    }
}
```

В примере мы сначала добавили в строку меню традиционные выпадающие меню (они, правда, для экономии места были оставлены пустыми), а затем, предварительно использовав заполнитель, поместили в строку меню самую обычную надпись со значком, установив для нее тисненую рамку. Мы специально использовали анимированный GIF-файл, чтобы пример произвел еще более сильный эффект. Не правда ли, напоминает современные браузеры? И все, что мы сделали для этого, — это добавили два компонента.




Отметьте также, что размещение строки меню в окне осуществляется не с помощью обычного метода add() и менеджера расположения, а с помощью специального метода setJMenuBar(). Этот метод на самом деле принадлежит не классу окна JFrame, а корневой панели JRootPane, которая и заботится о том, чтобы строка меню занимала подобающее ей положение на вершине окна, оставляя вам все пространство в панели содержимого.

Таким образом, знание того факта, что строка меню — это обычный контейнер, позволяет творить со строкой меню просто невероятные вещи. Это хороший пример того, как важна при разработке библиотеки ее прозрачность — никто не скрывает от вас тонкости реализации, и то же можно сказать обо всех компонентах. Знание основ устройства библиотеки — гораздо более мощное оружие, чем может показаться на первый взгляд.

Впрочем, не стоит сильно увлекаться созданием абсолютно нового облика меню в вашей программе, иначе пользователям трудно будет быстро освоиться с ней. Все-таки вид и поведение меню уже устоялись, и в привычном облике работать с меню проще.

## Выпадающие меню JMenu и разделители JSeparator

Как мы уже отмечали, для упорядочения элементов меню служат выпадающие меню JMenu. Интересно, что класс JMenu унаследован от обычного элемента меню JMenuItem, и его функция заключается только в том, чтобы при щелчке мышью показать в нужном месте экрана всплывающее меню, в котором и содержатся все добавленные элементы меню. Ничего хитрого в этом классе нет, и сказать о нем можно только то, что он позволяет организовывать вложение меню любой сложности. Для этого вы просто вкладываете выпадающие меню друг в друга, и получаете перечни элементов меню произвольной длины (каскадные меню).

Для того чтобы организовать в меню несколько групп элементов, используются разделители JSeparator. Мы можете добавлять их в выпадающие меню, создавая напрямую, или же использовать для этого специальный метод addSeparator(). Кроме того разделите-

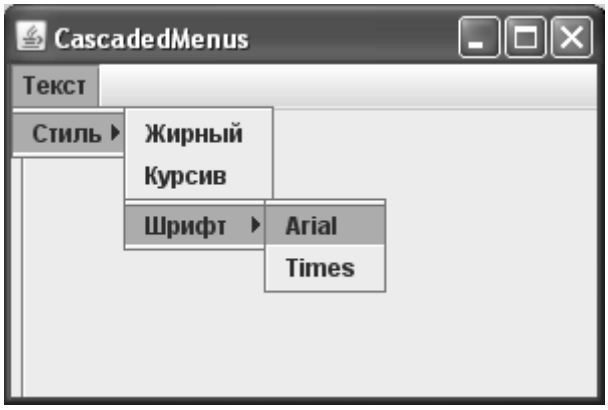
ли можно применять и вне меню в качестве разделительных линий, в том числе вертикальных. Рассмотрим пример:

```
// CascadedMenus.java
// Создание вложенных меню любой сложности
import javax.swing.*;
import java.awt.*;

public class CascadedMenus extends JFrame {
  public CascadedMenus() {
    super("CascadedMenus");
    setDefaultCloseOperation( EXIT_ON_CLOSE );
    // создаем строку главного меню
    JMenuBar menuBar = new JMenuBar();
    // создаем выпадающее меню
    JMenu text = new JMenu("Текст");
    // и несколько вложенных меню
    JMenu style = new JMenu("Стиль");
    JMenuItem bold = new JMenuItem("Жирный");
    JMenuItem italic = new JMenuItem("Курсив");
    JMenu font = new JMenu("Шрифт");
    JMenuItem arial = new JMenuItem("Arial");
    JMenuItem times = new JMenuItem("Times");
    font.add(arial);
    font.add(times);
    // размещаем все в нужном порядке
    style.add(bold);
    style.add(italic);
    style.addSeparator();
    style.add(font);
    text.add(style);
    menuBar.add(text);
    // помещаем меню в окно
    setJMenuBar(menuBar);
    // разделитель может быть полезен не только в меню
    ((JComponent)getContentPane()).setBorder(
      BorderFactory.createEmptyBorder(0, 5, 0, 0));
    add(new JSeparator(SwingConstants.VERTICAL), "West");
    // выводим окно на экран
    setSize(300, 200);
    setVisible(true);
  }
```

```
    public static void main(String[] args) {
      SwingUtilities.invokeLater(
       new Runnable() {
        public void run() { new CascadedMenus(); } });
    }
}
```

В этом примере мы создали весьма правдоподобное меню, которое содержит команды для работы с простым текстовым редактором. Для получения каскадных меню несколько выпадающих меню были просто вложены друг в друга в нужном порядке. Заодно мы еще раз использовали разделитель, который акцентировал различие между командами для выбора стиля текста и для выбора шрифта для него.

Посмотрите, как можно использовать разделитель вне меню. В примере мы создали вертикальный разделитель (для этого есть специальный конструктор), который добавили на запад своей панели содержимого. Получилось весьма неплохое украшение для возможного интерфейса. Правда, для того чтобы в печатной книге его было лучше видно, мы дополнительно отделили разделитель от границ окна с помощью рамки с пустым пространством EmptyBorder.




В примере была создана очень простая система меню, обычно она в несколько раз больше и сложнее. Но даже для такой простой системы код получился немного неряшливым (чего стоят одни только добавления одного в другого, а другого в третье). Действительно, инструкции создания меню в коде сильно «загрязняют» последний, это хорошо известный факт, что «декларативный» стиль описания объекта не слишком элегантно выглядит через призму вызовов объектно-ориентированных методов. Мы ведь просто хотим указать структуру меню, а не вызываем никаких действий. Чуть позже мы попытаемся решить эту проблему.

## Клавиатурные сокращения и мнемоники

Если часто работаешь с одним и тем же приложением, и раз за разом выполняешь одни и те же операции, поневоле хочется повысить свою производительность и не тратить время на то, что делалось уже много раз. Именно для этого в графических приложениях и появились *клавиатурные сокращения*, или *клавиши быстрого доступа* (accelerators) и *мнемоники* (mnemonics). Опытные пользователи высоко ценят возможность выполнить операцию, требующую в обычных условиях долгих «блуж-

даний» по системе меню, просто нажав комбинацию из двух-трех клавиш. К тому же, как мы уже отмечали, еще больше повышают привлекательность вашего приложения мнемоники, помогая пользователям компьютеров без мыши и пользователям с ограниченными возможностями.

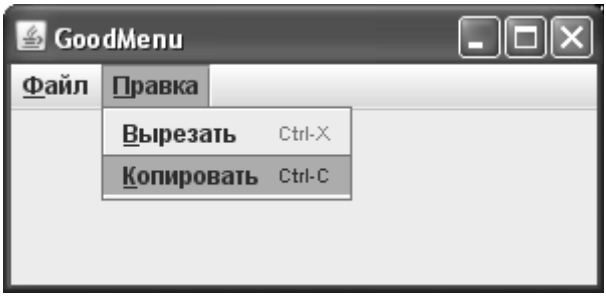
Если с мнемониками все более или менее ясно (мы уже встречались с ними в главах 8 и 9), то клавиатурные сокращения — привилегия элементов меню, где они чрезвычайно полезны. Клавиатурное сокращение — это произвольная комбинация всевозможных управляющих клавиш (Shift, Ctrl, Alt) с клавишей некоторого латинского символа, нажатие которой эквивалентно выбору элемента меню, что освобождает пользователя от необходимости разыскивать этот элемент в системе меню.

Вообще говоря, как мнемоники, так и клавиши быстрого доступа очень важны для меню. В хорошем приложении абсолютно все элементы меню, а также выпадающие и вложенные меню должны обладать мнемониками, а наиболее часто используемые команды — уникальной клавиатурной комбинацией. Попробуем создать такую «образцовую» систему меню:

```
// GoodMenu.java
// Клавиатурные комбинации и мнемоники для меню Swing
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class GoodMenu extends JFrame {
  public GoodMenu() {
    super("GoodMenu");
    setDefaultCloseOperation( EXIT_ON_CLOSE );
    // создаем строку главного меню
    JMenuBar menuBar = new JMenuBar();
    // некоторые весьма часто встречающиеся
    // выпадающие меню
    menuBar.add(createFileMenu());
    menuBar.add(createEditMenu());
    // поместим меню в наше окно
    setJMenuBar(menuBar);
    // выводим окно на экран
    setSize(300, 200);
    setVisible(true);
  }
  // создает меню "Файл"
  private JMenu createFileMenu() {
    // выпадающее меню
    JMenu file = new JMenu("Файл");
    file.setMnemonic('Ф');
    // пункт меню "Открыть"
```

```
    JMenuItem open = new JMenuItem("Открыть");
    open.setMnemonic('O'); // русская буква
    // установим клавишу быстрого доступа (латинская буква)
    open.setAccelerator(
      KeyStroke.getKeyStroke('O', KeyEvent.CTRL_MASK));
    // пункт меню "Сохранить"
    JMenuItem save = new JMenuItem("Сохранить");
    save.setMnemonic('C');
    save.setAccelerator(
      KeyStroke.getKeyStroke('S', KeyEvent.CTRL_MASK));
    // добавим все в меню
    file.add(open);
    file.add(save);
    return file;
  }
  // создает меню "Правка"
  private JMenu createEditMenu() {
    // выпадающее меню
    JMenu edit = new JMenu("Правка");
    edit.setMnemonic('П');
    // пункт меню "Вырезать"
    JMenuItem cut = new JMenuItem("Вырезать");
    cut.setMnemonic('В');
    cut.setAccelerator(
      KeyStroke.getKeyStroke('X', KeyEvent.CTRL_MASK));
    // пункт меню "Копировать"
    JMenuItem copy = new JMenuItem("Копировать");
    copy.setMnemonic('К');
    // клавишу быстрого доступа можно создать и так
    copy.setAccelerator(KeyStroke.getKeyStroke("ctrl C"));
    // готово
    edit.add(cut);
    edit.add(copy);
    return edit;
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
     new Runnable() {
      public void run() { new GoodMenu(); } });
  }
}
```

В этом примере создается небольшое меню с пунктами Файл и Правка, которые настолько часто встречаются, что стали уже стандартными. Для них мы, не жалея времени и усилий, сделали все необходимые настройки, а именно установили мнемоники и клавиатурные комбинации.




Вы можете видеть, что мнемоники устанавливаются хорошо знакомым нам методом setMnemonic(), а вот клавиши быстрого доступа для элементов меню создаются с помощью класса KeyStroke. В этом классе определено несколько перегруженных статических методов вида getKeyStroke(), которые возвращают экземпляр нужной клавиатурной комбинации. В примере демонстрируются два варианта этого метода: один требует указания символа для клавиши быстрого доступа и набора управляющих клавиш (вы можете указать несколько управляющих клавиш, используя соответствующие константы из класса KeyEvent, связав их операцией поразрядного «ИЛИ»), а другой позволяет получить нужное клавиатурное сокращение с помощью строки, в которой сначала указывают псевдонимы управляющих клавиш, а затем символ клавиши быстрого доступа.

Запустив программу с примером, вы столкнетесь с досадной проблемой мнемоник. Если вы уже находитесь в меню, то есть оно получило фокус ввода, мнемоники с русскими символами работают. Но если вы хотите получить доступ к одному из выпадающих меню первый раз (меню не обладает фокусом ввода), мнемоники не работают (меню файл не активируется с помощью клавиш Alt-Ф). Какие бы сочетания клавиш мы не нажимали, это не поможет, и меню так и не получит фокус ввода. Это – большой недостаток, и избежать его можно, либо включив в русские названия элементов меню латинские буквы, либо заранее предупредив пользователя о том, что строка меню в Swing получает фокус ввода с помощью клавиши F10 (по сути, она считается стандартной клавишей перехода на строку меню, но часто ли ее используют? Я не припомню, чтобы мне приходилось нажимать ее вместо мнемоники).

Но, как бы там ни было, клавиши быстрого доступа работают исправно, и довольно значительно повышают привлекательность приложения, так что не забывайте про них. Главное, помните о том, что в мире интерфейсов устоялись свои стандарты, пусть и молодые, и не стоит заново изобретать альтернативу старым добрым Control-S и Control-C, пусть они и набили уже всем оскомину. Даже на русском языке эти сочетания остаются собой. Для всех важных и частых операций своего приложения продумывайте наличие эффективных клавиш быстрого доступа[1].

[1] Программистам прекрасно знакомы клавиши быстрого доступа и их значение. Ни один потрясающей красоты дизайн не заменит возможности быстрым нажатием провернуть одну из множества нужных в данный момент операций, не отвлекаясь на их поиск в меню. Иногда список быстрых клавиш не умещается на нескольких страницах руководства средства разработки, такого как IDEA.

## Всплывающие меню JPopupMenu

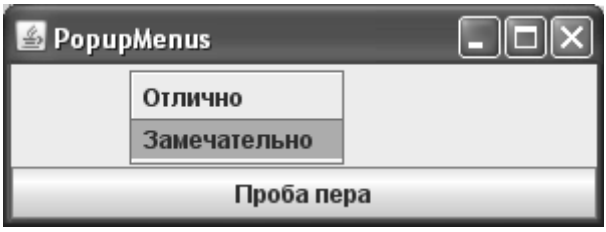
Меню не обязательно относятся ко всему приложению, они могут пригодиться и для частей этого приложения (например, для текстового поля в редакторе). В таком случае меню появляется на экране только по желанию пользователя, поэтому их называют *всплывающими* (popup menu), или *контекстными* (context-sensitive menu). Пользователь вызывает это меню, чтобы получить список команд для того объекта приложения, с которым он работает, не тратя время на поиск этих команд в главном меню.

Всплывающие меню в Swing реализованы классом JPopupMenu. На самом деле вы уже видели этот класс «в деле» — именно он применяется в выпадающем меню JMenu для вывода его элементов. Но использовать его можно и отдельно, в чем мы сейчас убедимся:

```
// PopupMenus.java
// Работа с всплывающими меню
import javax.swing.*;
import java.awt.*;

public class PopupMenus extends JFrame {
  public PopupMenus() {
    super("PopupMenus");
    setDefaultCloseOperation( EXIT_ON_CLOSE );
    // получаем всплывающее меню
    JPopupMenu popup = createPopupMenu();
    // и привязываем к нашей панели содержимого
    ((JComponent)getContentPane()).
        setComponentPopupMenu(popup);
    // "прозрачная" для меню кнопка
    JButton button = new JButton("Проба пера");
    button.setInheritsPopupMenu(true);
    add(button, "South");
    // выводим окно на экран
    setSize(300, 200);
    setVisible(true);
  }
  // создаем наше всплывающее меню
  private JPopupMenu createPopupMenu() {
    // создаем само всплывающее меню
    JPopupMenu pm = new JPopupMenu();
    // создаем его пункты
    JMenuItem good = new JMenuItem("Отлично");
    JMenuItem excellent = new JMenuItem("Замечательно");
    // и добавдяем все тем же методом add()
    pm.add(good);
```

```
        pm.add(excellent);
        return pm;
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(
         new Runnable() {
          public void run() { new PopupMenus(); } });
    }
}
```

В этом примере мы создаем простейшее всплывающее меню из двух элементов; это делает метод createPopupMenu(). Как видно, все действия аналогичны тем, что мы делали при создании выпадающих меню JMenu. После того как всплывающее меню подготовлено к работе, необходимо определить, где и при каком условии нужно вывести его на экран. Так как на различных платформах способ вызова всплывающего меню может быть разным, момент, когда это необходимо сделать, определяет в Swing текущий менеджер внешнего вида и поведения[2]. Ну а само меню задается для конкретного компонента, методом setComponentPopupMenu(). В примере мы задаем меню сразу же для всей панели содержимого нашего окна. Данный метод находится в базовом классе библиотеки JComponent, и таким образом доступен для всех компонентов. Есть еще один интересный метод для всплывающих меню с названием setInheritsPopupMenu(). Это булево свойство, по умолчанию отключенное (равное false). Означает оно, что компонент будет унаследовать всплывающее меню от своего компонента-предка (как правило контейнера). В нашем примере кнопка унаследует всплывающее меню от панели содержимого, в чем вы сможете легко убедиться, запустив программу.




Для вывода меню на экран чаще всего поступают так, как сделано у нас в примере. Однако есть и более низкоуровневый способ – создать в нужном компоненте слушателя событий от мыши, и при щелчке (обычно правой кнопкой) вывести всплывающее меню в месте этого щелчка.

Для вывода меню на экран в класс JPopupMenu специально был добавлен перегруженный метод show(), который позволяет одновременно указать, где (в каких координатах) и в каком компоненте будет показано меню. Особенно это пригодится в сложных случаях, когда у компонента может быть разное меню в зависимости от точки, в которой меню было вызвано. В общем же случае определять, как меню появится на экране, лучше доверить менеджеру внешнего вида.

Реализовано всплывающее меню в Swing достаточно разумно. В обычной ситуации оно представляет собой легковесный компонент, унаследованный от базового класса

[2] Для большинства платформ это щелчок правой кнопкой мыши или сочетание клавиш Shift-F10. Интересно, что сочетание клавиш вызовет меню для компонента, обладающего фокусом ввода, а иногда требуется вызывать его и для не способных обладать фокусом компонентов.

JComponent, и для вывода на экран его просто делают видимым и добавляют в специально предназначенный для этого слой POPUP_LAYER многослойной панели JLayeredPane. Однако как только всплывающее меню начинает «вылезать» за границы окна, в котором оно должно быть показано, или используется в AWT-приложении, все меняется — всплывающее меню выводится в собственном отдельном тяжеловесном окне без рамки. Для программиста вся эта кухня остается незаметной, что, конечно, облегчает программирование. Впрочем, вы можете включиться в процесс и с помощью метода setLightweightPopupEnabled() рекомендовать для всплывающего меню тип компонента (легковесный или тяжеловесный). Иногда это может быть полезно.

## Загрузка меню из файлов XML

Во всех примерах с меню, что мы разобрали, можно было заметить весьма неприятную тенденцию: создание нескольких полноценных пунктов меню (со значками, мнемониками, акселераторами и обработчиками событий) сильно раздувало код наших программ. Можно себе представить, насколько сильно разрастется код программы, обладающей более или менее большими возможностями, и соответственно большой системой меню. Выходом из этой ситуации может стать использование архитектуры Action, встроенной во все кнопки библиотеки, и такое решение будет *действительно* хорошим, особенно если эти же команды будут использоваться и для создания кнопок на панели инструментов, и для возможных контекстных меню. Но и у этого подхода есть свои недостатки — для каждой команды придется писать довольно объемный класс, и таких классов-близнецов может быть очень много.

С другой стороны, существует множество визуальных инструментов для построения пользовательского интерфейса, и в них почти всегда есть средства для создания меню произвольной сложности. Хотя после их использования качество и чистота кода оставляют желать лучшего, а стоимость поддержки программы возрастает (а иногда вообще невозможна без примененного визуального средства), в случае с меню они существенно сэкономят ваше время.

Наконец, можно просто вспомнить о том, что система меню представляет собой несложную иерархическую структуру, в которой в качестве узлов выступают выпадающие меню JMenu, листьями являются пункты меню, такие как JMenuItem и JRadioButtonMenuItem, а корнем всей системы является строка меню JMenuBar. У каждого члена этой иерархии есть некоторые атрибуты: название, значок, текст и т.п. Не правда ли, описанная структура так и просится в файл формата XML? Именно формат XML предназначен для описания произвольных иерархических структур, узлы которых могут иметь набор некоторых атрибутов. Придумав подходящее описание XML для меню Swing, а затем написав специальный инструмент для загрузки, мы сможем мгновенно загружать самые запутанные и сложные меню. Кстати, подобный подход часто используется во многих графических системах: меню описывается на некотором текстовом языке и помещается в «ресурсы» программы, откуда загрузить его не составляет особого труда.

Прежде всего необходимо определить то, как будет выглядеть файл XML с описанием системы меню. Наиболее логичным смотрится следующий вариант:

```
<?xml version="1.0" encoding="UTF-8"?>

<menubar name="mainMenu">
 <menu name="file" text="Файл" mnemonic="Ф">
     <menuitem name="create" text="Создать" mnemonic="Н"
         accelerator="control N"/>
     <menuitem name="open" text="Открыть" mnemonic="О"
```

```
        enabled="false"/>
    <menuitem name="close" text="Закрыть" mnemonic="З"
        enabled="false"/>
    <menuitem name="separator"/>
    <menu name="print" text="Печать" mnemonic="П">
        <menuitem name="preview"
            text="Предварительный просмотр" mnemonic="р"/>
    </menu>
    <menuitem name="separator"/>
    <menuitem name="exit" label="Выход" mnemonic="В"
        accelerator="alt X"/>
  </menu>
</menubar>
```

После стандартного заголовка файла XML следует описание элементов нашей системы меню. Корнем системы, как и следует ожидать, стал элемент с названием menubar: он будет описывать строку меню, в которой затем разместятся все выпадающие меню и пункты меню. У элемента menubar имеется только один атрибут name (имя), по этому имени мы затем сможем найти его[3]. Чтобы при поиске элементов меню не возникало ошибок, имена всех элементов должны быть уникальными. К строке меню присоединяются выпадающие меню: для XML это означает наличие элементов-потомков с названием menu. Как правило, для выпадающих меню настраивается два свойства: text (текст меню) и mnemonic (символ мнемоники). Данные свойства будут задаваться в виде атрибутов элемента menu. Выпадающие меню состоят из пунктов меню и, если это необходимо, из других выпадающих меню (это позволит организовать сложную иерархическую систему меню). Поэтому потомками элемента menu смогут быть элементы menuitem (пункт меню) или другие элементы menu. Больше всего атрибутов поддерживает элемент menuitem: для него часто задаются не только текст меню и символ мнемоники, но и сочетание клавиш быстрого доступа (accelerator)[4], а также то, доступен ли он в данный момент пользователю (enabled). Кроме того, в качестве особых элементов могут выступать разделители меню. Для них мы зарезервировали специальное имя separator. При написании файла XML, описывающего меню, не забывайте о том, что все элементы XML обязательно должны закрываться. Для системы меню это имеет дополнительное значение: пока вы не закроете элемент menu, все следующие за ним пункты меню будут попадать в описываемое им выпадающее меню.

Созданное нами описание системы меню на языке XML интуитивно понятно и позволяет мгновенно описать сложную систему меню, «на лету» задавая наиболее важные свойства пунктов меню. Теперь нам необходимо написать инструмент, распознающий подобный файл XML и создающий на его основе меню Swing. Стандартные библиотеки Java содержат все необходимые инструменты для работы с XML, в том числе и упрощенный интерфейс SAX (Simple API for XML). Именно этот интерфейс мы применим для распознавания системы меню. Вот что у нас получится:

```
// com/porty/swing/XMLMenuLoader.java
```

[3] Здесь может показаться, что имя для строки меню ни к чему, ведь она в нашем файле описана одна. Однако чуть позже мы увидим, что добавить можно сколь угодно строк меню, так что имя все же нужно.

[4] Нам не составит труда распознать строку, описывающую клавиши быстрого доступа: в классе KeyStroke имеется перегруженный метод getKeyStroke(), позволяющий получить клавиатурное сокращение на основе строки, составленной по определенным правилам. Описание этих правил вы сможете найти в интерактивной документации класса KeyStroke.

```
// Инструмент для загрузки меню из файла XML
package com.porty.swing;

import javax.swing.*;
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import java.io.*;
import javax.xml.parsers.*;
import java.awt.event.*;
import java.util.*;

public class XMLMenuLoader {
  // источник данных XML
  private InputSource source;
  // анализатор XML
  private SAXParser parser;
  // обработчик XML
  private DefaultHandler documentHandler;
  // хранилище для всех частей системы меню
  private Map<String, JComponent> menuStorage
      = new HashMap<String, JComponent>();

  // конструктор, требует задать поток данных с меню
  public XMLMenuLoader(InputStream stream) {
    // настраиваем источник данных XML
    try {
      Reader reader = new InputStreamReader(stream, "UTF-8");
      source = new InputSource(reader);
      parser = SAXParserFactory.
          newInstance().newSAXParser();
    } catch (Exception ex) {
      throw new RuntimeException(ex);
    }
    // создаем обработчик XML
    documentHandler = new XMLParser();
  }

  // считывает XML и создает систему меню
  public void parse() throws Exception {
    parser.parse(source, documentHandler);
  }
```

```
  // позволяет получить строку меню
  public JMenuBar getMenuBar(String name) {
    return (JMenuBar) menuStorage.get(name);
  }
  // позволяет получить выпадающее меню
  public JMenu getMenu(String name) {
    return (JMenu) menuStorage.get(name);
  }
  // позволяет получить элемент меню
  public JMenuItem getMenuItem(String name) {
    return (JMenuItem) menuStorage.get(name);
  }
  // удобный метод для быстрого добавления
  // слушателя событий
  public void addActionListener(String name, ActionListener
listener) {
    getMenuItem(name).addActionListener(listener);
  }

  // текущая строка меню
  private JMenuBar currentMenuBar;
  // список для упорядочения выпадающих меню
  private LinkedList<JMenu> menus = new LinkedList<JMenu>();

  // обработчик XML
  class XMLParser extends DefaultHandler {
    // новый узел XML
    public void startElement(String uri, String localName,
                             String qName, Attributes attributes) {
      // определяем тип узла
      if (qName.equals("menubar"))
        parseMenuBar(attributes);
      else if (qName.equals("menu"))
        parseMenu(attributes);
      else if (qName.equals("menuitem"))
        parseMenuItem(attributes);
    }
    // конец узла, используется для смены выпадающих меню
    public void endElement(String uri, String localName,
                           String qName) {
      if (qName.equals("menu")) menus.removeFirst();
    }
```

```
// создает новую строку меню
protected void parseMenuBar(Attributes attrs) {
  JMenuBar menuBar = new JMenuBar();
  // определяем имя
  String name = attrs.getValue("name");
  menuStorage.put(name, menuBar);
  currentMenuBar = menuBar;
}
// создает новое выпадающее меню
protected void parseMenu(Attributes attrs) {
  // создаем меню
  JMenu menu = new JMenu();
  String name = attrs.getValue("name");
  // настраиваем общие атрибуты
  adjustProperties(menu, attrs);
  menuStorage.put(name, menu);
  // добавляем меню к предыдущему выпадающему
  // меню или к строке меню
  if ( menus.size() != 0 ) {
    menus.getFirst().add(menu);
  } else {
    currentMenuBar.add(menu);
  }
  // добавляем в список выпадающих меню
  menus.addFirst(menu);
}
// новый пункт меню
protected void parseMenuItem(Attributes attrs) {
  // проверяем, не разделитель ли это
  String name = attrs.getValue("name");
  if (name.equals("separator")) {
    menus.getFirst().addSeparator();
    return;
  }
  // создаем пункт меню
  JMenuItem menuItem = new JMenuItem();
  // настраиваем свойства
  adjustProperties(menuItem, attrs);
  menuStorage.put(name, menuItem);
  // добавляем к текущему выпадающему меню
  menus.getFirst().add(menuItem);
}
```

```
    // настройка общих атрибутов пунктов меню
    private void adjustProperties(
        JMenuItem menuItem, Attributes attrs) {
      // получаем поддерживаемые атрибуты
      String text = attrs.getValue("text");
      String mnemonic = attrs.getValue("mnemonic");
      String accelerator = attrs.getValue("accelerator");
      String enabled = attrs.getValue("enabled");
      // настраиваем свойства меню
      menuItem.setText(text);
      if (mnemonic != null) {
        menuItem.setMnemonic(mnemonic.charAt(0));
      }
      if (accelerator != null) {
        menuItem.setAccelerator(
            KeyStroke.getKeyStroke(accelerator));
      }
      if (enabled != null) {
        menuItem.setEnabled(Boolean.valueOf(enabled));
      }
    }
  }
}
```

Наш инструмент для распознавания системы меню разместится в библиотечном пакете com.porty.swing — так вы сможете легко присоединять его к своим программам. Начинается работа с конструктора класса XMLMenuLoader: в качестве параметра конструктора необходимо указывать поток данных (InputStream), откуда будут считываться информация. Поток InputStream прежде всего преобразуется в символьный поток Reader: если мы этого не сделаем, то неминуемо получим проблемы с кодировками символов, отличных от ASCII. Так как мы применяем XML, инструмент подразумевает, что данные находятся в кодировке UTF-8. Далее необходимо получить источник данных InputSource, именно с источником данных такого типа работает любой распознаватель XML. Как видно из программы, сделать это легко: достаточно передать символьный поток Reader в конструктор класса InputSource. После этого нам остается настроить сам распознаватель XML — для этого пригодится фабрика классов SAXParserFactory. В конце конструктора мы создаем обработчик данных XML, именно в этот обработчик будет поступать вся информация от распознавателя.

После вызова конструктора инструмент XMLMenuLoader готов к работе. Для начала распознавания необходимо вызвать метод parse(). Данный метод очень прост: он «включает» распознаватель XML, передавая ему в качестве параметров источник данных и ссылку на обработчик распознанных элементов.

Таким образом, основная работа инструмента выполняется во внутреннем классе XMLParser, который обрабатывает распознанные элементы XML. Обработчик в интерфейсе SAX (а мы выбрали для обработки именно этот интерфейс) пассивен: он ожидает, пока распознаватель не вызовет один из определенных в обработчике методов. В нашем обработчике основной метод — startElement(), он вызывается распознавателем при обна-

ружении нового элемента XML. В качестве параметров данному методу передается название элемента и набор его атрибутов. Выяснив, какое название у текущего элемента, мы выполняем соответствующие действия.

Для элемента с названием menubar (он описывает строку меню) вызывается метод parseMenuBar(), в качестве параметров этого метода передаются атрибуты элемента. Прежде всего создается новая строка меню JMenuBar. Атрибуты используются для получения уникального имени элемента (как вы помните, при описании структуры меню на XML все элементы должны иметь уникальное имя в атрибуте name). С помощью уникального имени новая строка меню размещается в ассоциативном массиве menuStorage, откуда ее всегда можно получить обратно как раз по имени. Строка меню, распознанная последней, также сохраняется в ссылке currentMenuBar: с помощью этой ссылки в строку меню будут добавляться выпадающие меню.

Выпадающее меню, описанное в элементе с названием menu, создается и настраивается в методе parseMenu(). Для начала создается новый объект JMenu, после чего мы получаем уникальное имя нового выпадающего меню. Прежде чем разместить новое меню в массиве menuStorage, для него настраиваются общие для всех пунктов меню свойства (текст, символ мнемоники, клавиши быстрого доступа, доступность). Настройка выполняется в методе adjustProperties(): для всех поддерживаемых свойств мы получаем значения атрибутов с соответствующими именами, для каждого атрибута выясняем, не равен ли он пустой ссылке null (если атрибут для текущего элемента не описан, то вместо него будет возвращена пустая ссылка), и если нет, то настраиваем соответствующее свойства пункта меню. Заметьте, что метод adjustProperties() работает одинаково для выпадающих меню JMenu и пунктов меню JMenuItem, и в этом нет ничего удивительного — достаточно вспомнить, что класс JMenu унаследован от JMenuItem.

После настройки свойств остается выяснить, куда необходимо добавить новое выпадающее меню. Все дело в том, что выпадающее меню может быть добавлено как в строку меню, так и в другое выпадающее меню (при сложной иерархической системе меню). Чтобы поддерживать меню произвольной сложности, мы вводим список выпадающих меню LinkedList с названием menus. Каждое новое меню добавляется в этот список первым (методом addFirst()). Следующее выпадающее меню будет добавлено в то меню, что находится в списке на первой позиции, и после этого само станет первым, так что следующее за ним меню будет добавлено уже в него. Таким образом мы сможем поддерживать любое количество выпадающих меню. Остается только удалять из списка первое выпадающее меню, когда в файле XML закрывается элемент menu. Сделать это несложно: при закрытии элемента вызывается метод endElement(), именно в нем мы удаляем первое меню методом removeFirst(), если название элемента нам подходит (menu)[5]. В том случае если список выпадающих меню пуст, новое меню добавляется непосредственно в строку меню.

Наконец, обычный пункт меню, описываемый элементом с названием menuitem, обрабатывается в методе parseMenuItem(). Прежде всего мы проверяем, не описывает ли данный элемент разделитель (в таком случае атрибут name будет равен строке separator), и если это так, просто добавляем разделитель в текущее выпадающее меню (его позволяет получить список выпадающих меню menus). Если же элемент описывает «настоящий» пункт меню, то мы создаем объект JMenuItem, настраиваем его свойства (с помощью все того же метода adjustProperties()), добавляем в массив menuStorage и присоединяем к текущему выпадающему меню. Обратите внимание, что пункты меню присоединяются к выпадающему меню без проверки на его наличие, так что при написании XML следите за тем, чтобы пункты меню всегда находились внутри элементов menu.

---

[5] Обратите внимание, что в данном случае список LinkedList используется фактически в роли стека, благодаря наличию методов getFirst(), addFirst() и removeFirst() (эквивалентных просмотру верхнего элемента стека, «проталкиванию» элемента в стек и «выталкиванию» его оттуда).

После завершения анализа файла XML система меню создана, все выпадающие меню и пункты меню присоединены друг к другу, а ссылки на все созданные элементы меню хранятся в ассоциативном массиве menuStorage. Для дальнейшей работы с меню необходимо предоставить способ получения созданных элементов меню по их уникальному имени, так чтобы программа смогла добавить в окно строку меню и зарегистрировать слушателей событий для пунктов меню. Для этого в классе XMLMenuLoader имеется несколько удобных методов для получения элементов меню: метод getMenuBar() позволяет получить по имени строку меню, метод getMenu() возвращает выпадающее меню, наконец, метод getMenuItem() возвращает пункт меню. Еще один вспомогательный метод с названием addActionListener() чрезвычайно удобен для быстрого присоединения слушателя ActionListener к пункту меню с именем, указанным в качестве параметра метода. Применяя данный метод, вы сможете еще больше ускорить настройку системы меню.

Обратите внимание, что благодаря использованию анализатора SAX наш инструмент крайне нетребователен к структуре документа XML. Главное, чтобы элементы menubar, menu и menuitem были описаны в подходящем порядке. Используя эту возможность, вы можете описать в одном файле XML и строку меню для нашего инструмента (или даже несколько), и что-то еще, если вам это понадобится.

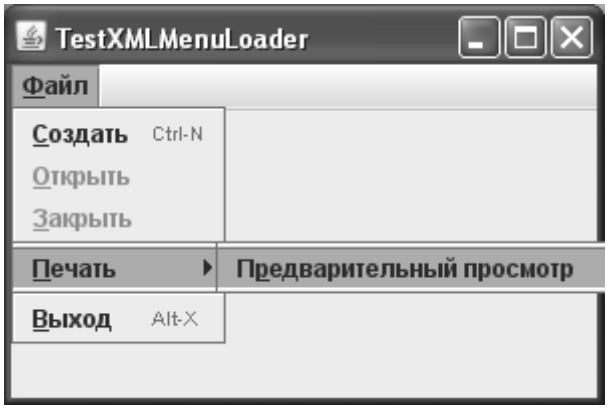
Ну а теперь остается проверить наш новый инструмент в работе. Теперь попробуем загрузить систему меню, описанную нами еще в начале раздела, в обычное окно JFrame. Описание меню на языке XML будет находиться в файле menu.xml, вы сможете найти его вместе с исходными текстами программы:

```
// TestXMLMenuLoader.java
// Проверка загрузки системы меню из файла XML
import javax.swing.*;
import com.porty.swing.*;
import java.io.*;
import java.awt.event.*;
import java.awt.*;

public class TestXMLMenuLoader extends JFrame {
  public TestXMLMenuLoader() {
    super("TestXMLMenuLoader");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // открываем файл XML с описанием меню
    try {
      InputStream stream =
          new FileInputStream("menu.xml");
      // загружаем меню
      XMLMenuLoader loader =
          new XMLMenuLoader(stream);
      loader.parse();
      // устанавливаем строку меню
      setJMenuBar(loader.getMenuBar("mainMenu"));
      // быстрое присоединение слушателя
      loader.addActionListener("exit",
          new ActionListener() {
```

```
            public void actionPerformed(ActionEvent e) {
              System.exit(0);
            }
          });
      } catch (Exception ex) {
        ex.printStackTrace();
      }
      // выводим окно на экран
      setSize(300, 200);
      setVisible(true);
    }
    public static void main(String[] args) {
      SwingUtilities.invokeLater(
          new Runnable() {
            public void run() { new TestXMLMenuLoader(); } });
    }
}
```

Пример чрезвычайно прост: мы наследуем класс своего приложения от окна с рамкой JFrame, и задаем для него небольшой размер. Система меню загружается прямо в конструкторе окна (приходится применять блок обработки исключений try/catch, так как происходит работа с вводом/выводом и всегда возможно исключение типа IOException): мы открываем файл menu.xml, передаем поток с данными в наш новый инструмент XMLMenuLoader и загружаем систему меню, вызывая метод parse(). После этого все элементы системы меню находятся в хранилище под своими уникальными именами. Далее нам остается получить строку меню (имя ее нам известно из описания меню на языке XML) и присоединить ее к своему окну. Также демонстрируется удобство метода addActionListener() — он одновременно получает необходимый пункт меню по имени и сразу же присоединяет к нему слушателя событий.

Запустив пример, вы увидите описанную нами на языке XML систему меню в окне, и оцените скорость, с которой можно создать сложную систему меню, настроенную до мелочей (включая мнемоники, клавиши быстрого доступа и возможность перевода на любой язык в кодировке UTF-8).



Создание аналогичной системы меню в коде вылилось бы в немалое количество строк кода, которые к тому же были бы трудно поддерживаемыми и не слишком хорошо читаемыми. Вы легко сможете доработать созданный нами инструмент, так чтобы он поддерживал больше свойств (понадобится доработать метод adjustProperties()) и остальные элементы системы меню, такие как меню с флажками. Только не забывайте, что наш инструмент манипулирует компонентами Swing, а именно элементами меню, это значит, что вся работа с ним должна идти из потока рассылки графических событий.

Идея применения декларативного языка, такого как XML, для описания простого интерфейса не нова, и на этой ниве испытывали свои силы многие программисты. Вы можете попробовать использовать довольно известный проект SwiXML, также позволяющий создать компоненты Swing из описания XML. Для системы меню или простых компонентов этот подход работает прекрасно, но вот со сложными интерфейсами поддержка XML вряд ли проще создания интерфейса прямо в коде. К тому же в коде, в хорошем средстве разработки (IDE) у вас будут и автоматическая проверка корректности типов, и автоматическое дополнение имен классов и переменных, и многое другое, что в обычном описании недоступно.

## *Панели инструментов*

Панели инструментов предназначены для вывода на экран набора кнопок (как правило, кнопок особого вида: с краткими надписями или вовсе без них, но с подсказками и с небольшими четко различимыми значками), инициирующих запуск наиболее часто используемых команд приложения. В панелях инструментов также встречаются наиболее востребованные пользователями компоненты, находить которые в меню или диалоговых окнах долго и неудобно. Продуманные панели инструментов значительно повышают привлекательность приложения и «привязывают» к себе пользователя, который мгновенно привыкает к ним.

В Swing панели инструментов представлены компонентом JToolBar. Этот компонент довольно прост в работе и настройке, тем не менее, с его помощью вы сможете создавать любые панели инструментов, способные на многое, а понадобится для этого всего несколько строк кода.

### Простые панели инструментов

Создание панели инструментов в Swing не таит в себе никаких трудностей. Вы создаете компонент JToolBar, добавляете в него свои кнопки или другие компоненты (особенно удобно использовать для панелей инструментов «команды» Action, которые позволяют в одном месте указать и параметры внешнего вида кнопки, и описать то, что должно происходить при щелчке на ней) и выводите панель инструментов на экран. Проиллюстрирует сказанное следующий пример:

```
// SimpleToolbars.java
// Простые панели инструментов
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class SimpleToolbars extends JFrame {
  public SimpleToolbars() {
    super("SimpleToolbars");
```

```
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // первая панель инструментов
    JToolBar toolbar1 = new JToolBar();
    // добавим кнопки
    toolbar1.add(new JButton(
        new ImageIcon("images/New16.gif")));
    toolbar1.add(new JButton(
        new ImageIcon("images/Open16.gif")));
    // разделитель
    toolbar1.addSeparator();
    // добавим команду
    toolbar1.add(new SaveAction());
    // вторая панель инструментов
    JToolBar toolbar2 = new JToolBar();
    // добавим команду
    toolbar2.add(new SaveAction());
    // раскрывающийся список
    toolbar2.add(new JComboBox(new String[] {
        "Жирный", "Обычный" }));
    // добавим панели инструментов в окно
    add(toolbar1, "North");
    add(toolbar2, "South");
    // выводим окно на экран
    setSize(400, 300);
    setVisible(true);
  }
  // команда для панели инструментов
  class SaveAction extends AbstractAction {
    public SaveAction() {
      // настроим значок команды
      putValue(AbstractAction.SMALL_ICON,
          new ImageIcon("images/Save16.gif"));
      // текст подсказки
      putValue(AbstractAction.SHORT_DESCRIPTION,
          "Сохранить документ...");
    }
    public void actionPerformed(ActionEvent e) {
      // ничего не делаем
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
```

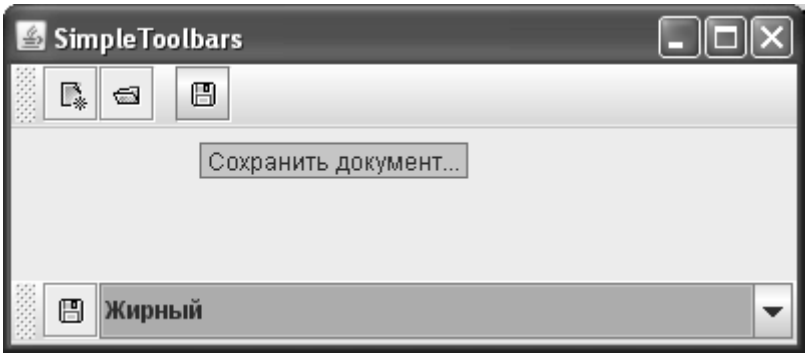
```
            new Runnable() {
              public void run() { new SimpleToolbars(); } });
    }
}
```

Мы создаем две панели инструментов, которые разместятся в небольшом окне JFrame. Сначала демонстрируется наиболее распространенный способ использования панели инструментов: создав компонент JToolBar, вы добавляете в него кнопки JButton, как правило, с небольшим значком. После двух кнопок мы добавляем разделитель, вызывая специальный метод addSeparator(). Внешний вид разделителя и его размеры зависят от задействованных внешнего вида и поведения, но вы можете повлиять на его размеры, вызвав перегруженную версию метода addSeparator() и передав в него объект Dimension, задающий размеры разделителя. В панели инструментов разделитель используется примерно так же, как в меню: для визуального отделения групп компонентов, выполняющих различные действия.

Третья кнопка добавляется не в виде компонента JButton, а как экземпляр команды Action, добавить команду позволяет специальная перегруженная версия метода add(). В главе 9 мы подробно рассмотрели команды Action и убедились в том, что во многих случаях они очень удобны, позволяя совмещать в одном месте настройку внешнего вида элемента управления и описание действия, которое он выполняет. Это особенно верно для панелей инструментов: в классе команды мы задаем значок и текст подсказки, и тут же можем описать действие, которое должна будет выполнить команда. После этого остается только добавить команду в панель инструментов. Если же вам понадобится модифицировать команду, будь это ее внешний вид или действие, вы знаете с чего начать — все скрыто в одном классе.

Вторая панель инструментов демонстрирует, что храниться в ней могут не только кнопки, но и любые другие компоненты. Сначала в панель добавляется команда, а затем раскрывающийся список JComboBox, созданный на основе массива строк. Раскрывающиеся списки довольно часто «гостят» в панелях инструментов, и не зря: они занимают немного места и позволяют организовать гибкий выбор одного варианта из многих.

Созданные нами панели инструментов добавляются в «пограничные» области окна, в котором по умолчанию используется полярное расположение BorderLayout. Первая панель размещается на севере, а вторая — на юге окна. Расположение BorderLayout специально создано для главных окон приложения с панелями инструментов. Запустив программу с примером, вы сможете оценить внешний вид панелей инструментов JToolBar.



Обратите внимание на специальные полосы с левого края панели инструментов (так называемые *вешки перемещения*): с их помощью вы сможете *перетаскивать* (drag) панели инструментов, *причаливая* (dock) их к разным края окна, или же оставлять «плавающими». Интересно, что перетаскивать панели инструментов и причаливать их к краям окна можно, *только* если в нем используется расположение BorderLayout. Отсюда следует нехитрый совет: если в вашем окне есть панели инструментов, и вы хотите дать пользователю возможность их перетаскивать к другим краям окна, применяйте менеджер расположения BorderLayout, в противном случае пользователь может быть весьма озадачен тем, что панели инструментов оказываются совсем не там, куда он их перетаскивает.

Напоследок давайте составим небольшую таблицу свойств панелей инструментов JToolBar, позволяющих настраивать их поведение. Свойств в табл. 10.2 немного, и это неудивительно — панели инструментов очень просты и фактически представляют собой контейнер особого вида с некоторыми дополнительными возможностями.

**Таблица 10.2.** Основные свойства панелей инструментов

| Свойство | Описание |
|---|---|
| orientation | Задает направление панели инструментов: вертикальное или горизонтальное (используемое по умолчанию). В заданном вами направлении в панель будут добавляться компоненты. Направление можно сменить и после создания панели инструментов |
| floatable | Управляет возможностью перетаскивания панели инструментов. По умолчанию перетаскивание включено. Если перетаскивание не нужно или способно запутать пользователя, задайте это свойство равным false |
| rollover | Включает эффект интерактивности — при наведении на компонент указателя мыши у компонента появляется специальная рамка или иной визуальный эффект (к примеру, особый значок). По умолчанию эффект интерактивности отключен. Работает только для кнопок (наследников класса AbstractButton) |
| borderPainted | Управляет прорисовкой рамки панели инструментов. Наличие рамки зависит от внешнего вида и поведения, и по умолчанию рамка отображается. Как правило, это специальная полоса, с помощью которой пользователь может перетаскивать панель инструментов. Малополезное свойство, применимое разве что для достижения некоторых визуальных эффектов, когда стандартная рамка не подходит |

Панели инструментов вызывают ожесточенные споры о том, какие же именно компоненты должны в них находиться, чтобы не запутывать пользователя. В нашем примере использована классическая схема — маленькие кнопки со значками и подсказками, но без текста. Однако такой дизайн хорош лишь для интуитивно понятных, привычных значков, таких как дискета для сохранения документа или лупа для поиска. Ознакомление пользователя с панелью инструментов пойдет гораздо лучше, если вы найдете на ней место для подписей на кнопках, а не станете заставлять вызывать всплывающую подсказку, чтобы понять, что же все-таки означает этот красивый значок.

## Комбинирование панелей инструментов

Совсем не обязательно ограничиваться созданием простейших панелей инструментов, располагающихся у границы окна вашего приложения и состоящих из кнопок и некоторых других компонентов. Если вспомнить, что панель инструментов JToolBar — са-

мый обыкновенный компонент библиотеки Swing, и его можно добавлять в любые панели с любым подходящим менеджером расположения, то легко конструировать панели инструментов произвольной сложности, динамически добавлять и удалять их, составляя набор панелей из любого их количества.

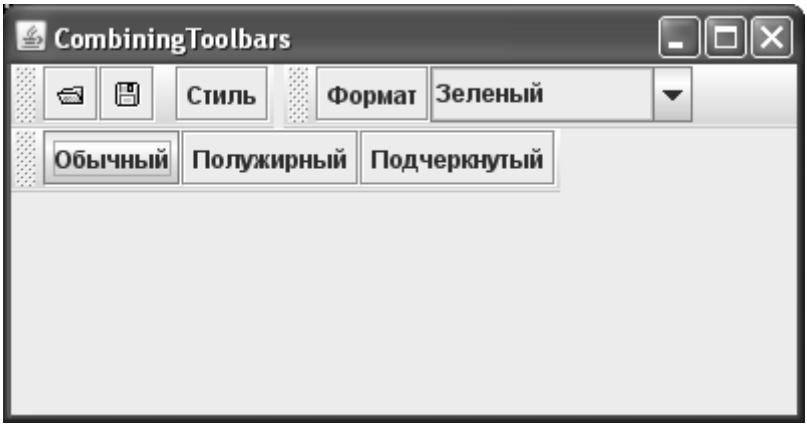
Давайте рассмотрим пример. В этом примере настроим несколько панелей инструментов и расположим их вместе так, что пользователь будет видеть перед собой одну сложную, состоящую из нескольких частей, панель инструментов, которую при том легко будет расширять, перетаскивать по частям и настраивать:

```
// CombiningToolbars.java
// Создание комбинированных панелей
// инструментов
import javax.swing.*;
import com.porty.swing.*;
import java.awt.event.*;
import java.awt.*;

public class CombiningToolbars extends JFrame {
  public CombiningToolbars() {
    super("CombiningToolbars");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // несколько панелей инструментов
    JToolBar toolbar1 = new JToolBar();
    toolbar1.add(new OpenAction());
    toolbar1.add(new SaveAction());
    toolbar1.addSeparator();
    toolbar1.add(new JButton("Стиль"));
    JToolBar toolbar2 = new JToolBar();
    toolbar2.add(new JButton("Формат"));
    toolbar2.add(new JComboBox(new String[] {
        "Красный", "Зеленый"}));
    JToolBar toolbar3 = new JToolBar();
    toolbar3.add(new JButton("Обычный"));
    toolbar3.add(new JButton("Полужирный"));
    toolbar3.add(new JButton("Подчеркнутый"));
    // выравнивание содержимого
    toolbar2.add(Box.createGlue());
    // добавим две панели инструментов сюда
    JPanel first =
        BoxLayoutUtils.createHorizontalPanel();
    first.add(toolbar1);
    first.add(Box.createHorizontalStrut(5));
    first.add(toolbar2);
    // комбинируем полученные панели
```

```
    JPanel all =
        BoxLayoutUtils.createVerticalPanel();
    all.add(first);
    all.add(toolbar3);
    // выравнивание содержимого
    BoxLayoutUtils.setGroupAlignmentX(
        JComponent.LEFT_ALIGNMENT, first, toolbar3);
    // добавим полученное на север окна
    add(all, "North");
    // выводим окно на экран
    setSize(400, 300);
    setVisible(true);
  }
  // несколько команд для панелей инструментов
  class OpenAction extends AbstractAction {
    public OpenAction() {
      // настроим значок команды
      putValue(AbstractAction.SMALL_ICON,
          new ImageIcon("images/Open16.gif"));
    }
    public void actionPerformed(ActionEvent e) {
      // ничего не делаем
    }
  }
  class SaveAction extends AbstractAction {
    public SaveAction() {
      // настроим значок команды
      putValue(AbstractAction.SMALL_ICON,
          new ImageIcon("images/Save16.gif"));
    }
    public void actionPerformed(ActionEvent e) {
      // ничего не делаем
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new CombiningToolbars(); } });
  }
}
```

В примере мы создаем целых три панели инструментов, которые в итоге должны выглядеть как единое целое. Сами панели инструментов несложны и не требуют особых комментариев: они состоят из настроенных команд Action, кнопок JButton, а также дополнительных компонентов. Здесь мы снова использовали раскрывающийся список JComboBox, но в панели инструментов найдется место для любого компонента, лишь бы он не был чересчур требовательным к пространству контейнера, в противном случае на его фоне потеряются остальные элементы панели. Обратите внимание, что после добавления во вторую панель инструментов раскрывающегося списка мы добавили *заполнитель* (glue): как оказывается, в качестве менеджера расположения компонента JToolBar используется прекрасно знакомый нам менеджер BoxLayout. Вы можете пользоваться этим фактом для того, чтобы особым образом выстраивать компоненты панели инструментов, если вам это понадобится.

Самое главное — правильно расположить панели инструментов, так чтобы их было удобно добавлять, удалять или перемещать. Первые две панели инструментов мы расположили в панели с горизонтальным блочным расположением, разделив распоркой размером 5 пикселов (для создания панели применялся класс BoxLayoutUtils, описанный в главе 7). Полученную панель и третью панель инструментов мы поместили в панель с вертикальным блочным расположением. Как вы помните, в главе 7 при обсуждении блочного расположения мы отмечали, что у компонентов в контейнерах с таким расположением должно быть согласованное выравнивание, и панели инструментов — не исключение. В нашем примере выравнивание компонентов производится по левому краю. Полученную конструкцию из трех панелей инструментов мы добавляем на север нашего окна, после чего остается запустить программу с примером и посмотреть на результат.




После запуска программы вы увидите картину, довольно часто встречающуюся в современных сложных приложениях: несколько панелей инструментов, отвечающих за решение разных задач, работают как одно целое. Их также можно перетаскивать, хотя перетаскивание (из-за блочного расположения BoxLayout, а не полярного расположения BorderLayout) реализуется не полностью: вам не удастся причалить панели инструментов к другим границам окна или поменять их местами. Впрочем, если вам понадобится подобное поведение, его достаточно просто реализовать самому: следить за перемещениями мыши в слушателе и назначать панелям инструментов подходящие места в контейнере. Также было бы очень удобно добавить к комбинированным панелям инструментов контекстное меню, позволяющее «на лету» выводить новую панель инструментов или прятать одну из присутствующих экране панелей. Вы можете также обратить внимание

на дополнительные библиотеки от энтузиастов Swing, к примеру, на библиотеку Flamingo, которая предоставляет некоторые довольно впечатляющие расширения для панелей инструментов.

## *Резюме*

Меню — это «визитная карточка» приложения, по нему пользователи судят о его масштабе и возможностях. Каким бы сложным набором функций ваше приложение не обладало, меню Swing позволят обеспечить доступ к ним, причем именно в том виде, который лучше подходит вашему приложению. Панель инструментов — еще одна часть вашего приложения, «бросающаяся в глаза» при знакомстве с ним. Компонент JToolBar библиотеки Swing делает все, чтобы панели инструментов вашего приложения поражали своим удобством и функциональностью.

# Глава 11. Списки

В этой главе будут рассмотрены обычные списки JList и раскрывающиеся списки JComboBox библиотеки Swing. Списки JList предназначены для вывода нескольких (чаще всего равноценных) вариантов данных; пользователь сразу видит весь список (или большую его часть), может провести быстрое сравнение альтернатив и выбрать то, что ему необходимо. Возможен выбор и нескольких альтернатив сразу. В отличие от обычного списка JList раскрывающийся список JComboBox чаще всего используется для выбора одного варианта из многих, примерно так же, как это происходит в группе переключателей. Он показывает только выбранный в данный момент элемент, а также имеет небольшую кнопку раскрытия списка, выводящую на экран всплывающее окно с набором альтернатив, которыми можно сменить текущий вариант выбора. Раскрывающийся список допускает и ввод пользователем собственных значений, не представленных во всплывающем окне с возможными альтернативами.

Мы постепенно продвигаемся по библиотеке Swing, и вот уже настает пора знакомиться с первыми компонентами, данные которых полностью хранятся в *моделях*. Именно в моделях хранятся данные списков и, как мы увидим, это на самом деле удобно и придает программе изящество и гибкость. Вы можете раздельно обрабатывать данные и только после тщательной подготовки выводить их на экран. Правильное использование моделей во всех аспектах улучшает качество ваших программ.

## *Обычные списки JList*

Списки JList позволяют выводить на экран перечень некоторых элементов, чаще всего строк, среди которых пользователь может выбрать то, что ему подходит. Список JList нуждается для правильной работы в двух поставщиках данных (*моделях*): первый поставщик сообщает, какие элементы и в каком порядке будет представлять список, второй — какие элементы являются выбранными. Первая модель, сообщающая списку, какие данные в нем хранятся, должна реализовывать интерфейс ListModel. Как и везде в Swing, вам не обязательно задействовать модель или реализовывать ее с нуля: у вас есть верные помощники в виде специальных конструкторов списка JList и уже имеющихся стандартных моделей. Вторая модель, следящая за состоянием выделенных элементов, реализует интерфейс ListSelectionModel. По умолчанию в списке JList используется стандартная реализация этой модели DefaultListSelectionModel, поддерживающая все самые важные режимы выделения, так что писать ее самому вряд ли понадобится.

Для начала давайте посмотрим, как проще всего создать список JList и как это будет выглядеть. Вот пример:

```
// SimpleLists.java
// Простейший способ создания списков
import javax.swing.*;
import java.util.*;
import java.awt.*;

public class SimpleLists extends JFrame {
```

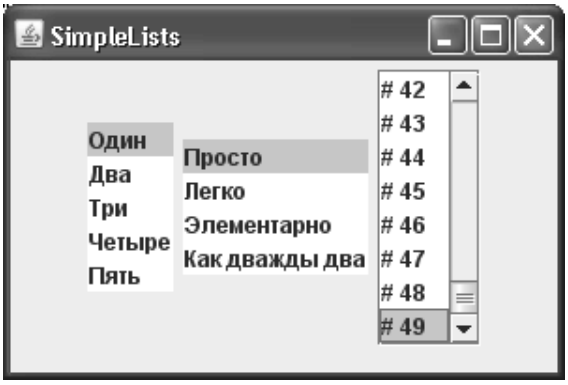
```
  // данные для списков
  private String[] data1 = { "Один", "Два",
      "Три", "Четыре", "Пять"};
  private String[] data2 = { "Просто", "Легко",
      "Элементарно", "Как дважды два"};
  public SimpleLists() {
    super("SimpleLists");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем списки
    JList list1 = new JList(data1);
    // для второго списка используем вектор
    Vector<String> data = new Vector<String>();
    data.addAll(Arrays.asList(data2));
    JList list2 = new JList(data);
    // динамически наполним вектор
    Vector<String> big = new Vector<String>();
    for (int i=0; i<50; i++) {
      big.add("# " + i);
    }
    JList bigList = new JList(big);
    bigList.setPrototypeCellValue("12345");
    // добавим списки в панель
    setLayout(new FlowLayout());
    add(list1);
    add(list2);
    add(new JScrollPane(bigList));
    // выведем окно на экран
    setSize(300, 200);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new SimpleLists(); } });
  }
}
```

В примере мы создаем небольшое окно и добавляем в него три списка JList. Первый список создается на основе массива строк String, второй использует в качестве данных вектор (динамический массив) Vector[1] из пакета java.util. Заметьте, как мы задействовали

[1] Класс Vector давно устарел, и все мы используем различные реализации списков List. Однако, по историческим причинам, Swing использует именно класс Vector, а поддержку современных списков добавлять в него не стали. Жаль, но ничего не поделаешь.

статический метод класса Arrays для создания вектора на основе еще одного массива строк. Ну и, наконец, третий список демонстрирует динамическое наполнение вектора данными прямо во время работы программы. С его помощью можно видеть, как список JList выводит большое количество элементов. Заметьте, что первые два списка сразу же добавляются в панель содержимого, а третий (большой) список предварительно включается в панель прокрутки JScrollPane. Компоненты Swing по умолчанию не предоставляют возможностей прокрутки содержимого, их необходимо помещать в панель прокрутки, и списки не являются исключением. Обратите также внимание на весьма полезный метод setPrototypeCellValue(). Элементы списка могут иметь разные размеры, поэтому перед выводом их на экран список JList запрашивает их размеры и в качестве своего горизонтального размера выбирает ширину самого большого элемента в списке. Метод setPrototypeCellValue() позволяет этого избежать. Вместо обращения ко всем элементам список будет использовать длину указанного вами объекта (чаще всего строки) и, таким образом, экономить время (это особенно верно для больших списков), а у вас появится возможность легко управлять внешним видом списка. Альтернативой данному методу также могут быть свойства fixedCellWidth и fixedCellHeight. С их помощью вы можете задать фиксированные размеры для ширины и высоты ячеек в пикселах, если уверены, что точно их знаете. Впрочем, задание размеров в пикселах — не лучшая идея в Java, так что метод setPrototypeCellValue() как правило предпочтительнее.

Нетрудно видеть, что в нашем примере нет и намека на модели, а списки тем не менее создаются и выводятся на экран. В этом вся прелесть Swing — вы выбираете тот подход к программированию, который вам больше по вкусу, и не задумываетесь о мелочах. На самом деле использованные нами конструкторы JList создают простую модель с помощью класса AbstractListModel на основе переданных массивов данных, так что от моделей все равно никуда не деться. То же самое можно сказать о модели, хранящей интервалы выделенных элементов списка: по умолчанию применяется стандартная модель, вполне удовлетворяющая основным нуждам.




Запустив программу с примером, вы сможете увидеть, как выглядят и функционируют списки. Заметьте, что во внешнем виде Metal списки без панели прокрутки или какой-либо рамки смотрятся не слишком эффектно, так что даже если список не велик, стоит поместить его в панель прокрутки. Это и улучшит его внешний вид, и позволит лучше контролировать размеры и состояние интерфейса в том случае, если элементы списка добавляются и удаляются динамически, прямо во время работы программы (а сделать это с помощью моделей легко). Работая с примером, вы сможете убедиться, что в списках по умолчанию можно выбирать произвольное количество элементов в произвольном порядке. Чуть позже мы увидим, как настроить параметры выбора элементов списка по своему вкусу.

## Модели

Примененные нами в первом примере конструкторы класса JList, наполняющие список данными из массива или вектора, конечно, замечательны, и в тех случаях, когда вам надо быстро и просто создать небольшой список, они незаменимы. Но по настоящему удобно работать с данными компонентов Swing, в том числе и с данными списков, позволяют модели. Мы уже прекрасно знаем, что они отделяют обработку данных от представления этих данных на экране, давая программисту возможность сосредоточиться поочередно на каждой задаче и таким образом применить знаменитый лозунг «разделяй и властвуй». Более того, конструкторы класса JList создают списки которые нельзя изменять и дополнять, что как правило, весьма некстати.

Обязанности моделей списков JList описаны в интерфейсе ListModel. Интерфейс этот очень прост, требуя от вас реализовать всего четыре метода. Два метода служат для присоединения и удаления слушателей событий, происходящих при обновлении данных списка (помните, при обсуждении модели MVC мы отмечали, что модель оповещает присоединенные к ней виды при изменении данных, и это позволяет виду всегда показывать верные данные); один метод возвращает элемент, находящийся на некоторой позиции списка; еще один метод позволяет списку узнать, сколько данных в данный момент содержит модель. Несложно, но достаточно для работы списка. Заметьте, что по умолчанию модель списка не подразумевает добавления и удаления элементов.

Сразу же бежать и писать свою собственную модель «с нуля» необязательно. Во многих ситуациях достаточно стандартной модели, поставляемой вместе с библиотекой Swing, она называется DefaultListModel. Фактически модель эта представляет собой динамический массив, который в дополнение к своим основным обязанностям еще и способен оповещать об изменениях в себе заинтересованных слушателей. Вы можете работать с этой моделью так же, как и с привычным вам контейнером данных: добавлять и удалять данные, вставлять их на любые позиции, производить перебор элементов и не заботиться о деталях, таких как выделение достаточного места под элементы, хранящиеся в массиве. Давайте рассмотрим пример:

```
// UsingListModel.java
// Использование стандартной модели списка
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class UsingListModel extends JFrame {
  // наша модель
  private DefaultListModel dlm;
  public UsingListModel() {
    super("UsingListModel");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // заполним модель данными
    dlm = new DefaultListModel();
    dlm.add(0, "Кое-что");
    dlm.add(0, "Кое-что еще");
    dlm.add(0, "Еще немного");
    // создаем кнопку и пару списков
    JButton add = new JButton("Обновить");
```

```
    add.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        dlm.add(0, "Новинка!");
      }
    });
    JList list1 = new JList(dlm);
    JList list2 = new JList(dlm);
    // добавляем компоненты
    setLayout(new FlowLayout());
    add(add);
    add(new JScrollPane(list1));
    add(new JScrollPane(list2));
    // выведем окно на экран
    setSize(400, 200);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new UsingListModel(); } });
  }
}
```

В примере мы создаем экземпляр модели DefaultListModel и используем его для хранения наших данных и вывода их на экран. Как видите, методы этой модели разительно напоминают методы стандартных контейнеров данных Java, и на самом деле всю работу по хранению данных модель DefaultListModel адресует находящемуся внутри нее вектору Vector, и даже названия методов для простоты сохранены. Мы добавляем в модель несколько элементов и передаем ее двум спискам для вывода данных на экран. Запустив программу с примером, вы увидите, как два списка, использующих одну и ту же модель, синхронно выводят одни и те же данные. В этом прелесть архитектуры MVC — неважно, откуда и как получены данные и как они будут выводиться: вы получаете данные и выбираете подходящий способ их отображения. Модели и виды сами позаботятся о том, чтобы все вовремя и правильно появлялось на экране.

Для демонстрации динамического поведения данных модели в панель содержимого была включена кнопка JButton. Щелкая на этой кнопке, вы будете добавлять данные в оба списка, потому что они используют одну и ту же модель, а все необходимые для обновления вида «переговоры» между моделью и списками будут проходить автоматически. Обновление экрана и проверка корректности компонентов также выполняются автоматически[2]. Однако помните, что динамически добавлять много данных в стандартную модель, которая уже присоединена к виду (или к нескольким видам), — нежелательная практика, ведущая к снижению производительности.

## СОВЕТ

> Основную часть данных подготавливайте и добавляйте в модель перед ее присоединением к виду (или видам). Это к тому же позволяет работать с моделью из отдельного потока, пока у ней нет связи с графическими компонентами. Присоединенная к виду модель оповещает его о каждом обновлении данных, и частое обновление данных может привести к серьезному снижению производительности.

В вашем приложении может быть множество диалоговых окон, особенно это касается приложений, работающих с базами данных. Здесь преимущества моделей просто неоценимы: чтобы каждый раз не получать данные, готовить их и выводить на экран, а затем запоминать новое состояние данных, вы просто храните данные в одной простой модели, которую и используете везде в своем приложении.

Стандартная модель DefaultListModel очень удобна и позволяет гибко хранить и изменять любые данные. Однако бывают случаи, когда лучше определить собственную модель данных, особенно там, где данные хранятся в нестандартных структурах или их необходимо получить из необычного источника, например из сетевого соединения или базы данных. Давайте рассмотрим небольшой пример, в котором данные списка мы будем получать из простой базы данных. Это весьма вероятная ситуация в больших приложениях, данные которые почти всегда хранятся в базах данных или в серверных компонентах (таких как Enterprise JavaBeans или .NET).

Нам не придется полностью реализовывать модель «с нуля». Создатели Swing уже позаботились о поддержке любыми моделями механизма оповещения (встроили списки слушателей и обеспечили рассылку событий) в абстрактном классе AbstractListModel. Нужно лишь унаследовать от этого класса и получить данные списка, а заботиться о поддержке слушателей будут другие:

```
// DatabaseListModel.java
// Модель списка, работающая с базой данных
package com.porty.swing;

import javax.swing.*;
```

[2] Списки увеличатся после добавления новых элементов, и самый длинный элемент не «влезет» в панель прокрутки после появления вертикальной полосы прокрутки, что приведет к появлению горизонтальной полосы прокрутки, крайне нежелательного гостя. На самом деле лучше составлять интерфейс так, чтобы размеров списка вместе с панелью прокрутки заранее хватало под новые элементы (в этом могут помочь подходящие менеджеры расположения и известный нам метод setPrototypeCellValue()). Внезапное изменение состояния интерфейса запутывает пользователя, и вы легко увидите это, поменяв размер окна — менеджер FlowLayout поменяет их размеры на предпочтительные. При работе со списками очень важно правильно определять максимальную длину их элементов — это обезопасит нас от неприятных сюрпризов.

```
import java.sql.*;
import java.util.*;

public class DatabaseListModel extends AbstractListModel {
  // здесь будем хранить данные
  private ArrayList<String> data = new ArrayList<String>();
  // загрузка из базы данных
  public void setDataSource(ResultSet rs, String column)
      throws SQLException {
    // получаем данные
    data.clear();
    while ( rs.next() ) {
      data.add(rs.getString(column));
    }
    // оповещаем виды (если они есть)
    fireIntervalAdded(this, 0, data.size());
  }
  // методы модели для выдачи данных списку
  public int getSize() {
    return data.size();
  }
  public Object getElementAt(int idx) {
    return data.get(idx);
  }
}
```

Сама по себе модель DatabaseListModel довольно проста. Она наследует от абстрактного класса моделей списков AbstractListModel и хранит данные в динамическом массиве ArrayList. Вся «соль» заключена в методе setDataSource(), который позволяет ей получать данные из соединения с базой данных. Для правильной работы этому методу необходимо передать результат запроса к базе данных (объект ResultSet), а также название столбца, данные которого будут использованы для заполнения списка (название столбца передается в строке column). Подразумевается, что в результате выполнения запроса столбец column появится в полученных данных (в объекте ResultSet), и в нем будут храниться текстовые данные. Впрочем, можно немного доработать нашу модель, чтобы она могла работать с любым типом данных.

Заметьте, что при получении всех записей из базы данных модель оповещает об этом своих слушателей. Для оповещения присоединенных к модели слушателей (а это чаще всего виды, отображающие ее данные) мы используем возможности базового класса AbstractListModel: вызываем методы fireXXX(), каждый из которых сообщает о некотором событии, как-то: обновление элементов списка, добавление в список новых данных или удаление данных из списка (эти методы просто создают подходящие экземпляры события ListDataEvent).

Наша модель очень проста, и отправляет события о изменениях в себе напрямую, что как правило означает перерисовку списка на экране. Это означает что мы должны вызывать метод setDataSource() только из потока рассылки событий, если модель уже пе-

редана в список JList. Однако до присоединения к списку мы вполне можем вызвать этот метод в отдельном потоке, чем мы вскоре и воспользуемся.

Более гибкая версия созданной в этом примере модели могла бы постепенно загружать данные, оповещая об этом список, и в случае если соединение с базой данных было бы медленным (например, сетевым), пользователь мог бы воочию видеть процесс загрузки данных в список и выбрать нужный ему элемент еще до полной загрузки. Однако это потребовало бы работы метода для загрузки данных в отдельном потоке, синхронизации списка data (в классе ArrayList для ускорения работы отключена синхронизация), и оповещения присоединенных к модели слушателей только из потока рассылки событий с помощью метода invokeLater(). Это несложно, и вы можете доработать нашу простую модель в качестве простого упражнения.

Созданная модель помещается в пакет com.porty.swing: так ее будет проще импортировать и многократно использовать в других программах. Не то чтобы новая модель была очень полезна, но для начального наброска интерфейса подходит весьма хорошо.

Остается проверить работу новой модели. Давайте напишем небольшой пример с ее использованием:

```
// DBListModelTest.java
// Использование модели списка
// для работы с базами данных
import javax.swing.*;
import java.sql.*;
import java.awt.*;

import com.porty.swing.*;

public class DBListModelTest {
  // параметры базы данных
  private static String
      dsn = "jdbc:odbc:Library",
      uid = "",
      pwd = "",
      query = "select * from readers.csv";
  public static void main(String[] args) throws Exception {
    // инициализация JDBC
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    // объект-соединение с базой данных
      Connection conn = DriverManager.getConnection(dsn, uid, pwd);
    Statement st = conn.createStatement();
    ResultSet rs = st.executeQuery(query);
    // создаем модель
    final DatabaseListModel dblm = new DatabaseListModel();
    // загружаем данные
    dblm.setDataSource(rs, "surname");
    rs.close();
```

```
        // интерфейс создаем в потоке рассылки событий
        SwingUtilities.invokeLater(
            new Runnable() {
              public void run() {
                // присоединяем список
                JList list = new JList(dblm);
                // помещаем список в окно
                JFrame frame = new JFrame("DBList");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setSize(200, 200);
                frame.add(new JScrollPane(list));
                frame.setVisible(true);
              } });
    }
}
```

Для работы модели нам понадобится соединение с базой данных. Подойдет практически любая база данных и ее любая таблица, в которой есть столбец с текстовыми данными. Для соединения с базой данных мы задействуем интерфейс JDBC (Java Database Connectivity)[3], настройка его для работы с вашей системой управления базами данных (СУБД) зависит от используемой платформы. Чаще всего это несложно и подробно описано в документации. Для правильной работы нам необходимо загрузить драйвер (мы применяем стандартный драйвер JDBC-ODBC от Sun, входящий в пакет JDK) и создать соединение с базой данных. Для этого требуется три параметра: идентификатор базы данных (Data Source Name DSN), имя пользователя (User ID UID) и его пароль (Password PWD). Эти параметры будут зависеть от базы данных и ее настроек. После этого, если все проходит успешно, производится запрос к базе данных (запрос находится в строке query, измените его так, чтобы он работал с вашей базой данных), и модель DatabaseListModel получает результат запроса ResultSet и название столбца, данные которого появятся в списке. Мы смело вызываем методы модели DatabaseListModel прямо из потока метода main(), потому что сама модель пока еще не присоединена ни к одному графическому компоненту.

Напоследок, уже из потока рассылки событий, к модели присоединяется вид — список JList, который мы помещаем в окно. Запустив приложение, вы увидите, как список выводит элементы, полученные моделью из базы данных. Использованный нами в примере подход гибок и удобен: модель выполняет все действия по получению и подготовке данных, и если у вас есть вопросы по данным, вы обращаетесь к ней. Список связывается с моделью и отображает ее информацию, и вы знаете, что для изменения внешнего вида программы нужно работать с видом, но не с данными. Все на своих местах.

## Модели для больших списков

Простота моделей списков без сомнения полезна при создании списков с небольшим или средним количеством данных. Чем проще в этом случае написан код, тем он быстрее работает и тем проще его обновлять и изменять в будущем. Однако, если речь

[3] Интерфейс JDBC — часть стандартной библиотеки JDK. Узнать об этом интерфейсе подробнее вы сможете в интерактивной документации пакета java.sql и на официальном сайте java.sun.com.

идет о списках с огромным количеством данных, где количество элементов списка исчисляется тысячами, а то и большим количеством, приходится задумываться о более сложных алгоритмах работы с моделью.

Прежде всего, необходимо помнить базовое правило работы с моделями: чем больше данных вы сможете загрузить в них до присоединения к списку, находящемуся на экране, тем проще с ними работать. Вы можете загружать данные в отдельном потоке, не заботясь о проблеме столкновения с потоком рассылки событий, и не замедлять реакции интерфейса в ответ на действия пользователя. Однако выполнить это правило не всегда возможно — если данные загружаются медленно, имеет смысл показывать на экране то, что уже есть, и постепенно добавлять новые элементы.

Постепенное добавление элементов в модель списка, который уже находится на экране, требует наличия отдельного потока, загружающего данные (это может быть объект SwingWorker, который мы рассматривали в главе 3). Чем реже мы будем сообщать списку о новых данных (блоками новых элементов), тем меньше он будет перерисовывать себя, и значит, тем быстрее будет отклик на действия пользователя, а это для последнего всегда важно. Сообщать списку о добавлении новых данных следует только из потока рассылки событий, иначе рано или поздно крах программы неминуем.

Список JList *никогда* не просит информацию об элементах списка, не видных в данный момент на экране, так же работают и остальные сложные компоненты Swing. Если приложение подразумевает работу в определенном диапазоне элементов списка, то список JList будет запрашивать информацию только о них и элементах, находящихся около них. Этот факт можно использовать для кэширования, если, например, элементы списка хранятся на диске. Все элементы большого списка невозможно хранить в памяти, так как она не безгранична, поэтому вы можете хранить их где-либо еще, в базе данных или подобном хранилище, и идея кэширования элементов, с которыми в данный момент работает пользователь, и близлежащих элементов позволит порядочно ускорить доступ к ним.

## Выделение

Если вы помните, мы говорили о том, что у списка JList имеется две модели: одну, поставляющую списку элементы для отображения, мы только что рассмотрели, а вторая хранит информацию о выделенных элементах списка. Вторая модель называется *моделью выделения списка*, она должна реализовывать методы, описанные в интерфейсе ListSelectionModel. Надо сказать, что модель выделения списка реализовать на порядок сложнее, чем модель с данными списка, пусть на первый взгляд это кажется удивительным. Все дело в существующем разнообразии вариантов выбора элементов списка пользователями: непрерывными интервалами, поодиночке, произвольно. Все эти ситуации модель выделения должна учитывать и хранить, а заодно оповещать своих слушателей (списки) об изменениях в выделении.

Главные два метода модели выделения — setSelectionInterval() и addSelectionInterval(). Первый метод устанавливает новый интервал выделения, а второй добавляет к уже имеющемуся выделению еще один интервал выделения. Интервал выделения задается двумя целыми числами: позицией первого выделенного элемента и позицией последнего выделенного элемента. Они могут совпадать, что означает выделение одного элемента, и необязательно первое число должно быть больше второго. В методе addSelectionInterval() вы сможете провести основную работу по выделению элементов, задав именно те режимы выделения, которые вам подходят. К примеру, используемая списком JList по умолчанию модель в режиме одиночного выделения позволяет выделять только один элемент, даже если пользователь попытался выделить большой интервал из нескольких элемен-

тов, и следит за этим она именно в упомянутых двух методах. В модели выделения также отдельно хранится начало последнего интервала выделения (selection anchor) и конец этого интервала выделения (selection lead). На некоторых платформах выделенные последними интервалы изображаются особым образом (так пользователи поймут, что же именно они выделили в последний раз), в расчете на такие случаи эти интервалы хранятся в модели отдельно. С помощью модели выделения вы также можете полностью отменить текущее выделение и задать специальный режим методом setValueIsAdjusting(). Такой режим полезен там, где выделение непрерывно обновляется (к примеру, выделение путем перетаскивания мыши), и каждый новый интервал можно считать частью предыдущего, не оповещая о нем слушателей.

Все описанные в интерфейсе ListSelectionModel возможности полностью реализованы в стандартной модели выделения DefaultListSelectionModel. Именно она используется в списках JList по умолчанию и менять ее чем-то особенным требуется редко. Она поддерживает три режима выделения элементов списка и позволяет получить исчерпывающую информацию о текущем выделении. Давайте посмотрим на примере, как с ней работать:

```
// ListSelectionModes.java
// Различные режимы выделения
import javax.swing.*;
import java.awt.*;

public class ListSelectionModes extends JFrame {
  private String[] data = { "Красный", "Синий",
      "Зеленый", "Желтый", "Белый"};
  public ListSelectionModes() {
    super("ListSelectionModes");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // заполним модель данными
    DefaultListModel dlm =
        new DefaultListModel();
    for (String next : data)
      dlm.addElement(next);
    // три списка с разным типом выделения
    JList list1 = new JList(dlm);
    list1.setSelectionMode(
        ListSelectionModel.SINGLE_SELECTION);
    JList list2 = new JList(dlm);
    list2.setSelectionMode(
        ListSelectionModel.SINGLE_INTERVAL_SELECTION);
    JList list3 = new JList(dlm);
    // аналогично предыдущему вызову
    list3.getSelectionModel().setSelectionMode(
        ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
    // добавляем компоненты
```

```
        setLayout(new FlowLayout());
        add(new JScrollPane(list1));
        add(new JScrollPane(list2));
        add(new JScrollPane(list3));
        // выведем окно на экран
        setSize(300, 200);
        setVisible(true);
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(
            new Runnable() {
              public void run() { new ListSelectionModes(); } });
    }
}
```

В примере мы заполняем стандартную модель DefaultListModel данными из массива (можно было бы передать массивы напрямую в конструкторы списков, но это не позволит нам сэкономить память) и на ее основе создаем три списка с одинаковыми элементами. Разница между списками лишь в режиме выделения элементов. Режим выделения можно сменить двумя способами: вызвав метод setSelectionMode() класса JList или вызвав метод с точно таким же названием уже для самой модели выделения. На самом деле это одно и тоже, метод класса JList просто обращается к модели выделения. В нашем распоряжении имеется три режима выделения (табл. 11.1).

**Таблица 11.1.** Режимы выделения элементов списка

| Режим | Описание |
|---|---|
| SINGLE_SELECTION | Позволяет пользователю выбирать только один элемент списка. Часто используемый режим, хотя для тех же целей могут быть успешно применены группы переключателей или раскрывающиеся списки |
| SINGLE_INTERVAL_SELECTION | Пользователь может одновременно выбирать несколько элементов списка, но только смежных |
| MULTIPLE_INTERVAL_SELECTION | Самый гибкий режим, используемый моделью выделения по умолчанию. Позволяет выбирать произвольное количество элементов списка в произвольном сочетании (в любом месте списка, поодиночке или интервалами) |

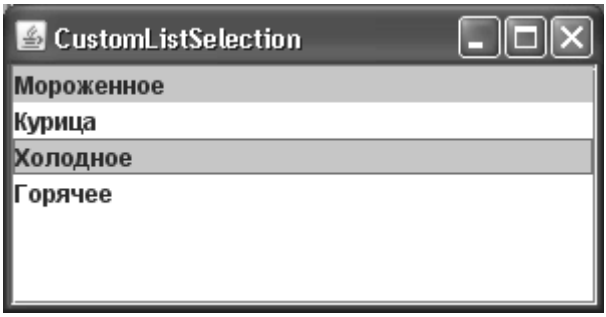
Запустив программу с примером, вы сможете оценить разницу в различных режимах выделения. Один элемент может быть выбран щелчком мыши или нажатием клавиш управления курсором, а интервалы и дополнительные элементы могут быть выбраны с помощью вспомогательных клавиш Shift и Ctrl. Стандартными внешними видами Swing поддерживаются и многие другие клавиши, особенно удобные для навигации в больших списках, их описание вы сможете найти в интерактивной документации Java.


Возможностей стандартной модели выделения и поддерживаемых ею трех режимов выделения с лихвой хватает для большей части приложений. Впрочем, иногда для максимального удобства пользователя бывает неплохо встроить в список особенный режим выделения, который каким-либо образом учитывает смысл отображаемых в списке элементов и при выделении одного элемента тут же предлагает пользователю выбрать еще один. Писать для таких ситуаций модель выделения «с нуля» неудобно и утомительно, а вот унаследовать от стандартной модели и немного вмешаться в ее работу может оказаться проще. Давайте рассмотрим небольшой пример:

```
// CustomListSelection.java
// Реализация особого режима выделения
import javax.swing.*;
import java.awt.*;

public class CustomListSelection extends JFrame {
  private String[] data = { "Мороженное", "Курица",
      "Холодное", "Горячее"};
  public CustomListSelection() {
    super("CustomListSelection");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // настроим список и добавим его в окно
    JList list = new JList(data);
    list.setSelectionModel(new CustomSelectionModel());
    add(new JScrollPane(list));
    // выведем окно на экран
    setSize(300, 200);
    setVisible(true);
  }
  // специальная модель выделения
  class CustomSelectionModel
```

```
      extends DefaultListSelectionModel {
    // добавление интервала выделения
    public void addSelectionInterval(int idx0, int idx1) {
      super.addSelectionInterval(idx0, idx1);
      adjustSelection(idx0);
    }
    // установка интервала выделения
    public void setSelectionInterval(int idx0, int idx1) {
      super.setSelectionInterval(idx0, idx1);
      adjustSelection(idx0);
    }
    // общий метод "подкрутки" выделения
    private void adjustSelection(int idx) {
      if ( idx == 0 )
        addSelectionInterval(2, 2);
      if ( idx == 1 )
        addSelectionInterval(3, 3);
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
             public void run() { new CustomListSelection(); } });
  }
}
```

В этом примере мы добавляем в центр окна список с элементами массива data. В списке нет ничего экстраординарного, за исключением специальной модели выделения CustomListSelectionModel, реализованной в виде внутреннего класса. Мы унаследовали эту модель от стандартной модели DefaultListSelectionModel, и переопределили два самых полезных для нас метода: метод addSelectionInterval() добавляет к выделению новый интервал, а метод setSelectionInterval() заменяет текущий интервал выделения новым (хотя на самом деле выделение меняет UI-представитель списка, когда пользователь щелкает мышью или нажимает клавиши выделения, но сейчас это не так важно). При вызове списком JList данных методов мы, во-первых, вызываем их базовые версии, так что выделение элементов идет обычным порядком, во-вторых, проверяем, какой именно элемент выбирается, в специальном методе adjustSelection(). Если это нужные нам первый или второй элемент списка, мы дополнительно к ним выделяем (программно, методом addSelectionInterval()) третий и четвертый элементы списка, соответственно. Это намек пользователю на оптимальное сочетание — мороженное должно быть холодным, а курица горячей. Хотя если ему больше по нраву холодная курица, он может и не согласиться с нашим вторым мнением и убрать не понравившийся ему выбранный элемент. В этом наша специальная модель выделения ему не мешает. Запустив программу с примером, вы увидите все своими глазами.




Подобное поведение можно было бы реализовать и более простым и интуитивно понятным способом: присоединением к модели выделения слушателя и выделением в нужной ситуации дополнительных элементов списка. Но у показанного подхода есть свои преимущества: вся работа по выделению сосредоточена в одном месте, а именно в модели выделения, так что вы можете легко настраивать и обновлять эту модель и ее поведение, применять ее к разным спискам, которых в приложении может быть множество (не забывайте, что модели могут присоединяться к любому числу видов, и модель выделения не является исключением).

Кроме того что модель выделения без труда настраивается и имеет широкие возможности, она позволяет легко получить выделенные в данный момент элементы списка. Для этого можно воспользоваться методами самой модели выделения или, как это обычно делается в Swing, аналогами этих методов, определенными в непосредственно списке JList (вы выбираете тот подход, который вам удобнее и лучше вписывается в концепцию вашего кода). Также в списке определены несколько очень удобных методов, позволяющих получить массив выбранных элементов или массив позиций этих элементов (табл. 11.2).

**Таблица 11.2.** Методы для получения информации о выделенных элементах

| Метод | Назначение |
|---|---|
| isSelectedIndex() | Метод модели выделения, который позволяет узнать, выбран ли некоторый элемент списка |
| getSelectedIndex(), getSelectedIndices() | Эти методы определены в списке JList и позволяют получить позицию первого выделенного элемента (если ничего не выделено, возвращается —1) или массив позиций выделенных элементов |
| getSelectedValue(), getSelectedValues() | Эти методы также определены в списке JList, с их помощью получают сам выбранный элемент списка или массив, составленный из всех выбранных элементов списка |

Используя описанные методы, вы сможете получить полную информацию о выборе пользователя.

## Внешний вид списка

До сих пор мы выясняли, какие данные показывают списки и как эти данные подготавливать и использовать, а затем получать информацию о выделенных элементах. Теперь мы можем посмотреть, что позволяют изменить списки в своем внешнем виде. Оказывается, изменить можно практически все.

Как и у всех компонентов Swing, унаследованных от базового класса JComponent, у списка JList есть возможность менять цвет фона (список является непрозрачным, его свойство opaque равно true, и как мы знаем из главы 4, это означает, что занимаемая им площадь закрашивается цветом фона), цвет шрифта и сам шрифт. Чтобы не ограничивать вашу фантазию, список также позволяет при необходимости изменить цвет выделенного элемента и сам цвет выделения. Свойства, позволяющие это сделать, кратко описаны в табл. 11.3.

**Таблица 11.3.** Свойства, меняющие внешний вид списка

| **Свойство (и методы get/set)** | **Описание** |
|---|---|
| background, foreground, font | Стандартные свойства любого компонента Swing. Позволяют задать фон, цвет шрифта и сам шрифт, соответственно |
| selectedBackground | Позволяет изменить цвет, которым прорисовывается фон выделенных элементов |
| selectedForeground | Определяет цвет шрифта выделенного элемента |

С помощью этих свойств вы сможете легко окрасить списки в любимые цвета, однако увлекаться этим занятием не стоит. Приложение чаще всего должно иметь выдержанный в едином стиле внешний вид, и по умолчанию UI-представители рисуют списки теми цветами, что больше всего соответствуют используемому менеджеру внешнего вида. Если вам нужен новый вид списков, вы можете написать собственного UI-представителя списков или переопределить «рисующий» метод paintComponent(), что позволит со вкусом приукрасить списки, к примеру, градиентными заливками. Однако чаще всего нужно по-особому нарисовать элементы списка.

Оказывается, список рисует содержащиеся в нем элементы не сам, он делегирует эту работу специальному объекту, реализующему интерфейс ListCellRenderer. Если вы вспомните методы модели ListModel, поставляющей списку данные для вывода на экран, то поймете, что списку элементы передаются в виде ссылок на базовый класс Object, то есть список способен «показывать» объекты любого типа. Как он показывает эти объекты, определяет его помощник по рисованию элементов ListCellRenderer. По умолчанию в списке используется стандартный рисующий объект DefaultListCellRenderer, который на самом деле представляет собой подкласс хорошо знакомой нам надписи JLabel. Объект DefaultListCellRenderer способен отображать значки Icon, а все остальные объекты, поставляемые моделью, он отображает в виде строк (вызывая для них метод toString()). При этом прорисовка элементов списка максимально оптимизируется, а все «ненужные» механизмы надписи, такие как оповещения об изменении текста и значка (привязанные свойства JavaBeans), автоматическая проверка корректности и другие, отключаются[4].

Реализовать прорисовывающий элементы списка объект несложно — в интерфейсе ListCellRenderer описан всего один метод getListCellRendererComponent(). Этот метод требу-

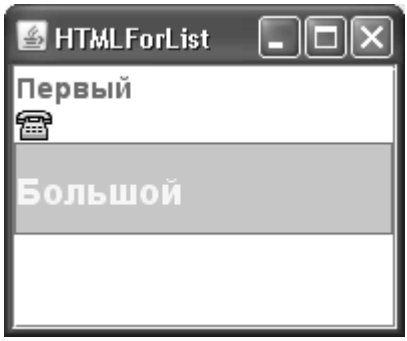
[4] Интересно, что на этом оптимизация для рисующих объектов ListCellRenderer не заканчивается. Список JList (а также, например, таблицы и деревья, которые делегируют работу по прорисовке элементов другим объектам) прорисовывает их на специальном «холсте» — особом компоненте CellRendererPane. Этот компонент отключает дополнительные возможности рисования (например, двойную буферизацию Swing) и встроенную проверку корректности компонентов. Создатели Swing выяснили, что элементы в списках (а также таблицах и деревьях) рисуются так часто, что обычный способ прорисовки приводит к значительному снижению быстродействия, и встроили в библиотеку особые механизмы работы с такими элементами.

ет от вас вернуть для определенного элемента списка (для которого еще известны его позиция в списке, а также наличие фокуса и его выделения) графический компонент Component — он и будет использован для прорисовки элемента в списке. Прежде чем пробовать свои силы в написании собственного отображающего объекта, можно до конца исследовать возможности стандартного объекта DefaultListCellRenderer. Нам известно, что это надпись JLabel, а с надписями мы работать умеем, и в том числе знаем о поддержке ими HTML. Иногда этого, вкупе с поддержкой значков, может быть достаточно:

```
// HTMLForList.java
// Использование стандартного объекта
// DefaultListCellRenderer
import javax.swing.*;
import java.awt.*;

public class HTMLForList {
  // данные списка
  private static Object[] data = {
      "<html><font size=4 color=red>Первый",
      new ImageIcon("bullet.gif"),
      "<html><h2><font color=yellow>Большой"
  };
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() {
            // создаем список
            JList list = new JList(data);
            // помещаем его в окно
            JFrame frame = new JFrame("HTMLForList");
            frame.setDefaultCloseOperation(
                JFrame.EXIT_ON_CLOSE);
            frame.setSize(200, 200);
            frame.add(new JScrollPane(list));
            frame.setVisible(true);
          } });
  }
}
```

Здесь мы просто создаем окно со списком, который наполняем данными из массива объектов. Отличительная особенность примера — в самих данных, которые должен будет отображать список. Вы видите, что для двух элементов мы используем некоторые возможности языка разметки HTML (и не забывайте о все тех же «волшебных» символах <html> в начале строки), а еще один элемент задаем в виде значка ImageIcon, получающего изображение из файла. Запустив программу с примером, вы увидите, что отображением элементов занимается на самом деле надпись: и HTML-текст, и значок выводятся правильно и в тоже время являются элементами списка. Продемонстрированных возмож-

ностей объекта DefaultListCellRenderer обычно вполне хватает для создания эффектных списков в вашем приложении. Единственное нарекание к внешнему виду — все же списки нам привычнее, когда все их элементы имеют одинаковую высоту. Чтобы исправить это, мы можем использовать уже знакомое нам свойство fixedCellHeight, и указать единую высоту для всех элементов, достаточную для их корректного отображения.




Тем не менее, создать собственный объект для отображения элементов списка также несложно и иногда удобно. Главное, что нужно при этом помнить, — список JList не размещает создаваемые вашим рисующим объектом компоненты на экране, он использует их как «кисти», прорисовывая их содержимое на положенном месте списка. Ни событий от пользователя, ни перехода в видимое состояние нет и не будет: просто вызывается метод paint() вашего компонента. Стандартный объект DefaultListCellRenderer использует этот факт: для всех элементов списка, какие бы данные они не содержали, имеется только один экземпляр надписи JLabel, который заново настраивается (меняется текст или значок) для каждого отображаемого элемента списка[5].

Несмотря на всю прелесть поддержки надписями языка HTML, мы еще в главе 8 узнали, что загружать с его помощью изображения неудобно. Давайте создадим библиотечный класс для отображения элементов списка, который будет способен отображать одновременно значок и текст (в том числе и HTML-текст). Чтобы не писать все «с нуля», а использовать достоинства стандартного класса DefaultListCellRenderer (он хорошо оптимизирован), унаследуем от него. Ну, а прежде всего, необходимо создать класс, который будет хранить значок и текст для нового рисующего объекта:

```
// com/porty/swing/ImageListElement.java
// Класс, хранящий значок и текст элемента
package com.porty.swing;

import javax.swing.*;

public class ImageListElement {
  // значок и текст
  private Icon icon;
```

[5] Такое использование надписи JLabel хорошо описывается шаблоном проектирования приспособленец (flyweight). Суть этого шаблона проста: некоторых объектов (надписей) нужно много (множество элементов списка), а это дорого и расточительно, поэтому вместо множества объектов остается один, состояние которого меняется в зависимости от контекста (отображаемого элемента), в котором он используется.

```
    private String text;
   // удобный конструктор
   public ImageListElement(Icon icon, String text) {
       this.icon = icon;
       this.text = text;
   }
    // методы для доступа к значку и тексту
    public Icon getIcon() {  return icon; }
    public String getText() { return text; }
    public void setIcon(Icon icon) { this.icon = icon; }
    public void setText(String text) { this.text = text; }
}
```

Это просто композиция из значка Icon и строки String с текстом. Чтобы было проще создавать этот объект, мы добавили удобный конструктор, позволяющий быстро задать нужные значения. Теперь нам понадобится рисующий объект, который будет прорисовывать для списка элементы ImageListElement:

```
// com/porty/swing/ImageListCellRenderer.java
// Класс для прорисовки в списке одновременно
// значка и текста
package com.porty.swing;

import javax.swing.*;
import java.awt.*;

public class ImageListCellRenderer
    extends DefaultListCellRenderer {
  // метод, возвращающий для элемента рисующий компонент
  public Component getListCellRendererComponent(
      JList list, Object data, int idx, boolean isSelected,
      boolean hasFocus) {
    // проверяем, нужного ли элемент типа
    if ( data instanceof ImageListElement ) {
      ImageListElement lie =
          (ImageListElement)data;
      // получаем текст и значок
      Icon icon = lie.getIcon();
      String text = lie.getText();
      // используем возможности базового класса
      JLabel label = (JLabel)
          super.getListCellRendererComponent(
              list, text, idx, isSelected, hasFocus);
      label.setIcon(icon);
```

```
        return label;
      } else
        return super.getListCellRendererComponent(
            list, data, idx, isSelected, hasFocus);
    }
  }
```

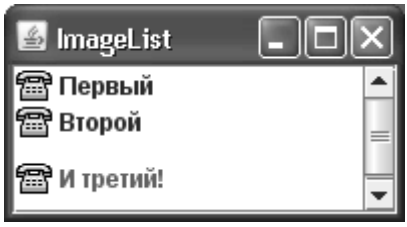
Мы наследуем класс ImageListCellRenderer от стандартного объекта DefaultListCellRenderer, и это дает нам возможность не беспокоиться о правильной прорисовке текста надписи с помощью установленных для списка цветов. Вместо этого мы можем сосредоточиться на обработке элементов нужного нам типа (а именно ImageListElement). Для начала мы выясняем, относится ли элемент списка к нужному типу, и если это так, начинаем его обработку. Прежде всего вызывается метод getListCellRendererComponent() базового класса DefaultListCellRenderer, но не для всего объекта ImageListElement, а только для хранящегося в нем текста (text). В результате мы получаем надпись JLabel (а мы знаем, что всегда получаем надпись от базового класса, поэтому преобразование типов безопасно) с установленным текстом. Нам остается лишь добавить к этому тексту наш значок и вернуть списку готовую надпись. Если же в списке окажется элемент незнакомого нам типа, мы просто перепоручим его обработку своему базовому классу.

Наконец, мы можем проверить работу нашего нового объекта для отображения элементов списка. Давайте попробуем:

```
// ImageList.java
// Список, использующий новый рисующий объект
import javax.swing.*;
import com.porty.swing.*;

import java.awt.*;

public class ImageList {
  // данные списка
  private static Icon bullet = new ImageIcon("bullet.gif");
  private static Object[] data = {
      new ImageListElement(bullet, "Первый"),
      new ImageListElement(bullet, "Второй"),
      new ImageListElement(bullet,
          "<html><h4><font color=green>И третий!")
  };
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() {
            // создаем список и настраиваем его
            JList list = new JList(data);
            list.setCellRenderer(new ImageListCellRenderer());
            // добавляем в окно
```

```
            JFrame frame = new JFrame("ImageList");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setSize(200, 200);
            frame.add(new JScrollPane(list));
            frame.setVisible(true);
          } });
    }
}
```

Мы создаем небольшое окно со списком в панели содержимого. Список создается на основе массива данных, и на этот раз мы записываем наши данные в виде объектов ImageListElement, одновременно указывая значок и текст. Заметьте, что благодаря использованию в качестве базового класса стандартного рисующего объекта DefaultListCellRenderer, возможности HTML для списка по-прежнему доступны. Остается не забыть установить в качестве рисующего объекта списка наш новый объект ImageListCellRenderer (с помощью метода setCellRenderer()) и вывести окно на экран. Вы убедитесь в том, что теперь список показывает значок и текст одновременно, а при необходимости может выводить и HTML-текст. Не забывайте только, что созданные нами классы ImageListElement и ImageListCellRenderer находятся в пакете com.porty.swing. Они могут сослужить вам хорошую службу — списки с элементами, состоящими из значка и текста, очень популярны в приложениях.




Аналогичным образом вы можете действовать, если вам понадобятся дальнейшие изменения во внешнем виде элементов списка. Унаследуйте от стандартного рисующего объекта и добавьте к надписи то, что вам необходимо, благо ее возможности нам хорошо известны и без сомнения велики. Если же вам понадобится выводить элементы списка в совершенно особом виде, вы можете написать рисующий объект ListCellRenderer «с нуля». Пример такого объекта (для создания списка с элементами-флажками) мы вскоре увидим.

Список JList также использует интерфейс ListCellRenderer, чтобы определить, где и какие следует выводить всплывающие подсказки. Для этого он переопределяет метод базового класса JComponent getToolTipText() и в момент вывода подсказки узнает, на каком элементе списка находится указатель мыши. Для этого элемента списка он получает с помощью объекта ListCellRenderer рисующий его компонент и пытается выяснить текст его подсказки. Если у компонента подсказка есть, она выводится, а если текст подсказки получить не удалось, она не появляется вовсе. Созданный нами класс ImageListCellRenderer легко доработать для поддержки всеми элементами списка подсказок: надо лишь добавить в класс ImageListElement строку подсказки, а в классе ImageListCellRenderer дополнительно вызывать для надписи метод setToolTipText(). После этого каждый элемент списка сможет не только показывать текст и значок, но и при необходимости выводить подсказку. Учитывая, что и в элементах списка, и в самих подсказках может присутствовать HTML-текст, возможности для создания любого интерфейса открываются просто захватывающие.

## События списка

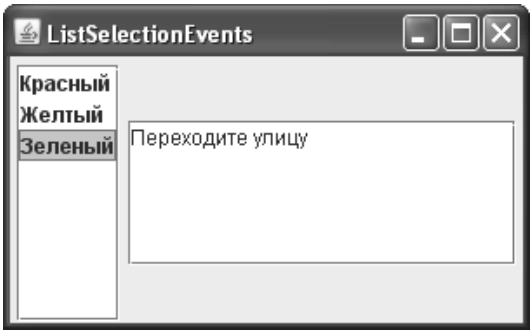
У списка JList нет собственных событий (за исключением стандартных событий, подобных событиям от мыши и клавиатуры, унаследованных им от базового компонента JComponent), все особенные события происходят в его моделях. Присоединив слушателя к модели ListModel, поставляющей данные для отображения в виде элементов списка, вы сможете узнать, когда и как меняются элементы списка и какие это элементы. Чаще всего эти события от модели обрабатывает как раз список, что позволяет ему вывести верные данные вовремя и в надлежащем виде, реализуя концепцию модели и вида. Впрочем, вы можете и сами использовать события модели списка, сделав из нее некий сообщающий об изменениях в себе контейнер данных, если задача того требует.

Гораздо более интересные события происходят в модели выделения списка ListSelectionModel; они сообщают об изменениях в выделении пользователем элементов списка. Конечно, никто не запрещает вам бесцеремонно вмешаться в работу модели выделения (мы уже так делали) и следить за выделением прямо из нее, но это сложнее и не так элегантно. Присоединив слушателя к модели выделения списка, вы сможете прямо на месте узнавать обо всех изменениях в выделенных элементах, которые делает пользователь, и в зависимости от его выбора предлагать ему дополнительную информацию или новые альтернативы. Давайте рассмотрим небольшой пример, в котором мы будем узнавать о выделенных элементах списка прямо «на лету»:

```
// ListSelectionEvents.java
// Слежение за выделением списка
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;

public class ListSelectionEvents extends JFrame {
  // данные списка
  private String[] data =
      { "Красный", "Желтый", "Зеленый"};
  private JTextArea jta;
  public ListSelectionEvents() {
    super("ListSelectionEvents");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем список и текстовое поле
    setLayout(new FlowLayout());
    JList list = new JList(data);
    list.setSelectionMode(
        ListSelectionModel.SINGLE_SELECTION);
    jta = new JTextArea(5, 20);
    // добавим слушателя событий выделения
    list.addListSelectionListener(new SelectionL());
    // добавляем компоненты
    add(new JScrollPane(list));
    add(new JScrollPane(jta));
    // выводим окно на экран
```

```
    setSize(300, 200);
    setVisible(true);
  }
  // слушатель событий от модели выделения
  class SelectionL implements ListSelectionListener {
    public void valueChanged(ListSelectionEvent lse) {
      int selected =
          ((JList)lse.getSource()).getSelectedIndex();
      switch ( selected ) {
        case 0: jta.setText("Переходить нельзя"); break;
        case 1: jta.setText("Будьте готовы"); break;
        case 2: jta.setText("Переходите улицу");
      }
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new ListSelectionEvents(); } });
  }
}
```

В примере мы добавляем в окно список на основе небольшого массива строк и текстовое поле JTextArea, которые мы будем использовать для вывода дополнительной информации. Затем к списку мы добавляем слушателя событий от модели выделения, вызывая метод addListSelectionListener(). Заметьте, что этот метод мы вызываем для самого списка JList, хотя точно такой же метод есть и в модели выделения ListSelectionModel. Разница лишь в источнике события, который будет возвращать нашему слушателю метод getSource(). При присоединении слушателя к списку (как в нашем примере) данный метод будет возвращать ссылку на него, и чаще всего это удобнее. В качестве слушателя событий будет использоваться внутренний класс SelectionL. В нем при получении извещения от модели выделения мы определяем позицию выделенного элемента списка (с помощью источника события — нашего списка) и в зависимости от этой позиции снабжаем пользователя дополнительной информацией. Запустите программу с примером и удостоверьтесь в том, что переходите улицу на нужный цвет светофора.



Очень часто слушателя событий от модели выделения достаточно для ваших нужд: вы знаете, как и когда меняются выделенные элементы списка, и можете оперативно изменить поведение программы в зависимости от выбора пользователя. Иногда никаких событий от списка вообще не обрабатывается: вы ждете нажатия кнопки или выбора пункта меню и только после этого смотрите, какие элементы списка выделены, считая их окончательным выбором пользователя.

Можно вспомнить, что в современных приложениях очень часто выбор (в том числе и в списках) легко сделать двойным щелчком мыши. Пользователь может быстро показать, что ему нужно, дважды щелкнув на некотором элементе списка. Список JList по умолчанию не поддерживает двойные щелчки мыши и не предоставляет средства по их определению, но поддержать такое поведение самостоятельно не составляет труда. Нам помогут два полезных метода списка JList: locationToIndex() и indexToLocation(). Они позволяют узнать, какому элементу списка принадлежит некоторая точка на экране, или наоборот выяснить, в каком месте экрана находится тот или иной элемент списка. С их помощью (для нашего примера понадобится первый метод) нам будет просто обеспечить поддержку двойных щелчков мыши:

```
// DoubleClickList.java
// Получение элемента списка по точке экрана
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class DoubleClickList extends JFrame {
  // данные списка
  private String[] data = { "Стейк", "Лобстер",
      "Борщ", "Севрюга"};
  private JList list;
  public DoubleClickList() {
    super("DoubleClickList");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем список
    list = new JList(data);
    list.setSelectionMode(
        ListSelectionModel.SINGLE_SELECTION);
    // добавим слушателя событий от мыши
    list.addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        if ( e.getClickCount() == 2 ) {
          // получаем элемент и покажем его
          int pos =
              list.locationToIndex(e.getPoint());
          JOptionPane.showMessageDialog(
              list, "Уже готовится: " +
                  list.getModel().getElementAt(pos));
        }
```

```
        }
      });
      // выведем окно на экран
      add(new JScrollPane(list));
      setSize(300, 200);
      setVisible(true);
    }
    public static void main(String[] args) {
      SwingUtilities.invokeLater(
          new Runnable() {
            public void run() { new DoubleClickList(); } });
    }
}
```

Мы создаем небольшой «кулинарный» список (пусть это будет меню небольшого ресторана фьюжн-кухни) и добавляем его в центр нашего окна. К списку присоединяется слушатель событий от мыши (конечно, в виде адаптера: к чему нам остальные методы слушателя), в котором мы следим за щелчками мыши на списке. Когда пользователь щелкает на списке дважды, мы получаем позицию элемента, на котором произошло это событие (как раз здесь нам и пригодится метод locationToIndex()), и сообщаем пользователю, что его выбор был получен и обработан. Поддержка двойных щелчков мыши интуитивно понятна пользователю («сильный», то есть двойной, щелчок равносилен выбору) и особенно полезна там, где выбирать можно только один элемент списка. В таких ситуациях всегда добавляйте поддержку двойных щелчков мыши: это уже становится нормой. Напоследок стоит сказать о том, что метод locationToIndex() работает в координатах списка JList. В нашем примере это не важно, потому что мы следили за щелчками мыши непосредственно в списке, но если вы присоединили слушатель мыши к другому компоненту, в котором находится список, не забудьте преобразовать позицию указателя мыши в координаты списка с помощью метода convertMouseEvent() класса SwingUtilities.

Список JList делает все, чтобы с ним можно было без проблем работать на самом низком уровне: при обработке событий от мыши и клавиатуры или в других подобных ситуациях. Он позволяет определить, какие элементы списка в данный момент видны на экране, прокрутить свое содержимое так, чтобы на экране появился определенный элемент, узнать размеры любого элемента и выяснить, на каких позициях экрана он находится. Вы можете смело манипулировать списком на всех уровнях: и посредством моделей, и посредством низкоуровневых событий. Вся необходимая информация у вас в руках.

## Список из флажков

В современных приложениях довольно часто встречаются списки, в качестве своих элементов показывающие флажки. Флажки позволяют упорядочить набор альтернатив и более наглядно представить глазам пользователя сделанный выбор. В качестве примера можно вспомнить разнообразные программы установки, в которых вы выбираете, какие компоненты устанавливаемой системы будут копироваться на вашу машину и сколько пространства на жестком диске в итоге потребуется, или всяческие «корзины покупателя», в которых аналогичным образом можно увидеть общую стоимость выбранных товаров и при необходимости от некоторых товаров тут же отказаться. Одним словом, наличие списка с флажками в вашем арсенале лишним не будет, и попробовать создать его стоит, заодно закрепив все полученные нами знания о списках JList.

Сразу ясно, что нам понадобится особый класс для хранения данных об элементе списка (включен или выключен элемент, а также строка с надписью) и специальный элемент ListCellRenderer, который будет отображать элементы списка в надлежащем виде, а именно как флажки JCheckBox. Но на этом работа не заканчивается: мы хорошо знаем, что список не помещает создаваемые объектом ListCellRenderer компоненты на экран, он всего лишь использует их как кисти для прорисовки, так что автоматически флажки устанавливаться или сбрасываться не будут (они не будут получать событий). Придется присоединить к списку слушателя событий от мыши и при щелчке сменять состояние элемента (включенное на выключенное и наоборот), чтобы флажок правильно отображал его. Давайте приступим. Сначала самое простое — класс с данными элементов нашего будущего списка:

```
// com/porty/swing/CheckBoxListElement.java
// Данные элемента списка с флажками
package com.porty.swing;

public class CheckBoxListElement {
  // данные элемента
  private boolean selected;
  private String text;
  // удобный конструктор
  public CheckBoxListElement(boolean selected, String text){
    this.selected = selected;
    this.text = text;
  }
  // методы для доступа к значку и тексту
  public boolean isSelected() {  return selected; }
  public String getText() { return text; }
  public void setSelected(boolean selected) {
    this.selected = selected;
  }
  public void setText(String text) { this.text = text; }
}
```

В классе CheckBoxListElement (мы поместили его в библиотечный пакет com.porty.swing) находятся данные для списка с флажками: признак выделения элемента и строка с текстом, которая этот элемент будет описывать. Для удобства был добавлен конструктор, который позволяет быстро задать нужные значения элемента. Теперь нужно написать отображающий объект ListCellRenderer, который на основе этого класса должен создавать подходящий флажок. Мы не станем писать его в виде отдельного класса (вряд ли он будет использоваться где-то еще помимо списка с флажками, потому что ему требуется только определенный тип данных), а сделаем частью класса нашего нового списка с флажками (опишем его в виде статического внутреннего класса). Вот что получится:

```
// CheckBoxList.java
// Список с флажками
package com.porty.swing;
```

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;

public class CheckBoxList extends JList {
  // сохраняем все конструкторы
  public CheckBoxList(ListModel model) {
    super(model);
    initList();
  }
  public CheckBoxList(Object[] data) {
    super(data);
    initList();
  }
  public CheckBoxList(Vector data) {
    super(data);
    initList();
  }
  // специальная настройка списка
  private void initList() {
    setCellRenderer(new CheckBoxCellRenderer());
    setSelectionMode(
        ListSelectionModel.SINGLE_SELECTION);
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        // следим за щелчками
        if ( e.getClickCount() == 1 &&
            SwingUtilities.isLeftMouseButton(e) ) {
          // нужный нам щелчок
          int pos =
              locationToIndex(e.getPoint());
          CheckBoxListElement cbel =
              (CheckBoxListElement)getModel().
                  getElementAt(pos);
          cbel.setSelected(! cbel.isSelected());
          // заново рисуем только эту ячейку списка
          repaint(getCellBounds(pos, pos));
        }
      }
    });
```

```
    }
    // отображающий флажки объект
    public static class CheckBoxCellRenderer
        extends JCheckBox implements ListCellRenderer {
      public Component getListCellRendererComponent(
          JList list, Object data, int idx,
          boolean isSelected, boolean hasFocus) {
        // полагаем, что данные всегда нужного типа
        CheckBoxListElement
            cbel = (CheckBoxListElement)data;
        // настраиваем флажок
        if ( isSelected ) {
          setBackground(list.getSelectionBackground());
          setForeground(list.getSelectionForeground());
        } else {
          setBackground(list.getBackground());
          setForeground(list.getForeground());
        }
        setSelected(cbel.isSelected());
        setText(cbel.getText());
        return this;
      }
    }
}
```

Мы создаем список с флажками CheckBoxList на основе обычного списка JList, наследуя от него, так что все возможности обычных списков остаются с вами. Прежде всего обратите внимание на реализацию специального рисующего объекта CheckBoxCellRenderer. Он унаследован от флажка JCheckBox и реализует необходимый для всех рисующих объектов списка интерфейс ListCellRenderer. Для отображения элементов списка требуется только один флажок — собственно сам рисующий объект. Он приводит нуждающийся в прорисовке элемент списка к нужному типу CheckBoxListElement (мы полагаем, что в списке с флажками будут элементы только такого типа, в противном случае возникнет исключение при приведении типов) и производит настройку флажка. Так как в этот раз нам приходится создавать рисующий объект «с нуля», нужно самостоятельно позаботится о соответствующих цветах элемента списка (которые могут меняться пользователем или текущим менеджером внешнего вида и поведения). Далее, используя специальные данные класса CheckBoxListElement, мы устанавливаем состояние флажка и текст для него.

Теперь остается только сам список с флажками. Он почти не отличается от обычного списка JList, за исключением того, что отдельно следит за нажатиями мыши, сразу же устанавливает специальный рисующий объект CheckBoxCellRenderer и по умолчанию задействует режим одиночного выделения элементов (это удобно для пользователя — он может сосредоточиться на смене состояния нужных ему элементов, а смена состояния возможна только для одного флажка одновременно). Мы сохранили все три конструктора обычного списка JList, так что вы не заметите разницы между функционированием обычного и нового списков CheckBoxList. Все что необходимо присоединяется в методе

initList(). Отдельного внимания заслуживает только специальный слушатель событий от мыши, реализованный прямо на месте. При подходящем событии (одно нажатие левой кнопкой мыши[6]) слушатель с помощью модели получает элемент, на котором было произведено нажатие, и меняет его состояние. Также он посылает запрос на перерисовку списка (repaint()), чтобы тот обновил свои элементы, заметьте, как мы эффективно это делаем, перерисовывая только один, изменившийся, элемент списка, определить его прямоугольник на экране помогает метод списка getCellBounds(). Это очень важно для списков больших размеров, со сложными рисующими объектами, иначе перерисовка будет слишком медленной. И здесь мы подразумеваем, что все элементы в списке имеют одинаковый тип CheckBoxListElement, так что других элементов лучше в него не помещать.

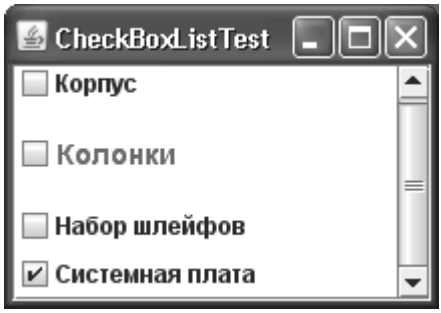
Мы сделали основу для правильной работы списка с флажками, и теперь нам остается только проверить его в деле. Проиллюстрирует все небольшой пример:

```
// CheckBoxListTest.java
// Проверка работы списка с флажками
import javax.swing.*;
import com.porty.swing.*;
import java.awt.*;

public class CheckBoxListTest {
  // данные списка
  private static Object[] data = {
      new CheckBoxListElement(false, "Корпус"),
      new CheckBoxListElement(false,
          "<html><h3><font color=red>Колонки"),
      new CheckBoxListElement(false, "Набор шлейфов"),
      new CheckBoxListElement(true, "Системная плата")
  };
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() {
            // создаем список и настраиваем его
            JList list = new CheckBoxList(data);
            // добавляем в окно
            JFrame frame = new JFrame("CheckBoxListTest");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setSize(200, 200);
```

[6] Обратите внимание, что в слушателе мы переопределяем метод mousePressed(), вызываемый при нажатиях кнопок мыши. Если бы мы выбрали метод mouseClicked (то есть щелчок), то у нас могли бы возникнуть небольшие затруднения. Щелчком мыши считается событие нажатия и отпускания кнопки мыши на одном и том же месте экрана. Если пользователь при этом хотя бы немного шевелит мышью, щелчка не получится. Все зависит от того, как конкретный пользователь работает с мышью, но иногда непредсказуемость щелчков мыши сильно раздражает, так что мы выбрали нажатия.

```
                frame.add(new JScrollPane(list));
                frame.setVisible(true);
            } });
    }
}
```

Мы создаем небольшое окно JFrame, в панель содержимого которого добавляем наш новый список CheckBoxList, вложенный в панель прокрутки. Для списка CheckBoxList годятся только данные типа CheckBoxListElement, их мы и подготавливаем в массиве data: задаем начальное состояние элемента (флажок может быть установленным или сброшенным) и надпись к нему. Если вспомнить все, что мы узнали о флажках JCheckBox в главе 9, то не останется сомнений, что флажки в Swing тоже поддерживают HTML. Вы видите, как мы используем HTML-текст для второго элемента списка. Запустив программу с примером, вы увидите хорошо знакомый по многим приложениям список с флажками, который к тому же способен задействовать всю мощь HTML.




Чаще всего к такому списку присоединяется слушатель событий от модели выделения, чтобы при выборе пользователем нового флажка что-то изменялось в выводимой программой информации, например, заново подсчитывалось количество свободного места на диске, требуемого для установки. Но мы и так уже хорошо знаем, как работать с событиями от модели выделения, так что для простоты в примере мы этого не делали.

Конечно, это лишь начальная заготовка для полнофункционального списка со флажками, который будет легко вписываться в любой отшлифованный интерфейс. К примеру, у нас не поддерживается выделение флажков с клавиатуры, нет эффекта «нажимания» флажка, а стоило бы его добавить, чтобы флажок не отличался от своих существующих на самом деле на экране собратьев, на элементе не прорисовываются фокус ввода. Но это все легко добавить всего несколькими мазками богатой палитры Swing.

Действительно, список JList позволяет тонко настраивать любые аспекты своей работы. Вы можете начать с модели данных, по своему вкусу выбирая источник и тип данных, которые должен будет отображать список. Совсем нетрудно изменить внешний вид списка, начиная с используемых им цветов и заканчивая любым нужным вам видом его элементов. Он позволяет работать с собой посредством низкоуровневых событий и полностью контролировать процесс выделения элементов. Список с флажками хорошо иллюстрирует всю гибкость списка JList — несколькими строками кода вы фактически создаете новый элемент пользовательского интерфейса, не прикладывая для этого титанических усилий.

## Списки JList в Java 7

Новая версия платформы Java 7 (или JDK 1.7) привнесла в списки JList некоторые изменения. Функциональность списков и все то, что мы рассмотрели в данной главе, осталось неизменным, а вот способ работы с элементами списка в модели ListModel стал несколько иным. Начиная с Java 7, списки JList являются типизированными компонентами, то есть при их создании вы можете указать точный тип, который они хранят. То же самое относится и к модели списков ListModel. Без сомнения это удобно, вспоминая наши специальные объекты для списков со значками и списков с флажками, где нам приходилось постоянно заниматься ручным приведением типов. В Java 7, задав тип хранимого в списке объекта, мы не только упрощаем реализацию, но и запрещаем хранение в списке объектов неподходящих типов еще на этапе компиляции. Типизированным стал и объект для отображения элементов списка ListCellRenderer, что также позволит избавиться от проверки конкретных типов в его коде.

## *Раскрывающиеся списки JComboBox*

Раскрывающиеся списки JComboBox служат для выбора одного из множества доступных вариантов, примерно так же, как это происходит в группе переключателей JRadioButton или выключателей JToggleButton. В отличие от групп кнопок, которые располагаются на экране все вместе, в раскрывающемся списке JComboBox виден только сам выбранный элемент, а список возможных альтернатив появляется, только когда этого захочет пользователь (щелкнув на специальной кнопке раскрытия списка). Список возможных альтернатив выводится в специальном всплывающем меню, которое после выбора скрывается. Таким образом раскрывающиеся списки (в отличие от групп элементов управления и обычных списков) позволяют экономить место в контейнере, даже если список альтернатив велик. Кроме того, раскрывающиеся списки допускают редактирование текущего элемента, и пользователь может не только выбрать что-то среди имеющихся альтернатив, но и ввести свое собственное значение. К тому же Swing автоматически делает работу с раскрывающимся списком весьма комфортной: список JComboBox обладает встроенной возможностью поиска элементов с клавиатуры, и как мы вскоре увидим, это значительно упрощает работу с большими списками.

В отличие от обычного списка JList, раскрывающийся список JComboBox вполне обходится услугами одной модели, поставляющей ему информацию об элементах списка; обязанности этой модели описаны в интерфейсе ComboBoxModel. На самом деле, вряд ли имело бы смысл использовать для раскрывающегося списка отдельную модель выделения — в любом случае в нем может быть только один выделенный элемент. Создатели Swing учли это и максимально упростили нам знакомство с моделью раскрывающегося списка ComboBoxModel: во-первых, эта модель унаследована от уже знакомой нам очень простой модели списка ListModel, во-вторых, в ней появилась лишь пара новых методов, предназначенных для смены выделенного элемента или его получения. Помимо интерфейса модели ComboBoxModel вы можете использовать унаследованный от нее интерфейс MutableComboBoxModel, который поддерживает вставку и удаление произвольных элементов списка.

Но прежде чем обратиться к моделям раскрывающегося списка, давайте посмотрим, как его проще всего создать и какие свойства чаще всего настраиваются при его создании. Вот простой пример:

```
// SimpleComboBoxes.java
// Создание простых раскрывающихся списков
import javax.swing.*;
import java.util.*;
```

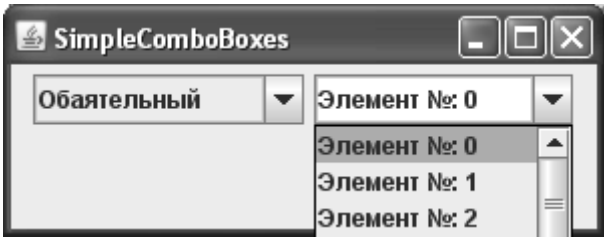
```
import java.awt.*;

public class SimpleComboBoxes extends JFrame {
  // массив с элементами списка
  public String[] elements = new String[] {
      "Обаятельный", "Умный", "Нежный", "Сильный"
  };
  public SimpleComboBoxes() {
    super("SimpleComboBoxes");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создадим пару раскрывающихся списков
    JComboBox combo1 = new JComboBox(elements);
    // задаем прототип элемента списка
    combo1.setPrototypeDisplayValue("Длинный элемент");
    // второй раскрывающийся список
    Vector<String> data = new Vector<String>();
    for (int i=0; i<10; i++)
      data.add("Элемент №: " + i);
    JComboBox combo2 = new JComboBox(data);
    // сделаем его редактируемым
    combo2.setEditable(true);
    combo2.setMaximumRowCount(6);
    // добавим списки и выведем окно на экран
    setLayout(new FlowLayout());
    add(combo1);
    add(combo2);
    setSize(300, 200);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new SimpleComboBoxes(); } });
  }
}
```

В примере мы создаем небольшое окно и размещаем в его панели содержимого пару раскрывающихся списков. Нетрудно заметить очевидное сходство раскрывающегося списка JComboBox с обычным списком JList: точно так же, как это было для него, вы можете задавать элементы раскрывающегося списка в конструкторе JComboBox в виде массива или динамического массива (вектора) Vector. Первый список мы создаем на основе массива строк elements. Его нужно просто передать в конструктор, который незаметно для нас создаст на его основе стандартную модель. Заметьте, как мы изменяем для первого списка свойство prototypeDisplayValue. С его помощью список получает возможность

определить, какой длины должны быть все элементы списка независимо от того, находятся они в списке сразу же или добавляются потом. Данное свойство особенно полезно в том случае, если вы не знаете заранее, какой длины элементы появятся в списке со временем. Задав прототип элемента достаточно большим, можно не опасаться, что добавляемые впоследствии длинные элементы будут отображаться неверно. Более того, наличие рядом с элементами списка небольшого свободного пространства эстетически воспринимается лучше, чем плотно упакованный набор таких элементов.

Второй раскрывающийся список демонстрирует работу с вектором Vector. Мы динамически наполняем вектор простыми строками, а затем передаем его в конструктор JComboBox. Во многих ситуациях, если вам не нравится модель, использование класса Vector — намного более гибкий способ быстро создать раскрывающийся список (впоследствии содержимое вектора гораздо проще изменить). Второй список мы делаем редактируемым, а также изменяем для него свойство maximumRowCount. Это свойство контролирует, сколько элементов будет одновременно выводиться во всплывающем меню списка. Если это значение меньше количества элементов в списке (а у нас так и есть: в списке десять элементов, а свойство мы устанавливаем равным шести), то список элементов во всплывающем меню автоматически включается в панель прокрутки.

Оба созданных нами раскрывающихся списка помещаются окно, после чего оно выводится на экран. Запустив программу с примером, вы сможете увидеть внешний вид раскрывающихся списков и оценить, как отразились на них сделанные нами изменения в свойствах.




Продемонстрированные в примере конструкторы и свойства раскрывающихся списков часто позволяют создать нужный вам список буквально за мгновение, так что стоит иметь их в виду. Обратите также внимание, что раскрывающийся список JComboBox обладает встроенной поддержкой клавиатуры для поиска элементов: раскрыв первый список и нажимая соответствующие элементам списка символьные клавиши, вы убедитесь в том, что соответствующие элементы выделяются. Встроенный поиск элементов с клавиатуры незаменим в больших раскрывающихся списках с множеством элементов — поиск элементов в них только с помощью мыши и кнопок прокрутки легко превращается в настоящую головную боль. Поддержка поиска с клавиатуры осуществляется внутренним классом JComboBox.KeySelectionManager, в случае необходимости вы можете заменить используемый списком экземпляр этого класса собственным.

## Модель ComboBoxModel

Для нас уже не секрет, что как бы быстро ни позволяли создавать раскрывающиеся списки их конструкторы, по настоящему гибкой и удобной в работе программу делают модели. Мы вскользь упоминали, что данные раскрывающихся списков хранит модель, описываемая несложным интерфейсом ComboBoxModel. Давайте познакомимся с этой моделью поближе.

Прежде всего стоит отметить, что модель ComboBoxModel, как и модель обычного списка ListModel, очень проста. Раскрывающийся список имеет много общего со списком обычным (он также хранит некоторую последовательность элементов, правда, в отличие от обычного списка, выбирать можно только один элемент), так что во избежание лишней работы создатели раскрывающегося списка решили унаследовать модель ComboBoxModel от модели обычного списка ListModel. Если вспомнить модель ListModel, то у нас в распоряжении оказывается четыре унаследованных метода: два для добавления и удаления заинтересованных в изменениях модели слушателей, один для получения элемента на некоторой позиции и еще один для получения количества хранящихся в списке элементов. Кроме того, в интерфейсе ComboBoxModel появилось пара новых методов: getSelectedItem() — для получения выбранного в данный момент элемента и setSelectedItem() — для смены выбранного элемента. У раскрывающегося списка JComboBox нет отдельной модели выделения (было бы слишком расточительно и сложно использовать отдельную модель для единственного выбранного элемента), так что информацию о выделенном элементе раскрывающийся список с помощью двух новых методов сохраняет в модели ComboBoxModel вместе с информацией обо всех своих элементах.

Создавать модель вы можете не только с помощью интерфейса ComboBoxModel, помимо него раскрывающийся список может использовать модель MutableComboBoxModel. Она унаследована от обычной модели ComboBoxModel и дополнительно описывает четыре метода, позволяющие динамически изменять содержимое раскрывающегося списка. С помощью этих методов вы сможете добавлять новые элементы (к концу списка или на произвольные позиции) или удалять уже имеющиеся элементы[7].

Прежде чем начать обдумывать, как реализовать собственную модель для раскрывающегося списка, вы можете обратить внимание на стандартную модель DefaultComboBoxModel. Она унаследована от вспомогательного класса AbstractListModel, позволяющего упростить создание моделей для обычных списков JList, и реализует интерфейс MutableComboBoxModel. В качестве хранилища для элементов списка в ней используется динамический массив (вектор) Vector[8]. Стандартная модель предоставляет вам соответствующий набор методов, так что вы практически не заметите разницы между работой с ней и с динамическим массивом, а все достоинства модели останутся между тем с вами. Давайте рассмотрим небольшой пример, в котором применим стандартную модель раскрывающегося списка:

```
// UsingComboBoxModel.java
// Использование стандартной модели раскрывающихся списков
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class UsingComboBoxModel extends JFrame {
```

[7] Появление для раскрывающегося списка отдельной «изменяемой» модели довольно странно и контрастирует с обычным списком JList, который обладает лишь одной «неизменяемой» моделью. Логичнее было бы предположить, что свойство «изменяемости» находится во власти конкретной модели, как это и было со списком JList. Похоже, что классы JComboBox и JList писали разные люди, и это впечатление только усиливается при взгляде на методы этих классов. Нам же от этого только сложнее знакомиться с очень похожими классами, работа с которыми «по идее» должна быть почти одинаковой.

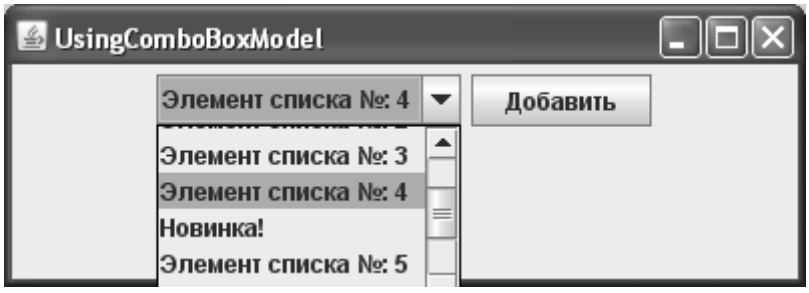
[8] Да-да, библиотека Swing так до сих пор и не удосужилась перейти на новую библиотеку коллекций, появившуюся в Java 2, и повсюду все еще используют часто становившийся объектом для нападок и споров довольно медленный динамический массив Vector. К счастью, это один из немногих очевидных просчетов создателей Swing.

```
  // наша стандартная модель
  private DefaultComboBoxModel cbm;
  public UsingComboBoxModel() {
    super("UsingComboBoxModel");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем стандартную модель и наполняем ее данными
    cbm = new DefaultComboBoxModel();
    for (int i=0; i<10; i++) {
      cbm.addElement("Элемент списка №: " + i);
    }
    // сменяем выбранный элемент
    cbm.setSelectedItem(cbm.getElementAt(4));
    // список на основе нашей модели
    JComboBox combo = new JComboBox(cbm);
    combo.setMaximumRowCount(5);
    // стандартная модель позволяет
    // динамически манипулировать данными
    JButton add = new JButton("Добавить");
    add.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        // случайно выбираем позицию
        int pos = (int) (Math.random()*cbm.getSize());
        cbm.insertElementAt("Новинка!", pos);
      }
    });
    // добавляем список и кнопку в панель
    setLayout(new FlowLayout());
    add(combo);
    add(add);
    // выводим окно на экран
    setSize(400, 200);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new UsingComboBoxModel(); } });
  }
}
```

В приведенном примере в окно добавляется раскрывающийся список JComboBox и простая кнопка JButton. Раскрывающийся список создается на основе стандартной модели DefaultComboBoxModel, которую мы создаем и наполняем данными еще до при-

соединения к раскрывающемуся списку. Как вы помните, это общее правило для всех стандартных моделей (чаще всего использующих для хранения данных динамические массивы Vector): основную часть данных следует добавлять в модель до присоединения к виду (или нескольким видам), потому что после присоединения модель начинает оповещать вид о каждом изменении в своих данных, что может значительно замедлить работу приложения, особенно если вид уже выведен на экран. Итак, мы наполняем модель элементами в цикле, используя для этого метод addElement() — полный аналог одноименного метода класса Vector. Заметьте, как модель позволяет сменить выбранный элемент с помощью метода setSelectedItem(). Правда, использовать его нужно немного необычно: метод требует передавать ему не позицию, а сам элемент (это необходимо для поддержки редактируемых списков, выбранным элементом в которых может быть любое значение). Поэтому нам приходится сначала получить элемент на нужной позиции (с помощью все той же модели), а затем только выбрать его. После этого мы присоединяем модель к списку JComboBox, передавая ее в конструктор (то же самое можно было бы сделать методом setModel()). Для удобства работы со списком (длинные выпадающие меню довольно неудобны, особенно если им не хватает места на экране) мы ограничили количество видимых на экране элементов меню пятью, изменив уже известное нам свойство maximumRowCount.

Для демонстрации динамической работы с нашей стандартной моделью мы добавили в окно кнопку add, слушатель ActionListener которой при щелчке на кнопке вводит в модель новый элемент, причем на случайную позицию, так что вы точно будете знать, что изменения происходят динамически. Запустив приложение, вы сможете увидеть, как стандартная модель поставляет данные раскрывающемуся списку и при этом позволяет без особых трудностей динамически изменять его содержимое. Созданную модель вы сможете разделять между несколькими видами и отдельно обрабатывать хранящиеся в ней данные.




Раскрывающийся список хранит довольно простые данные, и к тому же ему не нужна отдельная модель выделения (выделенный элемент хранится вместе с остальными элементами списка), так что талантов стандартной модели вполне хватает для большинства ситуаций. Создавать собственную модель стоит, только когда данные хранятся в особом источнике, требующем особого подхода, например, когда возможна некоторая оптимизация при получении и обработке данных. Мы уже создали для обычного списка JList модель, работающую с базами данных JDBC, давайте сделаем то же самое и для раскрывающегося списка JComboBox. Такая модель на самом деле будет очень полезна и может стать многократно использующимся элементом любого приложения: как только вам понадобится наполнить раскрывающийся список элементами из базы данных, вы сможете сделать это буквально парой строк кода. Удивительно, но создатели раскрывающегося списка не удосужились обеспечить нас абстрактным классом с поддержкой слушателей вроде AbstractComboBoxModel, хотя такие классы есть для всех остальных компонентов

с моделями. Создавать «модель» с нуля излишне сложно, вместо этого унаследуем ее от стандартной модели DefaultComboBoxModel, определив нужные нам методы. (Еще одним вариантом, если вы не хотите использовать стандартную модель, может быть наследование от базового класса модели списка AbstractListModel вместе с реализацией интерфейса модели выпадающего списка.) Давайте попробуем:

```
// com/porty/swing/DatabaseComboBoxModel.java
// Модель раскрывающегося списка для работы с базами данных
package com.porty.swing;

import javax.swing.*;
import java.sql.*;

public class DatabaseComboBoxModel
    extends DefaultComboBoxModel {
  // получение данных из запроса ResultSet
  public void setDataSource(ResultSet rs, int column)
      throws SQLException {
    // очистим список
    removeAllElements();
    // добавим новые элементы из базы данных
    while ( rs.next() ) {
      // получаем строки из столбца column
      addElement(rs.getString(column));
    }
  }
}
```

Здесь мы пошли по пути наименьшего сопротивления, максимально опираясь на уже имеющиеся возможности стандартной модели DefaultComboBoxModel. Мы не стали менять хранилище для данных и переопределять методы модели, а просто добавили новый метод setDataSource(), который получает результат запроса к базе данных ResultSet, удаляет все имеющиеся в модели элементы и заполняет ее новыми элементами из базы данных, получая их из столбца с определенным номером (его нужно указать в качестве второго параметра того же метода setDataSource()). Благодаря максимальному использованию механизмов стандартной модели, все делается за нас: при добавлении новых элементов методом addElement() модель будет оповещать об этом присоединенных слушателей, так что пользователь сможет своими глазами увидеть, как происходит заполнение раскрывающегося списка. Заметьте, что класс новой модели мы разместили в пакете com.porty.swing, откуда его проще извлечь.

Проверим работу нашей новой модели. Давайте напишем небольшое приложение, которое присоединяется к некоторой базе данных JDBC и полученные из нее данные показывает в раскрывающемся списке JComboBox. Вот что получается:

```
// DatabaseComboBox.java
// Пример использования модели DatabaseComboBoxModel
import javax.swing.*;
import com.porty.swing.DatabaseComboBoxModel;
```

```
import java.sql.*;
import java.awt.*;

public class DatabaseComboBox extends JFrame {
  // параметры подключения к базе данных
  private static String
      dsn = "jdbc:odbc:Library",
      uid = "",
      pwd = "";
  // наша модель
  private DatabaseComboBoxModel cbm;
  public DatabaseComboBox() {
    super("DatabaseComboBox");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // настраиваем соединение с базой данных
    Connection conn = null;
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      // объект-соединение с базой данных
      conn = DriverManager.getConnection(dsn, uid, pwd);
      Statement st = conn.createStatement();
      ResultSet rs = st.executeQuery(
          "select * from readers.csv");
      // передаем данные в модель
      cbm = new DatabaseComboBoxModel();
      cbm.setDataSource(rs, 2);
      rs.close();
    } catch (Exception ex) {
      throw new RuntimeException(ex);
    }
    // присоединяем модель к списку
    setLayout(new FlowLayout());
    add(new JComboBox(cbm));
    // выводим окно на экран
    setSize(300, 200);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new DatabaseComboBox(); } });
```

```
        }
    }
```

Роль базы данных у нас играет некоторая гипотетическая база данных JDBC для библиотеки книг без имени пользователя и пароля. Для присоединения к ней нам нужно, прежде всего, загрузить драйвер JDBC, мы задействуем стандартный драйвер JDBC-ODBC, поставляющийся вместе с пакетом JDK. После загрузки драйвера мы можем воспользоваться услугами класса DriverManager для создания соединения Connection с базой данных, и в случае успеха через созданное соединение выполнять запросы и получать данные. Полученные в результате запроса данные (в виде объекта ResultSet) мы передаем в нашу новую модель DatabaseComboBoxModel (дополнительно нужно указать номер столбца, данные которого станут элементами списка), после чего остается присоединить ее к раскрывающемуся списку, добавить последний в окно и вывести его на экран. Для запуска программы со своей базой данных вам нужно соответствующим образом изменить название базы данных, имя пользователя и пароль, указать другой драйвер, если это необходимо для подключения к вашей базе данных, и выбрать таблицу, в которой имеется столбец с текстовыми данными, указав ее в тексте запроса. Запустив правильно настроенную программу с примером, вы сможете увидеть, как раскрывающийся список выводит элементы из базы данных, на самом деле даже и не подозревая об этом: все детали по сбору и обработке данных, как это и должно быть, остаются в модели. Не забудьте, что манипулировать любыми моделями в Swing после их присоединения к компонентам можно только из потока рассылки событий, так как они напрямую влияют на визуальное состояние компонентов.

Продуманное использование специальных моделей, таких как DatabaseComboBoxModel, придает вашему приложению удивительную гибкость и легкость в расширении и поддержке. Вы сможете тщательно отделить части, работающие с данными (это части, соединяющиеся с базами данных и передающие результаты моделям), от частей, заведующих пользовательским интерфейсом (это компоненты пользовательского интерфейса, отображающие данные моделей и незаметно для вас позволяющие пользователю полностью изменять их, если это необходимо). Одну и ту же модель можно задействовать в нескольких частях сложного пользовательского интерфейса, не заботясь о правильном отображении данных, а при обработке итоговых данных знать, что все нужное скрыто именно в ней.

## Внешний вид списка

Внешний вид элементов, хранящихся в раскрывающемся списке JComboBox, также легко и полно настраивается, как и внешний вид элементов обычного списка JList. Более того, для вывода элементов обоих списков используется отображающий объект ListCellRenderer, знакомый нам по списку JList. На самом деле, было бы странно вводить для раскрывающегося списка новый отображающий объект — как и в случае со списком обычным, от него требуется лишь вывести некоторый элемент, находящийся на определенной позиции списка (про этот элемент также известно, выделен ли он и есть ли у него фокус ввода). Более того, список элементов в выпадающем списке выводит именно обычный список JList, обернутый в панель прокрутки и выведенный на экран во всплывающем окне.

Таким образом, все, что мы говорили про внешний вид обычных списков JList, в равной степени относится и к раскрывающимся спискам JComboBox. Поэтому мы со спокойной душой можем считать, что уже в совершенстве владеем самой изощренной техникой отображения элементов раскрывающегося списка (если вы пропустили раздел, посвященный внешнему виду списков JList, самое время к нему вернуться). Давайте рассмотрим пример, в котором используем возможности как стандартного отобра-

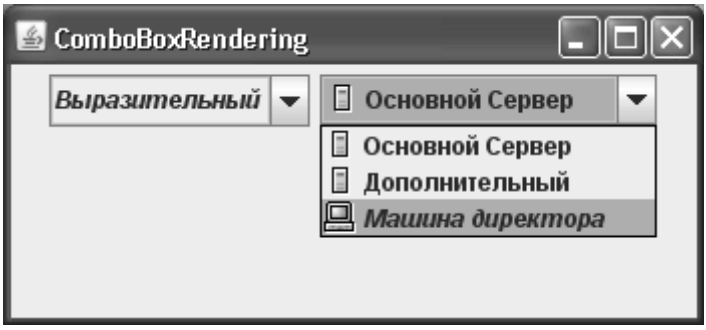
жающего объекта списков DefaultListCellRenderer, так и нашего специального объекта ImageListCellRenderer, способного отображать одновременно значки и текст. Итак:

```
// ComboBoxRendering.java
// Отображение элементов раскрывающегося списка
import javax.swing.*;
import com.porty.swing.*;

import java.awt.*;

public class ComboBoxRendering extends JFrame {
  // данные для первого списка
  private String[] textData = { "<html><code>Первый",
      "<html><b>Жирный", "<html><font color=red>Красный",
      "<html><em>Выразительный" };
  // значки
  private ImageIcon bullet1 =
      new ImageIcon("Server16.gif");
  private ImageIcon bullet2 =
      new ImageIcon("Host16.gif");
  // данные для второго списка со значками
  private ImageListElement[] iconData = {
      new ImageListElement(bullet1, "Основной Сервер"),
      new ImageListElement(bullet1, "Дополнительный"),
      new ImageListElement(bullet2,
          "<html><b><em>Машина директора")
  };
  public ComboBoxRendering() {
    super("ComboBoxRendering");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем пару раскрывающихся списков
    JComboBox combo1 = new JComboBox(textData);
    JComboBox combo2 = new JComboBox(iconData);
    // наш специальный отображающий объект
    combo2.setRenderer(new ImageListCellRenderer());
    // добавляем списки в окно
    setLayout(new FlowLayout());
    add(combo1);
    add(combo2);
    // выводим окно на экран
    setSize(350, 200);
    setVisible(true);
```

```
    }
    public static void main(String[] args) {
      SwingUtilities.invokeLater(
          new Runnable() {
            public void run() { new ComboBoxRendering(); } });
    }
}
```

В данном примере мы создаем два несложных раскрывающихся списка JComboBox, используя для хранения данных два массива textData и iconData. Первый список, создаваемый на основе массива строк textData, демонстрирует возможности стандартного отображающего объекта списков DefaultListCellRenderer. Как вы помните, стандартный отображающий объект унаследован от обычной надписи JLabel и способен отображать не только обычный текст, но и текст в формате HTML, а также значки. Значки мы для первого списка не используем, оставляя их для списка второго, а вот возможности HTML применяем «что есть сил», так что элементы первого списка спутать друг с другом будет непросто. Второй список использует в качестве данных объекты ImageListElement, созданные нами в разделе «Списки JList» и позволяющие хранить вместе и значок, и текст элемента списка, причем текст все также может быть приукрашен всеми доступными нам средствами HTML. Не забывайте только при применении объектов ImageListElement устанавливать подходящий отображающий объект ImageListCellRenderer методом setRenderer()[9].

После создания списков и незамысловатой настройки мы добавляем их в окно и выводим его на экран. Запустив программу с примером, вы сможете убедиться, что элементы списка теперь выглядят совсем иначе.




У вашей фантазии нет никаких ограничений: все, что связано с выводом элементов списка на экран, прорисовывается в отображающем объекте. Написав собственный отображающий объект, примерно так, как мы это сделали для объекта с текстом и значками ImageListCellRenderer, вы сможете любым образом расцветить свои списки. Более того, отображающие объекты «развязывают вам руки» при работе с моделями списка, позволяя хранить в них данные совершенно произвольных типов, даже самых сложных и запутанных. Предоставив в дополнение к таким данным отображающий объект, который знает, что именно из этих данных следует вывести на экран, вы получаете возможность

[9] И снова путаница: почему в обычном списке метод для смены отображающего объекта называется setCellRenderer(), а в списке JComboBox — setRenderer()? Похоже, что при создании двух похожих списков их разработчики не удосужились как следует скооперироваться. К счастью, в библиотеке Swing таких «проколов» мало, и работать с ней все же приятно.

использовать модель не только как «подносчик снарядов» для отображения элементов списка, но и как универсальное хранилище любых данных (которые к тому же легко разделять между видами и показывать пользователю).

## Редактирование

Как мы уже знаем, раскрывающийся список JComboBox позволяет пользователю не только выбирать что-либо в списке заранее имеющихся альтернатив, но и вводить собственные значения — это его основное отличие от обычного списка JList и его основное преимущество. Правда, по умолчанию раскрывающийся список редактирования не допускает, но в нескольких предыдущих примерах мы уже увидели, как включить для списка режим редактирования: этим управляет свойство editable. Установив последнее в true, вы предоставляете пользователю возможность самого широкого выбора: он сможет либо выбрать один из представленных в списке элементов, либо ввести собственное значение. Обратите внимание на маленький нюанс: редактировать элементы раскрывающегося списка *нельзя*, они всегда остаются теми, какими были при создании списка, можно лишь дополнительно к ним вводить новые значения.

Редактированием нового значения в списке JComboBox занимается отдельный объект, обязанности его описаны в интерфейсе ComboBoxEditor. Стандартная реализация этого интерфейса — класс с названием BasicComboBoxEditor из пакета javax.plaf.basic. В данном классе для редактирования применяется простое текстовое поле JTextField, которое прекрасно подходит для обычных строк[10]. Чаще всего возможностей стандартного объекта для редактирования вполне достаточно для обычных раскрывающихся списков, не использующих в качестве элементов какие-либо экзотические данные. Вы могли убедиться в этом на предыдущих примерах, где в дополнение к имеющимся в списке строкам можно было ввести собственную произвольную строку текста. Увы, но стандартный объект для редактирования хорошо справляется лишь с простыми строками: стоит только элементам списка стать немного сложнее, как идиллия нарушается. Даже такая несложная возможность, как обработка HTML-текста, уже не поддерживается стандартным объектом, который выводит его в виде кода с тегами, а внезапное предложение заняться написанием HTML-текста вручную вряд ли восхитит пользователя. Поэтому, если вы храните в раскрывающемся списке нестандартные данные, особым образом показываете их (даже если это «до боли» знакомый нам HTML-текст) и планируете сделать список редактируемым, придется заняться написанием своего объекта для редактирования, благо это не слишком сложно. В качестве варианта можно попробовать потоньше настроить стандартный отображающий объект BasicComboBoxEditor, «вырезая» из элемента списка все HTML-теги при передаче его для редактирования. Для этого будет достаточно унаследовать от него и обрабатывать проходящие через редактор элементы.

Впрочем, никто не заставляет нас оставаться в рамках стандартного объекта для редактирования и лишь немного настраивать его работу. Совсем несложно написать собственный объект для редактирования с нуля; давайте так и поступим, предоставив в качестве альтернативы простому HTML-коду, «очищенному» от тегов, объект для редактирования, позволяющий редактировать именно HTML-текст, со всеми его красочными атрибутами и широкими возможностями. С редактированием HTML нам поможет текстовый компонент JEditorPane, специально для этого и предназначенный. Нам остается только «втиснуть» его в рамки интерфейса ComboBoxEditor. Оказывается, это очень просто. Итак:

[10] Нетрудно понять, что объект для редактирования занимается не только собственно редактированием, но и отображает выбранный элемент, постоянно находясь на экране. За отображение остальных элементов, находящихся в выпадающем меню, как и прежде отвечает объект ListCellRenderer.

```
// com/porty/swing/HTMLComboBoxEditor.java
// Полнофункциональный редактор для списка
// JComboBox, использующего HTML
package com.porty.swing;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class HTMLComboBoxEditor
    implements ComboBoxEditor {
  // редактор для HTML
  private JEditorPane htmlEditor;
  public HTMLComboBoxEditor() {
    htmlEditor =
        new JEditorPane("text/html", "");
    htmlEditor.setBorder(BorderFactory.
        createEtchedBorder());
  }
  // возвращает редактор
  public Component getEditorComponent() {
    return htmlEditor;
  }
  // сигнал выбрать весь текст и приступить
  // к редактированию
  public void selectAll() {
    htmlEditor.selectAll();
    htmlEditor.requestFocus();
  }
  // возвращает редактируемый элемент
  public Object getItem() {
    return htmlEditor.getText();
  }
  // получает новый элемент для редактирования
  public void setItem(Object item) {
    htmlEditor.setText(item.toString());
  }
  // методы для присоединения слушателей
  // ActionListener, который оповещаются
  // об окончании редактирования
  public void addActionListener(
      ActionListener e) { }
```

```
    public void removeActionListener(
        ActionListener e) { }
}
```

Новый редактор для раскрывающегося списка с поддержкой HTML мы разместили в пакете com.porty.swing, так что при случае вам будет несложно использовать его в собственных приложениях. Для того чтобы некий класс смог действовать как редактор для списка JComboBox, ему необходимо реализовать интерфейс ComboBoxEditor. Интерфейс этот несложен, в нем четыре основных метода и еще пара методов служит для присоединения слушателей ActionListener. Наш новый объект для редактирования реализует данный интерфейс и работает следующим образом: метод getEditorComponent() возвращает компонент, который раскрывающийся список выведет на экран рядом с кнопкой, открывающей выпадающее меню, именно этот компонент и должен выполнять редактирование. В нашем классе это текстовый компонент JEditorPane, мы создаем и настраиваем его в конструкторе (указываем, что редактироваться будет HTML-текст и применяем специальную рамку, чтобы он не выглядел чересчур бледно). Метод setItem() вызывается раскрывающимся списком при смене элемента для редактирования, получаемый элемент мы преобразуем в строку и сразу же передаем компоненту JEditorPane, подразумевая, что все элементы списка записаны с помощью HTML-кода. Метод selectAll() является сигналом редактору выделить всю область редактирования и приступить к работе. В нем мы выделяем весь текст и на всякий случай запрашиваем в редактор фокус ввода, чтобы пользователь видел, что поле готово к вводу данных. Наконец, метод getItem() предназначен для возвращения набранного пользователем значения, мы возвращаем текст, находящийся в данный момент в редакторе.

Интерфейс ComboBoxEditor обязывает нас реализовать еще два метода, служащие для присоединения и отсоединения слушателей ActionListener. Данные слушатели должны оповещаться об окончании ввода в редакторе, так что программист-клиент раскрывающегося списка сможет узнать, когда пользователь заканчивает ввод нового значения. Стандартный объект для редактирования использует текстовое поле JTextField, а оно, как мы узнаем из главы 16, позволяет присоединять к себе слушателей ActionListener, которые оповещаются об окончании ввода. Эта возможность поля JTextField и применяется стандартным объектом для поддержки слушателей ActionListener. Увы, но текстовый редактор JEditorPane не позволяет легко узнать об окончании ввода, потому что он рассчитан на ввод неограниченного количества текста, так что мы оставляем наш редактор без поддержки слушателей. Это не слишком страшно — слушатели ActionListener применяются при работе с JComboBox не так уж и часто, к тому же вы будете знать о том, что на них не следует полагаться при работе с нашим новым редактором. Более того, при потере выпадающим списком фокуса ввода он сам генерирует искусственное событие ActionEvent, говорящее о новом значении в списке, что тоже неплохо. Событие не будет возникать лишь от самого редактора.

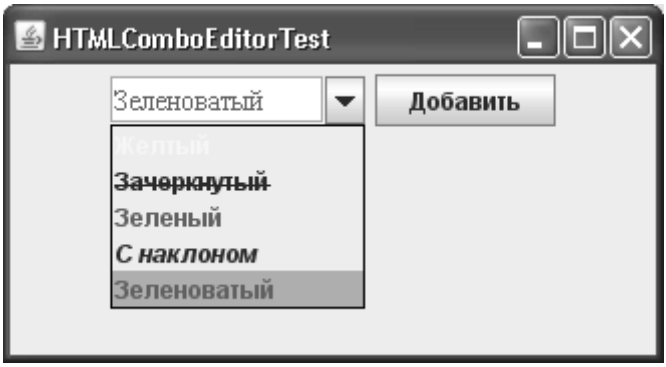
Ну а теперь посмотрим, как будет работать созданный своими руками объект для редактирования:

```
// HTMLComboEditorTest.java
// Пример использование специального объекта для
// редактирования
import javax.swing.*;
import com.porty.swing.HTMLComboBoxEditor;

import java.awt.*;
import java.awt.event.ActionListener;
```

```
import java.awt.event.ActionEvent;

public class HTMLComboEditorTest
    extends JFrame {
  // данные для раскрывающегося списка
  private String[] data = {
      "<html><font color=yellow>Желтый",
      "<html><strike>Зачеркнутый",
      "<html><font color=green>Зеленый",
      "<html><em>С наклоном" };
  public HTMLComboEditorTest() {
    super("HTMLComboEditorTest");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем список
    final JComboBox combo = new JComboBox(data);
    combo.setPrototypeDisplayValue("11223344556677");
    combo.setEditable(true);
    combo.setEditor(new HTMLComboBoxEditor());
    // добавляем список в окно
    setLayout(new FlowLayout());
    add(combo);
    // кнопка для добавления нового элемента в список
    JButton addButton = new JButton("Добавить");
    addButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        combo.addItem(combo.getSelectedItem());
      }
    });
    add(addButton);
    // выводим окно на экран
    setSize(330, 200);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new HTMLComboEditorTest(); } });
  }
}}
```

Мы создаем раскрывающийся список, используя для записи элементов HTML-код, и устанавливаем для списка новый объект для редактирования (не забывая включить редактирование свойством editable). Обратите внимание на маленький нюанс: длина

элемента раскрывающегося списка с помощью свойства prototypeCellValue делается довольно большой. Этому есть простое объяснение: HTML-текст, выводимый в выпадающем меню, выглядит гораздо компактнее, чем в редакторе JEditorPane, и если не сделать размер списка побольше (с помощью подходящего менеджера расположения или, как в примере, с помощью свойства prototypeCellValue), некоторые элементы при редактировании рискуют оказаться на нескольких строках, а это может быть неожиданностью для пользователя. Запустив программу с примером, вы сможете оценить новый объект для редактирования. Без сомнения, он прибавляет раскрывающемуся списку красок, к тому же с ним вы можете использовать для элементов все возможности HTML, не опасаясь неприятностей при редактировании.




Пример также демонстрирует, как получать только что набранную пользователем информацию, если выпадающий список редактируемый. При потере списком фокуса ввода (переходе его на другие компоненты) значение из редактора автоматически становится текущим элементом списка, которое мы можем получить методом getSelectedItem(). Чтобы убедиться в этом, мы тут же добавляем новое, только что набранное, значение в сам список. Но, как мы уже упоминали, наш редактор не поддерживает события ActionEvent, поэтому если вы хотите получить текущее значение из редактора без потери фокуса ввода списком (к примеру, проверяя его на корректность), лучше воспользоваться методом редактора getItem().

Аналогичным образом вы сможете применять для редактирования и другие компоненты, хотя чаще всего все сводится к использованию тех или иных текстовых компонентов Swing. Так вы сможете хранить, отображать и редактировать данные любых типов, и сделаете свои раскрывающиеся списки по-настоящему приятными в работе, а комфортная работа с пользовательским интерфейсом — одна из основных задач при его создании.

## События раскрывающегося списка

Раскрывающийся список JComboBox обладает куда более интересным набором событий в сравнении с обычным списком JList, где они происходят в основном в моделях. Прежде всего стоит обратить внимание на событие ItemEvent, которое происходит при смене выбранного элемента списка. С помощью этого события можно узнать не только о том, что выбранный элемент сменился, но и быстро получить данный элемент, а также узнать, какой элемент был выбран перед ним. Далее следует уже знакомое нам событие ActionEvent, которое должно сообщать о новом выбранном элементе или конце редактирования (если список допускает редактирование). Правда, после написания собственных редакторов для раскрывающегося списка мы видели, что правильная работа этого

события в значительной степени зависит от реализации объекта для редактирования, который может и не поддерживать слушателей окончания ввода (а раскрывающийся список присоединяет слушателей ActionListener именно к объекту для редактирования). При использовании нестандартных редакторов следует иметь это в виду и при необходимости применять для подтверждения ввода отдельные элементы интерфейса (как правило, кнопки).

Применение событий раскрывающегося списка проиллюстрирует небольшой пример:

```
// ComboBoxEvents.java
// События раскрывающихся списков
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class ComboBoxEvents extends JFrame {
  // данные для списков
  private String[] data = { "США", "Италия",
      "Швейцария", "Таиланд" };
  public ComboBoxEvents() {
    super("ComboBoxEvents");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // первый список
    JComboBox combo1 = new JComboBox(data);
    // слушатель смены выбранного элемента
    combo1.addItemListener(new ItemListener() {
      public void itemStateChanged(ItemEvent e) {
        // выясняем, что случилось
        if ( e.getStateChange() ==
            ItemEvent.SELECTED ) {
          // покажем выбранный элемент
          Object item = e.getItem();
          JOptionPane.showMessageDialog(
              ComboBoxEvents.this, item);
        }
      }
    });
    // список, позволяющий редактирование
    final JComboBox combo2 = new JComboBox(data);
    combo2.setEditable(true);
    // слушатель окончания редактирования
    combo2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
```

```
            // покажем выбор пользователя
            Object item = combo2.getModel().
                getSelectedItem();
            JOptionPane.showMessageDialog(
                ComboBoxEvents.this, item);
          }
        });
        // добавим списки в окно
        setLayout(new FlowLayout());
        add(combo1);
        add(combo2);
        // выведем окно на экран
        setSize(350, 250);
        setVisible(true);
      }
      public static void main(String[] args) {
        SwingUtilities.invokeLater(
            new Runnable() {
              public void run() { new ComboBoxEvents(); } });
      }
    }
```

В примере мы создаем пару раскрывающихся списков, используя в качестве данных массив строк String. Один из списков будет поддерживать редактирование, именно к нему мы присоединяем слушателя событий ActionListener, чтобы узнавать об окончании редактирования (здесь задействован стандартный объект для редактирования, так что это событие будет обрабатываться). К первому списку мы присоединяем слушателя событий ItemListener, метод itemStateChanged() этого слушателя будет вызываться при смене выбранного элемента списка, причем происходить такой вызов будет два раза: первый раз событие сообщит о том, что прежде выбранный элемент больше таковым не является, а второй раз оно позволит узнать все о новом выбранном элементе. Различить два типа события позволяет метод getStateChange(), возвращающий целое число, которое и указывает на тип события: первому типу соответствует константа DESELECTED из класса ItemEvent, второму типу соответствует константа SELECTED. Нас интересует выбранный элемент, так что мы выбираем событие типа SELECTED. Далее остается получить выбранный элемент методом getItem() и вывести его на экран (используя стандартное диалоговое окно JOptionPane, подробнее о нем рассказывается в главе 14). Экспериментируя с примером, попробуйте выбрать уже выбранный элемент — событие ItemEvent при этом не рассылается, хотя вы заново выбираете его в списке. Это некоторого рода оптимизация.

Ко второму списку (тому, что допускает редактирование) мы присоединяем слушателя ActionListener, он дает возможность узнать, когда пользователь заканчивает ввод нового значения, *или* когда он выбирает его из списка. После работы со вторым раскрывающимся списком вы убедитесь в том, что слушатель ActionListener оповещается не только об окончании ввода (окончанием ввода считается нажатие клавиши Enter в поле для редактирования), но и о смене выбранного элемента. В слушателе мы, как и для события ItemEvent, показываем новый выбранный элемент в стандартном диалоговом окне

JOptionPane. Если вы введете в поле для редактирования новое значение, то стандартное окно с сообщением увидите два раза — как сигнал о конце редактирования, и как сигнал о том, что в списке в качестве текущего значения используется новый элемент. Как видно, слушатель ActionListener более универсален, особенно при использовании его в списках с поддержкой редактирования, у слушателя ItemEvent перед ним лишь одно преимущество: он позволяет узнать о том, какой элемент был выбран перед новым выделенным элементом.

Чтобы подвести итог, составим маленькое резюме по не всегда очевидному действию событий выпадающих списков:

**Таблица 11.4.** Основные события выпадающего списка

| Свойство (и методы get/set) | Описание |
|---|---|
| ItemEvent | Возникает только при смене элемента. Не позволяет узнать о выборе того же самого элемента. Помогает узнать не только о выбранном, но и о предыдущем значении (выбор с которого перешел к новому элементу) |
| ActionEvent | Возникает при выборе элемента, даже если выбранный элемент не изменился (повторный выбор). Это позволяет применять данное событие для немедленного анализа выбранного значения, даже если оно не поменялось, и предпринять какие-то действия, например, обновить интерфейс, но следует помнить, что событие возникнет и от редактора списка, если будет введено новое значение и нажата клавиша (как правило Enter) или список потеряет фокус ввода. Таким образом, для редактируемых выпадающих списков событие стоит применять крайне осторожно. |

У раскрывающегося списка есть еще одно событие — PopupMenuEvent. Если говорить точнее, это событие выпадающего меню списка, с его помощью вы сможете узнать, в какой момент времени меню появляется на экране и исчезает с него. Во всех стандартных внешних видах, поставляемых вместе с JDK, UI-представитель JComboBox применяет в качестве всплывающего меню хорошо известный нам компонент JPopupMenu, к нему и присоединяется слушатель событий PopupMenuListener.

## Управление всплывающим меню

Класс JComboBox предоставляет нам некоторые методы, позволяющие ограниченно управлять своим всплывающим меню. Метод hidePopup() скрывает меню с экрана даже если пользователь не собирался его закрывать. У метода hidePopup() есть «брат-близнец» showPopup(); как нетрудно догадаться, он выводит всплывающее меню на экран. Кстати, функции обоих методов с успехом может выполнить и метод setPopupVisible() — передавая ему соответствующие булевы значения, вы сможете вывести меню на экран или скрыть его.

Кроме того, вы сможете рекомендовать раскрывающемуся списку JComboBox использовать в качестве контейнера для всплывающего меню легковесный компонент (один из тех, что затем разместятся в слое POPUP_LAYER многослойной панели) или его тяжеловесного собрата (окно без рамки JWindow или тяжеловесной панели Panel). По умолчанию всегда используется легковесный компонент (тяжеловесный компонент применяется, только если легковесному компоненту не хватает места в многослойной панели), однако вызвав метод setLightweightPopupEnabled(false), вы сможете рекомендовать списку применение тяжеловесных компонентов. В обычных ситуациях

это не оправдано и приведет лишь к снижению производительности, смысл в таком поведении имеется, только когда вы совмещаете в одном окне тяжеловесные и легковесные компоненты и хотели бы, что всплывающие меню оказались там, где им положено — над всеми остальными компонентами, даже если среди них есть тяжеловесные.

## *Резюме*

Списки Swing позволяют вывести на экран любой набор элементов, причем выводить их можно в любом виде, даже самом вычурном: прорисовка элементов списка целиком находится в ваших руках. Пользователь, в свою очередь, сможет без труда выбрать любые интересующие его элементы списка и сообщить об этом программе.

# Глава 12. Диапазоны значений

В этой главе мы познакомимся с компонентами, которые позволяют пользователю выбрать данные из некоторого диапазона или видеть, какая часть данных из некоторого диапазона обработана. Начнем мы с ползунка JSlider, который в наглядной форме представляет пользователю возможность плавного изменения выбранного значения, от минимального до максимального. Передвигая по шкале регулятор (специальный элемент интерфейса), пользователь, как правило, в реальном времени видит результат своих действий и может быстро оценить, какое именно значение из заданного диапазона подходит ему более всего.

Далее мы перейдем к изучению индикатора процесса JProgressBar, позволяющего предоставить пользователю информацию о выполнении какого-либо процесса в приложении в виде полосы, закрашенная область на которой увеличивается по мере выполнения процесса. Индикаторы процесса очень удобны для восприятия и дают пользователю возможность мгновенно оценить, на каком этапе выполнения находится интересующий его процесс и сколько времени он примерно может занять.

В завершение мы остановимся на счетчике JSpinner. Счетчик представляет пользователю набор из нескольких значений, который он в поиске нужного может «прокручивать» в обе стороны. Это своеобразный симбиоз раскрывающегося (JComboBox) и простого (JList) списков — вы сразу же можете прокручивать доступные альтернативы, как в обычном списке, не вызывая на экран всплывающее окно, и у вас остаются возможности раскрывающегося списка, как-то редактирование элемента и единственно возможный выбор. Есть у счетчика и свое уникальное свойство — в отличие от списков количество элементов в нем может быть неограниченным. Особенно полезен счетчик там, где по каким-то причинам нет места для списков или они нежелательны.

## *Ползунки JSlider*

Итак, как мы уже отметили, ползунок JSlider позволяет плавно изменять выбираемое в некотором диапазоне значение с помощью специального регулятора, перетаскивая который мышью по шкале с метками пользователь и производит выбор. Идея ползунков навеяна разного рода техническими устройствами; параметры этих устройств регулируются разнообразными ручками, которые вы можете вращать (или двигать, в зависимости от конструкции), выбирая подходящее вам значение. Например, можно вращать регулятор громкости, при этом ваш выбор ограничен: есть начальное значение (нет звука) и конечное (максимальная мощность колонок). Как раз выбор такого рода и позволяют делать ползунки JSlider.

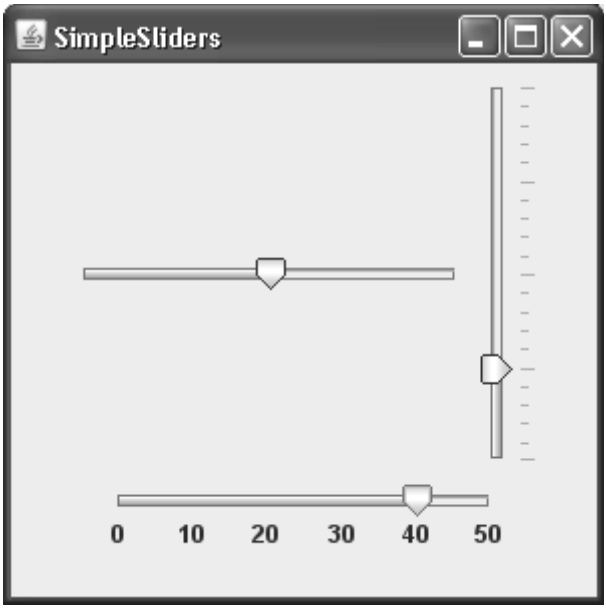
Для работы ползунка требуется специальная модель BoundedRangeModel, хранящая информацию об ограниченном наборе данных. Данные в ней хранятся в числовом виде в четырех целых (int) числах. Это минимальное значение (свойство с названием minimum), максимальное значение (maximum), текущее значение (value) и особое значение — внутренний диапазон (extent). При изменении любого из четырех значений модель запускает событие ChangeEvent, сообщающее о необходимости обновить вид (или провести другие действия). Чуть подробнее мы рассмотрим модель ограниченных данных позже, а сейчас давайте посмотрим, какие конструкторы предоставляет нам ползунок JSlider, и какие компоненты мы можем с их помощью создать. Рассмотрим простой пример:

```
// SimpleSliders.java
// Простые ползунки
import javax.swing.*;
import java.awt.*;

public class SimpleSliders extends JFrame {
  public SimpleSliders() {
    super("SimpleSliders");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем несколько ползунков
    JSlider s1 = new JSlider(0, 100);
    JSlider s2 = new JSlider(
        JSlider.VERTICAL, 0, 200, 50);
    // настройка внешнего вида
    s2.setPaintTicks(true);
    s2.setMajorTickSpacing(50);
    s2.setMinorTickSpacing(10);
    JSlider s3 = new JSlider(0, 50, 40);
    s3.setPaintLabels(true);
    s3.setMajorTickSpacing(10);
    // добавим их в панель содержимого
    setLayout(new FlowLayout());
    add(s1);
    add(s2);
    add(s3);
    // выводим окно на экран
    setSize(300, 300);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new SimpleSliders(); } });
  }
}
```

В примере мы создаем небольшое окно, в котором размещаем разнообразные ползунки. Пока модель BoundedRangeModel напрямую не используется — для демонстрации основных возможностей ползунков достаточно удобных конструкторов класса JSlider (впрочем, нетрудно догадаться, что эти конструкторы неявно задействуют стандартную модель DefaultBoundedRangeModel). Первый ползунок создается с помощью конструктора, принимающего два параметра: минимальное и максимальное значения. Текущим значением такого ползунка будет среднее (минимальное плюс максимальное разделить на два). Второй ползунок создается самым пол-

нофункциональным конструктором: он позволяет задать тип ползунка (ползунки JSlider могут быть двух видов: горизонтальными и вертикальными), минимальное и максимальное значения, а также текущее значение ползунка. Для второго ползунка мы производим настройку внешнего вида: включаем прорисовку делений (небольших штрихов, помечающих определенные значения ползунка) и определяем, через какие расстояния следует использовать большие и маленькие деления. Наконец, третий ползунок создается конструктором, позволяющим задать минимальное, максимальное и текущее значения. Для него мы включаем прорисовку надписей со значениями. Для того чтобы надписи появились, придется указать, через какие промежутки должны указываться все те же большие деления (надписи будут появляться под делениями).

Все три ползунка помещаются в окно (с последовательным расположением FlowLayout), и последнее выводится на экран. Запустив программу с примером, вы сможете увидеть, как работают и выглядят самые простые ползунки.




Обратите внимание на полную поддержку ими клавиатуры: значения ползунков можно менять, нажимая и клавиши управления курсором, и клавиши Page Up и Page Down. Где использовать вертикальные и горизонтальные ползунки, зависит от ситуации и от данных, которые они отображают. Например, громкость звука логичнее показать в виде вертикального ползунка, в то время как ползунок, позволяющий задавать расстояние, лучше изобразить горизонтальным. Иногда выбор варианта типа ползунка диктуется объемом свободного места в контейнере.

Почти всегда приходится дополнительно настраивать внешний вид ползунков, включая для них основные (большие) и промежуточные (маленькие) деления и/или метки, помечающие основные значения. Запуская программу с примером, вы могли убедиться в том, что первый ползунок (без делений и меток) выглядит весьма безлико и легко может ввести пользователя в заблуждение. Составим теперь перечень наиболее часто используемых свойств, изменяющих (путем вывода делений и меток) внешний вид ползунков (табл. 12.1).

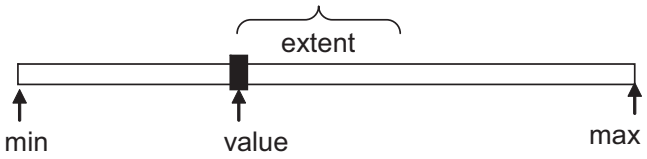
**Таблица 12.1.** Настройка отметок ползунков

| Свойство (и методы get/set) | Значение |
|---|---|
| majorTickSpacing | Позволяет задать расстояние, через которое будут рисоваться основные (большие) деления или метки (если они должны выводиться) |
| minorTickSpacing | Задает расстояние, через которое рисуются промежуточные (маленькие) деления. Логично делать это расстояние меньше, чем предыдущее, и так, чтобы между большими делениями было кратное количество маленьких делений |
| paintTicks | Включает или отключает прорисовку делений. Если вы включаете прорисовку, не забудьте настроить два предыдущих свойства (расстояния между большими и/или маленькими делениями) |
| paintLabels | Позволяет задать прорисовку меток под большими делениями. По умолчанию выводится значение, соответствующее делению, но как мы увидим далее, выводимые метки можно гибко настраивать |
| paintTrack | Управляет выводом на экран полосы (шкалы), по которой перемещается регулятор, меняющий выбранное значение. По умолчанию шкала выводится, и отключать ее не стоит — становится непонятно, к чему относится регулятор. Это свойство может быть полезным если, например, вы решите создать собственную шкалу, но не захотите писать UI-представителя ползунка «с нуля» |
| snapToTicks | Мы не использовали данное свойство в примере, но часто оно бывает весьма полезным. Если данное свойство равно true, регулятор ползунка всегда будет находиться только на метках (метки должны быть включены), «отскакивая» к ним, даже если пользователь остановит регулятор в произвольном месте шкалы. Задействовать это свойство можно там, где вы хотите применить ползунок в качестве своеобразного элемента для выбора одного из нескольких доступных вариантов (отмеченных метками) |

Используя конструкторы, описанные в примере, и свойства, перечисленные в таблице, можно получить большое количество разнообразных ползунков, вполне достаточное для большинства приложений. Впрочем, как и всегда, Swing не останавливается на полпути, и мы увидим, что ползунки способны на большее.

## Модель BoundedRangeModel

Мы уже упомянули о том, что отображаемые ползунком данные хранит модель, описываемая интерфейсом BoundedRangeModel. Она состоит из четырех целых чисел: минимального, максимального и текущего значений, а также внутреннего диапазона (extent). Первые три числа вряд ли нужно объяснять подробно, назначение их понятно, а вот зачем в модели внутренний диапазон? Внутренний диапазон начинается от текущего значения и занимает какой-то числовой интервал, хотя и не может быть больше максимального значения. Он используется не только в ползунках, но и в полосах прокрутки JScrollBar (они также хранят свои данные в модели BoundedRangeModel), где именно внутренний диапазон отвечает за длину полосы прокрутки. Грубо говоря, внутренний диапазон определяет, какая часть от целого в данный момент полностью доступна пользователю (это может быть видимая часть компонента в панели прокрутки JScrollPane). Подробнее мы обсудим полосы прокрутки JScrollBar в главе 13. Ползунки же используют внутренний диапазон, чтобы определить, на сколько должен «прыгнуть» регулятор, если пользователь щелкает на шкале ползунка, не трогая регулятор. Все это иллюстрирует следующая схема:




Если в вашем случае диапазон данных велик и подразумевает резкое изменение, вы можете сделать внутренний диапазон большим, так что пользователь сможет быстро менять его, щелкая на полосе. К сожалению, поведение ползунков с внутренним диапазоном отличается в зависимости от того, какой внешний вид вы используете, и в большинстве из реализаций вы не сможете выбрать максимальное значение, если установлен внутренний диапазон. Так что как правило при настройке ползунков внутренний диапазон лучше оставить нулевым.

Помимо хранения и при необходимости изменения (все четыре значения можно изменять, в модели описаны соответствующие методы get/set) четырех целых чисел модель BoundedRangeModel, как и можно было бы ожидать, обязана оповещать присоединенных к ней слушателей об изменениях в этих числах. Для этого она генерирует событие ChangeEvent. Модель BoundedRangeModel также поддерживает специальный режим valueIsAdjusting, в котором она перестает оповещать слушателей об изменениях в данных. Этот режим включается UI-представителем при перетаскивании пользователем регулятора, чтобы не замедлять программу запуском событий на каждое изменение положения указателя мыши. При завершении перетаскивания модель возвращается в обычное состояние и сообщает слушателям об изменении в данных, запуская событие ChangeEvent.

Вы видите, что модель BoundedRangeModel для ограниченных данных очень проста, и реализовывать ее самому вряд ли потребуется. Лучше задействовать класс DefaultBoundedRangeModel — стандартную реализацию, поставляемую вместе с библиотекой Swing. Стандартная модель позволяет задавать и изменять все четыре значения и прекрасно справляется со слушателями. Именно этот класс незаметно для вас создают конструкторы ползунка JSlider, передавая в него те же значения, которые вы передаете им. При этом остаются доступными все достоинства моделей: возможность присоединения модели к нескольким видам, отделение механизмов обработки данных от пользовательского интерфейса, и т. д. Рассмотрим простой пример с двумя ползунками:

```
// SliderDefaultModel.java
// Использование в ползунках стандартной модели
import javax.swing.*;
import java.awt.*;

public class SliderDefaultModel extends JFrame {
  // наша модель
  private BoundedRangeModel model;
  public SliderDefaultModel() {
    super("SliderDefaultModel");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создадем модель и пару ползунков
    model = new DefaultBoundedRangeModel(10, 0, 0, 100);
```

```
    JSlider slider1 = new JSlider(model);
    JSlider slider2 = new JSlider(JSlider.VERTICAL);
    slider2.setModel(model);
    // добавляем ползунки в окно
    setLayout(new FlowLayout());
    add(slider1);
    add(slider2);
    // выводим окно на экран
    setSize(300, 300);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new SliderDefaultModel(); } });
  }
}
```

Мы создаем стандартную модель DefaultBoundedRangeModel, используя удобный конструктор: в нем сразу задаются текущее значение, внутренний диапазон (он нам не понадобится, а вернее сказать, даже помешает, так что мы задали его равным нулю), а также минимальное и максимальное значения. Далее мы присоединяем модель к двум ползункам. Первый ползунок создается с помощью удобного конструктора, сразу же позволяющего задать модель. Впрочем, это не обязательно: второй ползунок мы создаем конструктором, в котором указываем, что ползунок будет вертикальным, а модель присоединяем позже методом setModel(). После этого ползунки помещаются в окно и появляются на экране.

Запустив программу с примером, вы увидите, как одна модель поставляет данные для двух разных компонентов, так что они меняются синхронно. Достоинства моделей налицо: вы храните в них данные и при необходимости меняете эти данные (или их меняет пользователь), не беспокоясь о правильном отображении или получении измененной информации — это задача вида, то есть компонента JSlider.

## События ползунков

Ползунки — очень простые компоненты, поэтому событий у них немного, а точнее одно. Это событие ChangeEvent, запускаемое моделью BoundedRangeModel при изменении пользователем текущего значения ползунка[1]. С его помощью вы сможете сразу узнавать о перемещении пользователем регулятора и соответствующим образом изменить состояние программы и ее интерфейса согласно новому текущему значению ползунка. Надо сказать, что для ползунков свойственен «мгновенный эффект»: любое перемещение регулятора по шкале ползунка заставляет пользователя смотреть (или слушать, например при перемещении регулятора громкости), что при этом изменилось, так что всегда следует присоединять к ползунку слушателя событий ChangeEvent и сразу же реагировать на изменения, иначе ваша программа рискует стать неудобной и нелогичной.

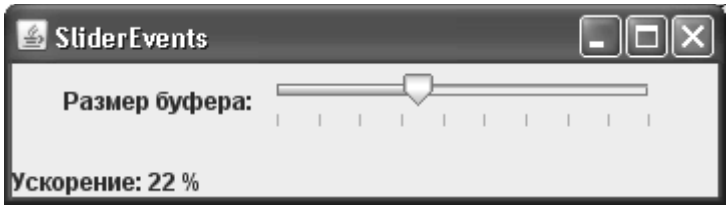
[1] Событие ChangeEvent генерируется также при изменении любого из четырех значений, хранимых в модели (вы легко можете изменить, к примеру, максимальное значение), но это происходит редко, чаще всего пользователю достаточно знать только об изменении текущего значения.

Присоединить слушателя событий ChangeEvent можно либо к модели BoundedRangeModel, используемой в ползунке, либо к самому ползунку JSlider. Разница будет лишь в источнике события, получаемого методом getSource() класса ChangeEvent: в первом случае источником будет модель, а во втором — ползунок. И модель, и ползунок позволяют получить все четыре значения и при необходимости изменить их, так что выбор того или другого — дело вкуса. Рассмотрим небольшой пример:

```
// SliderEvents.java
// События ползунков
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;

public class SliderEvents extends JFrame {
  public SliderEvents() {
    super("SliderEvents");
    setDefaultCloseOperation( EXIT_ON_CLOSE );
    // создаем ползунок и надписи
    JSlider slider = new JSlider(0, 900, 0);
    slider.setMajorTickSpacing(100);
    slider.setPaintTicks(true);
    boost = new JLabel("Ускорение: ");
    // присоединяем слушателя
    slider.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
        // меняем надпись
        int value =
            ((JSlider)e.getSource()).getValue();
        int percent = value/15;
        boost.setText("Ускорение: " + percent + " %");
      }
    });
    // добавляем компоненты в панель
    JPanel contents = new JPanel();
    contents.add(new JLabel("Размер буфера:"));
    contents.add(slider);
    add(contents);
    add(boost, "South");
    // выводим окно на экран
    setSize(360, 100);
    setVisible(true);
  }
  private JLabel boost;
```

```
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new SliderEvents(); } });
  }
}
```

Мы создаем ползунок, позволяющий регулировать размер некоторого буфера (возможно, для сетевого соединения или для воспроизведения звука). При этом с помощью динамически меняющейся надписи вы сможете видеть, какое ускорение можно получить при увеличении размера буфера. Для ползунка мы включаем прорисовку больших делений и присоединяем к нему слушателя события ChangeEvent. Событие ChangeEvent очень «легкое»: в нем не хранится никакой информации за исключением источника события (этому есть свои причины — ChangeEvent запускается так часто, что его пришлось сделать как можно более простым во избежание лишних затрат ресурсов), так что нужно знать, к чему вы присоединяете слушателя ChangeListener. Мы присоединили слушателя к ползунку JSlider, и можем смело преобразовать источник события к объекту JSlider, и с его помощью получить текущее значение. Полученное текущее значение обновляет надпись, так что пользователь мгновенно может оценить, как размер буфера влияет на ускорение, и выбрать оптимальный для него вариант. В заключение ползунок и надписи добавляются в окно (надпись boost на юг окна), и окно выводится на экран.




Это все, что можно сказать о событиях ползунка, который и впрямь очень простой компонент. Нужно лишь запомнить, что с точки зрения пользователя ползунок (вспомните, что он имитирует «ручки» приборов, которые дают мгновенный эффект) является динамическим элементом, поэтому при изменении текущего значения что-то сразу же должно меняться в программе и ее интерфейсе, а это значит, что к ползунку или его модели всегда должен быть присоединен слушатель событий ChangeEvent.

## Дополнительная настройка внешнего вида

Нам уже известны основные параметры внешнего вида ползунков: мы знаем, какие свойства отвечают за прорисовку основных и промежуточных делений и надписей под основными делениями, а также за отключение прорисовки шкалы ползунка (хотя это требуется редко). Оказывается, это не все, на что способен ползунок JSlider. Его внешний вид можно настроить еще точнее.

Наиболее полезна возможность создания собственных надписей для определенных значений, отображаемых ползунком. Вы передаете их в метод setLabelTable() в виде подкласса таблицы Dictionary, сопоставляя целому числу («обернутому» в класс Integer) соответствующую надпись, причем в качестве метки (надписи) могут выступать любые компоненты, унаследованные от базового класса JComponent. Это «развязывает вам руки»,

позволяя создавать ползунки самых причудливых форм и раскрасок. Далее у вас есть возможность включить для ползунка особое клиентского свойство, делающее его «наполняемым» по мере перемещения регулятора, то есть похожим на индикатор процесса (см. следующий раздел). Иногда такой визуальный эффект помогает лучше определить роль ползунка в приложении. Рассмотрим пример:

```
// SliderAdditionalFeatures.java
// Дополнительные возможности ползунков
import javax.swing.*;
import java.util.*;
import java.awt.*;

public class SliderAdditionalFeatures extends JFrame {
 public SliderAdditionalFeatures() {
     super("SliderAdditionalFeatures");
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     // используем стандартную модель
     BoundedRangeModel model =
         new DefaultBoundedRangeModel(10, 0, 0, 60);
     // таблица с надписями
     Dictionary labels = new Hashtable();
     labels.put(0, new JLabel(
         "<html><font color=red size=4>Ноль"));
     labels.put(10, new JLabel(
         "<html><font color=green size=3>Двадцать"));
     labels.put(50, new JLabel(
         "<html><font color=yellow size=5>Много"));
     labels.put(60, new JLabel(
         new ImageIcon("caution.gif")));
     // настройка первого ползунка
     JSlider slider1 = new JSlider(JSlider.VERTICAL);
     slider1.setModel(model);
     slider1.setLabelTable(labels);
     slider1.setPaintLabels(true);
     // "наполняемый" инвертированный ползунок
     JSlider slider2 = new JSlider(model);
     slider2.putClientProperty(
         "JSlider.isFilled", Boolean.TRUE);
     slider2.setInverted(true);
     slider2.setPaintTicks(true);
     slider2.setMajorTickSpacing(10);
     // добавляем компоненты в окно
     setLayout(new FlowLayout());
```

```
        add(slider1);
        add(slider2);
        // выводим окно на экран
        setSize(300, 300);
        setVisible(true);
    }
    public static void main(String[] args) {
       SwingUtilities.invokeLater(
           new Runnable() {
             public void run() { new SliderAdditionalFeatures(); } });
    }
}
```

Пример демонстрирует все дополнительные возможности ползунков JSlider. Данные мы будем хранить в стандартной модели DefaultBoundedRangeModel. Минимальное значение равно нулю, максимальное — шестидесяти, а текущее значение при запуске программы — десяти. Созданную модель мы используем для двух ползунков, которые и продемонстрируют вам дополнительные возможности. У первого ползунка мы создаем собственные надписи для наиболее важных значений, их необходимо поместить в любой подходящий подкласс абстрактной таблицы Dictionary[2], для примера была выбрана таблица Hashtable. Целым значениям, автоматически преобразованным в объекты Integer, можно сопоставлять любые компоненты, унаследованные от базового класса JComponent (в примере использованы прекрасно знакомые нам надписи JLabel, которые способны не только выводить текст и значки, но и поддерживают язык HTML). В первых трех метках мы применили надписи с красочным HTML-текстом, а последнее (максимальное) деление шкалы пометили особым значком. Заметьте, что создавать надписи можно для любых значений, и это позволяет получать весьма эффектные ползунки. После подготовки таблицы с надписями ее нужно передать в метод setLabelTable(). Первый ползунок является вертикальным, обратите внимание, несмотря на наличие нестандартных надписей их все равно приходится включать методом setPaintLabels().

Второй ползунок демонстрирует еще кое-что из арсенала класса JSlider. Он является «наполняемым»: по мере передвижения регулятора полоса (шкала), по которой он движется, заполняется специальным цветом. «Наполняемые» ползунки довольно эффектны — пользователь воочию видит, как выбирается все большее значение; особенно удобны такие ползунки, когда нужно показать объем или размер каких-либо объектов. Чтобы сделать ползунок «наполняемым», нужно изменить клиентское свойство с названием JSlider.isFilled, сопоставив ему объект Boolean со значением true. Правда, имейте в виду, что на данный момент «наполняемые» ползунки поддерживаются только внешним видом Metal, остальные поставляемые вместе с JDK внешние виды (Nimbus, Windows, Motif) это свойство игнорируют[3]. Помимо того что второй ползунок является «наполняемым», он еще и инвертированный, то есть значения в нем отсчитываются справа налево. Такое поведение чаще всего требуется, чтобы локализовать программу

[2] Начиная еще с покрытого древностью пакета JDK 1.2, класс Dictionary объявлен устаревшим, вместо него рекомендуется использовать карты Map. Но ползунки были разработаны раньше, и менять уже устоявшийся программный интерфейс (что привело бы к неработоспособности уже написанного кода) создатели Swing не стали, а нам остается лишь смириться с этим и использовать устаревший класс. Дополнительных методов за все это время также не добавилось.

[3] С другой стороны, почти все внешние виды от сторонних производителей свойство «наполняемости» поддерживают.

для языков с письмом справа налево, но можно использовать инвертированные ползунки и в обычных программах, в некоторых ситуациях это может внести в интерфейс разнообразие.


Оба ползунка добавляются в панель содержимого окна, которое после этого выводится на экран. Запустив программу с примером, вы сможете оценить описанные в этом разделе возможности ползунков JSlider и при желании применить их в своих пользовательских интерфейсах.

## *Индикаторы процесса JProgressBar*

Индикаторы процесса JProgressBar наглядно демонстрируют пользователю, на каком этапе выполнения находится некий процесс, так что пользователь сможет понять, как долго остается ждать его завершения. Такие индикаторы выводятся на экран в виде непрерывно меняющейся полосы, чаще всего с текстом, который показывает (обычно в процентах), насколько выполнен процесс, или каким-либо другим образом поясняет, что в данный момент происходит. Индикаторы процесса — незаменимы для создания качественного пользовательского интерфейса. Время пользователя следует ценить, а благодаря индикаторам процесса он увидит, сколько этого времени ему понадобится для завершения какого-либо действия (и решит, стоит ли ждать завершения этого действия вообще).

Индикаторы процесса JProgressBar — чрезвычайно незатейливые компоненты, одни из самых простых в Swing. Данные они черпают из модели BoundedRangeModel, предназначенной для хранения ограниченного диапазона чисел. Модель BoundedRangeModel уже знакома нам по ползункам JSlider. Как вы помните, модель BoundedRangeModel хранит четыре целых (int) числа: максимальное, минимальное, текущее, а также внутренний диапазон. Индикаторам процесса JProgressBar требуются лишь три из этих значений: максимальное, минимальное и текущее. Именно на основе трех чисел и прорисовывается индикатор JProgressBar. Как правило, минимальное и максимальное значения задаются сразу же (вы знаете либо промежуток времени, который займет процесс, либо диапазон данных, которыми он оперирует), а текущее значение постоянно обновляется по мере выполнения процесса. Писать свою реализацию модели BoundedRangeModel вряд ли имеет смысл, она слишком проста: со всеми возможными случаями ее использования прекрасно справится стандартная реализация, описанная в знакомом нам по тем же ползункам классе DefaultBoundedRangeModel.

Процесс, выполнение которого наглядно выводит на экран индикатор JProgressBar, необходимо поместить в отдельный поток выполнения (Thread) или использовать инструмент SwingWorker, чтобы остальной интерфейс программы не страдал, если процесс окажется требовательным к загрузке процессора и других ресурсов, а значение индикатора (и при необходимости его текст) можно было непрерывно обновлять. Все это мы подробно обсуждали в главе 3. Рассмотрим теперь несложный пример, в котором попробуем применить самые полезные возможности индикатора процесса JProgressBar:

```
// UsingProgressBars.java
// Использование основных возможностей
// компонента JProgressBar
import javax.swing.*;
import java.awt.*;

public class UsingProgressBars extends JFrame {
  // максимальное значение (100%)
  private int MAX = 100;
  // будем использовать общую модель
  private BoundedRangeModel model;
  public UsingProgressBars() {
    super("UsingProgressBars");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем стандартную модель
    model = new DefaultBoundedRangeModel(5, 0, 0, MAX);
    // горизонтальный индикатор
    JProgressBar progress1 = new JProgressBar(model);
    progress1.setStringPainted(true);
    // вертикальный индикатор
    JProgressBar progress2 =
        new JProgressBar(JProgressBar.VERTICAL);
    progress2.setModel(model);
    progress2.setStringPainted(true);
    progress2.setString("Немного терпения...");
    // добавляем индикаторы в окно
    setLayout(new FlowLayout());
    add(progress1);
    add(progress2);
    // выводим окно на экран
    setSize(400, 300);
    setVisible(true);
    // запускаем "процесс"
    new LongProcess().execute();
  }
```

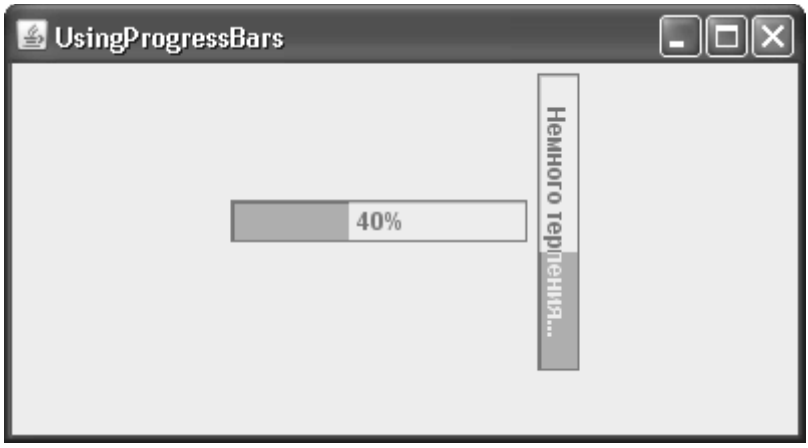
```
    // эмуляция долгого процесса
    class LongProcess extends SwingWorker<String,Integer> {
      // работаем до завершения процесса
      protected String doInBackground() throws Exception {
        int current = 5;
        while ( current <= MAX ) {
          // обновляем внешний вид
          publish(current++);
          // случайная задержка
          Thread.sleep((int)(Math.random()*1000));
        }
        return "Готово";
      }
      // выполняется в потоке рассылки событий
      public void process(java.util.List<Integer> chunks) {
        // увеличиваем текущее значение
        model.setValue(chunks.get(0));
      }
    }
    public static void main(String[] args) {
      SwingUtilities.invokeLater(
          new Runnable() {
            public void run() { new UsingProgressBars(); } });
    }
}
```

Итак, мы создаем небольшое окно, в котором разместятся два индикатора процесса. Прежде всего необходимо создать поставщика данных для индикаторов — модель BoundedRangeModel. Мы задействуем стандартную модель, задав в конструкторе текущее значение равным пяти, внутренний диапазон — нулю (он нам не понадобится), минимальное значение — также равным нулю, а максимальное значение — ста. Полученная модель потребуется для обоих индикаторов, так что они будут синхронно показывать ход процесса. С другой стороны, как это принято в Swing, вы можете делать вид, что модели нет, задавая минимальное и максимальное значения в особом конструкторе и при необходимости меняя их или текущее значение методами класса JProgressBar. Но это не так элегантно.

Далее мы создаем сами индикаторы процесса JProgressBar. Индикаторы могут быть вертикальными и горизонтальными, управляет этим свойство orientation, но задать ориентацию индикатора можно и в конструкторе. Если вы не указываете ориентацию индикатора процесса явно, он считается горизонтальным. Первый индикатор создается конструктором, принимающим в качестве параметра нашу модель BoundedRangeModel. По умолчанию никакого текста индикатор прорисовывать не будет, чтобы он начал это делать, следует вызвать метод setStringPainted(true). Стандартно прорисовывается строка, показывающая, сколько в процентах от максимального занимает текущее значение, то есть какая часть процесса уже позади (значения эти хранятся в модели). Второй индикатор создается конструктором, в котором мы указываем его ориентацию: он будет вертикальным. Далее мы присоединяем к нему нашу модель методом setModel(), включаем прорисовку текста уже известным мето-

дом setStringPainted(true), и задаем нестандартный текст с помощью метода setString(). Текст, несмотря даже на то, что индикатор является вертикальным, может быть порядочного размера: текст прорисовывается так же, как и сам индикатор, то есть вертикально. Впрочем, стоит заметить, что читать в таком положении длинный текст не слишком удобно (может заболеть шея). Длинный текст лучше оставить для горизонтальных индикаторов, а в вертикальных ограничиться чем-то лаконичным, вроде символа процентов.

После настройки модели и индикаторов нужно добавить последние в панель содержимого нашего окна и вывести окно на экран. Так мы и делаем. Если больше ничего не предпринимать, индикаторы процесса застынут на отметке в пять процентов (помните, текущее значение модели изначально у нас равно пяти, а максимальное — ста, так что все справедливо) и будут малоинтересны. Действительно, в любом приложении, использующем индикаторы, должен быть некоторый процесс, по мере выполнения которого текущее значение индикатора непрерывно обновляется. Роль такого процесса в нашем примере сыграет внутренний класс LongProcess, созданный на основе инструмента SwingWorker и, следовательно, представляющий собой отдельный поток выполнения с возможностью вызова промежуточных действий в графическом потоке рассылке событий. В реальных приложениях процессом может быть загрузка данных из сети или файла с диска, установка компонентов или сетевых соединений, и еще много самых разных действий. Мы сразу же запускаем нашу задачу (вызывая метод execute()), и параллельно с потоком рассылки событий начинает выполняться ее метод doInBackground(). Работа задачи проста: пока текущее значение модели не превышает максимального (то есть пока наш воображаемый процесс не закончен), в цикле while текущее значение увеличивается на единицу, после чего процесс «засыпает» на случайный промежуток времени, а затем все повторяется. Случайный промежуток простоя процесса внесет в пример свою изюминку: вы увидите, как индикатор то бежит быстрее, то замедляется. Обратите внимание, что нам не приходится заботиться о синхронизации: мы применяем метод publish() для вызова кода из потока рассылки событий, передавая туда текущее значение для модели, и работаем с моделью, а не с компонентом, так что модель сама оповестит все присоединенные к ней виды о том, что пора обновить изображение. Просто и эффектно. Не забывайте, что работа с моделью, несмотря на то, что вы не работаете с графическими компонентами напрямую, подразумевает отсылку событий этим компонентам, и, соответственно, их обновление. Так что с моделями работать необходимо только из потока рассылки событий.




Напоследок выпишем все самые полезные свойства индикаторов JProgressBar, использованные нами в примере, чтобы они всегда были «под рукой» и могли послужить простым справочником (табл. 12.2).

**Таблица 12.2.** Основные свойства индикаторов JProgressBar

| Свойства | Описание |
|---|---|
| orientation | Управляет ориентацией индикатора. Ориентация может быть вертикальной или горизонтальной. Чаще всего это свойство задается при создании индикатора и не меняется впоследствии. Если же вы меняете ориентацию для компонента, который уже находится на экране, не забудьте провести проверку корректности контейнера: смена ориентации повлияет на расположение компонентов |
| stringPainted | Позволяет включить или выключить прорисовку текста на индикаторе процесса. По умолчанию прорисовка текста выключена. Если вы ее включаете, то стандартно будет прорисовываться значение в процентах, показывающее, какая часть процесса закончена (значения берутся из модели). Сменить текст позволяет следующее свойство |
| string | Задает текст, который будет прорисовываться на индикаторе процесса. Не забывайте только включать прорисовку текста: по умолчанию она отключена |
| value, maximum, minimum | Эти свойства позволяют манипулировать данными модели BoundedRangeModel напрямую с помощью самого индикатора (модель в любом случае есть, данные свойства просто делегируют ей свою работу). Управляют соответственно текущим, максимальным и минимальным значениями. Благодаря им (а также благодаря конструкторам JProgressBar, которые сразу позволяют задать эти значения и неявно создают стандартную модель) вы сможете работать с индикатором так, как будто модели вовсе и нет. Впрочем, преимущества моделей нам прекрасно известны, так что лучше ее задействовать |

Описанных свойств индикаторов вполне хватает для большей части приложений: роль этих компонентов на самом деле невелика. Используются они именно так, как было показано нами в примере: есть отдельный поток выполнения с долгими вычислениями, который обновляет модель (конечно же, из потока рассылки событий), а индикатор каждое изменение исправно отображает на экране.

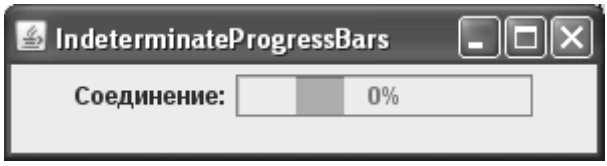
## Когда ничего не ясно

Бывают (и, надо сказать, довольно часто) ситуации, когда сказать точно, сколько осталось до завершения процесса и на каком этапе он в данный момент находится, вы не в состоянии. В таких случаях не слишком удобно использовать индикатор: непонятно, каковы же значения модели и как их обновлять. С другой стороны, лишать пользователя хоть какого-то намека на то, что процесс идет, тоже не следует. В расчете на такие ситуации индикаторы процесса JProgressBar поддерживают состояние «неопределенности»: периодическое движение на полосе есть, но, сколько осталось до конца, не ясно. В том случае, если процесс переходит в более «определенное» состояние, и вы, наконец, сможете выяснить, на каком этапе он находится и с какой скоростью продвигается, индикатор можно перевести в обычное состояние, которое нам уже знакомо. Давайте рассмотрим «неопределенный» пример:

```
// IndeterminateProgressBars.java
// Индикаторы в состоянии "неопределенности"
import javax.swing.*;
import java.awt.*;

public class IndeterminateProgressBars extends JFrame {
  public IndeterminateProgressBars() {
```

```
    super("IndeterminateProgressBars");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // неопределенный индикатор
    JProgressBar progress = new JProgressBar(0, 100);
    progress.setIndeterminate(true);
    progress.setStringPainted(true);
    // добавляем его в окно и выводим на экран
    setLayout(new FlowLayout());
    add(new JLabel("Соединение:"));
    add(progress);
    setSize(300, 200);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() {
            new IndeterminateProgressBars(); } });
  }
}
```

В панель содержимого окна мы добавляем индикатор JProgressBar, у которого нет даже самой простой отдельной модели: нам помогает конструктор, позволяющий задать минимальное и максимальное значения, а стандартная модель будет создана автоматически, за кулисами. Большего нам не понадобится: индикатор в состоянии неопределенности не использует никаких данных модели. Как видно из примера, состояние «неопределенности» включается свойством indeterminate. Никаких действий предпринимать больше не требуется: индикатор сам позаботится о том, чтобы вид полосы периодически менялся, изображая продолжающуюся деятельность процесса и подбадривая этим пользователя. Дополнительно мы включили прорисовку текста свойством stringPainted, он появится даже в «неопределенном состоянии» (по умолчанию будет изображаться выполненная в процентах часть процесса — в зависимости от значения, установленного в модели). Однако чаще всего «неопределенное» состояние не использует стандартное отображение в процентах, как дополнительное подтверждение того, что на данный момент информация о выполнении процесса отсутствует. Лучше заменить строку на свою собственную, которая подчеркнет что процесс занимает неопределенное время. Запустив программу с примером, вы сможете оценить, как смотрится «неопределенная» ситуация в исполнении индикатора JProgressBar.




Пример довольно реален: сетевые соединения зачастую не позволяют сказать определенно, что именно произошло и когда возобновится прием информации.

## Небольшие хитрости

Индикаторы процесса на самом деле очень просты, и работа с ними сводится в основном к обновлению текущего значения модели. Даже их внешний вид толком не настраивается (изменить можно лишь отображаемую индикатором строку). Впрочем, с помощью некоторых незамысловатых свойств UI-представителя индикаторов JProgressBar, не попавших в класс самого компонента, вы сможете добиться более изысканных визуальных эффектов, в некоторых случаях больше отвечающих стилю вашего приложения. Свойства эти отвечают за вид полосы индикатора. Рассмотрим небольшой пример, в котором эти свойства будут использоваться для достижения нового внешнего вида:

```
// ProgressBarTricks.java
// Небольшие хитрости индикаторов процесса
import javax.swing.*;
import java.awt.*;

public class ProgressBarTricks extends JFrame {
  // максимальное значение индикатора
  private final int MAX = 100;
  public ProgressBarTricks() {
    super("ProgressBarTricks");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // настроим параметры для UI-представителей
    UIManager.put("ProgressBar.cellSpacing", 2);
    UIManager.put("ProgressBar.cellLength", 6);
    // стандартная модель
    final DefaultBoundedRangeModel model =
        new DefaultBoundedRangeModel(0, 0, 0, MAX);
    // создадим простой индикатор процесса
    // на основе полученной модели
    JProgressBar progress = new JProgressBar(model);
    // добавим его в окно
    setLayout(new FlowLayout());
    add(progress);
    // выводим окно на экран
    setSize(300, 200);
    setVisible(true);
    // создадим "процесс"
    Thread process = new Thread(new Runnable() {
      public void run() {
        // увеличиваем текущее значение модели до
        // достижения максимального значения
        int value = 0;
        while ( ++value <= MAX ) {
```

```
            final int passValue = value;
            SwingUtilities.invokeLater(
                new Runnable() {
                  public void run() {
                    model.setValue(passValue); }});
            try {
              Thread.sleep(200);
            } catch (Exception ex) { }
          }
        }
      });
      // запустим поток
      process.start();
    }
    public static void main(String[] args) {
      SwingUtilities.invokeLater(
          new Runnable() {
            public void run() { new ProgressBarTricks(); } });
    }
}
```

Здесь мы создаем небольшое окно, в котором разместится незатейливый индикатор процесса. Данные этому индикатору будет поставлять стандартная модель, начальное значение ее равно нулю, минимальное также равно нулю, а максимальное значение равно ста (вы можете видеть, как мы задаем значения в конструкторе модели). Самое интересное происходит в паре строк, которые настраивают внутренние свойства UI-представителей индикаторов процесса (а мы знаем, что именно эти представители и выполняют фактическую прорисовку компонента на экране). Свойства UI-представителей всех компонентов Swing для любого внешнего вида хранятся в специальной таблице класса UIManager, и мы можем менять их, используя методы put(). Оказывается, что с помощью данных свойств можно тонко настроить внешний вид индикаторов процесса (точнее, перемещающейся в них полосы). Свойство с названием cellLength управляет тем, какой длины будут ячейки полосы индикатора (полоса индикатора не обязательно прорисовывается сплошной, она может состоять из набора ячеек одинакового размера), а свойство cellSpacing задает расстояние между ячейками (если это расстояние приравнять нулю, то полоса будет сплошной). Запустив программу с примером, вы увидите, как описанные два свойства влияют на внешний вид индикатора процесса (в современных пользовательских интерфейсах более популярны именно такие, ячеистые, индикаторы). Правда, стоит иметь в виду, что данные свойства действуют, только если в индикаторе процесса не прорисовывается информационная строка. Если строка есть, то полоса индикатора будет сплошной в любом случае.

Более того, данные свойства зависят от того внешнего вида, который вы используете, и если они работают для внешнего вида Swing по умолчанию (Metal), это не значит, что они работают для других внешних видов. Всегда, если вы собираетесь применить внутренние настройки внешнего вида, проверьте, работают ли они в вашей ситуации и со всеми внешними видами, которые вы планируете использовать.

Ну и напоследок обратите внимание на «процесс», в котором обновляется значение модели индикатора, — это обычный поток Thread; в нем с некоторой задержкой текущее значение модели увеличивается до максимального. Как вы помните, для каждого индикатора процесса такой процесс должен быть, и он должен постоянно обновлять значение индикатора, в противном случае пользователь просто не поймет, что же вы хотели ему сообщить неподвижным индикатором. В отличие от одного из предыдущих примеров, где мы применили инструмент SwingWorker, здесь создается самый банальный поток. С одной стороны, создать его проще, меньше параметров и всего один метод run(). С другой стороны, любые действия с нашим интерфейсом и его моделями мы должны выполнять из потока рассылки событий, а для этого приходится вызывать громоздкий метод invokeLater() очереди событий EventQueue (мы вызываем его через класс SwingUtilities). В него, как правило, приходится передавать внутренний анонимный класс, реализующий интерфейс Runnable, и чтобы последний смог получить доступ к переменным и объектам, их необходимо объявлять неизменными (final). Код получается немного грязноватым, и применение SwingWorker все же дает более приличный результат. До тех пор пока в Java нет другого способа передать фрагмент кода кроме как обернуть его в интерфейс Runnable, код обычного потока в Swing-приложении выглядит не совсем элегантно, и нам приходится с этим смириться.

## *Счетчики JSpinner*

Счетчики JSpinner позволяют пользователю сделать выбор из набора альтернатив, которые он в поиске нужного значения может «прокручивать» в любую сторону. Счетчик вместе с полем внешне похож на раскрывающийся список JComboBox: на экране в поле виден только выбранный в данный момент элемент, но если раскрывающийся список показывает набор альтернатив в выпадающем меню, то счетчик набора своих элементов вообще не показывает, предлагая вместо этого пользователю перейти к следующему или предыдущему значению с помощью маленьких кнопок со стрелками. Такое свойство счетчика позволяет ему хранить неограниченное количество альтернатив (списки должны иметь конечный набор альтернатив, потому что они выводят его на экран либо сразу, как JList, либо в выпадающем меню, как JComboBox).

Данные счетчика хранятся в его модели, описываемой интерфейсом SpinnerModel. Модель эта довольно проста: она позволяет узнать текущее значение счетчика, следующее и предыдущее значения, сменить текущее значение новым, а также поддерживает списки слушателей ChangeListener, которые оповещаются, когда меняется текущее значение модели. В модели счетчика JSpinner нет и намека на методы, определяющие размер списка или получающие позицию элемента списка. Есть только три значения, и следующим значением может быть все что угодно. Благодаря такой модели данных счетчик позволяет организовать выбор из практически произвольного диапазона данных, тем не менее в нем остаются удобные ограничения списка (такие ограничения уменьшают количество ошибок при вводе данных пользователем: последний всегда может увидеть, что предлагается в качестве следующей или предыдущей альтернативы). Более того, в зависимости от настройки счетчик может допускать ввод произвольных значений.

Счетчики и хранимые ими данные довольно необычны, так что создавать их с помощью простых конструкторов не слишком удобно. Давайте сразу же посмотрим, какие стандартные модели для счетчиков JSpinner имеются в Swing. Таких моделей целых три: модель SpinnerListModel организует выбор из конечного числа элементов (при использо-

вании этой модели счетчик становится оригинальной копией списка JComboBox), модель NumberSpinnerModel дает возможность сделать из этого элемента интерфейса «настоящий» счетчик (позволяющий выбирать только цифры из определенного диапазона), наконец, модель SpinnerDateModel, более экзотичная и сложная, чем первые две модели, позволяет организовать гибкий выбор дат. К третьей модели мы вернемся немного позже, а сейчас рассмотрим небольшой пример, в котором используем счетчик и первые две модели:

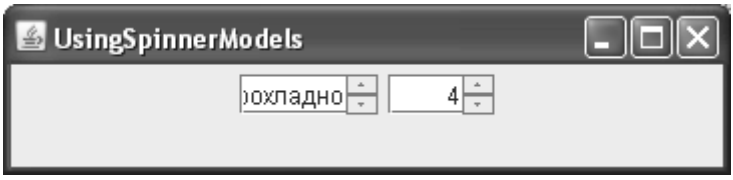
```
// UsingSpinnerModels.java
// Использование стандартных моделей счетчика
import javax.swing.*;
import java.awt.*;

public class UsingSpinnerModels extends JFrame {
  // набор данных для счетчика
  private String[] data = {
    "Холодно", "Прохладно", "Тепло", "Жарко"
  };
  public UsingSpinnerModels() {
    super("UsingSpinnerModels");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // модель счетчика для выбора из набора данных
    SpinnerModel list = new SpinnerListModel(data);
    JSpinner spinner1 = new JSpinner(list);
    // модель счетчика для выбора целых чисел
    SpinnerModel numbers = new SpinnerNumberModel(
      4, 0, 100, 1);
    JSpinner spinner2 = new JSpinner(numbers);
    // добавим счетчики в панель содержимого
    setLayout(new FlowLayout());
    add(spinner1);
    add(spinner2);
    // выводим окно на экран
    setSize(400, 300);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new UsingSpinnerModels(); } });
  }
}
```

В примере создается небольшое окно, в котором разместятся два счетчика JSpinner. Первый создается на основе модели SpinnerListModel, которая позволяет организовать вы-

бор из нескольких альтернатив, в точности повторяя поведение раскрывающегося списка JComboBox. Задать элементы для этой модели можно либо в виде массива (в примере мы так и поступаем), либо в виде любого динамического списка с данными List (к примеру, можно использовать список ArrayList). Нетрудно сменить список элементов уже после того, как вы создали модель или даже успели присоединить ее к счетчику: для этого предназначен метод модели setList() (заметьте, что передавать данные в виде массива позволяет только конструктор модели). Созданную модель мы передаем в конструктор счетчика, так что последний будет знать, что именно надо отображать (модель, как мы уже давно привыкли, можно задать и после создания счетчика с помощью метода setModel()).

Второй счетчик использует модель для выбора чисел SpinnerNumberModel. В конструкторе этой модели требуется задать четыре значения: начальное значение (у нас оно равно четырем), минимальное и максимальное значения, между которыми должен лежать выбор пользователя (в нашем примере — ноль и сто), и, наконец, приращение, на которое будет увеличиваться или уменьшаться текущее значение при выборе пользователем следующего или предыдущего значения, соответственно. Приращение в нашем примере равно единице. У модели SpinnerNumberModel есть несколько перегруженных конструкторов: для выбора целых чисел (int), чисел с плавающей запятой (double), и, наконец, произвольных чисел, представленных объектами Number, сравнение этих объектов выполняется с помощью задаваемых в конструкторе объектов Comparable[4]. Как нетрудно видеть, мы использовали конструктор, позволяющий организовать выбор целых чисел. Как и в случае с предыдущей моделью, вы можете менять числовые диапазоны модели и после ее создания и присоединения (правда, в таком случае можно задействовать только объекты Number и Comparable).

Запустив программу с примером, вы увидите, как работают два созданных нами счетчика, обратите внимание, что оба допускают редактирование выбранного в данный момент значения. Вы даже можете ввести в поле число, не попадающее в заданный нами для модели диапазон, только в этом случае кнопки перехода к следующему или предыдущему значению перестанут работать. Прокручивая первый счетчик, вы столкнетесь с не слишком приятной ситуацией: поле счетчика имеет размер, соответствующий первому элементу («Холодно»), и следующий элемент в него не поместится, что для пользователя будет совсем не кстати. Более того, при смене элементов разной длины поле счетчика динамически меняет свой предпочтительный размер[5], что вполне может сделать поведение вашего интерфейса непредсказуемым.




[4] Объект, реализующий интерфейс Comparable, поддерживает сравнение себя с другими объектами, обычно такого же типа. В интерфейсе Comparable определен один метод с названием compareTo(), который возвращает целое число: положительное, если данный объект больше объекта, с которым он сравнивается, отрицательное, если он меньше, и ноль, если объекты равны. В конструктор SpinnerNumberModel можно передавать не любые объекты, реализующие Comparable, а подклассы Number, хотя это и не афишируется интерактивной документацией.

[5] Как видно, счетчик подводит нас дважды: при выводе окна на экран он имеет размер, подходящий для размещения первого элемента, а после смены элемента меняет свой размер, «подгоняя» его под новый элемент. Однако пользователь не увидит смены размера счетчика до тех пор пока не произойдет проверка корректности контейнера (к примеру, это может произойти при изменении размеров окна или после добавления нового компонента), и это еще больше увеличит так нежелательный в пользовательских интерфейсах эффект неожиданности.

Логично было бы предположить, что в классе JSpinner имеется какой-либо метод, позволяющий задать наилучшую длину поля счетчика (аналогично методу setPrototypeValue() списка JComboBox), но увы, такого метода нет[6]. Так что использование счетчика для выбора одного из нескольких элементов (с помощью модели SpinnerListModel) не слишком удобно, особенно в случае небольшого количества этих элементов, с такими ситуациями гораздо лучше справится раскрывающийся список JComboBox. Да и пользователю будет намного легче справиться с задачей поиска, когда он видит хотя бы малую часть множества, из которого следует выбирать значение. Правда, у счетчика есть приятное дополнение — он пытается автоматически дополнить текст, который вы набираете в нем, если он частично совпадает с одним из элементов списка.

С другой стороны, счетчик для выбора чисел идеально справляется со своей ролью: он ограничивает диапазон доступных чисел, позволяет быстро найти нужное пользователю число и, наконец, прекрасно справляется с вводом новых значений, не позволяя пользователю вводить ничего, кроме чисел нужного формата. Всегда, когда у вас в приложении возникает необходимости ввода чисел, применяйте счетчик: он намного эффективнее простого текстового поля.

## Выбор дат

У счетчиков JSpinner есть и третья модель с названием SpinnerDateModel, позволяющая организовать гибкий выбор дат из определенного диапазона. Учитывая отсутствие в Swing стандартного компонента-«календаря»[7], данная модель находит в большинстве программ довольно широкое применение. Для использования модели для выбора дат необходимы некоторые знания о том, как в Java проводятся манипуляции датами. Классы для таких манипуляций находятся в пакете java.util, модель SpinnerDateModel использует два класса: класс Date описывает некоторый момент во времени (с точностью до миллисекунды), а класс Calendar позволяет менять даты по любому из параметров, както: день недели, день месяца, год, месяц, или даже эра.

Для того чтобы организовать с помощью класса SpinnerDateModel выбор дат, требуется совсем немного: задать с помощью объекта Date текущее значение, с помощью объектов Comparable указать минимальную и максимальную даты, между которыми должен лежать выбор пользователя (если задать эти объекты равными null, то выбор будет неограничен), и указать, по какому полю календаря будет осуществляться прокрутка. Рассмотрим пример:

```
// ChoosingDates.java
// Выбор дат с помощью SpinnerDateModel
import javax.swing.*;
import java.util.*;
import java.awt.*;

public class ChoosingDates extends JFrame {
  public ChoosingDates() {
    super("ChoosingDates");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
```

[6] На самом деле способ «твердо» задать длину поля счетчика все-таки есть, хотя и не самый элегантный. Мы узнаем о нем чуть позже, когда будем обсуждать редакторы для счетчиков.

[7] Хотя в наборе дополнительных компонентов SwingX «календарь» есть (он называется JXDatePicker), и счетчик для выбора дат не так актуален, если вы решитесь использовать это дополнение.

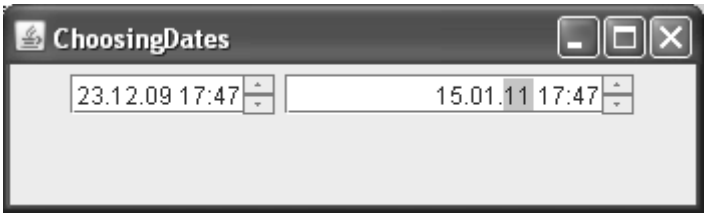
```
    // настраиваем модель для выбора дня месяца
    SpinnerModel monthDay = new SpinnerDateModel(
      new Date(), null, null, Calendar.DAY_OF_MONTH);
    JSpinner spinner1 = new JSpinner(monthDay);
    // модель для выбора месяца, с ограничениями
    SpinnerModel month = new SpinnerDateModel(
         new Date(), new MinDate(), new MaxDate(), Calendar.MONTH);
    JSpinner spinner2 = new JSpinner(month);
    // добавляем списки в панель
    setLayout(new FlowLayout());
    add(spinner1);
    add(spinner2);
    // выводим окно на экран
    setSize(350, 300);
    setVisible(true);
  }
  // вспомогательный объект для проверок дат
  private Calendar calendar = Calendar.getInstance();
  // проверяет минимальную дату (по году)
  class MinDate extends Date implements Comparable<Date> {
    public int compareTo(Date d) {
      calendar.setTime(d);
      int year = calendar.get(Calendar.YEAR);
      // год не меньше 2005
      return (year < 2005) ? 1 : -1;
    }
  }
  // проверяет максимальную дату (по году)
  class MaxDate extends Date implements Comparable<Date> {
    public int compareTo(Date d) {
      calendar.setTime(d);
      calendar.get(Calendar.YEAR);
      int year = calendar.get(Calendar.YEAR);
      // год не больше 2011
      if ( year > 2011 ) return -1;
        else return 1;
    }
  }
  public static void main(String[] args) {
     SwingUtilities.invokeLater(
        new Runnable() {
```

```
            public void run() { new ChoosingDates(); } });
    }
}
```

В примере мы создаем пару счетчиков JSpinner, которые позволят пользователю выбирать дату (точнее, один из параметров даты, такой как день недели). Как мы уже знаем, для выбора даты используется специальная модель счетчика SpinnerDateModel. Модель для первого счетчика позволит нам организовать выбор дня месяца, причем без ограничений: пользователь сможет выбрать тот день месяца, который ему приглянется, даже если тот прошел несколько столетий назад. Конструктор модели SpinnerDateModel требует задать четыре параметра: первым идет начальное значение в виде объекта-даты Date, которое будет отображать поле счетчика. Как правило, начальным значением выбирают настоящий момент времени, получить его не составляет труда: надо просто создать объект Date[8]. Так мы и поступаем. Далее в конструкторе надо указать диапазон дат, минимально возможную и максимально возможную даты. Мы передаем вместо этих параметров пустые ссылки null, это означает, что выбор дат будет неограничен. Наконец, последним параметром должно идти поле, по которому будет изменяться дата. Доступные поля перечислены в виде целочисленных констант в классе Calendar, для нашего первого счетчика мы выбираем поле «день месяца» (DAY_OF_MONTH). Настроенную модель мы передаем в конструктор счетчика.

Вторая модель позволяет выбирать месяц, причем она сложнее, так как ограничивает доступный для выбора диапазон дат. В качестве «ограничителей» должны выступать объекты, реализующие интерфейс Comparable и выполняющие сравнение выбранной пользователем даты с минимальной и максимальной границами диапазона. В принципе, в качестве «ограничителя» должен был бы годиться любой объект, реализующий интерфейс Comparable, но не тут то было. Объект-«ограничитель» обязательно должен представлять собой экземпляр класса Date (или экземпляр его подкласса). В противном случае вас вместо поля со счетчиком поджидает маловразумительная цепочка исключений.

В нашем примере мы наследуем наши объекты от класса Date и реализуем интерфейс Comparable. Сравнение дат проводится по году, вы видите, что год даты позволяет получить класс Calendar (в классе Date есть, правда, метод getYear(), но он плохо справляется с временными зонами и не рекомендован к использованию). Класс MinDate следит за тем, чтобы год выбираемой пользователем даты был не меньше 2005 (если год меньше, возвращается единица, а это значит, что минимальная дата больше той, с которой проводится сравнение, то есть такая дата не подходит). Аналогичным образом класс MaxDate требует, чтобы год выбираемой даты не превышал 2011. «Ограничители» мы передаем в конструктор модели и последним параметром указываем, что прокрутка должна производиться помесячно. Вы можете и не писать методы compareTo(), а просто указать даты, между которыми должен лежать выбор пользователя. В классе Date уже имеется реализация интерфейса Comparable. Как мы уже упоминали, настроить время, заданное в объекте Date, позволяет все тот же класс Calendar.




[8] Получить объект Date, представляющий другой момент во времени, позволяет класс Calendar. Метод set() данного класса дает возможность изменить любое поле даты, к примеру, изменить день месяца можно так: set(Calendar.DAY_OF_MONTH, 1).

Созданные на основе настроенных моделей счетчики мы добавляем в панель содержимого и выводим на экран. Запустив программу с примером, вы увидите, как первый счетчик позволяет выбирать даты из неограниченного диапазона и как ограничивает выбор второй счетчик, дающий вам возможность «прокрутить» месяцы только до конца 2011 года, но и не позволяющий уходить в прошлое на срок раньше 2005 года. Счетчики очень полезны при выборе дат, хотя вам может показаться, что формат, в котором выводится дата, не слишком удобен для восприятия и подходит не ко всем ситуациям. Однако эту проблему легко обойти, настроив редактор, используемый счетчиком для отображения данных модели SpinnerDateModel, мы вскоре увидим, как это делается.

Остается отметить, что компонент JSpinner иногда способен «удивить» пользователя, а в деловых интерфейсах это не приветствуется. К примеру, когда в нашем примере фокус ввода попадает на второй счетчик, прокрутка все равно ведется с первого поля, то есть по дням, несмотря на то, чтобы просили счетчик прокручивать дату по месяцам. Более того, счетчик всегда позволяет ввести в поле ввода любую строку, и только после того как пользователь покинет поле счетчика, сбрасывается на последнее подходящее значение, например дату. Все это делает счетчики, по крайней мере стандартные счетчики JDK, не совсем удобными для эффектных интерфейсов. Но кое-что улучшить всегда возможно, это мы сейчас и увидим.

## Редактор элементов

Во всех списках библиотеки Swing отображение и при необходимости редактирование элементов списка выполняется сторонним объектом, что позволяет тонко настраивать внешний вид элементов списка по своему вкусу. Не остался в стороне и счетчик JSpinner: отображение выбранного в данный момент элемента (а других элементов, как мы знаем, поле счетчика и не отображает) выполняется специальным редактором, который должен быть унаследован от базового класса JComponent библиотеки Swing. Таким образом, в качестве редактора для элементов счетчика может быть использован любой компонент библиотеки Swing. Впрочем, название «редактор» достаточно условно: главной его функцией является отображение элемента, а редактирование он может разрешать или нет в зависимости от своей настройки.

Как нетрудно догадаться, по умолчанию для отображения своих элементов счетчик использует стандартные редакторы, поставляемые вместе с ним. Для каждой рассмотренной нами стандартной модели (как мы знаем, для класса JSpinner их три) имеется свой стандартный редактор. Правда, все стандартные редакторы JSpinner чрезвычайно похожи друг на друга: все они унаследованы от редактора JSpinner.DefaultEditor, который представляет собой текстовое поле с поддержкой данных в специальном формате JFormattedTextField. Унаследованные от редактора DefaultEditor подклассы просто задают для текстового поля подходящее форматирующее выражение, например, выражение, описывающее дату. В табл. 12.3 перечислены редакторы, соответствующие стандартным моделям.

**Таблица 12.3.** Редакторы для стандартных моделей

| Модель | Редактор |
|---|---|
| SpinnerListModel | JSpinner.ListEditor |
| SpinnerNumberModel | JSpinner.NumberEditor |
| SpinnerDateModel | JSpinner.DateEditor |

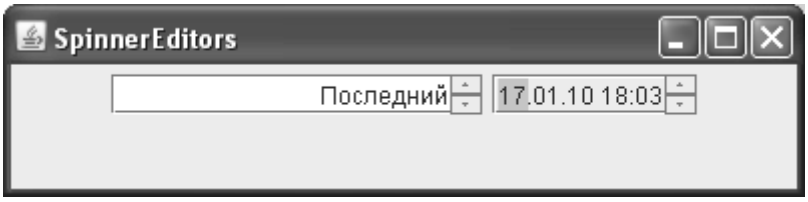
Как мы уже отметили, стандартные редакторы отличаются лишь форматом информации, которая будет появляться в текстовом поле JFormattedTextField. Их довольно легко настроить: вы получаете стандартный редактор методом getEditor(), преобразуете его к базовому классу DefaultEditor, получаете текстовое поле и настраиваете его по своему вкусу. Иногда это единственный способ придать счетчику нужный вам вид, не обращаясь к написанию собственного редактора. Попробуем теперь настроить стандартные редакторы в следующем примере:

```
// SpinnerEditors.java
// Стандартные редакторы счетчиков
import javax.swing.*;
import java.util.*;
import java.awt.*;

public class SpinnerEditors extends JFrame {
  // данные для первого счетчика
  private String[] data = {
      "Первый", "Второй", "Последний"
  };
  public SpinnerEditors() {
    super("SpinnerEditors");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // счетчик на основе массива
    JSpinner spinner1 = new JSpinner(
        new SpinnerListModel(data));
    // настраиваем редактор
    ((JSpinner.ListEditor)spinner1.getEditor()).
        getTextField().setColumns(15);
    // выбор дат
    SpinnerDateModel dates = new SpinnerDateModel(
        new Date(), null, null, Calendar.DAY_OF_MONTH);
    JSpinner spinner2 = new JSpinner(dates);
    // настраиваем редактор
    ((JSpinner.DateEditor)spinner2.getEditor()).
        getTextField().setEditable(false);
    // добавляем счетчики в панель содержимого
    setLayout(new FlowLayout());
    add(spinner1);
    add(spinner2);
    // выводим окно на экран
    setSize(400, 300);
    setVisible(true);
  }
```

```
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new SpinnerEditors(); } });
  }
}
```

Мы создаем два счетчика, один на основе модели SpinnerListModel, данные ему мы передали в виде массива строк, другой на основе модели SpinnerDateModel, так что он послужит для выбора дат (легко видеть, что выбор будет проводиться по дню месяца и без ограничений). После создания моделей и присоединения их к счетчикам производится простая настройка внешнего вида и поведения редакторов счетчиков. Для первого счетчика мы устанавливаем новое количество столбцов текстового поля (заметьте, что текстовое поле JFormattedTextField можно получить методом getTextField()) и таким образом сразу же избавляемся от неприятной проблемы, замеченной нами еще в самом начале знакомства со списками JSpinner: элементы различной длины могут не «влезать» в счетчик или динамически менять размеры счетчика. После задания определенного количества столбцов счетчик будет иметь твердо заданный размер, подходящий под ваши нужды.

Для редактора второго счетчика мы отключаем редактирование, это часто приходится делать, чтобы пользователь применял именно «прокрутку», а не вводил собственные значения (при этом не учитываются ограничения, установленные моделью счетчика). Текстовое поле JFormattedTextField способно на большее, особенно это касается сложных форматов данных, заданных регулярными выражениями или их разновидностями, масками. В том числе можно полностью настроить формат даты, отображаемой в поле счетчика для выбора дат. Текстовое поле JFormattedTextField мы подробнее изучим в главе 16.




Действия с редакторами не ограничиваются настройкой стандартных редакторов класса JSpinner, вы с легкостью сможете написать свой редактор. Мы уже знаем, что в качестве редактора счетчика JSpinner может выступать любой компонент, унаследованный от базового класса библиотеки JComponent. Особенно полезно создание нового редактора в случае применения собственной модели с экзотичными данными: поставляя вместе с такой моделью подходящий редактор, вы полностью настраиваете счетчик, говоря, какие данные он должен «прокручивать» и как их надо отображать. В качестве примера давайте попробуем создать редактор на основе надписи JLabel — нам прекрасно известны ее возможности по отображению красочного содержимого. Для того чтобы редактор смог вовремя узнавать о смене текущего элемента счетчика, он должен присоединить к нему слушателя ChangeListener, который будет оповещаться при смене элементов. Итак, вот пример:

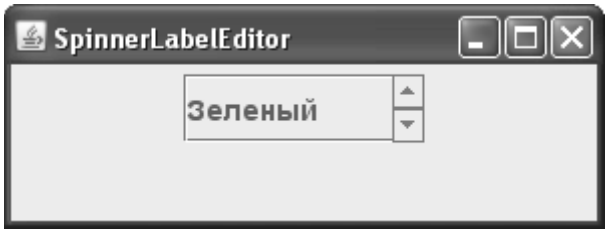
```
// SpinnerLabelEditor.java
// Редактор счетчика JSpinner на основе надписи
import javax.swing.*;
```

```
import javax.swing.event.*;
import java.awt.*;

public class SpinnerLabelEditor extends JFrame {
  // данные для списка
  private String[] data = {
      "Красный", "Зеленый", "Синий"
  };
  public SpinnerLabelEditor() {
    super("SpinnerLabelEditor");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем счетчик
    JSpinner spinner = new JSpinner(
        new SpinnerListModel(data));
    // присоединяем наш редактор
    LabelEditor editor = new LabelEditor();
    spinner.setEditor(editor);
    // регистрируем слушателя
    spinner.addChangeListener(editor);
    // для появления на экране необходимо
    // чтобы до редактора дошло событие
    spinner.getModel().setValue(data[1]);
    // выводим окно на экран
    setLayout(new FlowLayout());
    add(spinner);
    setSize(300, 200);
    setVisible(true);
  }
  // специальный редактор для счетчика
  class LabelEditor extends JLabel
      implements ChangeListener {
    // метод слушателя событий
    public void stateChanged(ChangeEvent e) {
      // получаем счетчик
      JSpinner spinner = (JSpinner)e.getSource();
      // получаем текущий элемент
      Object value = spinner.getValue();
      // устанавливаем новое значение
      if ( value.equals(data[0]) ) {
        setText("<html><h2><font color=\"red\">"
            + value);
```

```
        }
        if ( value.equals(data[1]) ) {
          setText("<html><h3><font color=\"green\">"
              + value);
        }
        if ( value.equals(data[2]) ) {
          setText("<html><h4><font color=\"blue\">"
              + value);
        }
      }
      // размер редактора
      public Dimension getPreferredSize() {
        return new Dimension(100, 30);
      }
    }
    public static void main(String[] args) {
      SwingUtilities.invokeLater(
          new Runnable() {
            public void run() { new SpinnerLabelEditor(); } });
    }
}
```

Здесь в окно добавляется один простой счетчик на основе модели SpinnerListModel, данные которой хранятся в массиве. Самое интересное — это внутренний класс LabelEditor, унаследованный от надписи JLabel и реализующий интерфейс слушателя событий ChangeListener (интерфейс этого слушателя обязательно нужно реализовывать любому редактору, иначе он не сможет узнавать об изменении текущего элемента счетчика). Вся работа происходит в методе слушателя stateChanged(), он вызывается при смене текущего значения счетчика. Мы находим источник события, подразумевая, что это счетчик JSpinner (хотя можно присоединить слушателя и к модели SpinnerModel), получаем текущее значение счетчика и обрабатываем его. Нам известно, что значения у нас всего три, так что каждое из них редактор обрабатывает по своему, «выкрашивая» в подходящий цвет. Обратите внимание на метод getPreferredSize(): он возвращает намеренно больший размер, в противном случае размер надписи менялся бы динамически в зависимости от отображаемого элемента, а это привело бы не к самому лучшему результату на экране.

После создания редактора остается присоединить его к счетчику. Для этого нужно не только передать его счетчику методом setEditor(), но и зарегистрировать как слушателя событий ChangeListener. В примере это сделано не слишком элегантно, лучше было бы передать ссылку на счетчик в конструктор редактора, и уже в этом конструкторе провести регистрацию слушателя. Также имеется и еще одна маленькая хитрость — наша надпись отображает что-либо только после получения события ChangeListener, а пока пользователь не начнет прокручивать счетчик, это событие не произойдет. Нам приходится вручную поменять значение модели, что приведет к запуску события ChangeListener, и обновлению нашей надписи (в противном случае после запуска счетчик отображал бы пустое поле). Пожалуй, проще было бы присоединить модель *после* того, как присоединен редактор. Запустив программу с примером, вы увидите, как отображаются разные элементы счетчика, и это только начало — нам прекрасно известно, на что способна надпись JLabel.




В качестве красочного редактора можно использовать и компонент JEditorPane, примерно так, как мы это делали в главе 11 для раскрывающихся списков JComboBox, и тем самым еще больше повысить гибкость и элегантность своего приложения.

## *Резюме*

Плавно меняющиеся диапазоны данных, сколько бы их ни было в вашем приложении, всегда можно представить с помощью компонентов, описанных в данной главе. Простота компонентов и модели с одной стороны и возможность исчерпывающей настройки всех аспектов работы этих компонентов – с другой позволят вам реализовать именно то поведение, которое необходимо, а пользователю – получить максимальное удовлетворение от работы с вашим приложением.

# Глава 13. Управление пространством

В главе 7 мы уже знакомились с вариантами размещения компонентов в контейнере и знаем, что для этого используются разнообразные менеджеры расположения. С их помощью вы сможете создать любой необходимый вам пользовательский интерфейс и расположить компоненты в контейнере так, как планировалось. Однако не все ситуации можно разрешить только средствами контейнеров и подходящих менеджеров расположения — в вашем приложении, особенно если оно сложное или обладает множеством вариантов настройки, может быть огромное количество разнообразных компонентов, которые просто физически не смогут уместиться на экране. Кроме того, существуют ситуации, когда размер какого-либо компонента намного превышает доступное пространство экрана. В решении проблемы иногда может помочь применение дополнительных диалоговых окон, но их излишнее количество запутывает пользователя, особенно если одни диалоговые окна приходится вызывать из других.

С недостатком места в контейнере призваны справляться специальные компоненты, которые мы и будем рассматривать в этой главе. К таким компонентам, прежде всего, относится панель с вкладками JTabbedPane, которая в удобной и наглядной форме компактно размещает несколько элементов пользовательского интерфейса так, что работать вы можете только с одним из них, пока остальные скрыты. Таким образом можно сэкономить пространство контейнера и при этом разбить его на логически связанные группы.

Далее мы рассмотрим разделяемую панель JSplitPane, в обязанности которой входит гибкое распределение пространства между двумя компонентами. С ее помощью пользователь может сделать больше тот компонент, который его в данный момент интересует, произвольным образом меняя соотношение доступных размеров компонентов.

Наконец, остается панель прокрутки JScrollPane, отображающая только часть некоторого компонента и с помощью специальных полос прокрутки (JScrollBar) легко позволяющая перемещаться к любой другой его части. Первые два компонента довольно просты, и их функции очевидны, так что их изучение не займет у нас много времени, а вот панель прокрутки настолько часто гостит в пользовательских интерфейсах, что ее стоит рассмотреть подробнее, к тому же, как вы узнаете, она способна на многое.

## *Панель с вкладками JTabbedPane*

Как нетрудно догадаться по названию, панель с вкладками JTabbedPane позволяет выводить на экран так называемыми *вкладки* (tabs) — панели с небольшими ярлычками с краю (это может быть любой из краев панелей — верхний, нижний, левый или правый). При щелчке пользователем на ярлычке панель JTabbedPane выводит на экран соответствующие выбранной вкладке элементы пользовательского интерфейса. Таким образом у вас появляется возможность сэкономить место в контейнере: вместо размещения всех компонентов пользовательского интерфейса в одной панели содержимого вы разбиваете их (чаще всего по смыслу и предназначению) на несколько групп, добавляете каждую группу в отдельную панель, и передаете эти панели в компонент JTabbedPane, заодно

указывая, какая надпись (и значок, если захотите) должна соответствовать данной группе компонентов. Пользователь будет видеть только одну группу компонентов (которая занимает гораздо меньше места на экране), а чтобы увидеть и воспользоваться другой группой, нужно перейти на подходящую вкладку. Панель JTabbedPane позаботится о том, чтобы при выборе вкладки на экране появилась соответствующая группа компонентов. Особенно часто панели с вкладками используются в диалоговых окнах настройки приложения: в них бывает много компонентов с различным предназначением, и вкладки не только помогают компактно разместить эти компоненты, но и дают пользователю возможность быстро найти то, что его интересует.

Использовать панель с вкладками JTabbedPane очень просто. В основном работа с ней заключается в вызове метода add() или addTab() (лучше остановить свой выбор на втором методе, его название лучше соответствует производимому действию и параметры этого метода удобнее настраивать), которому необходимо передать компонент, соответствующий вкладке, надпись для ярлычка вкладки и значок, если вы его используете. После добавления всех вкладок панель JTabbedPane выводится на экран, при этом стоит проследить за тем, чтобы занимаемого ею места было достаточно не только для тех вкладок, которые вы добавили в нее, но и для соответствующих этим вкладкам компонентов. Рассмотрим небольшой пример использования панели JTabbedPane и убедимся, что она на самом деле проста:

```
// SimpleTabbedPanes.java
// Использование панелей с вкладками
import javax.swing.*;
import java.awt.*;

public class SimpleTabbedPanes extends JFrame {
  public SimpleTabbedPanes() {
    super("SimpleTabbedPanes");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // первая панель с вкладками
    JTabbedPane tabsOne = new JTabbedPane(
        JTabbedPane.BOTTOM, JTabbedPane.SCROLL_TAB_LAYOUT);
    // добавляем вкладки
    for (int i=1; i<8; i++) {
      JPanel tab = new JPanel();
      tab.add(new JButton("Просто кнопка " + i));
      tabsOne.addTab("Вкладка №: " + i, tab);
    }
    // вторая панель с вкладками
    JTabbedPane tabsTwo = new
        JTabbedPane(JTabbedPane.TOP);
    // добавляем вкладки
    for (int i=1; i<8; i++) {
      JPanel tab = new JPanel();
      tab.add(new JButton("Снова кнопка " + i));
      tabsTwo.addTab("<html><i>Вкладка №: " + i,
```

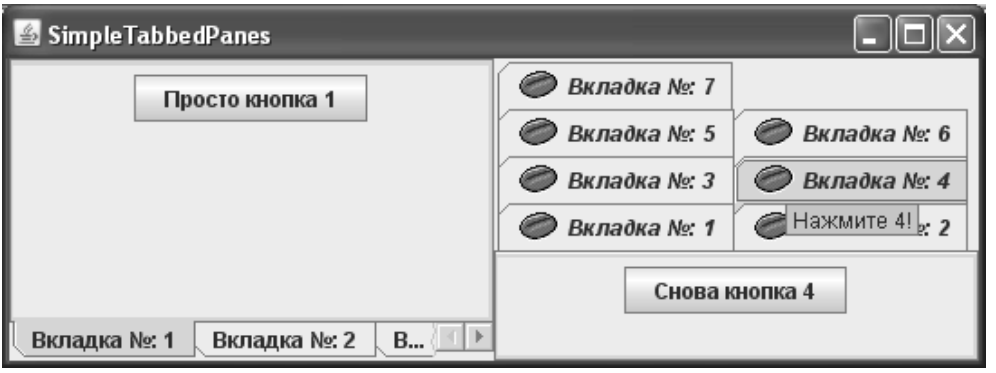
```
            new ImageIcon("icon.gif"),
            tab, "Нажмите " + i + "!");
      }
      // добавляем вкладки в панель содержимого
      setLayout(new GridLayout());
      add(tabsOne);
      add(tabsTwo);
      // выводим окно на экран
      setSize(600, 250);
      setVisible(true);
    }
    public static void main(String[] args) {
      SwingUtilities.invokeLater(
          new Runnable() {
            public void run() { new SimpleTabbedPanes(); } });
    }
}
```

В примере мы создаем окно (довольно большого размера — нам нужно, чтобы в нем поместились все наши вкладки) и размещаем в нем две панели с вкладками JTabbedPane. Первая панель создается с помощью конструктора, принимающего два параметра: первый отвечает за расположение ярлычков вкладок (мы выбрали расположение внизу панели, но вы можете выбрать любую из четырех сторон панели, каждой из сторон соответствует своя константа класса JTabbedPane), а второй позволяет указать, как вкладки будут располагаться на экране в том случае, если их не удастся разместить в один ряд. По умолчанию вкладки располагаются в несколько рядов (аналогично действует менеджер последовательного расположения FlowLayout, когда ему не хватает места для размещения всех компонентов в одну строку), такому поведению соответствует константа WRAP_TAB_LAYOUT, но для первой панели мы выбрали второй вариант расположения вкладок (которому отвечает константа SCROLL_TAB_LAYOUT): в одну строку вместе с небольшими кнопками прокрутки. Такое расположение вкладок позволяет сэкономить место, хотя и не дает возможности видеть ярлычки сразу всех вкладок. Вкладки мы добавляем в цикле методом addTab(), в качестве содержимого каждой вкладки используется панель JPanel, в которую помещена кнопка. Чтобы вы могли отличать вкладки и их содержимое друг от друга, мы добавили в надписи на ярлычках вкладок и на кнопках их порядковые номера. Метод addTab() очень прост: ему нужно передать строку с названием вкладки и ее содержимое.

Вторая панель JTabbedPane создается с помощью более простого конструктора: в нем мы указываем лишь расположение ярлычков вкладок (наверху панели), а способ размещения вкладок на экране оставляем заданным по умолчанию (как мы уже упоминали, по умолчанию вкладки располагаются в несколько рядов). Вкладки мы добавляем в цикле, так же как и в первом случае, но на этот раз используем более функциональную перегруженную версию метода addTab(). Она принимает сразу четыре параметра: надпись для ярлычка вкладки, значок, компонент, который будет соответствовать вкладке, и текст всплывающей подсказки (панель JTabbedPane позаботится о том, чтобы у каждой вкладки была собственная подсказка, если вы ее задаете). Кроме того что вы можете задать текст вместе со значком, для текста доступен прекрасно вам «знакомый» формат HTML: надо лишь указать в начале строки «волшебные» символы <html> (впрочем, текст может быть и пустой ссылкой null, в таком случае у вкладки будет только значок). Учитывая,

что язык HTML можно также использовать в тексте подсказки для вкладки, а к тексту ярлычка вкладки нетрудно сразу же присоединить значок, у вас есть отличный способ сообщить пользователю все необходимое о вкладках в вашем интерфейсе, так что ему не придется слишком часто обращаться к системе помощи и работать он сможет гораздо эффективнее.

Созданные панели с вкладками JTabbedPane мы добавляем в наше окно, для которого предварительно устанавливается табличное расположение GridLayout (конструктор GridLayout без параметров позволяет располагать компоненты в одну строку). Благодаря этому окно будет разделено на две одинаковые части, в которых и расположатся наши панели с вкладками. Остается только вывести окно на экран. Запустив программу с примером, вы сможете оценить различные варианты расположения вкладок и их внешний вид, а в случае со второй панелью с вкладками посмотреть на всплывающие подсказки.




Описанные нами параметры панели JTabbedPane и отдельных вкладок можно изменять не только при вызове конструктора или метода addTab(). В классе JTabbedPane все свойства, использованные нами, описаны как свойства JavaBeans с набором методов get/set, так что параметры панели с вкладками и сами вкладки в любой момент нетрудно изменить, вызвав соответствующие методы. Опишем теперь все задействованные нами свойства, чтобы они всегда были у вас «под рукой» (табл. 13.1).

**Таблица 13.1.** Основные свойства панели с вкладками JTabbedPane

| Свойства (и методы get/set) | Описание |
|---|---|
| tabPlacement | Задает расположение ярлычков вкладок у одной из четырех сторон панели. По умолчанию (если вызывается конструктор JTabbedPane без параметров) ярлычки располагаются вверху панели (TOP). Располагать ярлычки справа или слева лучше, если связанные с вашими вкладками компоненты занимают по оси Y больше места, чем по оси X, в противном случае ярлычки удобнее поместить сверху или снизу |
| tabLayoutPolicy | Определяет, как будут располагаться вкладки в том случае, если в контейнере не хватит места для их размещения в один ряд. Как мы уже отмечали, режим WRAP_TAB_LAYOUT (используемый по умолчанию) позволяет разместить вкладки в несколько рядов, а режим SCROLL_TAB_LAYOUT — оставить их в одном ряду с добавлением кнопки прокрутки. В большинстве случаев пользователю лучше подойдет режим с прокруткой, так как несколько рядов вкладок занимают слишком много места на экране (особенно когда вкладок много), что затрудняет переход на нужную вкладку |

**Таблица 13.1** *(продолжение)*

| Свойства (и методы get/set) | Описание |
|---|---|
| titleAt, iconAt | Эти свойства позволяют задать надпись и значок для ярлычка любой вкладки (не забывайте о том, что у вас в руках вся «магия» HTML). В качестве первого параметра требуется указать индекс вкладки (отсчет ведется с нуля) |
| toolTipTextAt | Это свойство отвечает за текст всплывающей подсказки к каждой вкладке. Первым параметром метода также является индекс вкладки. |
| componentAt | Данное свойство позволяет получить или заменить компонент, который показывает вкладка уже после того, как вы настроили ее методом addTab(). Не следует путать со свойством tabComponentAt, которое отвечает за саму вкладку и которое мы рассмотрим чуть позже |
| foregroundAt, backgroundAt | Удобные свойства для задания цвета шрифта и фона отдельной вкладки (которая, как нетрудно догадаться, задается индексом). В примере мы не использовали эти свойства, но иногда они помогают просто и быстро выделить одну из вкладок (или щедро расцветить все вкладки, если это соответствует интерфейсу вашего приложения) |

Все свойства панели с вкладками JTabbedPane довольно стандартны, отметить можно лишь необычный формат этих свойств и суффикс «At» в конце их названий. Эти свойства индексированные (в качестве первого параметра методов get/set нужно указывать числовой индекс), так как вкладок в панели может быть много, а свойства надо задавать отдельно для каждой из них. Вам не обязательно добавлять все вкладки и соответствующие им компоненты перед выводом панели с вкладками на экран, это вполне можно делать динамически, прямо во время работы программы, вызывая метод addTab() в нужный момент. Это может пригодиться, если какая-то часть вашего пользовательского интерфейса должна стать доступной позже остальных (хотя, как мы увидим чуть позже, можно отключить вкладку и включать ее при необходимости). Добавлять вкладки можно не только методом addTab(). Вы можете попробовать метод insertTab(), параметры которого аналогичны параметрам метода addTab() и который позволяет вставить вкладку в произвольное место панели JTabbedPane.

Таким образом, обычно работа с панелью JTabbedPane сводится к заданию текста (и/или значка) для ярлычка вкладки и ее содержимого, в качестве которого чаще всего выступает панель JPanel с группой компонентов вашего пользовательского интерфейса (хотя содержимым вкладки может быть любой компонент, унаследованный от базового класса JComponent). При необходимости в панель нетрудно динамически добавить новые вкладки или изменить все свойства уже имеющихся вкладок.

## Модель выделения и обработка событий

Панель с вкладками JTabbedPane чаще всего используется в качестве вспомогательного инструмента, позволяющего эффективно распределить доступное пространство контейнера, и поэтому дополнительные действия с ней проводить требуется крайне редко. На самом деле, вы сопоставляете вкладкам набор компонентов вашего пользовательского интерфейса, а о переключении вкладок вам заботиться не нужно. Когда пользователь переходит на другую вкладку (щелчком мыши или нажатием клавиш), все необходимые действия выполняют внутренние механизмы класса JTabbedPane, так что для программиста все в итоге сводится к выводу панели с вкладками на экран.

Впрочем, при необходимости вы можете расширить «свое сотрудничество» с панелью JTabbedPane и более подробно отслеживать, что с ней происходит. Относится это

прежде всего к переключению вкладок. Вы можете следить за этим и в любой момент времени узнать, какая вкладка активна, поскольку у класса JTabbedPane есть помощник — модель выделения единственного элемента SingleSelectionModel. Эта модель и хранит информацию об активной вкладке и при необходимости (когда ей сообщает об этом UI-представитель) меняет текущий индекс вкладки. Модель SingleSelectionModel является, пожалуй, самой простой моделью из всех моделей Swing. Она хранит целое число — индекс активной вкладки (когда это число равно –1, считается, что в данный момент активных вкладок нет) и поддерживает методы для присоединения и отсоединения слушателей ChangeListener, которые оповещаются при смене активной вкладки. Действительно, невозможно придумать модель проще. По умолчанию в панели JTabbedPane используется стандартная реализация этой модели, класс с именем DefaultSingleSelectionModel, и вряд ли вам понадобится вмешиваться в его работу или писать свою модель «с нуля». Хранение одного числа не стоит таких усилий, с ним вполне может справиться и стандартная модель. Тем не менее, это самая настоящая модель, так что вы можете разделять ее между несколькими панелями с вкладками, сохраняя в ней информацию о выделенном элементе. Такое поведение может вам пригодится, если у вас в программе есть несколько одинаковых панелей JTabbedPane, активная вкладка в которых должна быть всегда одной и той же.

Единственное событие, поддерживаемое панелью с вкладками JTabbedPane (за исключением низкоуровневых событий, общих для всех графических компонентов), относится именно к модели ее активизации. Добавив к панели JTabbedPane (или к ее модели выделения, это одно и то же, различным будет лишь источник события) слушателя ChangeListener, вы сможете узнавать об активизации следующей вкладки. Практическое применение такому событию придумать непросто, однако оно может быть полезно при реализации разного рода отложенных вычислений. Например, если какая-либо вкладка содержит сложные компоненты, выводить на экран которые дорого и долго, вы можете отложить их создание до того момента, когда пользователь реально перейдет на эту вкладку. Он может этого и не сделать — тогда вы сэкономите его время и ресурсы компьютера. Кроме того, панель JTabbedPane позволяет узнать, в каком месте экрана находится та или иная вкладка, а также к какой вкладке относится некоторая точка экрана. Иногда, при нестандартной обработке событий от мыши, такая возможность бывает полезной, например, при реализации своей системы помощи, когда для каждой вкладки имеются свои советы и свои разделы в справочной системе (прототип подобной системы был создан нами в главе 6).

Создадим несложный пример и посмотрим, как можно использовать события панели с вкладками JTabbedPane:

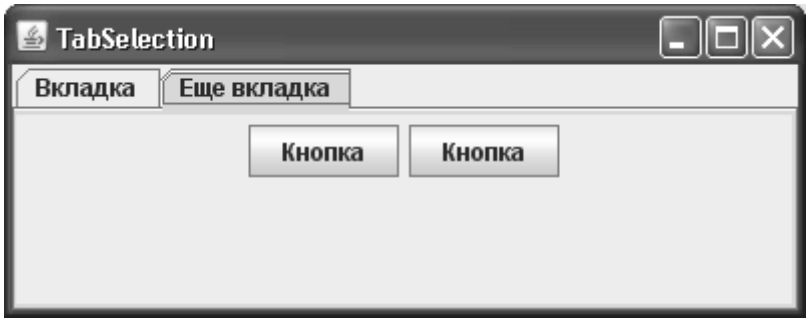
```
// TabSelection.java
// Работа с активными вкладками и обработка событий
import javax.swing.*;
import javax.swing.event.*;
import java.awt.event.*;
import java.awt.*;

public class TabSelection extends JFrame {
  public TabSelection() {
    super("TabSelection");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем нашу панель с вкладками
    JTabbedPane tabs = new JTabbedPane();
```

```
    tabs.addTab("Вкладка", new JPanel());
    tabs.addTab("Еще вкладка", new JPanel());
    // добавляем слушателя событий
    tabs.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
        // добавляем на вкладку новый компонент
        JPanel panel = (JPanel)
            ((JTabbedPane)e.getSource()).
                getSelectedComponent();
        panel.add(new JButton("Кнопка"));
      }
    });
    // работа с низкоуровневыми событиями
    tabs.addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        // узнаем, на какой вкладке был щелчок
        int idx = ((JTabbedPane)e.getSource()).
            indexAtLocation(e.getX(), e.getY());
        JOptionPane.showMessageDialog(
            null, "Index: " + idx);
      }
    });
    // выводим окно на экран
    add(tabs);
    setSize(400, 300);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new TabSelection(); } });
  }
}
```

Здесь мы создаем небольшое окно, в которого добавляем панель с вкладками JTabbedPane. Для создания последней используется конструктор без параметров, так что ярлычки вкладок будут располагаться наверху панели (так принято по умолчанию). Сама панель с вкладками очень проста: мы добавляем в нее только две вкладки, которым сопоставляем пустые панели JPanel (как вы помните, по умолчанию в таких панелях имеет место последовательное расположение FlowLayout). В эти панели мы будем динамически добавлять компоненты. Интереснее посмотреть на то, как к панели с вкладками присоединяются слушатели событий. Первый слушатель (ChangeListener) следит за переходами между вкладками. Можно было бы присоединить его и к модели выделения SingleSelectionModel, но нам в примере удобнее получать в качестве источника события

саму панель с вкладками, так что слушателя мы присоединяем к ней. Когда пользователь переходит на другую вкладку, вызывается метод stateChanged() нашего слушателя, в котором мы определяем источник события (объект JTabbedPane), с помощью удобного метода getSelectedComponent() получаем компонент, соответствующий активной вкладке, преобразуем его к панели JPanel (а мы знаем, что нашим вкладкам соответствуют панели) и добавляем в нее новую кнопку JButton. Таким образом каждый раз при смене вкладки в нее добавляется еще одна кнопка. Не слишком полезное поведение программы, но как мы уже упомянули, в реальных приложениях примерно таким же образом можно реализовать отложенные вычисления.

Далее демонстрируется, как можно работать с панелью JTabbedPane на самом низком уровне. К ней присоединяется слушатель событий от мыши, позволяющий следить за щелчками мыши внутри нашей панели с вкладками. Когда пользователь щелкает мышью где-то на панели с вкладками, нам передаются только координаты щелчка. Определить, на какой вкладке щелкнул пользователь, позволяет метод indexAtLocation(). Он возвращает индекс вкладки, если пользователь щелкнул на вкладке, или –1, если щелчок был не на вкладке, а где-то в другом месте панели. Вы можете применять подобный подход, к примеру, если вам потребуется отдельное выпадающее меню для каждой вкладки, а не единственное меню для компонента JTabbedPane целиком.




Запустив программу с примером, вы сможете увидеть, как панель с вкладками позволяет следить за активной вкладкой и динамически обновлять панель содержимого при переходе на другую вкладку. Вы также сможете увидеть, как работает метод indexAtLocation(), и при случае использовать его в своих программах, хотя такая необходимость возникает редко.

## Дополнительные возможности компонента JTabbedPane

Рассмотренных нами возможностей панели с вкладками JTabbedPane с лихвой хватает для ее традиционной работы в хорошем пользовательском интерфейсе: вы можете задавать любые надписи и значки для ярлычков вкладок, сопоставлять с вкладками любые компоненты, следить за изменениями состояния вкладок, и пользователь без затруднений сможет переключаться между вкладками щелчками мыши или нажатиями клавиш (список поддерживаемых панелью JTabbedPane сочетаний клавиш вы можете найти в интерактивной документации Java, эти клавиатурные сокращения практически идентичны для всех стандартных внешних видов). Тем не менее, создатели Swing на этом не остановились и добавили в класс JTabbedPane несколько возможностей, которые помогут сделать ваш пользовательский интерфейс отточенным и продуманным до мелочей.

Очень полезной является возможность задать для вкладки мнемонику, знакомую нам «со времен» изучения глав 8 и 9. Как вы помните, это сочетание управляющей клави-

ши (как правило, Alt) и клавиши латинского символа (проблемы с русскими символами мы уже обсудили), при нажатии которых подходящая вкладка мгновенно активизируется. Мнемоники позволяют максимально ускорить процесс работы с приложением для опытных пользователей, позволяя перемещаться по интерфейсу молниеносными комбинациями клавиш. Однако, как мы уже знаем, в Swing из-за проблем с символами других языков число мнемоник довольно ограничено (приходится укладываться в 26 символов латинского алфавита). Впрочем, панель с вкладками чаще всего встречается в диалоговых окнах, а в отдельных окнах все сочетания клавиш собственные, так что за нехватку символов, как правило, можно не опасаться.

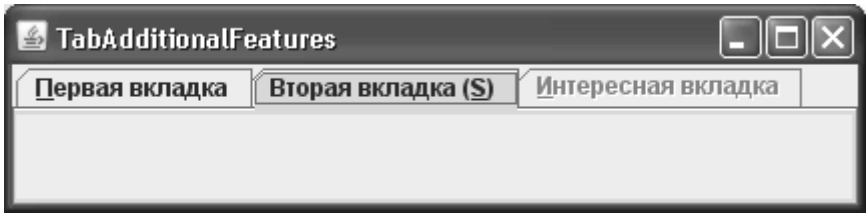
Также весьма полезной является возможность отключения вкладок. Она особенно эффективна в приложениях, работающих с несколькими видами документов или с многочисленными пользователями, когда доступность некоторых частей пользовательского интерфейса определяется тем, кто или как работает с приложением. Отключив вкладку, вы отключаете сразу целую группу элементов пользовательского интерфейса и недвусмысленно даете понять пользователю, что данные функции в текущем режиме работы приложения недоступны.

Рассмотрим небольшой пример использования описанных возможностей панели с вкладками, чтобы сразу же освоиться с ними:

```
// TabAdditionalFeatures.java
// Дополнительные возможности панелей с вкладками
import javax.swing.*;
import java.awt.*;

public class TabAdditionalFeatures extends JFrame {
  public TabAdditionalFeatures() {
    super("TabAdditionalFeatures");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // панель с вкладками
    JTabbedPane tabs = new JTabbedPane();
    tabs.addTab("Первая вкладка", new JPanel());
    tabs.addTab("Вторая вкладка (S)", new JPanel());
    tabs.addTab("Интересная вкладка", new JPanel());
    // зададаем мнемоники
    tabs.setMnemonicAt(0, 'П');
    tabs.setMnemonicAt(1, 'S');
    tabs.setMnemonicAt(2, 'И');
    // активизируем последнюю вкладку
    tabs.setEnabledAt(2, false);
    // выводим окно на экран
    add(tabs);
    setSize(430, 300);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
```

```
            new Runnable() {
              public void run() { new TabAdditionalFeatures(); } });
    }
}
```

Мы наследуем наш класс от окна с рамкой JFrame и добавляем в панель содержимого панель с вкладками JTabbedPane. У нас будет три вкладки, для простоты мы сопоставляем им пустые панели JPanel. Для каждой из вкладок мы устанавливаем мнемоники методом setMnemonicAt(); этому методу помимо символа мнемоники нужно указать индекс вкладки, для которой меняется мнемоника. Для двух вкладок мы передали в качестве мнемоник русские символы (и вы увидите их подчеркнутыми на экране, только вот работать они, увы, не будут; подробно причины и возможные решения этой проблемы мы обсуждали в главе 9), а для третьей использовали не слишком эстетичный, но зато надежный способ, включив в мнемонику латинский символ, который мы к тому же добавили к русской надписи в скобках, так что пользователь сможет увидеть мнемонику вкладки. Помимо мнемоник пример демонстрирует и отключение вкладок: методом setEnabledAt() мы отключили последнюю (судя по надписи, самую интересную) вкладку, так что перейти на нее пользователь не сможет.




Запустив программу с примером, вы сможете увидеть, как работает мнемоника с латинским символом и как не работают мнемоники с символами русскими (к этому мы уже успели привыкнуть), а также убедитесь в том, что на отключенную вкладку перейти на самом деле нельзя, как бы ни было интересно ее содержимое. Мы уже обсудили ситуации, в которых можно применить это свойство панелей с вкладками, оно на самом деле довольно полезно.

## Размещение компонентов во вкладках

До сих пор рассматриваемые нами вкладки были полностью информационными компонентами — мы могли размещать на них текст и значки, а компонент JTabbedPane сам отслеживал способы их переключения и решал, как рисовать на них. Однако в последнее время вкладки наделяются все большими полномочиями, в качестве самого популярного расширения можно вспомнить маленькие кнопки на вкладках, закрывающие их, если интерфейс приложения подразумевает такое поведение (вспомните все популярные браузеры). Библиотека Swing также не осталась в стороне и позволяет тонко настроить внешний вид вкладок.

Вспомнив другие компоненты библиотеки Swing, можно было бы подумать, что компонент JTabbedPane предлагает рисовать вкладки отдельному рисующему объекту (renderer), как мы это уже видели в списках и еще не раз увидим. Однако в случае с вкладками все одновременно и проще и сложнее. Как таковой «прорисовки» нет, вы просто можете добавить в то место, где рисуется вкладка, свой собственный компонент, будь то простая кнопка или плотно упакованная компонентами сложная составная панель. Казалось бы, нет ничего проще, чем добавить свой компонент и нарисовать в нем все, что нам нужно, однако здесь есть свои весьма острые подводные камни. Давайте сразу же

рассмотрим несложный пример, в котором попробуем добавить кнопку для закрытия вкладки и флажок, а затем обсудим все его тонкости:

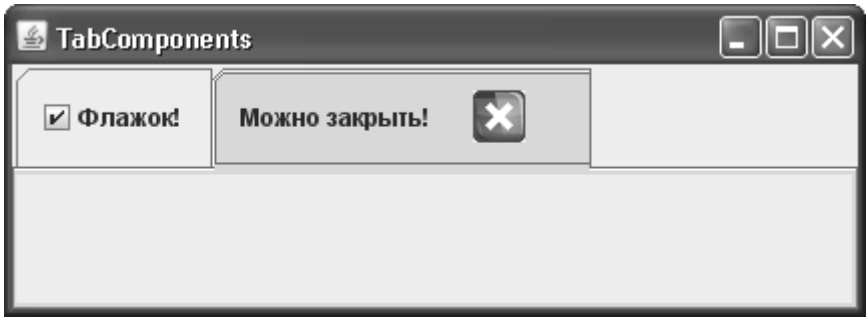
```
// TabComponents.java
// Размещение компонентов во вкладках
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;

public class TabComponents extends JFrame {
  public TabComponents() {
    super("TabComponents");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // панель с вкладками
    final JTabbedPane tabs = new JTabbedPane();
    tabs.addTab(null, new JPanel());
    tabs.addTab(null, new JPanel());
    // флажок во вкладке
    JCheckBox checkBox = new JCheckBox("Флажок!");
    checkBox.setOpaque(false);
    tabs.setTabComponentAt(0, checkBox);
    // вкладка с надписью и кнопкой закрытия
    final JPanel panel = new JPanel();
    JLabel label = new JLabel("Можно закрыть!");
    JButton closeButton = new JButton(new AbstractAction() {
      {
        putValue(SMALL_ICON, new ImageIcon("close.png"));
      }
      public void actionPerformed(ActionEvent e) {
        // нужно определить вкладку, в которой находится кнопка
        tabs.removeTabAt(
            tabs.indexOfTabComponent(panel));
      }
    });
    closeButton.setBorderPainted(false);
    closeButton.setContentAreaFilled(false);
    panel.setOpaque(false);
    panel.add(label);
    panel.add(closeButton);
    tabs.setTabComponentAt(1, panel);
    // выводим окно на экран
    add(tabs);
```

```
    setSize(430, 300);
    setVisible(true);
  }

  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new TabComponents(); } });
  }
}
```

Пример очень прост — мы создаем всего лишь одну панель со вкладками JTabbedPane и добавляем ее в центр окна. В ней будут находиться две вкладки, содержащие две пустые панели. Так как мы собираемся отдать отображение вкладок отдельным компонентам, вместо названия вкладок в метод addTab() первым параметром передаются пустые ссылки null.

Далее настает черед настроить компоненты. Первой вкладкой будет заниматься флажок JCheckBox. Мы создаем его и устанавливаем свойство непрозрачности (opaque) в false, чтобы он своим цветом фона не перекрывал фон и внешний вид вкладки, в противном случае картина получится не слишком привлекательная. Не во всех внешних видах Swing флажки непрозрачны, однако во внешнем виде по умолчанию это именно так, и на всякий случай стоит всегда менять это свойство. После этого нам остается указать, что для первой вкладки мы будем использовать флажок методом setTabComponentAt(), требующим в качестве параметров номер вкладки и ссылку на компонент.

Для второй вкладки настраивается отдельная панель, которая будет отображать название вкладки (с помощью надписи) и хранить кнопку для закрытия вкладки. Если с надписью все очевидно, то кнопка создается на основе команды Action, что позволяет нам «одним махом» указать и значок кнопки, и описать, что будет происходить при нажатии кнопки[1]. Закрыть вкладку оказывается не совсем просто — как нам узнать, на какой вкладке пользовать нажал кнопку? Следить за курсором мыши в этом случае не слишком-то рационально — в разных внешних видах и операционных системах кнопку можно нажать по-разному. В примере нам помогает метод indexOfTabComponent() класса JTabbedPane, который позволяет узнать, на какой вкладке находится данный компонент. Так как наша кнопка находится во вспомогательной панели, нам приходится передавать ее в этот метод, чтобы узнать, в какой же вкладке мы находимся, и наконец удалять ее. Если бы у нас было много вкладок с кнопками, нам бы пришлось хранить в слушателе нажатия кнопки ссылку на панель, которая размещается во вкладке.




[1] Мы подробно обсуждали команды Action в главе 9, посвященной кнопкам.

Запустив пример, вы увидите совершенно отдельные компоненты на вкладках, и сможете с легкостью закрыть вторую вкладку. Однако нетрудно будет заметить довольно много мелочей, которые делают работу с вкладками не такой гладкой, как нам хотелось бы:

- Флажок «отбирает» у вкладки нажатие мыши для того чтобы сменить свое состояние, и вкладка в этом случае не переключается, что с точки зрения пользователя неудобно и недопустимо. Решением станет дополнительный слушатель событий от мыши, который в этом случае станет переключать вкладку, даже если пользователь щелкнул на флажке. Еще один вопрос — если пользователь переключился на вкладку, не нажимая на флажок, нужно ли переключать последний? Если да, нужно будет отдельно следить и за этим событием.
- У флажка и у кнопки закрытия есть свой фокус ввода. Эффект получается такой, как если бы вкладка получала фокус несколько раз. Чтобы сделать работу более приятной, скорее всего, имеет смысл отключить фокус ввода для кнопок, если это приемлемо.
- Мы задаем текст для флажка и надписи при их создании, но что будет, если надпись для вкладки поменяется динамически во время работы программы? Опять-таки необходимо следить за этим вручную, неплохим вариантом может быть слежение за привязанным свойством «indexForTitle», с помощью метода JavaBeans addPropertyChangeListener(), доступного в каждом компоненте Swing.

Как видно, для того чтобы вкладки с отдельными компонентами работали неотличимо от своих стандартных собратьев, нужны немалые усилия и тонкая настройка внешнего вида. Наверняка некоторые готовые решения можно найти среди бесплатно доступных библиотек, однако в общем случае стоит применять стандартные вкладки, а возможность размещать на них компоненты, как бы она не была привлекательна, использовать лишь в случае, когда это действительно оправдано, а не просто для украшения интерфейса.

## *Разделяемая панель JSplitPane*

Разделяемая панель JSplitPane служит для гибкого распределения пространства между двумя компонентами, разместить которые на экране одновременно обычным порядком (просто добавляя в контейнер) по каким-либо причинам не получается (чаще всего из-за нехватки места в этом самом контейнере). Сама по себе разделяемая панель незатейлива и не производит особого впечатления: это разделенный тонкой полосой контейнер с двумя содержащимися в нем компонентами. Перетаскивая мышью или назначенными внешним видом клавишами полосу (так называемую *вешку разбивки*), пользователь может увеличивать размер одного компонента, «отбирая» пространство у другого компонента.

Использовать разделяемую панель особенно удобно там, где есть требовательные к пространству контейнера компоненты, одновременная работа с которыми маловероятна. Работая с одним компонентом, пользователь сможет для удобства увеличить его, а при переключении на другой компонент увеличить размер последнего. В качестве примера можно вспомнить разнообразные среды разработки программ, использующие разделяемые панели для размещения редактора кода или форм и дерева проекта. При работе с проектом вам нужно больше места под его дерево и таблицу свойств, а при написании кода место требуется уже под редактор и диагностические сообщения компилятора. Разделяемая панель в этом случае случает удачной альтернативой многооконному интерфейсу (MDI), который в последнее время потерял свои позиции при проектировании интерфейсов.

Теперь рассмотрим пример, в котором создадим несколько разделяемых панелей JSplitPane, применив самые полезные и часто используемые их возможности, после чего обсудим все поподробнее:

```
// UsingSplitPanes.java
// Использование разделяемых панелей
import javax.swing.*;
import java.awt.*;

public class UsingSplitPanes extends JFrame {
  // этот значок будем использовать в надписях
  private Icon icon = new ImageIcon("image.jpg");
  public UsingSplitPanes() {
    super("UsingSplitPanes");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // первая разделяемая панель
    JSplitPane splitMain = new JSplitPane();
    splitMain.setOneTouchExpandable(true);
    // размер полосы
    splitMain.setDividerSize(20);
    // вертикальная разделяемая панель
    JSplitPane split2 = new JSplitPane(
        JSplitPane.VERTICAL_SPLIT, true);
    // настроим ее компоненты
    split2.setTopComponent(
        new JScrollPane(new JLabel(icon)));
    split2.setBottomComponent(
        new JScrollPane(new JLabel(icon)));
    // настроим компоненты первой панели
    splitMain.setLeftComponent(
        new JScrollPane(new JLabel(icon)));
    splitMain.setRightComponent(split2);
    // добавим панель и выведем окно на экран
    add(splitMain);
    setSize(600, 400);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new UsingSplitPanes(); } });
  }
}
```

В примере мы создаем небольшое окно, в центре панели содержимого которого разместится разделяемая панель JSplitPane. Разделяемые панели могут быть вертикальными и горизонтальными, в вертикальных панелях компоненты располагаются друг над другом, а в горизонтальных — в ряд, слева и справа. По умолчанию создается горизонтальная разделяемая панель. Для первой разделяемой панели (splitMain) мы использовали конструктор без параметров, он создает горизонтальную разделяемую панель, а в качестве компонентов задействует две кнопки (чтобы не оставлять место пустым, но вы всегда можете установить в разделяемой панели собственные компоненты). Сразу же после создания компонента мы изменили два свойства разделяемой панели. Свойство oneTouchExpandable мы установили в true; оно позволяет добавить к вешке разбивки компонентов две маленькие кнопки со стрелками, щелкая на которых пользователь сможет мгновенно переместить вешку в крайнее положение, полностью предоставив все пространство разделяемой панели одному из компонентов. Мы также поменяли значение свойства dividerSize, которое управляет размером (в пикселах) полосы, образующей вешку разбивки. По умолчанию размер полосы задается UI-представителем разделяемой панели, мы выбрали его на свой вкус, немного большим, чем обычно.

Вторая разделяемая панель создается гораздо более интересным конструктором. Он принимает два параметра: первый задает тип панели (вертикальная или горизонтальная), а второй позволяет сразу же определить, будут ли компоненты при перемещении вешки разбивки непрерывно обновляться (перерисовываться и, если это сложный компонент, проводить проверку корректности). По умолчанию непрерывное обновление отключено (как для первой созданной нами разделяемой панели), мы включаем его. Непрерывное обновление можно включить (или выключить) и после создания разделяемой панели, меняя значение свойства continuousLayout. Как правило, непрерывное обновление стоит сразу же включать, тогда при перемещении вешки разбивки пользователь сразу увидит, как открываются новые части компонента, которому он предоставляет дополнительное место, так что оценить, сколько места понадобится компоненту для комфортной работы, будет гораздо проще.

После создания разделяемых панелей и нехитрой настройки настает пора задать компоненты, которые они должны разделять. Для панели, созданной второй по счету (вертикальной), мы задействуем для задания компонентов методы с красноречивыми именами setTopComponent() и setBottomComponent(), позволяющие указать верхний и нижний компонент, соответственно. Компонентами являются две надписи JLabel с одинаковыми значками. Обратите внимание на то, что даже для таких несложных компонентов, как надписи, были применены панели прокрутки JScrollPane. Использовать панели прокрутки для компонентов, содержащихся в разделяемой панели, приходится почти всегда: дело в том, что разделяемая панель учитывает минимальный размер компонента и не позволяет делать его размер меньше. Если минимальный размер компонента значим (к примеру, минимальный размер надписи со значком равен размеру значка), толку от разделяемой панели будет мало — она просто не позволит изменять размеры компонентов и гибко распределять пространство контейнера. Учитывая это, необходимо включать компоненты в панели прокрутки. Исключение составляют лишь компоненты, минимальный размер которых равен нулю или невелик: к таким компонентам относятся панели JPanel, «рабочий стол» JDesktopPane и практически все остальные вспомогательные контейнеры, служащие для размещения других компонентов, в том числе и сама разделяемая панель JSplitPane. Их вы можете добавлять в разделяемую панель напрямую.

Далее мы задаем компоненты для горизонтальной разделяемой панели, созданной нами первой. Названия методов здесь также красноречивы: методы setLeftComponent() и setRightComponent() позволяют задать левый и правый компоненты панели. Интересно, что уже использованные нами для вертикальной панели методы setTopComponent()

и setBottomComponent() на самом деле вызывают методы, задающие левый и правый компоненты, и определены только для ясности и удобства программиста. Левым компонентом будет надпись JLabel с тем же значком, включенная в панель прокрутки, а вот правым компонентом станет ничто иное, как уже созданная и настроенная нами вертикальная разделяемая панель. Такое поведение вполне возможно, так что вы без труда сможете сделать из своего контейнера настоящую мозаику, добавляя в разделяемые панели JSplitPane такие же панели. Вот только увлекаться этим не стоит: если в разделяемую панель вложено больше одной такой же панели, интерфейс запутает пользователя и не придаст вашему приложению привлекательности. Лучше вообще ограничиться единственным экземпляром JSplitPane.


Интересно, что разделяемая панель JSplitPane относится к тем немногим контейнерам, которые переопределяют метод isValidateRoot() базового класса JComponent и возвращают в нем true. Мы обсуждали это свойство и улучшенный механизм проверки корректности Swing в главе 5. Здесь все логично: при изменении размеров одного из компонентов разделяемой панели может измениться положение полосы и, соответственно, изменятся размеры другого компонента, но все изменения произойдут в пределах разделяемой панели, и проводить проверку корректности «за ее рубежами» не понадобится.

## Свойства разделяемой панели

Запустив программу с примером, вы сможете увидеть разделяемые панели в действии, а также оценить удобство мгновенного перемещения полосы прокрутки (маленькими кнопками со стрелками) и непрерывного обновления компонентов. Ну а все использованные нами в примере свойства мы перепишем, чтобы держать их «под рукой» в качестве справочника (табл. 13.2).

**Таблица 13.2.** Самые полезные свойства разделяемой панели JSplitPane

| Свойства | Описание |
|---|---|
| orientation | Задает ориентацию разделяемой панели. Разделяемая панель может быть горизонтальной (по умолчанию создается именно горизонтальная панель) или вертикальной. В горизонтальной панели компоненты располагаются слева и справа вешки разбивки, а в вертикальной — сверху и снизу |

**Таблица 13.2** *(продолжение)*

| Свойства | Описание |
|---|---|
| continuousLayout | Позволяет включить или выключить режим непрерывного обновления компонентов при перемещении вешки разбивки. По умолчанию непрерывное обновление выключено, но его имеет смысл включать: непрерывное обновление позволяет создать более «отзывчивый» интерфейс, потому что пользователь сможет в реальном времени видеть, какая часть компонента открывается при перемещении вешки разбивки. Отключение обновления имеет смысл лишь при серьезных проблемах с производительностью перерисовки. |
| dividerSize | Управляет размером (в пикселах) вешки разбивки |
| oneTouchExpandable | Установив данное свойство в true, вы добавляете к вешке разбивки две маленькие кнопки, позволяющие «одним мановением руки» предоставить все пространство панели только одному из компонентов. По умолчанию это свойство равно false, но лучше установить его в true: особого труда это не требует, а мгновенное распределение пространства удобно для пользователя |
| leftComponent, topComponent | С помощью этих свойств (и методов get/set) вы сможете установить левый (в терминах горизонтальной панели) или верхний (в терминах вертикальной) компонент. Используйте то свойство, которое вам больше по душе; как мы уже знаем, они устанавливают один и тот же компонент, различны лишь названия |
| rightComponent, bottomComponent | Эти свойства позволяют задать компонент для правой (или, что то же самое, для нижней) части разделяемой панели. Как и в случае с предыдущими свойствами, компонент один и тот же, а выбрать вы можете то свойство, название которого лучше подходит для вашей панели (вертикальной или горизонтальной) |

Действуя примерно так, как мы это делали в примере, и манипулируя описанными в таблице свойствами, вы сможете разделить любые компоненты и предоставить пользователю возможность в любой момент перераспределить занятое ими пространство контейнера.

Управлять положением вешки разбивки разделяемой панели можно и программно с помощью нескольких дополнительных свойств класса JSplitPane (табл. 13.3). Также есть возможность указать, как должно распределяться свободное место в том случае, если размер разделяемой панели увеличится.

**Таблица 13.3.** Дополнительные свойства разделяемой панели

| Свойства | Описание |
|---|---|
| dividerLocation | Для этого свойства имеется два перегруженных метода set: первый принимает в качестве параметра целое число (int), указывающее, сколько пикселов от левой (или, что то же самое, верхней) границы разделяемой панели должно быть до полосы. Вторая версия метода требует параметр типа double, причем значение его должно лежать в границах от нуля до единицы. Фактически это значение в процентах (если умножить его на сто, конечно), показывающее, какую часть разделяемой панели будет занимать левая (верхняя) ее часть. Указав, к примеру, значение 0,35, вы выделите под левую часть панели 35% доступного пространства, правой части достанется 65% этого пространства, соответственно расположится и полоса |

**Таблица 13.3** *(продолжение)*

| Свойства | Описание |
|---|---|
| resizeWeight | Позволяет указать, как должно распределяться избыточное пространство, если разделяемая панель увеличивается в размерах. Это свойство имеет тип double и работает точно так же, как и вторая версия предыдущего свойства. Значение свойства должно лежать в пределах от нуля до единицы, оно задает, сколько (в процентах, если умножить значение на сто) от избыточного пространства достанется левой (верхней) части разделяемой панели. Остальная часть достанется правой (нижней) панели.<br><br>Интересный факт — если вы установите значение этого свойства в 0 или 1, это будет означать, что все дополнительное пространство будет доставаться исключительно одной из частей разделяемой панели (левой или правой, соответственно), а вторая не будет менять свой размер. Однако это не отменяет тот факт, что пользователь сможет менять размер компонентов панели. |

## События разделяемой панели

У разделяемой панели JSplitPane нет собственных событий. «Как же так?» — спросите вы: «А если понадобится узнать о перемещениях полосы разделяемой панели, чтобы предпринять при этом какие-либо особые действия? Как это сделать?». Тут надо вспомнить, что все свойства компонентов Swing являются *привязанными* (мы обсуждали основы Swing и JavaBeans в главе 1). Это означает, что каждый компонент библиотеки поддерживает слушателей PropertyChangeListener, которые оповещаются при смене *любого* свойства компонента, имеющего значение для его создателя-программиста. Создатели класса JSplitPane решили, что не стоит плодить новые классы и интерфейсы только для того, чтобы следить за свойством dividerLocation. Ведь сделать это можно и с помощью слушателя PropertyChangeListener. Принцип действия прост: вы регистрируете в компоненте своего слушателя, так что при смене какого-либо свойства вызывается метод этого слушателя propertyChange(). В качестве параметра этого метода выступает объект PropertyChangeEvent. Вызвав его метод getPropertyName(), вы узнаете имя свойства, а метод getNewValue() поможет получить новое значение свойства (но вы должны знать его тип, для свойства dividerLocation это будет объект Integer). Следя за событием с именем dividerLocation, вы сможете вовремя узнавать, куда и когда пользователь перетаскивает вешку разбивки.

Хотя мы и отмечали, что события PropertyChangeEvent в основном предназначены для визуальных инструментов построения GUI и обмена информацией между компонентами и UI-представителями, они могут быть полезны и в повседневном программировании. Если вам вдруг понадобится следить за изменениями какого-либо свойства, не стоит сетовать на судьбу, если отдельного события для этого свойства создатели компонента не предусмотрели. К вашим услугам всегда механизм привязанных свойств JavaBeans, которому четко следует библиотека Swing.

## Разделяемая панель в насыщенных интерфейсах

Одним из признаков пользовательского интерфейса, спроектированного не самым лучшим образом, является переизбыток так называемого «визуального шума», разнообразных рамок, линий, жирного текста, подчеркиваний и тому подобных эффектов. Наш разум достаточно хорош, чтобы структурировать интерфейс и просто по тому, что его компоненты разделены некоторым пустым пространством. Эта

проблема часто возникает с разделяемыми панелями — внутри них, как правило, находятся панели прокрутки со своими собственными рамками или панели с какими-нибудь украшениями, и ко всему этому великолепию разделяемая панель добавляет свою вешку разбивки. Интерфейс чересчур сильно насыщается рамками и границами.

Чтобы избежать негативного эффекта, достаточно считать, что вешка разбивки разделяемой панели представляет собой и так достаточно сильную визуальную рамку, и ни в коем случае не применять внутри разделяемой панели других рамок и эффектов отделения содержимого. На крайний случай, если вам совершенно необходима рамка или что-то подобное в компоненте, который содержится в разделяемой панели, вы можете отделить его от вешки разбивки с помощью пустой рамки EmptyBorder определенного размера, соблюдая при этом рекомендации дизайнеров вашего внешнего вида. Ну а панель прокрутки можно избавить от рамки по умолчанию, заменив ее на все ту же пустую рамку нулевого размера, и вы будете избавлены от лишних линий и «шума».

## *Панель прокрутки JScrollPane*

Панель прокрутки JScrollPane — настоящая «волшебная палочка» пользовательских интерфейсов Swing. Она уже не раз позволяла нам добиваться многого всего строчкой кода и, как мы сейчас увидим, способна на большее. Несложно догадаться по названию, что панель JScrollPane дает возможность наделить компонент (обычно большого размера) возможностью прокрутки (на экране появляется одна или две полосы прокрутки, используя которые, вы можете переходить к той или иной части компонента, выводя ее на экран). Как правило, для добавления панели прокрутки достаточно следующей записи:

```
add(new JScrollPane(ваш_компонент));
```

Этой строкой вы добавляете в контейнер некоторый компонент, размеры которого не позволяют ему разместиться на экране полностью[2]. Передав компонент в простейший конструктор класса JScrollPane, вы получаете прекрасную возможность ограничить размеры компонента (применив для этого подходящие менеджеры расположения или вспомогательные компоненты), не беспокоясь о том, что пользователь не сможет получить доступ к какой-либо его части.

Прежде чем начать обсуждение более сложных деталей, рассмотрим небольшой пример. В этом примере мы настроим самые простые свойства панели прокрутки JScrollPane, которые, тем не менее, могут быть весьма полезны для тонкой настройки пользовательского интерфейса:

```
// SimpleScrollPanes.java
// Настройка некоторых простых свойств панелей прокрутки
import javax.swing.*;
import java.awt.*;

public class SimpleScrollPanes extends JFrame {
  public SimpleScrollPanes() {
    super("SimpleScrollPanes");
```

[2] Впрочем, это не обязательно — компонент может и полностью умещаться на экране. В таком случае панель прокрутки обеспечит гибкость вашему пользовательскому интерфейсу на тот случай, если размеры компонента изменятся (в нем появится больше данных для отображения), и к тому же изменит внешность компонента (панель прокрутки обладает специальной рамкой, зависящей от ее UI-представителя).

```
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // надпись
    JLabel label = new JLabel(new ImageIcon("image.jpg"));
    // особый конструктор панели прокрутки
    JScrollPane scrollPane = new JScrollPane(label,
        JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
        JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
    // некоторые свойства
    scrollPane.setViewportBorder(
        BorderFactory.createLineBorder(Color.BLUE));
    scrollPane.setWheelScrollingEnabled(false);
    // выводим окно на экран
    add(scrollPane);
    setSize(400, 300);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new SimpleScrollPanes(); } });
  }
}
```

Пример замечательно прост: мы создаем надпись JLabel со значком ImageIcon, который был получен из файла с изображением в формате JPEG. Предположительно надпись большого размера. Мы размещаем ее в панели прокрутки, используя довольно сложный конструктор с тремя параметрами. Первый параметр, как нетрудно видеть, — это сам компонент, нуждающийся в прокрутке, то есть наша надпись с изображением. Вторые два параметра, прежде не применяемые нами при создании панелей прокрутки, управляют полосами прокрутки (JScrollBar), которые собственно и позволяют пользователю выполнять прокрутку. Первый из этих двух параметров определяет поведение вертикальной полосы прокрутки, второй — горизонтальной. По умолчанию используются описанные в классе JScrollPane в качестве констант значения XXX_SCROLLBAR_AS_NEEDED (где XXX может принимать значение VERTICAL или HORIZONTAL). Они указывают, что полосы прокрутки должны появляться на экране только в том случае, если компонент в панели прокрутки действительно велик и ему требуется прокрутка. Мы использовали значение SCROLLBAR_ALWAYS, которое означает, что полосы прокрутки будут находиться на экране в любом случае, даже если прокрутка не требуется. Иногда это позволяет ясно показать пользователю, что в случае необходимости компонент поддерживает прокрутку (к примеру, такое поведение вполне подходит текстовым компонентам: текста в них может быть немного, но полосы прокрутки должны быть всегда, как показатель того, что объем текста ничем не ограничен). Есть и еще один режим вывода полос прокрутки — XXX_SCROLLBAR_NEVER. При выборе этого значения полосы прокрутки не появляются на экране вообще, даже если в этом возникает необходимость. Обычно такое поведение требуется там, где прокрутка реализована программно. Помимо задания режима появления полос прокрутки в конструкторе вы можете изменять его с помощью свойств verticalScrollBarPolicy и horizontalScrollBarPolicy.

Далее мы настраиваем еще несколько свойств панели прокрутки. Как известно, панель прокрутки всегда добавляет особую рамку к видимой части компонента (видимая часть компонента выводится на экран окном просмотра JViewport, незаменимым помощником класса JScrollPane — вскоре мы о нем узнаем), которая определяется UI-представителем. Иногда именно эта рамка позволяет придать компоненту, добавленному в панель прокрутки, законченный вид (в главе 11 мы отмечали, что списки JList из-за отсутствия своей собственной рамки выглядят без панели прокрутки не слишком презентабельно, то же относится к большинству текстовых компонентов). Менять ее стоит лишь тогда, когда у вас и так хватает визуального подчеркивания компонента, как мы говорили в этой главе про разделяемые панели. Как правило, лишнюю рамку можно просто убрать методом setBorder(), передав в него экземпляр пустой рамки EmptyBorder нулевого размера.

Однако рамка самой панели прокрутки не одинока. Есть еще и рамка вокруг содержимого панели прокрутки. Ее можно сменить с помощью свойства viewportBorder, в примере мы меняем ее рамкой LineBorder синего цвета, так что вы сможете оценить то, где она находится и как выглядит. Не сказать, чтобы это было очень полезное свойство, но знать все возможности своих компонентов определенно стоит.

Наконец, свойство wheelScrollingEnabled определяет, будет ли поддерживаться прокрутка колесиком мыши, по умолчанию оно включено. Если такое поведение по какой-либо причине вас не устраивает, вы можете легко отключить его и использовать движение колесика так, как вам хочется. В нашем примере мы как раз отключили поддержку колесика мыши.


Запустив программу с примером, вы сможете увидеть (вернее, «пощупать») те небольшие, но часто требующиеся изменения, которые мы произвели с панелью прокрутки JScrollPane. Обратите внимание, что когда компоненту в панели прокрутки хватает места, он все равно занимает все доступное место в панели (растягивается), и то, как он при этом будет выглядеть, зависит уже исключительно от него. Надпись JLabel, к примеру, при избытке пространства по умолчанию помещает значок по центру, а текст бы разместила слева. При разработке интерфейса следует учесть все подобные «краевые условия», и удостовериться в том, что он всегда ведет себе подобающе. Для этого панель прокрутки предоставляет специальные инструменты, о которых мы сейчас узнаем.

## Управление прокруткой

По умолчанию процесс прокрутки компонента, добавленного в панель прокрутки JScrollPane, проходит довольно очевидно: при нажатии на одну из кнопок полос прокрутки компонент сдвигается в ту или иную сторону на один пиксел. При щелчке на полосе прокрутки видимая часть изображения меняется «блоком», при этом прежняя видимая часть исчезает, а вместо нее появляется следующая или предыдущая (в зависимости от того, по какую сторону полосы прокрутки был сделан щелчок) часть такого же размера, если это возможно. С другой стороны, можно вспомнить разнообразные сложные компоненты Swing, такие как списки или таблицы — уж они-то точно прокручиваются не по одному пикселу, даже если пользователь щелкает на кнопке полосы прокрутки. Вместо этого содержимое этих компонентов перемещается на одну логическую единицу: на один элемент списка или столбец таблицы, и это именно то поведение, что ожидает от них пользователь. Как же реализовать особое поведение при прокрутке?

Оказывается, ничего сложного в этом нет. Все компоненты Swing, поддерживающие нестандартную прокрутку, просто пользуются предоставленным им панелью прокрутки JScrollPane механизмом управления. Это интерфейс Scrollable, который необходимо реализовать компоненту, требующему особый режим прокрутки. Когда такой компонент попадает в панель прокрутки JScrollPane, последняя видит это и получает (вызывая определенные в интерфейсе методы Scrollable) дополнительную информацию о компоненте, которая и позволяет ей проводить прокрутку компонента так, как тому необходимо. Интерфейс Scrollable несложен и назначение методов его очевидно. Рассмотрим пример, в котором реализуем особый режим прокрутки своего компонента:

```
// ControllingScrolling.java
// Управление процессом прокрутки с помощью
// интерфейса Scrollable
import javax.swing.*;
import java.awt.*;

public class ControllingScrolling extends JFrame {
  public ControllingScrolling() {
    super("ControllingScrolling");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // добавим в центр панель прокрутки с
    // нашим компонентом
    add(new JScrollPane(new GridComponent()));
    // выводим окно на экран
    setSize(400, 300);
    setVisible(true);
  }
  // компонент-"сетка" с особым режимом прокрутки
  class GridComponent extends JPanel
      implements Scrollable {
    // размер ячейки сетки
    private int CELL_SIZE = 45;
```

```
// количество ячеек сетки
private int CELL_COUNT = 50;
// предпочтительный размер компонента
public Dimension getPreferredSize() {
  return new Dimension(CELL_SIZE*CELL_COUNT,
      CELL_SIZE*CELL_COUNT);
}
// прорисовка компонента
public void paintComponent(Graphics g) {
  // нужно вызвать метод базового класса
  super.paintComponent(g);
  for (int x=0; x<CELL_COUNT; x++) {
    for (int y=0; y<CELL_COUNT; y++) {
      // рисуем ячейку
      g.setColor(Color.BLACK);
      g.drawRect(x*CELL_SIZE, y*CELL_SIZE,
          CELL_SIZE, CELL_SIZE);
      // текст с координатами
      g.drawString(""+x+","+y,
          x*CELL_SIZE + 5, y*CELL_SIZE + CELL_SIZE/2);
    }
  }
}
// предпочтительный размер области прокрутки
public Dimension
getPreferredScrollableViewportSize() {
  return getPreferredSize();
}
// приращение при прокрутке на один элемент
public int getScrollableUnitIncrement(
    Rectangle visible, int or, int dir) {
  return CELL_SIZE;
}
// приращение при прокрутке "блоком"
public int getScrollableBlockIncrement(
    Rectangle visible, int or, int dir) {
  return CELL_SIZE*10;
}
// нужно ли следить за размером области прокрутки
public boolean getScrollableTracksViewportWidth() {
  return false;
```

```
    }
    public boolean getScrollableTracksViewportHeight() {
      return false;
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
             public void run() { new ControllingScrolling(); } });
  }
}
```

Пример совсем прост: мы создаем окно с рамкой небольших размеров и размещаем в нем панель прокрутки, в которой, в свою очередь, размещается специальный компонент GridComponent, реализованный в виде внутреннего класса. Сам по себе новый компонент также прост: он унаследован от панели JPanel и переопределяет процедуру прорисовки paintComponent(), в которой рисуется сетка черного цвета. Размеры и количество ячеек сетки (одинаковые по вертикали и горизонтали) задаются специальными переменными. В каждой ячейке также выводится текст с ее координатами, чтобы вам было проще отслеживать процесс прокрутки. Обратите внимание, что непосредственно перед прорисовкой ячеек (которая выполняется в двойном цикле по осям X и Y) необходимо вызвать метод paintComponent() базового класса, в противном случае компонент будет рисоваться неверно[3]. В панели JPanel, от которой унаследован наш компонент, свойство непрозрачности включено по умолчанию, поэтому перед прорисовкой нужно закрашивать панель цветом фона или очищать ее от «мусора» каким-либо другим способом. Это и делает метод базового класса. Также нам необходимо указать размер нового компонента (по умолчанию размеры компонентов нулевые). Для этого мы переопределяем метод getPreferredSize() и прямо в нем вычисляем размер нашего компонента.

Но самое интересное в том, как компонент GridComponent реализует особый режим прокрутки с помощью интерфейса Scrollable. Как мы уже отмечали, когда панель прокрутки JScrollPane обнаруживает, что размещенный в ней компонент реализует данный интерфейс, она настраивает процесс прокрутки согласно информации, возвращаемой описанными в нем методами. Давайте кратко опишем методы Scrollable (табл. 13.4), а заодно поясним, как эти методы действуют на компонент GridComponent.

**Таблица 13.4.** Методы интерфейса Scrollable

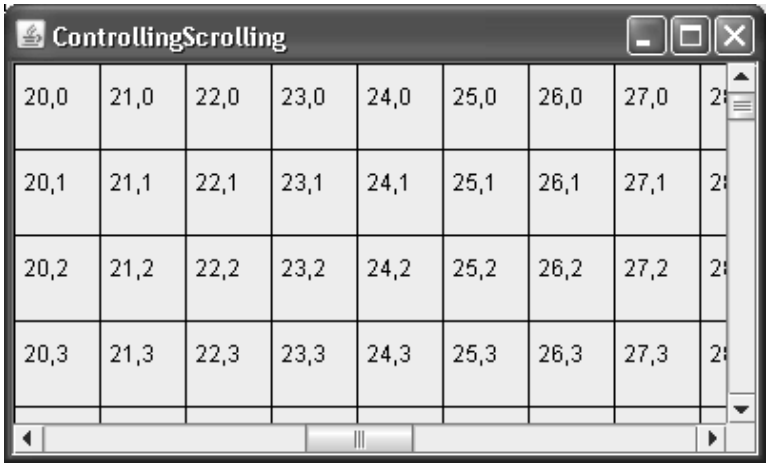
| Метод | Предназначение |
|---|---|
| getPreferredScrollableViewportSize() | Возвращает предпочтительный размер для области прокрутки данного компонента. Как правило, этот размер совпадает с предпочтительным размером самого компонента, как в нашей программе, но бывают и исключения. За примером долго ходить не надо — можно вспомнить раскрывающийся список JComboBox, который выводит перечень своих элементов именно в панели прокрутки. |

[3] Детали процесса рисования компонентов Swing всегда можно найти в Главе 4.

**Таблица 13.4** *(продолжение)*

| Метод | Предназначение |
|---|---|
| | С помощью свойства maximumRowCount вы можете ограничить количество выводимых элементов, даже если есть возможность вывести на экран их все. Роль ограничителя играет размер, возвращаемый данным методом: больше этого размера область прокрутки не станет (конечно, если ее не заставит это сделать какой-нибудь менеджер расположения). Обычно данный параметр употребляется большими компонентами, которые даже при наличии достаточного пространства предпочитают занимать небольшую его часть, а остальное показывать за счет прокрутки |
| getScrollableUnitIncrement() | Данный метод возвращает приращение (в пикселах), на которое должно сместиться изображение в панели прокрутки, когда пользователь нажимает кнопку полосы прокрутки (и тем самым «заказывает» переход к следующему логическому элементу компонента). Как и следовало ожидать, в нашем примере мы возвращаем размер ячейки сетки — на самом деле, сетку логичнее всего прокручивать по одной ячейке. В качестве параметров данного метода выступают видимая в данный момент часть изображения, ориентация полосы прокрутки JScrollBar (вертикальная или горизонтальная) и тип прокрутки (назад или вперед, соответственно, положительное или отрицательное число). Проанализировав данные параметры, вы сможете реализовать очень «тонкую» прокрутку, зависящую от позиции изображения и направления прокрутки |
| getScrollableBlockIncrement() | Возвращает приращение (в пикселах) при прокрутке «блоком». Подобное приращение происходит, когда пользователь щелкает на полосе прокрутки, а не нажимает ее кнопки. В нашем примере блок состоит из 10 ячеек и не зависит от параметров. Параметры данного метода аналогичны параметрам метода предыдущего, на их основании вы также сможете возвращать различные приращения в зависимости от позиции изображения и направления прокрутки |
| getScrollableTracksViewportWidth(), getScrollableTracks ViewportHeight() | Пара этих методов возвращает булевы значения, которые говорят, зависит ли содержимое компонента, находящегося в области прокрутки, от размеров последней (первый метод говорит о ширине области прокрутки, второй — о высоте). Означает это буквально следующее: вместо того чтобы положиться на механизм прокрутки и не менять свой размер при изменении размеров контейнера компонент следит за изменениями размеров и соответственным образом меняет свое содержимое, чтобы оно в доступные размеры помещалось. Как правило, методы эти возвращают true только по отдельности, так как фактически это означает отказ от прокрутки — компонент вместо этого заявляет, что постарается вместить содержимое в то пространство, что ему доступно. Чаще всего эти методы применяются там, где компонент отказывается от прокрутки по одному из направлений, вертикальному или горизонтальному. Прекрасный пример — текстовое поле с переносом по словам. Несмотря на то, что вертикальная прокрутка такому полю нужна, горизонтальная ему уже не понадобится, так как слова переносятся автоматически и длина строки может быть любой |

Запустив программу с примером, вы оцените особый режим прокрутки созданного нами компонента-«сетки», и теперь всегда сможете без особых трудностей реализовать специальный режим для собственных компонентов.




Как правило, требуется особое поведение прокрутки не так уж и часто: в Swing богатейший набор стандартных компонентов, а те из них, что нуждаются в особых режимах прокрутки, уже реализуют интерфейс Scrollable и прокручиваются так, как нужно.

## Компонент JViewport

Если вы думаете, что процесс прокрутки полностью управляется панелью прокрутки JScrollPane, то ошибаетесь. На самом деле компонент JScrollPane — не более чем удобный «фасад»[4] для целой системы, занимающейся прокруткой вашего компонента. Эта система состоит из специальным образом расположенных[5] вспомогательных компонентов для прокрутки, вертикальной и горизонтальной полос прокрутки, заголовков и «уголков» панели. Отображение лишь части от целого компонента и переход к другим его частям лежит на плечах компонента JViewport, основного рабочего «винтика» системы прокрутки, предоставленной нам Swing.

Компонент JViewport удобно представлять себе в виде своеобразного «видоискателя», работающего примерно так же, как в фотоаппарате: имеется общая картина, которую вы хотите запечатлеть (компонент большого размера), но доступного пространства меньше. JViewport позволяет показывать часть компонента и при необходимости перемещаться к любой другой его части. К основным возможностям класса JViewport относится задание компонента для отображения, называемого *видом*, задание размера области для отображения, называемой *видимым диапазоном* (extent), а также задание точки левого верхнего угла в области координат вида, с которой

[4] Панель прокрутки JScrollPane — прекрасный пример применения шаблона проектирования *фасад* (facade). Фасад предоставляет простой интерфейс к сложной системе, предположительно составленной из множества различным образом взаимодействующих компонентов.

[5] Специальным расположением компонентов, составляющих панель прокрутки, занимается специализированный менеджер расположения ScrollPaneLayout. Вы не сможете установить в панели прокрутки другой менеджер расположения (а если попытаетесь, то непременно наткнетесь на сообщение об ошибке).

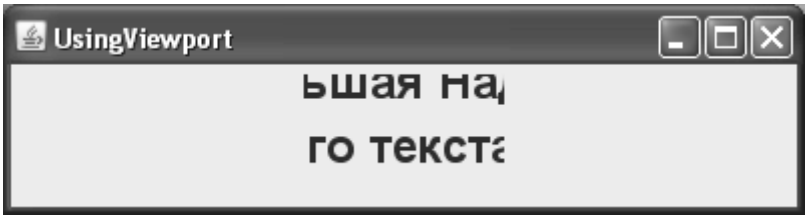
и отсчитывается видимая область. Меняя последнюю точку, вы отображаете различные части компонента.

«Видоискатель» JViewport крайне редко применяется изолированно, как правило, вы и не задумываетесь о том, что именно он исправно трудится за кулисами класса JScrollPane, отображая различные части вашего компонента. Иногда, впрочем, особенно если вы реализуете нестандартный способ прокрутки или вам нужно отображать часть компонента, не затрагивая остальных частей панели прокрутки, компонент JViewport может быть полезен и сам по себе. В следующем примере мы познакомимся с ним поближе, отобразив с его помощью части большой надписи:

```
// UsingViewport.java
// Работа с "видоискателем" JViewport
import javax.swing.*;
import java.awt.*;

public class UsingViewport extends JFrame {
  public UsingViewport() {
    super("UsingViewport");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // надпись с текстом большого размера
    JLabel bigLabel = new JLabel(
        "<html><h1>Большая Надпись!<br>Много текста!");
    // "видоискатель"
    JViewport viewport = new JViewport();
    // устанавливаем вид и видимый диапазон
    viewport.setView(bigLabel);
    viewport.setExtentSize(new Dimension(100, 60));
    // точка начала видимой области
    viewport.setViewPosition(new Point(50, 50));
    // ограничим размер "видоискателя"
    viewport.setPreferredSize(new Dimension(100, 60));
    // выводим окно на экран
    setLayout(new FlowLayout());
    add(viewport);
    setSize(400, 300);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new UsingViewport(); } });
  }
}
```

В примере у нас имеется надпись большого размера (в том, что она большого размера, можно не сомневаться: мы создали ее с помощью встроенного языка HTML, написав пару строк текста большим заголовком — H1). С помощью «видоискателя» JViewport легко отобразить на экране только часть нашей большой надписи, причем любую часть любого размера.

В классе JViewport определен только один конструктор — конструктор по умолчанию (без параметров). После создания компонента JViewport необходимо провести его настройку. Прежде всего задается собственно компонент (называемый видом), который будет отображать «видоискатель», за это отвечает свойство view. Затем необходимо указать, область какого размера будет показывать компонент JViewport (если вы хотите увидеть его в действии, размер видимой области должен быть меньше, чем размер вида). Установить размер видимой области позволяет свойство extentSize. Ну и наконец, остается задать точку левого верхнего угла (в координатах вида), с которой будет начинаться видимая область (свойство viewPosition). Обратите внимание еще на пару нюансов примера: в качестве менеджера расположения панели содержимого мы устанавливаем менеджер FlowLayout и вручную указываем предпочтительный размер компонента JViewport — он будет совпадать с размером видимой области. Эти два действия гарантируют, что сам компонент JViewport не станет больше видимой области вида. В противном случае, если ему хватит места (а в нашем окне ему места хватит), он отобразит весь компонент вне зависимости от настройки. В проведенных дополнительных мерах нет ничего удивительного: JViewport обычно используется при нехватке места в контейнере, поэтому нам пришлось организовать эту нехватку искусственно.




Запустив программу с примером, вы увидите, как созданный и настроенный нами «видоискатель» JViewport отображает лишь часть большой надписи. Именно так он и используется панелью прокрутки JScrollPane: отображает большой компонент, а когда пользователь щелкает на полосах прокрутки, панель прокрутки JScrollPane говорит «видоискателю», что надо сменить координаты левого верхнего угла изображения (viewPosition) и прорисовать новую его часть[6]. Подобные действия вы сможете проводить и сами, если возникнет необходимость в написании собственной процедуры прокрутки. Например, вам может понадобиться сделать ее подобием прокрутки в известных продуктах компании Apple. Вам придется продумать систему ускорения и замедления прокрутки для создания нужного визуального эффекта, а вот заменять «видоискатель» JViewport при этом вряд ли понадобится: он прекрасно справляется со своей задачей.

JViewport можно задействовать и при создании интерактивной справочной системы в качестве «видеокамеры», плавно перемещающейся по пользовательским интерфейсам вашего приложения. Вариантом может быть размещение всей панели содержимого вашего главного окна в компоненте JViewport, отключение от него всех событий от пользователей с помощью класса JXLayer, который был нами изучен в главе 6, и перемещение «видоискателя» по интерфейсу в зависимости от раздела справки.

[6] Получить используемый в JScrollPane «видоискатель» можно с помощью метода getViewport().

### Особенности реализации класса JViewport

Прокрутка часто становится «узким местом» графических программ, особенно если прокручиваются большие сложные компоненты, прорисовка которых целиком стоит дорого и отнимает немало ресурсов. В компоненте JViewport для смены изображения используется *блиттинг (blitting)* — побитовое копирование изображения. Вместо перерисовки всего изображения та часть его, которая после смены изображения все равно остается на экране (а такая часть почти всегда есть, особенно при постепенной прокрутке) копируется со старого изображения (посредством блиттинга), а новые части изображения прорисовываются заново. Так удается намного ускорить производительность прокрутки, и требования к памяти практически не повышаются: мы помним, что прорисовка компонентов Swing происходит с двойной буферизацией, и класс RepaintManager хранит общее внеэкранное изображение всего интерфейса. Из этого изображения JViewport и копирует старые, но все еще необходимые, части.

Компонент, часть которого отображает класс JViewport (вид), является его единственным потомком. Чтобы избежать вмешательства базовой системы прорисовки в свои действия, класс JViewport переопределяет два метода. Метод isOptimizingDrawingEnabled() возвращает false, а это означает, что механизмы прорисовки не будут бездумно вызывать прорисовку всех потомков компонента JViewport, и возложат их прорисовку на метод paint(). Переопределяется и метод paint(): вместо перерисовки всего «видоискателя» он копирует все еще необходимые части (применяя блиттинг), а остальное дополняет, рисуя новые части заново. Правда, из-за переопределения метода paint() класс JViewport не может обладать рамкой — метод paintBorder() больше не вызывается основным рисующим методом paint(). Так что если вы попытаетесь сменить рамку «видоискателя» методом setBorder(), то получите исключение. Но мы уже знаем способ, как обойти эту проблему: в классе JScrollPane имеется свойство viewportBorder, именно оно позволит вам задать рамку для «видоискателя» вашей панели прокрутки. Если же вы используете класс JViewport сам по себе, то рамку можно задать для вспомогательной панели, в которой затем разместится «видоискатель».

Для программиста-клиента особая настройка системы рисования компонента JViewport остается за кадром, главное — подходящая производительность. Правда, у вас есть возможность повлиять на выбор метода прокрутки: как мы уже видели, по умолчанию используется блиттинг, но есть и еще два средства: специальное внеэкранное изображение для всего вида (это средство может работать быстрее, но в некоторых случаях, если компонент велик, ведет к неоправданным затратам памяти) или отказ от оптимизации и полная перерисовка всего компонента при прокрутке (вряд ли стоит полагаться на этот режим, иначе вы рискуете значительно снизить производительность своего приложения). Выбрать алгоритм прокрутки позволяет метод setScrollMode() класса JViewport.

## Заголовки и уголки панели прокрутки JScrollPane

Панель прокрутки JScrollPane позволяет не только организовать прокрутку компонента, но при необходимости дополнить ее чрезвычайно полезными возможностями. *Заголовки* (headers) панели прокрутки дополняют прокручиваемый компонент сверху и слева, предоставляя пользователю вспомогательную информацию о том содержимом, что он прокручивает. Как правило, заголовки используются в качестве навигационных или информационных дополнений: линеек или координатных осей, которые помогают быстро определить размеры прокручиваемой области, текущую позицию или получить иную вспомогательную информацию. Интересно, что заголовки находятся в отдельных «видоискателях» JViewport, так же как и основное содержимое панели прокрутки. Это позволяет им иметь большой размер, как правило,

совпадающий с размером области прокрутки (что весьма логично: заголовки должны предоставлять вспомогательную информацию обо всей прокручиваемой области). Отличие заголовков в том, что если позиция прокручиваемого компонента меняется пользователем с помощью полос прокрутки, то позиция заголовка меняется панелью прокрутки JScrollPane автоматически, синхронно с позицией основного содержимого. Такое поведение как нельзя лучше подходит вспомогательному информационному компоненту.

*Уголки* (corners) панели прокрутки реже применяются в приложениях. Это обычные компоненты, которые занимают пустые места по углам панели прокрутки. Интересно, что этих пустых мест может и не быть — они появляются только при наличии полос прокрутки и заголовков. В уголки обычно добавляются компоненты, которые выполняют некоторое масштабирование прокручиваемого изображения, переход на позицию по некоторому признаку или вызов контекстного меню с подходящим списком команд. Заголовки и уголки располагаются так, как показано на рис. 13.1.

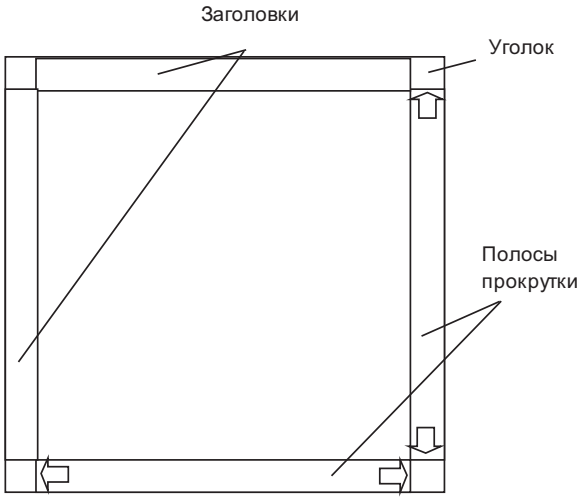



**Рис. 13.1.** Заголовки и уголки панели прокрутки

Рисунок наглядно демонстрирует, как располагаются заголовки (как располагаются полосы прокрутки, вы могли видеть и в предыдущих примерах), и где находятся уголки (их может быть четыре). Правый нижний уголок появляется, если видны обе полосы прокрутки, остальные уголки могут рассчитывать на свое появление только при наличии в панели прокрутки заголовков. Все четыре уголка видны на экране, когда видны обе полосы прокрутки и оба заголовка.

Рассмотрим теперь пример, в котором снабдим прокручиваемый компонент заголовками и парой уголков:

```
// HeadersAndCorners.java
// Заголовки и "уголки" панели прокрутки JScrollPane
import javax.swing.*;
import java.awt.*;
```

```
public class HeadersAndCorners extends JFrame {
  // надпись с большим изображением
  private JLabel label = new JLabel(
      new ImageIcon("image.jpg"));
  public HeadersAndCorners() {
    super("HeadersAndCorners");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем панель прокрутки
    JScrollPane scroll = new JScrollPane(label);
    // устанавливаем заголовки
    scroll.setColumnHeaderView(new XHeader());
    scroll.setRowHeaderView(new YHeader());
    // устанавливаем левый верхний "уголок"
    scroll.setCorner(JScrollPane.UPPER_LEFT_CORNER,
        new JButton(new ImageIcon("Print16.gif")));
    // выводим окно на экран
    add(scroll);
    setSize(400, 300);
    setVisible(true);
  }
  // заголовок по оси X
  class XHeader extends JPanel {
    // размер заголовка
    public Dimension getPreferredSize() {
      return new Dimension(
          label.getPreferredSize().width, 20);
    }
    // прорисовываем линейку
    public void paintComponent(Graphics g) {
      super.paintComponent(g);
      int width = getWidth();
      for (int i=0; i<width; i+=50) {
        g.drawString("" + i, i, 15);
      }
    }
  }
  // заголовок по оси Y
  class YHeader extends JPanel {
    // размер заголовка
    public Dimension getPreferredSize() {
      return new Dimension(
```

```
            20, label.getPreferredSize().height);
      }
      // прорисовываем линейку
      public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int height = getHeight();
        for (int i=0; i<height; i+=50) {
          g.drawString("" + i, 0, i);
        }
      }
    }
    public static void main(String[] args) {
      SwingUtilities.invokeLater(
          new Runnable() {
            public void run() { new HeadersAndCorners(); } });
    }
}
```

Мы создаем небольшое окно, в центре панели содержимого которого разместится панель прокрутки JScrollPane. «Прокручивать» мы будем надпись JLabel с изображением большого размера, которое загружается из файла. В дополнение мы добавляем к панели прокрутки пару заголовков и один уголок.

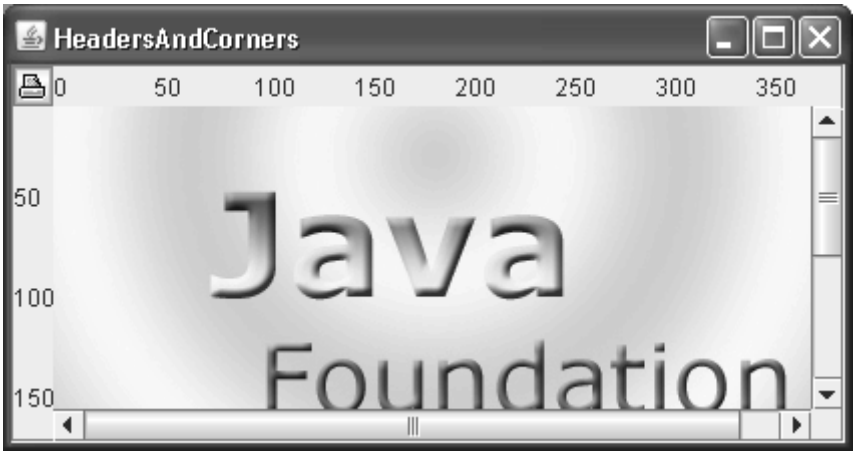
Задать заголовки позволяют свойства columnHeaderView и rowHeaderView[7] (верхний и левый заголовки, соответственно, или, что то же самое, заголовки для столбцов и строк). В качестве заголовков будут использованы специальные компоненты, написанные нами в виде внутренних классов. Они унаследованы от панелей JPanel и прорисовывают координатную сетку по оси X и Y с шагом в 50 пикселов. Обратите внимание на то, какой размер возвращается методами getPreferredSize(): по одной из осей он постоянен, а по второй зависит от компонента, находящегося в панели прокрутки. При создании заголовков чаще всего приходится поступать именно так: размеры заголовков должны совпадать с размерами области прокрутки, иначе они будут не в состоянии сообщить информацию обо всей прокручиваемой области.

Далее мы задаем левый верхний уголок панели прокрутки, добавляя в него кнопку с маленьким значком печати[8] (вполне удобно для пользователя, поскольку заголовки с координатами позволят ему оценить размеры изображения, а щелкнув на кнопке, он мгновенно получит печатную копию и, несомненно, оценит качество вашего приложения). Для задания уголков служит метод setCorner(), принимающий два параметра: идентификатор уголка (все четыре уголка описаны соответствующими константами класса JScrollPane) и собственно компонент, который будет уголком.

---

[7] Здесь мы видим все ту же схему именования, используемую в классе JScrollPane — основное название принадлежит «видоискателю» (например, viewport для основного «видоискателя» панели прокрутки и columnHeader для «видоискателя» верхнего заголовка), а компоненты, которые в этих «видоискателях» располагаются, управляются свойствами с суффиксом «View», например viewportView.

[8] Этот значок взят из коллекции Java Look And Feel Graphics Repository (java.sun.com).




Запустив программу с примером, вы увидите, как заголовки помогают моментально оценить размеры изображения, а кнопка печати недвусмысленно показывает возможность быстрой печати изображения. В качестве заголовков можно использовать компоненты с любой информацией об области прокрутки, хотя, без сомнения, линейки с разнообразными размерами встречаются чаще всего. Интересно, что на заголовки панели прокрутки JScrollPane полагается таблица JTable[9]: она помещает заголовки своих столбцов именно в верхнем заголовке панели прокрутки, так что если вы располагаете таблицу вне панели прокрутки, то ее заголовка вообще не увидите.

## Полосы прокрутки JScrollBar

Пара полос прокрутки (вертикальная и горизонтальная) играет важную роль в панели прокрутки JScrollPane, поскольку именно полосы прокрутки предоставляют пользователю возможность перемещаться по области прокрутки, различными способами: щелчками на кнопках полос, щелчками на полосе или просто перетаскиванием бегунка. Полосы прокрутки в Swing реализованы классом JScrollBar. Это самый настоящий компонент, который вы можете добавлять в контейнеры и использовать в своих программах, тем не менее отдельно от панели прокрутки он применяется редко.

Полоса прокрутки JScrollBar чрезвычайно похожа на исследованный нами в главе 12 ползунок JSlider: выполнять прокрутку можно, перетаскивая бегунок (в ползунке — регулятор) или щелкая мышью на полосе (шкале). Более того, оба компонента используют модель ограниченного диапазона данных BoundedRangeModel. Правда, есть и отличия: полоса прокрутки обладает кнопками, а ползунок — нет, кроме того, по-разному обрабатывается значение внутреннего диапазона, хранимое в модели. Ползунки им не пользуются, а полоса прокрутки на основании этого значения прорисовывает свою видимую часть. Собственно, эти несколько отличий и объясняют различное назначение двух схожих компонентов: ползунок используется как самостоятельный компонент, а полоса прокрутки — в составе панели прокрутки, как вспомогательный компонент.

Применять полосы прокрутки JScrollBar очень легко, отличий от ползунков практически нет. Вы настраиваете модель (как правило, используя стандартную реализацию DefaultBoundedRangeModel), передаете ее в конструктор и добавляете к модели слушателя изменений значений ChangeListener. Столь же легко задать или сменить ориентацию полосы: она может быть горизонтальной или вертикальной. Именно таким образом на-

[9] Мы подробно обсудим таблицы в главе 17.

страивает свои полосы прокрутки панель прокрутки JScrollPane. Числа для модели она либо выбирает по умолчанию (в зависимости от своих размеров и размеров прокручиваемого компонента), либо с помощью методов интерфейса Scrollable, если прокручиваемый компонент его реализует. В следующем примере мы проверим полосы прокрутки в действии:

```
// Scrollbars.java
// Полосы прокрутки JScrollBar
import javax.swing.*;
import java.awt.*;

public class Scrollbars extends JFrame {
  public Scrollbars() {
    super("Scrollbars");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем стандартную модель
    BoundedRangeModel model =
        new DefaultBoundedRangeModel(10, 40, 0, 100);
    // пара полос прокрутки
    JScrollBar scrollbar1 = new JScrollBar(
        JScrollBar.HORIZONTAL);
    JScrollBar scrollbar2 = new JScrollBar(
        JScrollBar.VERTICAL);
    // присоединяем модель
    scrollbar1.setModel(model);
    scrollbar2.setModel(model);
    // добавляем компоненты
    add(scrollbar1, "South");
    add(scrollbar2, "East");
    // выводим окно на экран
    setSize(400, 300);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new Scrollbars(); } });
  }
}
```

Здесь мы создаем пару полос прокрутки JScrollBar, одна из них будет горизонтальной, вторая — вертикальной. Данные для обеих полос будет поставлять стандартная модель DefaultBoundedRangeModel. Обратите внимание на ее значения: текущее значение равно 10, внутренний диапазон велик и равен 40 (как вы помните, внутренний диапазон отвечает за величину полосы прокрутки), а минимальное и максимальное значения равны

нулю и ста, соответственно. Благодаря разделению одной и той же модели полосы прокрутки будут работать синхронно.


Созданные полосы добавляются в окно, горизонтальная — на юг, вертикальная — на восток окна. Запустив программу с примером, вы увидите синхронную работу полос, связанных одной моделью, и оцените размер полосы, заданный большим внутренним диапазоном. Размер полосы, как можно видеть из изображения, относительный, и зависит от размера всей полосы прокрутки, то есть от внутреннего диапазона относительно всего диапазона модели.

Возможности настройки полос прокрутки этим не исчерпываются: вы можете также задать значения приращений для прокрутки на один элемент (при щелчке на кнопке полосы) и для прокрутки блоком (при щелчке на самой полосе). Впрочем, использовать полосы прокрутки при создании пользовательских интерфейсов приходится редко: с большинством задач подобного рода справляются ползунки. Как правило, полосы прокрутки требуются при создании собственных компонентов для специализированной прокрутки.

## *Резюме*

После прочтения этой главы можно забыть о проблемах с нехваткой места в контейнере и не думать о слишком больших компонентах. Вспомогательные компоненты, с которыми мы познакомились, позволят решить любую проблему с размерами и пространством, и сделают это замечательно элегантно, не требуя от вас титанических усилий.

# Глава 14. Стандартные диалоговые окна

Во всех современных графических системах имеется набор так называемых стандартных диалоговых окон. Это оформленные особым образом диалоговые окна, которые позволяют быстро выводить разнообразную информацию для пользователя (чаще всего всяческие сообщения, поступающие от приложений) или же получать от него некоторые наиболее распространенные типы данных. Использование стандартных диалоговых окон, во-первых, облегчает программирование вашего приложения (как правило, очень просто настроить стандартное диалоговое окно и вывести его на экран), во-вторых, намного ускоряет процесс знакомства пользователя с интерфейсом, поскольку встречая стандартные диалоговые окна не только в вашем приложении, но и во многих других, он хорошо усваивает правила работы с ними.

Библиотека Swing не является исключением из общего правила — в ней представлен богатый выбор стандартных диалоговых окон, намного упрощающих и ускоряющих вывод простой информации и выбор наиболее часто встречающихся в графических приложениях форматов данных. Для вывода разнообразной полезной информации (как правило, сообщений о работе программы, ошибках и нестандартных ситуациях) и ввода несложных данных предназначен класс JOptionPane, частый «гость» любого приложения, созданного средствами Swing. Работа с ним обычно заключается в вызове одного из многочисленных статических методов, создающих и выводящих на экран модальное диалоговое окно стандартного вида (который определяется текущим менеджером внешнего вида и поведения). В таких диалоговых окнах вы можете выводить самую разнообразную информацию и при необходимости размещать в них дополнительные компоненты, тонко настраивая все аспекты их поведения и внешнего вида. С другой стороны, класс JOptionPane представляет собой просто еще один компонент, унаследованный от базового класса JComponent библиотеки Swing, так что вы можете работать с ним напрямую, как мы это делали с остальными компонентами Swing: создавать экземпляры класса JOptionPane и настраивать их свойства. С настроенным компонентом JOptionPane дальше можно поступать по-разному: размещать его в каких-либо контейнерах своего приложения или с помощью вспомогательных методов быстро выводить его в диалоговом окне подходящего вида.

Подавляющее большинство графических приложений так или иначе работают с файлами, даже если манипуляция файлами не входит в сферу их основных обязанностей. Поэтому наличие в Swing мощного средства для удобного выбора файлов — компонента JFileChooser — приходится весьма кстати. Как и в случае с классом JOptionPane, компонент JFileChooser, хотя и представляет собой обычный компонент (а точнее контейнер, в котором расположены несколько компонентов, списков и кнопок, «ведающих» выбором файлов), чаще всего выводится на экран с помощью легко настраиваемого модального диалогового окна. Компонент JFileChooser вы можете добавить в любое место своего пользовательского интерфейса. Он весьма гибок и позволит вам до мелочей настраивать внешний вид. При необходимости можно полностью изменить стандартное расположение входящих в него компонентов и добавить дополнительные элементы, такие как панели предварительного просмотра файлов. Сказанное о компоненте JFileChooser в полной мере относится и к компоненту JColorChooser, который, с одной стороны, «одним мановением руки» позволяет организовать удобный и наглядный выбор цвета в модальном

диалоговом окне, а с другой может играть роль обычного компонента, то есть вы можете настроить его свойства и добавить компонент в нужное место своего интерфейса.

У всех стандартных диалоговых окон Swing есть собственные UI-представители, которые отвечают за то, чтобы диалоговые окна выглядели подобающим образом в используемом приложением внешнем виде. Это особенно важно для внешних видов, имитирующих известные платформы, пользователи которых не должны ощущать разницы (по крайней мере, значительной) при переходе от «родных» приложений к Java-приложениям. Этому прекрасно служат стандартные диалоговые окна, хорошо известные пользователям .

## *Многоликий класс JOptionPane*

С помощью одного только класса JOptionPane уже можно «состряпать» приличное приложение, настолько он разнообразен и столько всяческих операций по сбору и выводу данных позволяет проводить. Мы уже поняли, что JOptionPane представляет собой обычный компонент Swing, и работать с ним можно так же, как с другими компонентами, настроив и добавив в подходящее место своего контейнера. Выглядит экземпляр класса JOptionPane, даже настроенный самым экзотичным образом, всегда примерно так, как показано на рис. 14.1 (хотя значок может отсутствовать).


**Рис. 14.1.** Стандартное диалоговое окно

Впрочем, гораздо удобнее использовать статические методы класса JOptionPane, быстро настраивающие и выводящие на экран модальные диалоговые окна стандартного вида[1]. У класса JOptionPane целый букет таких методов, причем пользоваться ими, несмотря на все разнообразие, довольно просто, так как все они разительно похожи друг на друга и запомнить способы их вызова не сложно. Рассмотрим по порядку все возможности класса JOptionPane.

## Вывод сообщений

Перегруженные статические методы с общим именем showMessageDialog() позволяют быстро сообщить пользователю некоторую информацию, как правило, довольно важную (модальное диалоговое окно подразумевает, что информация заслуживает хоть какого-то внимания пользователя, потому что приостанавливает работу основного окна приложения). Рассмотрим пример использования таких методов, а потом обсудим их подробнее:

[1] По поводу модальных окон идут жаркие споры, и чрезмерное их использование действительно раздражает, так как они полностью блокируют основное окно приложения, а значит, отрывают пользователя от важных дел. Однако для действительно важных сообщений или вопросов модальные окна по-прежнему хороши.

```
// MessageDialogs.java
// Методы JOptionPane для вывода сооообщений
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class MessageDialogs extends JFrame {
  // этот значок выведем в одном из сообщений
  private ImageIcon icon = new ImageIcon("question.gif");
  public MessageDialogs() {
    super("MessageDialogs");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // кнопки, после щелчков на которых
    // выводятся сообщения
    JButton message1 = new JButton("2 параметра");
    message1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(
            MessageDialogs.this,
            "<html><h2>Привет!<br>HTML есть и здесь!");
      }
    });
    JButton message2 = new JButton("4 параметра");
    message2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(
            MessageDialogs.this,new String[] {
            "Сообщение может быть",
            "записано массивом!" }, "Свой заголовок",
            JOptionPane.ERROR_MESSAGE);
      }
    });
    JButton message3 = new JButton("5 параметров");
    message3.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(
            MessageDialogs.this,
            "Настроено все что можно", "Свой заголовок",
            JOptionPane.INFORMATION_MESSAGE, icon);
      }
    });
```

```
    // выведем окно на экран
    setLayout(new FlowLayout());
    add(message1);
    add(message2);
    add(message3);
    setSize(400, 200);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new MessageDialogs(); } });
  }
}
```

В примере мы создаем небольшое окно, в котором разместятся три кнопки JButton. После щелчка на каждой из них вашему взору будет представлен соответствующий вариант стандартного диалогового окна с некоторым сообщением. Первая кнопка демонстрирует минималистский способ вывода сообщений: все, что здесь требуется методу showMessageDialog() — это сослаться на «родительский» компонент (которым может быть любое окно, унаследованное от класса Window[2]) и показать собственно сам объект-сообщение. В качестве сообщения для первого диалогового окна мы использовали строку в формате HTML (начав ее с «волшебных» символов <html>). Запустив программу с примером, вы сможете убедиться в том, что стандартные диалоговые окна прекрасно знакомы с HTML. Это неудивительно: если вы передаете стандартному диалоговому окну строку, для вывода ее на экран оно использует нашу «старую знакомую», надпись JLabel, а уж о ее знании HTML нам прекрасно известно.

Второй вызов метода showMessageDialog() демонстрирует, как можно точнее настроить стандартное диалоговое окно. Первые два параметра не меняются — это все те же родительский компонент и сообщение, а вот следующие два параметра позволяют задать заголовок стандартного диалогового окна и его тип. Поддерживаемые классом JOptionPane типы стандартных диалоговых окон описаны в нем несколькими константами (табл. 14.1).

**Таблица 14.1.** Типы стандартных диалоговых окон

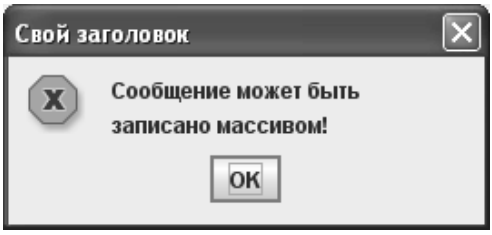
| Тип диалогового окна | Описание |
|---|---|
| INFORMATION_ MESSAGE | Это стандартное диалоговое окно выводит информацию общего назначения. Диалоговое окно именно такого типа создает метод showMessageDialog() с двумя параметрами. Все стандартные диалоговые окна, относящиеся к данному типу, выводятся на экран со значком соответствующего вида |
| WARNING_MESSAGE | Стандартное диалоговое окно такого типа выводит на экран предупреждающую информацию. Также снабжается соответствующим значком |

[2] В качестве родительского компонента можно использовать любой унаследованный от базового класса Component графический компонент, по центру этого компонента диалоговое окно и будет выведено на экран. Просто чаще всего таким компонентом является ссылка на главное окно приложения, так что все сообщения выводятся по центру окна, где пользователю работать с ними удобнее всего. Вы можете передать в метод пустую (null) ссылку, в таком случае диалоговое окно окажется в центре рабочего стола пользователя.

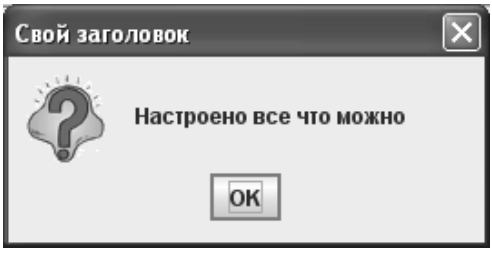
**Таблица 14.1** *(продолжение)*

| Тип диалогового окна | Описание |
|---|---|
| QUESTION_MESSAGE | Это стандартное диалоговое окно для запроса информации от пользователя (мы его вскоре рассмотрим). Как правило, не используется для информационных сообщений |
| ERROR_MESSAGE | Это диалоговое окно выводит на экран информацию об ошибке. Тоже имеет свой значок, чаще всего довольно «грозного» вида |
| PLAIN_MESSAGE | Указывает, что диалоговое окно не принадлежит ни к одному из вышеперечисленных типов. Выводится на экран без стандартного значка |

В нашем примере для второго диалогового окна мы использовали тип ERROR_MESSAGE, так что вы сможете увидеть, как выглядят на экране стандартные диалоговые окна с сообщениями об ошибках. Заголовок диалогового окна задается обычной строкой, а вот в качестве сообщения мы указали кое-что необычное — массив строк. Запустив программу с примером, вы увидите, что класс JOptionPane с массивом прекрасно справляется, выводя его содержимое на экран в несколько рядов. Благодаря поддержке массивов вы сможете методом showMessageDialog()выводить самые пространные сообщения, используя при необходимости в некоторых из них язык HTML.




Наконец, третий вариант метода showMessageDialog() показывает, как можно настроить все элементы стандартного диалогового окна для вывода сообщений. Первые четыре параметра неизменны — это родительский компонент, сообщение, заголовок и тип диалогового окна, а вот пятым идет значок Icon, который появится на месте стандартного значка. Если вы задаете такой значок (а мы в примере так и делаем), тип диалогового окна уже не так важен: ведь он отвечает только за значок, а мы его меняем[3]. После запуска программы с примером вы сможете увидеть, как окно, выведенное самым пространным вариантом метода showMessageDialog(), выглядит на экране.




[3] Правда, в случае специального оформления окон и наличия внешнего вида, который такое оформление поддерживает (это может быть используемый по умолчанию внешний вид Metal), тип диалогового окна может влиять на вид его рамок и его элементы управления. Мы рассматривали специальное оформление окон в главе 6.

Мы уже познакомились с тем, как в качестве сообщений используются строки и массивы строк, но оказывается, это еще не все способности класса JOptionPane. Сообщениями могут быть также любые графические компоненты, унаследованные от класса Component, которые без изменений вводятся в стандартные диалоговые окна. Давайте опишем все поддерживаемые типы сообщений, чтобы их удобно было держать «под рукой» (табл. 14.2).

**Таблица 14.2.** Поддерживаемые JOptionPane типы сообщений

| Тип сообщения | Способ обработки |
|---|---|
| Любой объект (то есть унаследованный от класса Object, но не от Component) | Вызывается метод объекта toString(), и полученный результат (строка) выводится на экран с помощью «рабочей лошадки» библиотеки — надписи JLabel (отсюда и поддержка HTML) |
| Графический объект (то есть унаследованный от Component) | Добавляется в область сообщений методом контейнера add() и без изменений выводится на экран. В области сообщений по умолчанию используется расположение GridBagLayout, компоненты оно размещает по рядам, один над другим. Несколько компонентов в одном ряду вы может разместить сами, передав в качестве сообщения заполненную ими панель JPanel |
| Значок Icon | Еще один особый случай. Этот значок также выводится с помощью надписи JLabel, причем располагается он по центру надписи. Если вы хотите другого поведения, создайте и настройте надпись со значком самостоятельно и передайте в качестве сообщения уже ее |
| Массив объектов Object | Обрабатывается рекурсивно: для каждого элемента массива, принадлежащего к одному из вышеперечисленных типов, выполняется соответствующая процедура. Массив может содержать другой массив (ведь массив является самым обыкновенным объектом), для него повторяется та же рекурсивная процедура |

Теперь, когда у вас в руках есть вся информация о доступных типах стандартных диалоговых окон и данных, которые они могут выводить (а могут они, как показано в таблице, довольно много), вы ничем не ограничены в выводе самой разнообразной информации, причем для этого достаточно всего нескольких строк кода. Это незаменимая возможность, действительно ускоряющая разработку любого приложения, да еще к тому же унифицирующая его внешний вид.

## Ввод данных

Замечательно, что мы уже умеем всего несколькими строками выводить разнообразные данные (методом showMessageDialog()), обратимся теперь к возможностям класса JOptionPane по вводу данных. Для этого служат несколько перегруженных методов showInputDialog(), их довольно много, но все они очень похожи друг на друга. Сначала проиллюстрируем работу этих методов несложным примером, а затем разберем их немного подробнее:

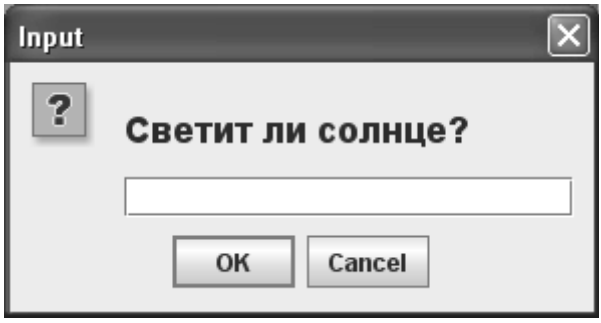
```
// InputDialogs.java
// Стандартные диалоговые окна JOptionPane для ввода данных
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class InputDialogs extends JFrame {
  // значок для одного из сообщений
```

```
private ImageIcon icon = new ImageIcon("question.gif");
// данные для выбора
private String[] values = {"Белый","Красный","Зеленый" };
public InputDialogs() {
  super("InputDialogs");
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  // после щелчков на кнопках выводятся сообщения
  JButton input1 = new JButton("2 и 3 параметра");
  input1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      // ввод строки в текстовом поле
      String res = JOptionPane.showInputDialog(
          InputDialogs.this,
          "<html><h2>Светит ли солнце?");
      res = JOptionPane.showInputDialog(
          InputDialogs.this,
          "Ваш ответ был таким?", res);
    }
  });
  JButton input2 = new JButton("4 параметра");
  input2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      // позволяет задавать тип и заголовок
      String res = JOptionPane.showInputDialog(
          InputDialogs.this, new String[] {
          "Пароль введен неверно!", "Повторите ввод:"},
          "Пароль", JOptionPane.WARNING_MESSAGE);
    }
  });
  JButton input3 = new JButton("7 параметров");
  input3.addActionListener(new ActionListener() {
    // выбор из нескольких альтернатив
    public void actionPerformed(ActionEvent e) {
      Object res = JOptionPane.showInputDialog(
          InputDialogs.this,"Выберите любимый цвет:",
          "Выбор цвета", JOptionPane.QUESTION_MESSAGE,
          icon, values, values[0]);
      JOptionPane.showMessageDialog(
          InputDialogs.this, res);
    }
  });
```

```
    // добавляем кнопки в окно
    setLayout(new FlowLayout());
    add(input1);
    add(input2);
    add(input3);
    setSize(400, 200);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new InputDialogs(); } });
  }
}
```

В окне JFrame, от которого в примере мы наследуем класс, размещаются три кнопки. Щелкая на них, вы сможете выводить на экран соответствующим образом настроенные стандартные диалоговые окна JOptionPane для ввода информации.

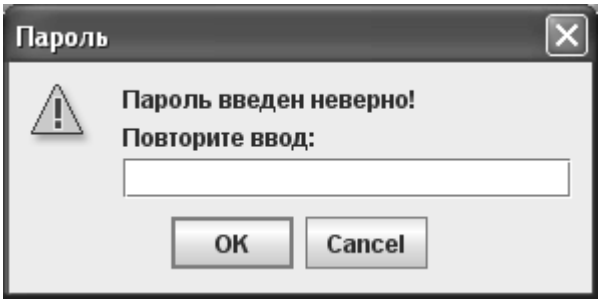
Первая кнопка создает самый простой и поэтому чаще всего используемый вариант диалогового окна для ввода несложной информации. В качестве параметров этому диалоговому окну нужно передать родительский компонент (любой графический компонент, по центру которого диалоговое окно будет размещаться на экране) и объект-надпись, сообщающий пользователю, что от него требуется. Поддерживаемые классом JOptionPane виды надписей мы только что обсудили (если вы уже успели забыть их, вернитесь к табл. 14.2). Для первого диалогового окна сообщением стала строка в формате HTML. Создаваемые первой кнопкой диалоговые окна используют для ввода данных простое текстовое поле, которое следует сразу за текстом сообщения. Запустив программу с примером, вы сможете увидеть, как выглядит подобное диалоговое окно. Кроме того, есть похожий, но чуть более удобный способ получения информации из текстового поля. Для него требуется дополнительный параметр — начальное значение, которое появляется в текстовом поле. Обратите внимание, как мы этим способом подтверждаем ввод, в качестве дополнительного параметра передавая только что полученную строку.




В том случае если пользователь подтвердил свой выбор щелчком на кнопке OK, первые два рассмотренные нами варианта метода showInputDialog() возвращают строку String, которую он набрал в текстовом поле. Если ничего набрано не было, возвращается

пустая строка. Если же пользователь отменил ввод щелчком на кнопке Cancel[4], методы showInputDialog() вернут вам пустую (null) ссылку.

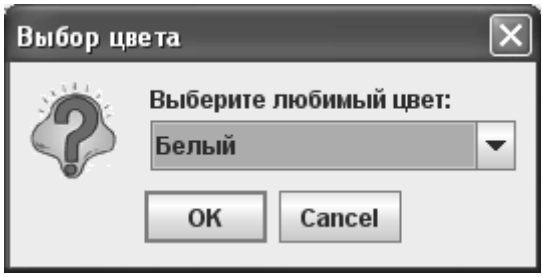
Вторая кнопка выводит на экран диалоговое окно, также получающее данные с помощью текстового поля; для его создания вам понадобится четыре параметра. Первые два параметра будут такими же, как и для только что рассмотренных диалоговых окон — это родительский компонент и сообщение (отметьте, кстати, что ради разнообразия мы записали сообщение массивом строк), в вот вторые два для нас внове. Третьим параметром является заголовок создаваемого диалогового окна, а четвертым — его тип (см. табл. 14.1). По умолчанию в методах showInputDialog() используются тип QUESTION_MESSAGE и соответствующий стандартный значок, но, как видите, сменить его можно. В примере показано вполне реальное применение этой возможности: вы можете повысить уровень «опасности» диалогового окна для ввода данных (таким окном может быть, например, окно для ввода паролей).




Создаваемые с помощью первых двух кнопок диалоговые окна обеспечивают ввод данных с помощью текстовых полей, а вот третья кнопка иллюстрирует кое-что необычное — выбор варианта из нескольких известных альтернатив. Тут же вспоминается раскрывающийся список JComboBox, как раз и позволяющий выбирать из нескольких альтернатив. И на самом деле, класс JOptionPane использует для этой почетной миссии именно его. Правда, создание диалогового окна для такого выбора требует дополнительных хлопот: придется указывать целых семь параметров. Первые четыре нам знакомы — все тот же родительский компонент, сообщение, заголовок окна и его тип. Далее следует значок, вы можете и не задавать его, передав вместо него пустую (null) ссылку, в таком случае будет использован стандартный значок. Наконец, последние два параметра ведают непосредственно выбором — это массив доступных альтернатив (и в нем вы можете задействовать «вездесущий» язык HTML) и элемент, выбранный изначально. Запустив программу с примером и щелкнув на третьей кнопке, вы оцените, как работает данный вариант стандартного диалогового окна. Кстати, возвращается в этом случае выбранный элемент, один из тех, что вы передали ему в качестве доступных альтернатив, или null, если пользователь щелкнул на кнопке Cancel. Для демонстрации этого мы тут же, сразу после выбора, показываем выбранный элемент уже знакомым нам методом showMessageDialog().

---

[4] «А как же локализованные приложения?» — спросите вы: «Можно ли изменять надписи на кнопках в стандартных диалоговых окнах?». Конечно, сделать это можно, и довольно просто. Надписи для кнопок UI-представитель извлекает из специальной таблицы с параметрами внешнего вида, UIDefaults. Получить доступ к этой таблице, то есть получить используемые в данный момент надписи или сменить их, позволяют особые методы get/set класса UIManager. К примеру, заменить кнопку Cancel кнопкой Отмена позволит строка UIManager.put(«OptionPane.cancelButtonText», «Отмена»).




В примере были представлены не все доступные варианты метода showInputDialog(). Так, варианты этого метода, используемые при щелчке на первой кнопке, могут быть еще проще — создаваемые диалоговые окна не требуют передачи ссылок на родительские компоненты и размещаются по центру рабочего стола пользователя. Но такой же эффект можно получить, передавая вместо родительских компонентов ссылки null, так что мы ничего не упустили.

## Получение подтверждений

Следующая группа статических методов класса JOptionPane с общим названием showConfirmDialog() служит для вывода на экран стандартных диалоговых окон, запрашивающих у пользователя разрешение на совершение некоторого действия. Как правило, это запрос вида «Вы действительно хотите удалить файл?», который класс JOptionPane снабжает парой кнопок Да и Нет. Можно прибавить к этим трем кнопкам и кнопку отмены. Получив от пользователя положительный ответ, вы вправе считать, что он понимает последствия совершаемого действия и согласен на него. Рассмотрим несложный пример, в котором изучим все варианты метода showConfirmDialog():

```
// ConfirmDialogs.java
// Стандартные диалоговые окна для получения от
// пользователя подтверждения совершаемого действия
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class ConfirmDialogs extends JFrame {
  // значок для одного из сообщений
  private Icon icon = new ImageIcon("goblet.png");
  public ConfirmDialogs() {
    super("ConfirmDialogs");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // после щелчков на кнопках выводятся сообщения
    JButton confirm1 = new JButton("2 и 4 параметра");
    confirm1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        // диалоговое окно с 4-мя параметрами
        int res = JOptionPane.showConfirmDialog(
```

```
            ConfirmDialogs.this, "Вы этого хотите?",
            "Вопрос",
            JOptionPane.YES_NO_CANCEL_OPTION);
        // простейшие диалоговые окна
        if ( res == JOptionPane.YES_OPTION )
          JOptionPane.showConfirmDialog(
              ConfirmDialogs.this, "Точно хотите?");
        if ( res == JOptionPane.NO_OPTION )
          JOptionPane.showConfirmDialog(
              ConfirmDialogs.this,
              "Значит, не хотите?");
      }
    });
    JButton confirm2 = new JButton("5 параметров");
    confirm2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        int res = JOptionPane.showConfirmDialog(
            ConfirmDialogs.this,
            "Думайте тщательно, итак...",
            "Может произойти ошибка!",
            JOptionPane.YES_NO_OPTION,
            JOptionPane.ERROR_MESSAGE);
      }
    });
    JButton confirm3 = new JButton("6 параметров");
    confirm3.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        int res = JOptionPane.showConfirmDialog(
            ConfirmDialogs.this, "Вам нравится значок?",
            "Полностью настроенное окно!",
            JOptionPane.YES_NO_OPTION,
            JOptionPane.ERROR_MESSAGE, icon);
      }
    });
    // добавляем кнопки в окно
    setLayout(new FlowLayout());
    add(confirm1);
    add(confirm2);
    add(confirm3);
    // выводим окно на экран
    setSize(300, 200);
```

```
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new ConfirmDialogs(); } });
  }
}
```

В примере мы создаем небольшое окно с рамкой JFrame, в котором разместятся три кнопки. Щелкая на этих кнопках, вы будете вызывать на экран стандартные диалоговые окна, созданные различными вариантами метода showConfirmDialog() класса JOptionPane.

Первая кнопка демонстрирует сразу два вида диалоговых окон, созданных самыми простыми вариантами метода с названием showConfirmDialog(). Для начала на экране появляется диалоговое окно, созданное методом, принимающим четыре параметра — это родительский компонент, сообщение для отображения (в качестве сообщения могут выступать любые объекты, перечисленные нами в табл. 14.2, или массив таких объектов), заголовок окна, а также константа из класса JOptionPane, указывающая, набор каких альтернатив будет представлен пользователю для выбора (эти альтернативы появятся внизу диалогового окна в виде набора кнопок). Константы, которые обычно применяются для диалоговых окон, создаваемых методом showConfirmDialog(), перечислены в табл. 14.3.

**Таблица 14.3.** Константы класса JOptionPane, задающие набор альтернатив для диалоговых окон, запрашивающих подтверждение

| Константа | Описание |
|---|---|
| YES_NO_CANCEL_OPTION | Этот набор альтернатив по умолчанию применяется в диалоговых окнах, создаваемых методом showConfirmDialog(). При использовании данного набора в окне появляются кнопки Да, Нет и Отмена |
| YES_NO_OPTION | Набор альтернатив практически идентичен предыдущему за одним исключением — в окне не будет кнопки Отмена |
| OK_CANCEL_OPTION | При использовании этого набора альтернатив в диалоговом окне вместо кнопок Да и Нет появятся кнопки ОК и Отмена |

Как правило, вариант метода showConfirmDialog() с четырьмя параметрами применяется в программах чаще остальных: он сочетает в себе легкость и полноту настройки с простотой вызова. Значения, которые могут возвращать варианты метода showConfirmDialog(), перечислены в табл. 14.4.

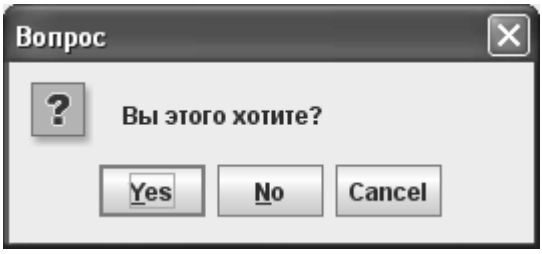
**Таблица 14.4.** Значения, возвращаемые методом showConfirmDialog()

| Значение | Описание |
|---|---|
| YES_OPTION (OK_OPTION) | Это значение возвращается, если пользователь подтвердил свое намерение произвести некоторое действие, щелкнув на соответствующей кнопке (Да или ОК) |
| NO_OPTION | Это значение вы получите в том случае, если пользователь отказался от своего намерения, щелкнув на кнопке Нет |

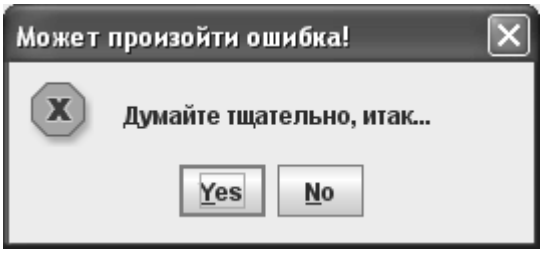
**Таблица 14.4** *(продолжение)*

| Значение | Описание |
|---|---|
| CANCEL_OPTION (CLOSED_OPTION) | Первое значение возвращается, если пользователь не пожелал в данный момент отвечать на вопрос диалогового окна, щелкнув на кнопке Отмена. Второе значение показывает, что пользователь просто закрыл диалоговое окно. Как правило, этот вариант ничем не отличается от отмены |

В примере мы проверяем, какой выбор сделал пользователь, и в зависимости от него предлагаем еще одно диалоговое окно с окончательным подтверждением сделанного выбора. Второе диалоговое окно создается самым лаконичным вариантом метода showConfirmDialog(): он принимает всего два параметра: родительский компонент (вы можете указать и пустую ссылку null, в этом случае диалоговое окно будет размещено по центру главного окна или рабочего стола пользователя, в зависимости от версии JDK) и само сообщение. В качестве набора альтернатив такого окна будет использован набор YES_NO_OPTION. Заметьте, параметры для метода showConfirmDialog() класса JOptionPane идентичны параметрам других уже рассмотренных нами методов данного класса, что без сомнения облегчает работу с ними.




Вторая кнопка демонстрирует, как выглядит диалоговое окно для получения подтверждения, созданное методом showConfirmDialog() с пятью параметрами. Первые четыре параметра неизменны, а в пятом параметре вы сможете указать тип диалогового окна; доступные типы стандартных диалоговых окон мы уже описали в табл. 14.1. В примере мы создаем диалоговое окно для получения подтверждения пользователя, одновременно (с помощью пятого параметра) придавая ему стиль окна, сообщающего об ошибке. В некоторых ситуациях подобная комбинация способна лучше выразить важность задаваемого пользователю вопроса, нежели простая форма диалогового окна.




Ну и наконец, самая полная форма метода showConfirmDialog(), с шестью параметрами, практически ничем не отличается от формы с пятью параметрами, за исключением именно шестого параметра — это значок Icon, который будет отображен в диалоговом

окне взамен стандартного значка, поставляемого вместе с текущими внешним видом и поведением Swing.

Запустив программу с примером, вы воочию увидите, как стандартные диалоговые окна JOptionPane позволяют получать от пользователя подтверждение о проведении дальнейших действий, и сможете применять их в собственных программах, так что ваш пользователь не останется в неведении.

## Дополнительные возможности

Чаще всего класс JOptionPane применяется так, как мы использовали его в рассмотренных примерах: вы вызываете один из многочисленных статических методов, выводящих на экран стандартное диалоговое окно, не забывая передать в метод список подходящих параметров, полностью определяющих внешний вид будущего окна. Но у класса JOptionPane имеются и другие достоинства. Прежде всего это способность работать в виде самого обыкновенного компонента: вы создаете объект класса JOptionPane, используя при этом подходящий конструктор, настраиваете остальные свойства с помощью методов get/set (а свойствами класса являются все свойства диалоговых окон, прежде задаваемых нами в виде параметров статических методов, в том числе тип сообщения, само сообщение, значок и т. п.), а затем выводите полученный компонент на экран так, как вам угодно (к примеру, в диалоговом окне, созданном своими руками, если вам это необходимо). Создание компонента имеет свои преимущества: вы получаете больший контроль над тем, где и когда появляются стандартные диалоговые окна, а также экономию памяти — стандартные диалоговые окна одного вида и предназначения могут создаваться один раз, меняться будут лишь текст сообщения и заголовок окна.

Далее можно использовать те методы showXXXDialog(), которые мы изучили в данной главе, а также методы с общим названием showInternalXXXDialog(). В последнем случае стандартные диалоговые окна появляются в специальном компоненте Swing с названием JDesktopPane, предназначенном для создания приложений с многодокументным интерфейсом (MDI). Но чтобы методы showInternalXXXDialog() работали, необходима поддержка в вашем приложении многодокументного интерфейса и рабочего стола (компонента JDesktopPane), а это, учитывая все снижающуюся популярность многодокументного интерфейса, встречается уже не так часто, к тому же в таком виде стандартные диалоговые окна привлекают к себе недостаточно внимания. Поэтому в подавляющем большинстве случаев вполне хватает изученных нами методов (и в большинстве случаев они лучше соответствуют нуждам приложений).

Помимо рассмотренных нами методов showXXXDialog(), предназначенных для вывода сообщений, ввода несложных данных и получения подтверждений, в классе JOptionPane имеется еще один метод с названием showOptionDialog(). С его помощью вы можете вывести на экран любое сообщение, разместить в нем любые компоненты и снабдить его нужными типом и набором альтернатив для выбора. Вот только метод этот довольно громоздкий, и, если вам потребуется совершенно особое диалоговое окно с необычным набором компонентов, лучше это диалоговое окно создать «с нуля», и заполнить компонентами окно JDialog.

# *Выбор файлов в компоненте JFileChooser*

Выбор файлов необходим в любом более или менее приличном приложении: в самом деле, все, что пользователями делается, чаще всего сохраняется в файлах, даже если расположены они на удаленных серверах. Компонент JFileChooser — отличный и легко настраиваемый инструмент для выбора файлов и при необходимости каталогов. Особенности различных файловых систем скрыты в подклассах абстрактного

класса FileSystemView, и беспокоиться вам об этих различиях не придется: выбранный вами для приложения внешний вид отобразит файловую структуру как подобает – соответственно текущей операционной системе[5]. В самом начале главы мы упомянули, что JFileChooser – это обычный компонент, унаследованный от класса JComponent, так что вы можете включить его в любое место своего интерфейса. Так оно и есть, но удобнее использовать пару методов, показывающих компонент для выбора файлов в собственном модальном диалоговом окне. Настроить и вывести на экран несложное диалоговое окно для открытия файла или сохранения в нем данных совсем легко. Рассмотрим пример:

```
// SimpleFileChooser.java
// Создание простых диалоговых окон
// открытия и сохранения файлов
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class SimpleFileChooser extends JFrame {
  // создадим общий экземпляр JFileChooser
  private JFileChooser fc = new JFileChooser();
  public SimpleFileChooser() {
    super("SimpleFileChooser");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // кнопка, создающая диалоговое окно для открытия файла
    JButton open = new JButton("Открыть...");
    open.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        fc.setDialogTitle("Выберите каталог");
        // настроим для выбора каталога
        fc.setFileSelectionMode(
            JFileChooser.DIRECTORIES_ONLY);
        int res = fc.showOpenDialog(
            SimpleFileChooser.this);
        // если файл выбран, покажем его
        if ( res == JFileChooser.APPROVE_OPTION )
          JOptionPane.showMessageDialog(
              SimpleFileChooser.this,
              fc.getSelectedFile());
      }
    });
```

[5] Все операционные системы поддерживаются только одним стандартным внешним видом, поставляющимся вместе с JDK, – внешним видом Metal, который используется в Swing по умолчанию. Остальные внешние виды могут показывать только соответствующую файловую систему. Так что запускать под Windows программу с внешним видом CDE или Motif можно, но вот выбирать файлы в ней будет затруднительно – файловая система Unix для Windows не слишком подходит.

```
    // кнопка, создающая диалоговое окно
    // для сохранения файла
    JButton save = new JButton("Сохранить...");
    save.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        fc.setDialogTitle("Сохранение файла");
        // настройка режима
        fc.setFileSelectionMode(
            JFileChooser.FILES_ONLY);
        int res = fc.showSaveDialog(
            SimpleFileChooser.this);
        // сообщим об успехе
        if ( res == JFileChooser.APPROVE_OPTION )
          JOptionPane.showMessageDialog(
              SimpleFileChooser.this,
              "Файл сохранен");
      }
    });
    // добавим кнопки и выведем окно на экран
    setLayout(new FlowLayout());
    add(open);
    add(save);
    setSize(300, 200);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new SimpleFileChooser(); } });
  }
}
```

В примере мы создаем небольшое окно с двумя кнопками. После щелчков на этих кнопках на экране появляются диалоговые окна для открытия и сохранения файлов. Заметьте, что на весь пример у нас всего один экземпляр компонента для выбора файлов JFileChooser, хотя мы и сохраняем файлы, и открываем их. Более того, эти действия можно совершать многократно, поскольку, как уже отмечалось, JFileChooser представляет собой обычный компонент, и вы можете создать его один раз, а затем после соответствующей настройки снова и снова выводить в подходящих диалоговых окнах. При щелчке на первой кнопке на экран выводится диалоговое окно открытия файлов. Посмотрите, как можно задать соответствующий заголовок для диалогового окна методом setDialogTitle(). Перед выводом диалогового окна для выбора файлов на экран мы настраиваем режим выбора. Компонент JFileChooser может работать в одном из трех режимов (режим выбора хранится в свойстве fileSelectionMode), перечисленных в табл. 14.5.

**Таблица 14.5.** Доступные режимы работы JFileChooser

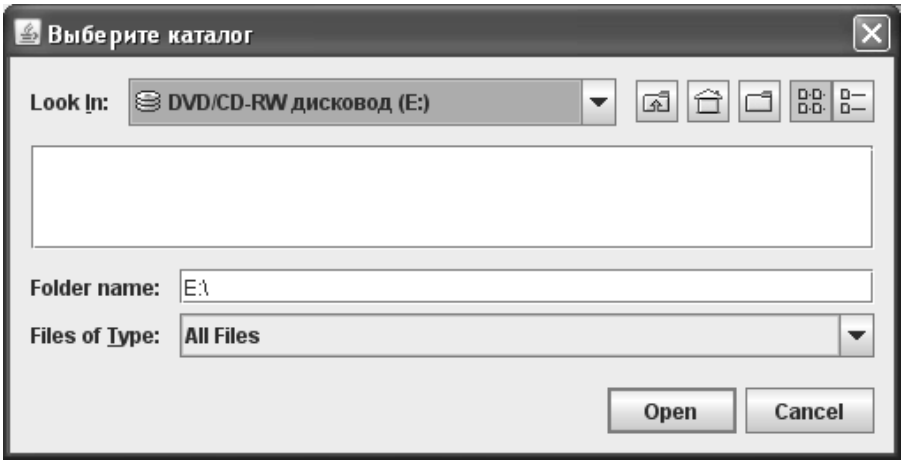
| Режим работы | Описание |
|---|---|
| FILES_ONLY | Пользователю для выбора (независимо от того, сохраняется файл или открывается) будут доступны только файлы, но не каталоги. По умолчанию JFileChooser работает именно в этом режиме и правильно делает, поскольку подобный режим необходим чаще остальных. Именно в этом режиме пользователь сохраняет свою работу в файлах |
| FILES_AND_DIRECTORIES | В этом режиме пользователь может выбирать и каталоги, и файлы. Как правило, этот режим хорош там, где вы меняете общие свойства файловой системы, и файлы для вас не отличаются от каталогов |
| DIRECTORIES_ONLY | Этот весьма ценный режим разрешает пользователю выбирать исключительно каталоги. Особенно хорош он там, где нужно выбирать каталоги под временные файлы, указывать каталоги с исходными текстами и т. п. |

Мы выбираем последний режим, так что открыть вы сможете только каталог. Для вывода диалогового окна открытия файлов (в нашем случае каталогов) служит удобный метод showOpenDialog(). Этому методу нужно только передать ссылку на «родительский» компонент, относительно которого окно будет располагаться на экране. В ответ метод возвращает целое число, определяющее сделанный пользователем выбор (табл. 14.6).

**Таблица 14.6.** Значения, возвращаемые методами класса JFileChooser

| Константа | Описание |
|---|---|
| APPROVE_OPTION | Выбор файла в диалоговом окне прошел успешно, можно получить выбранный файл методом getFile() |
| CANCEL_OPTION | Пользователь отменил выбор файла, щелкнув на кнопке Cancel |
| ERROR_OPTION | При выборе файла произошла ошибка, или пользователь закрыл диалоговое окно для выбора файлов |

В нашем примере мы проверяем, успешен ли был выбор файла, и если да (метод showOpenDialog() вернул нам значение APPROVE_OPTION), выводим на экран имя выбранного каталога (с помощью класса JOptionPane).



Щелкнув на второй кнопке, вы сможете вызвать на экран диалоговое окно для сохранения файлов. Разница между ним и создаваемым первой кнопкой окном для открытия файлов невелика, всего лишь в надписях, используемых для компонентов JFileChooser. Мы также устанавливаем режим выбора файлов (FILES_ONLY), задаем собственный заголовок для создаваемого окна методом setDialogTitle() и выводим диалоговое окно на экран методом showSaveDialog(). Так же как и в случае окна для открытия файлов, для этого требуется только один параметр — «родительский» компонент. Если выбор файла для сохранения проходит успешно (нам возвращают значение APPROVE_OPTION), на экране появляется краткое сообщение, подтверждающее успешное сохранение файла (которое, правда, беззастенчиво лукавит, поскольку никакого сохранения не происходит).

Запустив программу с примером, вы сможете выбрать любой файл в своей файловой системе с помощью компонента JFileChooser. Для этого, как нетрудно видеть, нужно всего несколько строк кода. Но не забывайте, что компонент JFileChooser предназначен всего лишь для выбора файла, а проводить с выбранным файлом соответствующие манипуляции (записывать в него информацию или анализировать его содержимое) вам придется своими руками.

Вопрос с переводом англоязычных строк в приложении, использующим компонент JFileChooser, не отличается от подхода в компоненте JOptionPane. По умолчанию все строки написаны на английском языке (что легко видеть на снимке экрана), они берутся они из специальных ресурсов класса UIManager. Например, заменить надпись Open можно так: UIManager.put(«FileChooser.openButtonText», «Открыть»). Заменив эти ресурсы на строки, написанные на нужном вам языке, вы сможете перевести стандартные диалоги (но заменять строки необходимо *перед* созданием компонентов). Ну а полный список ключей для этих строк размещен в интерактивной документации JDK и Swing.

## Фильтры файлов

По умолчанию в списке файлов, который компонент JFileChooser предоставляет пользователю, содержатся все файлы из текущего каталога (того, что пользователь просматривает в данный момент). Порой это очень неудобно: как правило, пользователю необходим файл с определенным именем или типом, и ему приходится «рыскать» по всему списку в поиске нужного (только представьте себе поиск нужного файла среди нескольких тысяч). Мы можем сократить круг поиска пользователя с помощью *фильтра файлов*, который легко встраивается в компонент JFileChooser.

Фильтры файлов для JFileChooser описаны абстрактным классом FileFilter из пакета javax.swing.filechooser. Дело происходит следующим образом: вы наследуете от класса FileFilter, основным методом которого является accept(). Этот метод класс JFileChooser вызывает для каждого файла, перед тем, как добавить его в список файлов, предоставляемых для выбора пользователю. Вы, в методе accept(), анализируете файл (как правило, по имени, хотя никто не запрещает вам анализировать файл по его содержимому, если вы захотите удостовериться, что оно соответствует определенному формату) и решаете, «достоин» ли файл появления перед глазами пользователя[6]. Рассмотрим несложный пример, в котором попробуем написать собственный фильтр файлов:

```
// FilteringFiles.java
// Фильтры файлов в компоненте JFileChooser
import javax.swing.*;
import javax.swing.filechooser.*;
```

[6] Это не что иное, как шаблон проектирования *стратегия (strategy)*., один из самых распространенных в Swing.

```
import java.awt.*;

public class FilteringFiles extends JFrame {
  public FilteringFiles() {
    super("FilteringFiles");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // выводим окно на экран
    setSize(300, 200);
    setVisible(true);
    // настраиваем компонент для выбора файла
    JFileChooser chooser = new JFileChooser();
    chooser.setDialogTitle("Выберите текстовый файл");
    // присоединяем фильтр
    chooser.addChoosableFileFilter(
        new TextFilesFilter());
    // выводим диалоговое окно на экран
    int res = chooser.showOpenDialog(this);
    if ( res == JFileChooser.APPROVE_OPTION )
      JOptionPane.showMessageDialog(
          this, chooser.getSelectedFile());
  }
  // фильтр, отбирающий текстовые файлы
  class TextFilesFilter extends FileFilter {
    // принимает файл или отказывает ему
    public boolean accept(java.io.File file) {
      // все каталоги принимаем
      if ( file.isDirectory() ) return true;
      // для файлов смотрим на расширение
      return ( file.getName().endsWith(".txt") );
    }
    // возвращает описание фильтра
    public String getDescription() {
      return "Текстовые файлы (*.txt)";
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new FilteringFiles(); } });
  }
}
```

Мы создаем небольшое окно, и сразу же после того как это окно появляется на экране, выводим компонент для выбора файлов JFileChooser с подключенным к нему фильтром файлов. Настройка нам уже знакома: мы просто устанавливаем заголовок диалогового окна для открытия файлов, а затем выводим это окно на экран методом showOpenDialog().

Посмотрите, как создается фильтр файлов. Мы унаследовали внутренний класс от абстрактного класса FileFilter и переопределили два метода базового класса. Метод getDescription() не таит в себе никаких тайн и должен просто сообщать краткое описание фильтра файлов. Как правило, в это описание добавляется правило, по которому действует фильтр. В нашем случае мы просто указали маску, по которой отбираются файлы. Главная «рабочая лошадка» фильтра — метод accept(). Именно в нем нам необходимо проанализировать имя файла (или даже его содержимое) и сделать вывод о его пригодности для включения в список файлов, который окажется перед глазами пользователя. Обратите внимание, что в данный метод попадают абсолютно *все* файлы и каталоги, поэтому мы прежде всего проверяем, не каталог ли нам попался, и все каталоги включаем в список (в противном случае, при отсутствии в списке каталогов, пользователь просто не сможет перемещаться по файловой системе). Для обычных файлов мы проводим анализ имени: наш фильтр должен отбирать текстовые файлы, а их имена (точнее расширения) заканчиваются символами «.txt».

Присоединить фильтр файлов к компоненту JFileChooser можно двумя способами. Оба способа предназначены для присоединения нового фильтра к списку фильтров, уже имеющихся в компоненте JFileChooser. В примере мы применили первый способ, вызвав метод addChoosableFileFilter(). Данный метод присоединит наш фильтр файлов к списку остальных фильтров (а в компоненте JFileChooser всегда имеется, по крайней мере, один фильтр — тот, что пропускает все файлы и каталоги «без разбора»), но не выберет новый фильтр, так что пользователю самому придется его выбирать . Есть и второй способ — вызов метода setFileFilter(). Он, так же как и метод addChoosableFileFilter(), добавит фильтр в список имеющихся фильтров, но к тому же выберет его текущим фильтром и незамедлительно отфильтрует все файлы в текущей директории, так что пользователь сразу же увидит что позволяет выбирать данный фильтр. Какой метод использовать зависит от конкретной ситуации, главным образом от того, какой фильтр вы планируете использовать по умолчанию.

Запустив программу с примером, вы увидите, что после присоединения фильтра файлов компонент JFileChooser стал отображать только текстовые файлы (после того, как фильтр выбран в списке фильтров), и выбирать среди них нужный файл стало гораздо проще. Видно, как описание, переданное нами из метода getDescription(), появляется в раскрывающемся списке доступных фильтров файлов. Фильтры файлов намного облегчают жизнь пользователя, особенно при выборе из большого количества файлов, так что стоит всегда держать их «под рукой».

Список фильтров можно не только дополнять новыми участниками, но и уменьшать и узнавать, какие фильтры доступны в данный момент. Полезные методы для работы с фильтрами изложены в следующей таблице:

**Таблица 14.7.** Методы для работы с фильтрами файлов в компоненте JFileChooser

| Метод | Описание |
|---|---|
| getChoosableFileFilters() | Возвращает массив всех добавленных в JFileChooser фильтров файлов, включая фильтр по умолчанию (для выбора без ограничения) |
| getFileFilter() | Возвращает фильтр, выбранный в данный момент. Полезный метод, иногда именно выбранный фильтр определяет, в каком формате будет сохранен файл |

**Таблица 14.7** *(продолжение)*

| Метод | Описание |
|---|---|
| setAcceptAllFileFilterUsed(boolean) | Метод позволяет указать, надо ли показывать универсальный фильтр для выбора без ограничения. Отличная замена ручному удалению этого фильтра |
| removeChoosableFileFilter(FileFilter) | Удаляет указанный в качестве параметра фильтр из списка фильтров |

## Внешний вид файлов

Возможность фильтрации файлов перед тем, как они попадут в список файлов компонента JFileChooser, и их увидит пользователь – не единственное дополнение к стандартному диалоговому окну для выбора файлов. Вы также можете настроить внешний вид файлов, отображаемых компонентом JFileChooser. Оказывается, нет ничего невозможного в том, чтобы указать, какие у файлов должны быть имена, значки и описания.

Для описания внешнего вида файлов компонент JFileChooser применяет объект специального класса FileView из пакета javax.swing.filechooser. Данный объект служит для получения исчерпывающей информации о внешнем виде отображаемого файла[7]. С помощью объекта FileView вы можете задать имена файлов (если вам нужно, чтобы они отличались от имен файлов в файловой системе, например вы можете хранить название документа непосредственно в файле и вместо имени файла выводить на экран это название), значки для файлов и каталогов, описания типов файлов и многое другое. Для каждого атрибута внешнего вида файла предназначен свой метод, вы наследуете от класса FileView и переопределяете необходимые вам методы.

Рассмотрим теперь небольшой пример, в котором попробуем по-своему отобразить некоторые файлы и каталоги:

```
// CustomFileView.java
// Внешний вид файлов в компоненте JFileChooser
import javax.swing.*;
import javax.swing.filechooser.*;
import java.awt.*;

public class CustomFileView extends JFrame {
  public CustomFileView() {
    super("CustomFileView");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // выводим окно на экран
    setSize(300, 200);
    setVisible(true);
    // настраиваем компонент для выбора файлов
    JFileChooser chooser = new JFileChooser();
    chooser.setDialogTitle("Выберите файл");
```

[7] Подобная ситуация типична для компонентов Swing и их внешнего вида: как правило, внешний вид определяется сторонними объектами. Это снова тот самый шаблон проектирования *стратегия (strategy)*.

```
    chooser.setFileView(new NewFileView(
        chooser.getFileSystemView()));
    // показываем диалоговое окно
    int res = chooser.showOpenDialog(this);
  }
  // объект, определяющий внешний вид файлов
  class NewFileView extends FileView {
    // значки, применяемые для файлов
    private Icon fileIcon = new ImageIcon("file.gif");
    private Icon folderIcon = new ImageIcon("folder.gif");
    // конструктору необходимо описание файловой системы
    public NewFileView(FileSystemView fileSystem) {
      this.fileSystem = fileSystem;
    }
    private FileSystemView fileSystem;
    // возвращает значок для файла
    public Icon getIcon(java.io.File file) {
      // основные части файловой системы пропускаем
      if ( fileSystem.isRoot(file) ||
          fileSystem.isDrive(file) )
        return null;
      if ( file.isDirectory() )
        return folderIcon;
      else return fileIcon;
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new CustomFileView(); } });
  }
}
```

Мы создаем небольшое окно, после его вывода на экран появится стандартное диалоговое окно для выбора файлов со специальным объектом, задающим внешний вид файлов. Настройка компонента JFileChooser здесь аналогична настройке в уже рассмотренных примерах: для диалогового окна задается заголовок, после чего оно выводится на экран.

Во внутреннем классе NewFileView, унаследованном от абстрактного класса FileView, мы описываем, как должны выглядеть файлы в нашем компоненте JFileChooser. В классе FileView имеется несколько методов, отвечающих в том числе за описание файла и его типа, однако мы ограничиваемся настройкой значков, которые будут соответствовать файлам, переопределяя только один метод getIcon(). В этом методе необходимо вернуть значок Icon. Мы будем возвращать собственные значки для всех файлов и каталогов. Правда, при работе с методами класса FileView надо иметь в виду следующее: в них передаются *все* отображаемые компонентом JFileChooser элементы, в том числе файлы, представляющие со-

бой корни файловой системы, устройства чтения гибких дисков и компакт-дисков, особые каталоги пользователя и т. п. Если для таких особых файлов возвращать те же значки, что применяются для обычных файлов и каталогов, пользователю станет неудобно работать с компонентом для выбора файлов и выглядеть такой компонент будет не лучшим образом. Именно поэтому мы передаем в конструктор внутреннего класса ссылку на используемое компонентом JFileChooser описание файловой системы FileSystemView. Вызывая методы объекта FileSystemView, мы узнаем, не принадлежит ли файл к «особенным», и для таких «особенных» файлов возвращаем вместо значка пустую (null) ссылку. Это означает, что значок для таких элементов подыщет текущий UI-представитель компонента JFileChooser. Обычные файлы и каталоги будут отображаться с нашими собственными значками.

Запустив программу с примером, вы увидите, как теперь выглядят каталоги и файлы в списке файлов компонента JFileChooser. Как правило, объект, отвечающий за внешний вид файлов, работает в тандеме с фильтром файлов определенного типа, так что пользователь сразу оценивает, какие файлы ему предстоит открывать и быстро запоминает их внешний вид. Как мы уже отметили, в классе FileView имеются и другие методы, которые позволят вам, к примеру, на свой лад описать файл или его тип (по умолчанию описания у файла либо вообще нет, либо используется описание файла, получаемое от операционной системы). Все они легко найдутся в интерактивной документации JDK.

## Дополнительные компоненты

Несмотря на то, что компонент для выбора файлов JFileChooser чаще всего используется в своем стандартном виде, вы не ограничены тем, что вам предлагает библиотека, и можете добавить к стандартным компонентам, составляющим JFileChooser, свои собственные компоненты. Дополнительные компоненты, как правило, служат для предварительного просмотра содержимого файлов или для произведения над ними каких-либо действий без выхода из модального диалогового окна, в котором располагается JFileChooser. Специально для поддержки дополнительных компонентов в классе JFileChooser определен метод setAccessory(), который вы можете вызвать для добавления своего компонента (добавить, правда, можно только один компонент, но он может состоять из многих других компонентов). Куда будет добавлен ваш компонент, зависит от используемого внешнего вида, обычно он размещается рядом со списком файлов.

Как правило, при добавлении своих собственных компонентов вас начинает интересовать, что происходит в компоненте JFileChooser, например, какой именно файл в данный момент выбран пользователем. Это необходимо для совершения действий над файлом или его предварительного просмотра, даже если выбор пользователя не окончательный. Специального события, сообщающего о выборе очередного файла из списка, в классе JFileChooser нет, однако не стоит забывать о механизме привязанных свойств JavaBeans. Все наиболее важные свойства компонентов Swing являются привязанными, и узнать о любом их изменении вы сможете, присоединив к компоненту слушателя PropertyChangeListener. Компонент JFileChooser не является исключением: свойства, управляющие выделенным в данный момент файлом или каталогом, также относятся к привязанным, поэтому вы всегда будете знать, к какому файлу или каталогу перешел пользователь, и сможете произвести с этим файлом или каталогом необходимые действия.

А теперь добавим к стандартному компоненту для выбора файлов панель для предварительного просмотра (в нашем примере она будет поддерживать предварительный просмотр изображений). Мы будем следить за сменой выбранного файла (с помощью привязанного свойства) и, если выбранный файл будет хранить изображение, выводить на экран уменьшенный вариант этого изображения. К компоненту JFileChooser мы также присоединим фильтр файлов, который будет выбирать только файлы с изображениями — так просматривать и выбирать изображения будет еще удобнее.

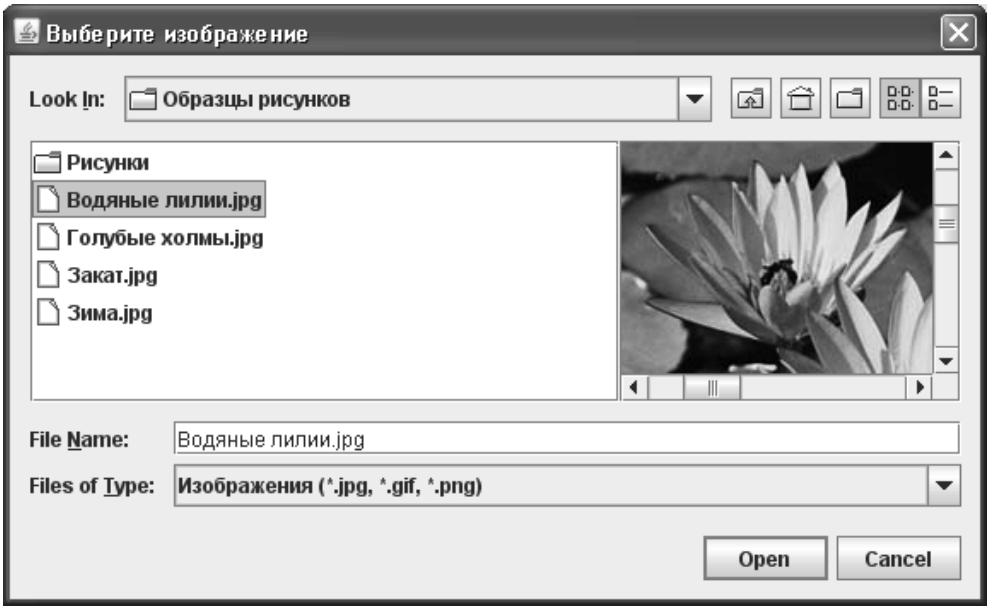
```
// PreviewingFiles.java
// Предварительный просмотр файлов
// в компоненте JFileChooser
import javax.swing.*;
import javax.swing.filechooser.*;
import java.beans.*;
import java.awt.*;

public class PreviewingFiles extends JFrame {
  public PreviewingFiles() {
    super("PreviewingFiles");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // выводим окно на экран
    setSize(300, 200);
    setVisible(true);
    // настраиваем компонент для выбора файла
    JFileChooser chooser = new JFileChooser();
    chooser.setDialogTitle("Выберите изображение");
    // присоединяем фильтр
    chooser.setFileFilter(new ImageFilesFilter());
    // убираем универсальный фильтр
    chooser.setAcceptAllFileFilterUsed(false);
    // присоединяем дополнительный компонент
    Previewer previewer = new Previewer();
    chooser.setAccessory(previewer);
    // регистрируем в качестве слушателя
    chooser.addPropertyChangeListener(previewer);
    // выводим диалоговое окно на экран
    int res = chooser.showOpenDialog(this);
    if ( res == JFileChooser.APPROVE_OPTION )
      JOptionPane.showMessageDialog(
          this, chooser.getSelectedFile());
  }
  // компонент для предварительного просмотра
  class Previewer extends JPanel
      implements PropertyChangeListener {
    private JLabel label;
    public Previewer() {
      // настраиваем контейнер
      setLayout(new BorderLayout());
      setPreferredSize(new Dimension(200, 200));
      // создаем надпись в панели прокрутки
```

```
        label = new JLabel();
        JScrollPane scroller = new JScrollPane(label);
        add(scroller);
      }
      public void propertyChange(PropertyChangeEvent e) {
        if ( e.getPropertyName().equals(
            JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)) {
          // сменился выбранный файл, покажем его
          if ( e.getNewValue() != null )
            label.setIcon(new ImageIcon(
                e.getNewValue().toString()));
        }
      }
    }
    // фильтр, отбирающий файлы с изображениями
    class ImageFilesFilter extends FileFilter {
      // принимает файл или отказывает ему
      public boolean accept(java.io.File file) {
        // все каталоги принимаем
        if ( file.isDirectory() ) return true;
        // имя файла не будет учитывать регистр
        String name = file.getName().toLowerCase();
        // для файлов смотрим на расширение
        return ( name.endsWith(".jpg") ||
            name.endsWith(".gif") || name.endsWith(".png") );
      }
      // возвращает описание фильтра
      public String getDescription() {
        return "Изображения (*.jpg, *.gif, *.png)";
      }
    }
    public static void main(String[] args) {
      SwingUtilities.invokeLater(
          new Runnable() {
            public void run() { new PreviewingFiles(); } });
    }
  }
```

Мы создаем небольшое окно, после его вывода на экран появится модальное диалоговое окно с компонентом для выбора файлов. Настройка компонента для выбора файлов происходит в несколько этапов: для начала мы создаем объект JFileChooser и задаем заголовок для будущего диалогового окна. Затем присоединяется фильтр файлов: он описан во внутреннем классе и будет принимать только файлы, хранящие изображе-

ния (отбор производится на основе расширения файла). Так как изображения некоторые программы и аппаратные устройства именуют по-разному, имеет смысл проверять окончание файлов вне зависимости от регистра. Для этого мы делаем имя файла строчным. Чтобы пользователь не смог выбрать фильтр без ограничений, мы отключаем его методом setAcceptAllFileFilterUsed(false). Напоследок производится настройка компонента для предварительного просмотра изображений Previewer: после создания мы присоединяем его к компоненту для выбора файлов, применяя метод setAccessory(). Прежде чем вывести компонент JFileChooser на экран, мы регистрируем в нем слушателя событий PropertyChangeEvent, которые запускаются при смене значений привязанных свойств. Для удобства слушатель реализован все в том же классе Previewer.

Компонент, реализующий предварительный просмотр изображений, описан во внутреннем классе Previewer и унаследован от панели JPanel. Состоит он из панели прокрутки, добавленной в центр компонента (в компоненте задействуется менеджер расположения BorderLayout), в панели прокрутки располагается надпись JLabel, которая и будет использоваться для вывода изображения. Заметьте, что мы вручную указываем предпочтительный размер компонента: если этого не сделать, он будет динамически менять свой размер в зависимости от размеров просматриваемого изображения, затрагивая остальной интерфейс, а это сделает работу пользователя некомфортной. Самое интересное происходит в методе propertyChange(), вызываемом при смене значения какого-либо привязанного свойства компонента JFileChooser. Мы следим за свойством, которое хранит имя выделенного в данный момент файла, название этого свойства содержится в константе SELECTED_FILE_CHANGED_PROPERTY класса JFileChooser. Если свойство с этим названием поменялось, мы проводим следующие действия: проверяем, что новое значение свойства не равно пустой ссылке null (это происходит, когда пользователь обнуляет выделенный файл, например выделяя каталог или полностью сбрасывая выделение), и если это так, создаем новый значок ImageIcon, передавая ему имя нового выделенного файла, и устанавливаем для надписи JLabel полученный значок. Если вы помните, в главе 8 мы говорили, что надписи проводят автоматическую проверку корректности и перерисовку при смене своих свойств, так что вызова метода setIcon() будет достаточно для того, чтобы в нашем компоненте появилось новое изображение.



Запустив пример, вы оцените, насколько удобнее выбирать изображения, когда есть возможность не только увидеть название файла, в котором они хранятся, но и быстро взглянуть на само изображение. Создавая подобным образом любой необходимый вашему приложению компонент, вы сможете присоединить его к компоненту JFileChooser и намного повысить удобство работы с последним.

На самом деле останавливаться на только продемонстрированном подходе вовсе не обязательно. Не забывайте о том, что компонент JFileChooser — самый обычный компонент, пусть и состоящий из множества других компонентов, так что вы вполне можете добавить его в свои диалоговые окна наряду с многими другими компонентами. Функциональность JFileChooser легко дополнить с помощью нескольких дополнительных компонентов. Вы даже сможете убрать из компонента JFileChooser ряд кнопок («OK» и «Отмена»), вызвав метод setControlButtonsAreShown(false), и заменить их более подходящим для вашей ситуации вариантом. Всего этого достаточно для того, чтобы сделать выбор файлов в компоненте JFileChooser не только простым и быстрым (как для пользователя, так и для программиста), но и удивительно гибким.

## *Выбор цвета в компоненте JColorChooser*

Еще одно стандартное диалоговое окно, любезно предоставляемое нам Swing, реализовано в классе JColorChooser и предназначено для максимально быстрого (для программиста) и удобного (для пользователя) выбора цвета. Работать с компонентом JColorChooser так же просто, как и с компонентом для выбора файлов JFileChooser, а во многом и проще, так как возможностей тонкой настройки у JColorChooser куда меньше. Типичная схема выбора цвета такова: вы создаете объект класса JColorChooser и по мере необходимости выводите его на экран в диалоговом окне стандартного вида с помощью метода showDialog(), попутно задавая родительский компонент и заголовок окна. Рассмотрев пример, мы увидим, каким образом JColorChooser чаще всего применяется в программах:

```
// SimpleColorChooser.java
// Выбор цвета с помощью JColorChooser
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SimpleColorChooser extends JFrame {
 public SimpleColorChooser() {
     super("SimpleColorChooser");
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     // кнопка выводит на экран окно выбора цвета
     JButton choose = new JButton("Выбор цвета фона");
     choose.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
             Color color = JColorChooser.showDialog(
              SimpleColorChooser.this,
               "Выберите цвет фона",
                 getContentPane().getBackground());
             // если цвет выбран, используем его
```

```
            if ( color != null)
                getContentPane().setBackground(color);
            repaint();
        }
    });
    // выводим окно на экран
    setLayout(new FlowLayout());
    add(choose);
    setSize(300, 200);
    setVisible(true);
}
public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new SimpleColorChooser(); } });
}
}
```

В примере мы создаем небольшое окно, а компонент JColorChooser позволит нам выбрать цвет фона для панели содержимого. В качестве панели содержимого по умолчанию выступает простая панель JPanel, а она по умолчанию непрозрачна, то есть закрашивает все свое пространства цветом фона. Так что для смены цвета фона всего нашего окна нам достаточно будет сменить цвет фона панели содержимого, вызвав метод setBackground().

Объект JColorChooser, так же как и объект JFileChooser, может создаваться в программе единожды, а затем с большими или меньшими вариациями в настройке использоваться на протяжении всего времени ее работы. Это значительно ускоряет процесс появления диалогового окна на экране (если сравнивать с созданием объекта JColorChooser прямо перед выводом диалогового окна на экран) и экономит ресурсы, но требует от вас самостоятельно найти для него место на экране или создать диалоговое окно. С другой стороны, вывести стандартное диалоговое окно для выбора цвета на экран совсем просто: надо всего лишь вызвать статический метод showDialog(), которому передается три параметра. Первым параметром является родительский компонент диалогового окна (это может быть любой компонент, относительно него диалоговое окно располагается на экране), в качестве второго параметра вы задаете заголовок диалогового окна, а третьим параметром является цвет, изначально выбранный в компоненте JColorChooser. Третий параметр можно задать и пустой (null) ссылкой, в таком случае изначально цвета выбрано не будет. В ответ метод showDialog() (он выводит на экран модальное диалоговое окно, так что вам придется подождать, пока оно закончит свою работу) вернет вам цвет Color, который выбрал пользователь, или пустую (null) ссылку, если пользователь передумал и решил цвет не выбирать. В примере мы проверяем, что выбор был сделан, и устанавливаем выбранный цвет в качестве цвета фона.

В компоненте JColorChooser имеется возможность дополнительной настройки: к стандартным способам выбора цвета (из вполне приличного заранее определенного набора, с помощью цветовых моделей от RGB до CMYK) вы можете добавить собственную панель для выбора цвета, унаследовав ее от абстрактного класса AbstractColorChooserPanel из пакета javax.swing.colorchooser. Однако требуется это крайне редко: стандартные способы выбора цвета, представленные нам компонентом JColorChooser, достаточно хороши. Кроме того, вы можете присоединить к компоненту JColorChooser панель просмотра

(методом setPreviewPanel()), в которой принято показывать, как смена цвета скажется на работе пользователя еще до того, как он окончательно сделает свой выбор. В панели просмотра вы можете показать, к примеру, уменьшенную копию изображения с новыми цветами. Для того чтобы узнать о смене цвета в компоненте JColorChooser еще до окончательного выбора, используйте модель выбора цвета ColorSelectionModel. Присоединив к этой модели слушателя ChangeListener, вы будете узнавать, какие цвета «опробует» пользователь перед тем, как принять окончательное решение.

## *Резюме*

Стандартные диалоговые окна легко настраивать и выводить на экран, они хорошо знакомы пользователю и наглядно показывают ему, что ваше приложение не сложнее и не запутаннее других. Использование в подходящих местах стандартных диалоговых окон упрощает и ускоряет не только процесс знакомства пользователя с приложением, но и жизнь программиста, позволяя выводить и получать информацию всего несколькими строками кода.

# Глава 15. Уход за деревьями

Разнообразную информацию можно представить в виде иерархических отношений «родитель-потомок», когда некоторые «младшие» данные хранятся как часть «старших» данных. «Старшие» данные в свою очередь могут иметь своих «родителей», и так может продолжаться бесконечно. Иерархические отношения данных принято отображать в специальных компонентах пользовательского интерфейса, *деревьях* (trees), идея которых навеяна генеалогическими деревьями, издавна используемыми для отображения семейных отношений. Компьютерное дерево и на самом деле похоже на дерево, только поставленное с «ног» на «голову»: вверху располагается *корень* (root), которого, впрочем, может и не быть, а ниже находятся его потомки (первые ветви), у которых могут быть свои потомки, и т.д.

У деревьев, используемых в мире компьютеров, есть своя нехитрая терминология. Информация о ветвях дерева хранится в специальных объектах, *узлах* (nodes). Узел, у которого нет потомков, справедливо называется *листом* (leaf) дерева. Все узлы-потомки одного узла-предка называются *элементами одного уровня* (siblings). *Путь* (path) в дереве определяет, как добраться до некоторого узла, и представляет собой последовательность узлов, начиная от корня, по которым следует пройти, чтобы найти нужный нам узел.

Деревья в Swing реализованы компонентом JTree. Класс JTree буквально напичкан различными методами, более того, существует целый вспомогательный пакет javax.swing.tree с добрым десятком классов и интерфейсов, обеспечивающих правильную работу деревьев Swing. Это неудивительно: деревья Swing обладают впечатляющими возможностями, и все что возможно делать с деревьями, они делать позволяют. Пакет javax.swing.tree хранит основные «строительные кирпичики» деревьев Swing: интерфейс TreeNode описывает узел дерева; модель дерева, хранящая его данные, описана интерфейсом TreeModel; путь в дереве описывается объектом TreePath. Пока нам этого хватит, ну а по мере чтения этой главы мы узнаем и об остальных частях деревьев Swing.

Впрочем, как и всегда в Swing, сказанное не означает, что вам придется долго и тщательно изучать возможности класса JTree, прежде чем «вырастить» даже самое простое дерево. Простые, но довольно эффектные деревья вполне можно создать несколькими строками кода, а все остальные их возможности вы будете привлекать только по мере необходимости. Итак, начнем по порядку.

## *Простые деревья*

Для вывода несложных деревьев на экран не обязательно окунаться в хитросплетения модели деревьев TreeModel или начинать изучение узлов TreeNode, вы можете создавать простые деревья с помощью нескольких вспомогательных конструкторов класса JTree. Правда, возможности созданных этими конструкторами деревьев довольно ограничены, потому что иерархическая структура данных деревьев весьма специфична, и полностью представить ее простыми данными (динамическими или ассоциативными массивами) не так просто, как, к примеру, в случае со списками. Тем не менее, способа создать быстрее настоящее дерево не существует, так что конструкторы эти стоит держать «под рукой».

В качестве источника данных при создании деревьев с помощью простых конструкторов можно использовать одномерные массивы, динамические массивы Vector или ассоциативные массивы Hashtable. Мы уже располагаем опытом работы с библиотекой Swing (если вы прочли все предыдущие главы, конечно), поэтому не трудно догадаться,

что на самом деле незаметно для нас эти данные «оборачиваются» в некую стандартную модель дерева JTree. Особую роль в создании деревьев на основе разнообразных массивов играет специальный тип узла дерева DynamicUtilTreeNode. Он позволяет быстро соорудить список узлов-потомков для какого-либо узла на основе массива (простого или динамического), причем потомки эти добавляются к узлу только при его раскрытии пользователем, так что если узел никогда не раскроется, потомков у него не будет.

Ну а теперь можно рассмотреть пример использования конструкторов класса JTree, позволяющих быстро наполнить дерево данными:

```
// SimpleTrees.java
// Создание самых простых деревьев
import javax.swing.*;
import java.util.*;

public class SimpleTrees extends JFrame {
  public SimpleTrees() {
    super("SimpleTrees");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создание дерева на основе массива
    Object[] data = new Object[] { "Первый", "Второй",
      "Третий", new String[] { "Чей-то потомок",
        "Еще потомок" }
    };
    JTree tree1 = new JTree(data);
    // дерево на основе вектора
    Vector vector = new Vector();
    for (int i=0; i<5; i++) vector.add("Лист № " + i);
    JTree tree2 = new JTree(vector);
    // дерево на основе таблицы
    Hashtable table = new Hashtable();
    table.put("Одна", "пара");
    table.put("Еще одна", "тоже пара");
    JTree tree3 = new JTree(table);
    // можно включить показ корня дерева
    tree3.setRootVisible(true);
    // добавляем деревья в панель содержимого
    JPanel contents = new JPanel();
    contents.add(tree1);
    contents.add(tree2);
    contents.add(tree3);
    setContentPane(contents);
    // выводим окно на экран
    setSize(400, 300);
    setVisible(true);
```

```
  }
  public static void main(String[] args) {
    new SimpleTrees();
  }
}
```

Мы создаем три разных дерева с помощью разных конструкторов класса JTree: первое — на основе обычного массива, второе — на основе динамического массива (вектора) Vector, в третьем для наполнения дерева данными используется таблица (ассоциативный массив) Hashtable. Все просто: заполнив подходящий контейнер данными, вы передаете этот контейнер в конструктор и размещаете на экране созданное дерево. Заметьте, что мы задействовали для первого дерева вложенные массивы объектов — это возможно, поскольку массив в Java также является обычным объектом. В результате мы получим не только листья, относящиеся к корню дерева, но и новые узлы с вложенными в них потомками, вот только, к сожалению, у нас нет возможности задать для этих узлов названия, так что вместо названий будут использованы адреса соответствующего массива в памяти, а это вряд ли восхитит пользователя. Так что вложенные массивы, скорее всего, не пригодятся вам при создании простых деревьев, хотя с их помощью можно быстро создавать ветви самой разной степени сложности (вы также можете вкладывать в массив списки Vector для создания отдельных ветвей).

Аналогичным образом обстоит дело и с динамическими массивами: вы также можете использовать в качестве их элементов такие же массивы, но отображаться на экране они будут как адреса в памяти, так что лучше эту затею оставить (как мы вскоре увидим, проще обратиться к модели дерева). Вектор мы заполняем элементами (будущими листами дерева) динамически, в цикле, а в таблице размещаем две пары «ключ-значение». Запустив программу с примером, вы увидите, что на экран выводятся только ключи, а про значения можно забыть, причем ключи могут появиться на экране в произвольном порядке, а не в том, в котором вы их добавляли в таблицу; все зависит от того, как пары физически хранятся в таблице.

Все простые деревья, создаваемые применяемыми в примере конструкторами класса JTree, не показывают корень дерева, так что дерево на самом деле на дерево не слишком похоже (оно скорее напоминаем своеобразный список). Вы можете вывести корень на экран с помощью свойства rootVisible, что мы и сделали для третьего дерева в примере, но при этом название корня будет стандартным («root»), потому что мы не можем задать его в наших простых конструкторах. Впрочем, вы сможете исправить ситуацию после изучения возможностей стандартной модели дерева, которая неявно используется классом JTree в нашем примере: стандартная модель позволяет изменить названия всех узлов дерева и после его создания.

В итоге все три созданных нами дерева помещаются в окно (с последовательными расположением FlowLayout) и выводятся на экран. Заметьте, что мы не задействовали панель прокрутки именно из-за последовательного расположения, для которого потребовалось бы слишком много места (панели прокрутки с деревьями довольно требовательны к месту, даже если дерево невелико). При использовании для дерева панели прокрутки лучше проектировать интерфейс таким образом, чтобы панель прокрутки занимала в контейнере определенную фиксированную область. Без панелей прокрутки деревья ведут себя не лучшим образом: раскрытие или свертывание любого узла приводит к изменению всего интерфейса, что для пользователя станет сюрпризом, так что в дальнейшем мы всегда будем «оборачивать» деревья в панели прокрутки JScrollPane.

Из всех компонентов библиотеки Swing деревья, пожалуй, единственные не позволяют создать более или менее приличный компонент с помощью простого конструктора (согласитесь, что созданные нами в примере деревья больше напоминают необычные списки). Иногда такие деревья могут вам пригодиться, но все же требуются они редко. Так что не будем медлить и перейдем к рассмотрению модели дерева.

## *Модель дерева TreeModel*

Хотя простые деревья и можно создавать с помощью массивов данных и конструкторов класса JTree, на настоящие деревья они похожи мало. Показать иерархические данные любой сложности дерево сможет, лишь черпая эти данные из специально предназначенной для этого модели TreeModel. Модель дерева TreeModel позволяет описать любую иерархическую структуру данных с единственным корнем. С ее помощью можно указать, сколько потомков имеется у определенного узла дерева, получить потомка узла по его порядковому номеру и наоборот, получить порядковый номер потомка по его значению, легко выяснить, является ли некоторый узел листом дерева. Кроме того, модель TreeModel поддерживает списки слушателей TreeModelListener, которые оповещаются при изменениях в модели. Всего этого оказывается вполне достаточно для вывода компонентом JTree самых пышных деревьев.

В качестве узлов дерева в модели TreeModel применяются самые обычные объекты Object, так что использовать для хранения информации об узлах в своей реализации модели дерева можно все что угодно (любой объект Java унаследован от Object). Давайте попробуем написать несложную модель дерева с обычными строками в качестве узлов, которые и будут выводиться деревом на экран[1]. А для того чтобы упростить поддержку древовидной структуры, хранить эти строки мы будем в динамических списках ArrayList (так проще определять порядковые номера потомков). Вы тут же можете спросить: «А как же стандартная модель, наверное, с ее-то помощью можно создать дерево быстрее и проще?». На самом деле, мы всегда сначала рассматривали стандартную модель в качестве более простого варианта размещения данных в модели, но с деревом все обстоит не так, не как обычно. Интерфейс модели дерева TreeModel позволяет описать иерархическую структуру с использованием любых объектов, и стандартная модель дерева выбирает в качестве таких объектов узлы, описываемые интерфейсом TreeNode. Мы вскоре с ними познакомимся, и тогда уже перейдем к стандартной модели дерева.

Ну а теперь, как мы и хотели, создадим собственную модель дерева «с нуля», реализуя интерфейс TreeModel:

```
// SimpleTreeModel.java
// Создание простой модели для дерева
```

[1] По умолчанию в качестве узла дерева на экран выводится строка, полученная методом узла toString(). Впрочем, как мы вскоре выясним, это поведение настраивается.

```
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;
import java.util.*;
import java.awt.*;

public class SimpleTreeModel extends JFrame {
  public SimpleTreeModel() {
    super("SimpleTreeModel");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // дерево на основе нашей модели
    JTree tree = new JTree(new SimpleModel());
    // добавляем его в окно
    add(new JScrollPane(tree));
    setSize(300, 200);
    setVisible(true);
  }

  // наша модель для дерева
  class SimpleModel implements TreeModel {
    // корень дерева и основные узлы
    private String root = "Кое-что интересное";
    private String
        colors = "Цвета",
        food = "Еда";
    // хранилища данных
    private ArrayList<String> rootList = new ArrayList<String>(),
        colorsList = new ArrayList<String>(),
        foodList = new ArrayList<String>();
    public SimpleModel() {
      // заполняем списки данными
      rootList.add(colors);
      rootList.add(food);
      colorsList.add("Красный");
      colorsList.add("Зеленый");
      foodList.add("Мороженое");
      foodList.add("Бутерброд");
    }

    // возвращает корень дерева
    public Object getRoot() {
```

```
  return root;
}

// сообщает о количестве потомков узла
public int getChildCount(Object parent) {
  if ( parent == root ) return rootList.size();
  else if ( parent == colors )
    return colorsList.size();
  else if ( parent == food ) return foodList.size();
  return 0;
}

// возвращает потомка узла по порядковому номеру
public Object getChild(Object parent, int index) {
  if ( parent == root )
    return rootList.get(index);
  else if ( parent == colors )
    return colorsList.get(index);
  else if ( parent == food )
    return foodList.get(index);
  return null;
}

// позволяет получить порядковый номер потомка
public int getIndexOfChild(
    Object parent, Object child) {
  if ( parent == root )
    return rootList.indexOf(child);
  else if ( parent == colors )
    return colorsList.indexOf(child);
  else if ( parent == food )
    return foodList.indexOf(child);
  return 0;
}

// определяет, какие узлы являются листьями
public boolean isLeaf(Object node) {
  if ( colorsList.contains(node) ||
      foodList.contains(node) ) return true;
  else return false;
}
```

```
    // вызывается при изменении значения некоторого узла
    // для нашей модели не понадобится
    public void valueForPathChanged(
        TreePath path, Object value) {
    }

    // методы для присоединения и удаления слушателей
    // нашей простой модели не потребуются
    public void addTreeModelListener(
        TreeModelListener tml) {
    }
    public void removeTreeModelListener(
        TreeModelListener tml) {
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new SimpleTreeModel(); } });
  }
}
```

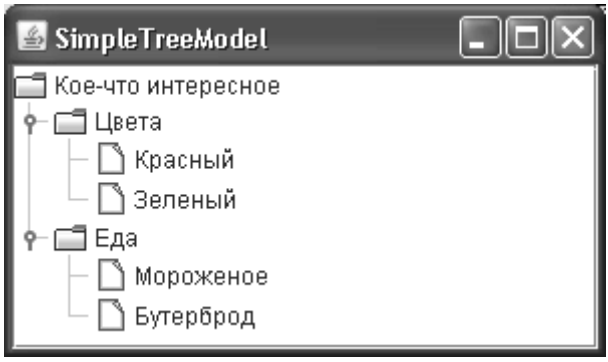
Уже по объему кода в нашем примере можно предположить, что даже такую простую модель для дерева, которую мы хотели создать здесь, на самом деле создать не так уж и просто (по крайней мере, приходится довольно много писать). Так оно и есть: не забывайте, что дерево хранит иерархические данные, и правильная организация таких данных и их гибкое хранение требуют от нас определенных стараний. Сам по себе пример очень прост: мы наследуем от окна JFrame, создаем на основе нашей модели дерево, добавляем его в центр окна (не забывая включить дерево в панель прокрутки JScrollPane, поскольку предыдущий пример показал, что без нее дерево не слишком-то изящно) и выводим окно на экран. Все самое интересное находится в классе модели SimpleModel.

Для того чтобы объект мог стать моделью для дерева, ему нужно реализовывать интерфейс TreeModel. В этом интерфейсе довольно много методов, применять большую часть которых непросто, поэтому в примере каждый метод снабжен подробным комментарием, так что вы не запутаетесь в его назначении. В качестве узлов в нашей модели используются обычные строки String. Заметьте, что данные любого узла, обладающего потомками, хранятся в списках ArrayList, которые позволяют с легкостью определять порядковые номера потомков или получать потомков по их порядковым номерам. Кроме того, с помощью списков мы сможем без труда определить, является ли узел листом (то есть узлом, не обладающим потомками): все листы у нас также находятся в соответствующих списках. Корнем дерева является строка root. Все принадлежащие корню дерева узлы-потомки хранятся в списке rootList (все списки мы наполняем в конструкторе). Корень дерева возвращает метод getRoot() модели. Следующие три метода служат для определения порядковых номеров потомков узла и получения самих потомков. Мы реализовали их с помощью удобных методов списка ArrayList. Заметьте, что даже у такого небольшого дерева, которое мы создаем в примере, для распознания узлов приходится использовать каскадные операторы if-else, что не придает коду изящества и гибкости. Следующий метод (метод isLeaf()) позволяет выяснить, является ли узел листом. Здесь

нам также помогают свойства списка ArrayList (все листы у нас хранятся в списках, так что мы можем легко найти их). Дереву JTree необходимо знать про листы, чтобы оптимизировать производительность модели и прорисовки.

Наконец, последние три метода для нашей простой модели излишни, так что реализовывать мы их не стали. Первый метод вызывается деревом при изменении значения некоторого узла (это может произойти, если ваше дерево допускает редактирование). В примере узлы дерева редактировать нельзя, так что в нашей модели этот метод ни к чему. Тем не менее, стоит помнить о том, что метод valueForPathChanged() вызывается при редактировании любого узла дерева. Если в ваших узлах хранятся нестандартные данные (даже если вы используете стандартную модель), данный метод имеет смысл переопределить и правильно изменять эти данные. Пример, показывающий, как и зачем это делается, мы рассмотрим при подробном обсуждении процесса редактирования узлов дерева.

Два заключительных метода служат для присоединения и отсоединения слушателей, которых мы должны оповещать при изменениях в данных нашей модели. Данные нашей модели во время работы программы не меняются, так что списки слушателей нам тоже не пригодятся (впрочем, мы еще вспомним об этой задаче чуть позже). Обратите внимание, что создатели модели дерева не предоставили нам абстрактный класс с названием вроде AbstractTreeModel, в который была бы встроена поддержка списков слушателей (аналоги подобных классов есть практически для всех компонентов Swing с моделями). Причина здесь все та же — дерево очень гибко, и создать класс для общих нужд не получается. Запустив программу с примером, вы увидите, как данные модели превращаются в дерево.




Согласитесь, что для таких простых данных, какие мы использовали в модели, пришлось выполнить чересчур много работы. У нас в дереве выводится всего один корень, два узла с двумя потомками, а написать пришлось порядочно. Без сомнения, обычные, неструктурированные данные мало подходят для описания древовидных структур. Очевидно, что в дополнение к самим данным необходимо добавлять ссылки и на их потомков, и так до окончательных листьев, иначе описание дерева превращается в пытку, а код в «спагетти».

Проще было бы раз и навсегда написать класс, позволяющий создавать любые сочетания узлов и листьев, который при необходимости сам сможет справиться с запросами модели и управлять всеми списками с данными, вставлять новые узлы и удалять уже имеющиеся. С одной стороны, можно доработать нашу простую модель, но использовать в качестве узлов простые строки не слишком разумно: чаще всего в деревьях хранятся куда более сложные данные. С другой стороны, нам стоит вспомнить о стандартной модели дерева DefaultTreeModel. Она позволяет делать с узлами и листьями все то, о чем мы говорили, и использует для хранения информации об узлах специальные объекты TreeNode, прекрасно для этого подходящие и позволяющие хранить любую ин-

формацию. Но забывать о реализации модели дерева «с нуля», как бы запутано это ни было, не стоит: если в вашем приложении данные хранятся в специальных древовидных структурах, будет проще написать собственную модель дерева, а не переносить данные в стандартную модель. Это позволит свести работу к минимуму и максимально оптимизировать ее, например, сообщать о потомках узла динамически, только когда этого потребует дерево. Ну а теперь познакомимся с узлами TreeNode.

## Узлы TreeNode

Интерфейс TreeNode из пакета javax.swing.tree описывает характеристики единственного узла (в модели описание всех узлов «раскидано» сразу по нескольким методам, что и вносит в нее дополнительную сложность). В характеристики узла входят перечисление (Enumeration) его потомков и получение их по порядковому номеру (потомки возвращаются также в виде объектов TreeNode), а также информация о предке узла и о том, является ли данный узел листом. Таким образом, если в модели дерева TreeModel вы описываете все узлы сразу, то, реализуя интерфейс TreeNode (и в дальнейшем используя его в стандартной модели дерева), вы получаете возможность говорить только об одном узле. Без сомнения, это проще.

Впрочем, интерфейс TreeNode используется не так уж и часто. У него есть гораздо более популярный потомок — унаследованный от него интерфейс MutableTreeNode. Последний определяет еще несколько методов, позволяющих динамически добавлять к узлу новых потомков, удалять их (по номеру или по значению), менять предков узла или удалять данный узел из списка потомков предка. Кроме того, в этом интерфейсе есть метод setUserObject(), который позволяет быстро сменить данные, хранящиеся в узле. Данные могут иметь любой тип, так что в ваших узлах может храниться все что угодно, даже очень сложные данные (например, содержимое файлов, соответствующих узлу дерева).

Для использования возможностей узлов TreeNode (и впоследствии стандартной модели дерева) вам не придется скрупулезно реализовывать все методы описанных выше интерфейсов. Библиотека предоставляет нам стандартную реализацию интерфейса MutableTreeNode — класс с названием DefaultMutableTreeNode, который разрешает делать с узлом все, что только можно вообразить. Более того, он плотно «набит» различными полезными методами, которые не раз пригодятся при работе с деревом: эти методы позволят без труда определить, принадлежит ли некоторый узел вашему дереву, является ли он потомком или предком для другого узла, определить общего предка нескольких узлов, получить путь до корня дерева, работать с листьями, принадлежащими узлу, и делать еще многое с помощью нескольких простых вызовов. В дополнение к этому класс DefaultMutableTreeNode позволяет получать различные перечисления узлов и их потомков, в том числе и перечисления всего дерева по известным дисциплинам «в глубину» (depth first) и «в ширину» (breadth first). Благодаря всему этому класс DefaultMutableTreeNode может быть полезен и просто как универсальное средство описания узлов любой иерархической структуры данных (которое к тому же элементарно вывести на экран, как мы вскоре увидим).

Создать древовидную структуру с помощью класса DefaultMutableTreeNode очень просто. Для каждого узла своего дерева вы создаете отдельный объект этого класса, указывая в конструкторе данные, которые он будет хранить. Именно эти данные (а, точнее, результат вызова метода toString(), переданного вами в узел в качестве данных объекта) будут использоваться деревом для вывода узла на экран. Далее с помощью методов add() (для добавления узла к концу списка потомков другого узла) или insert() (для вставки узла на произвольную позицию) вы организуете иерархические отношения узлов. Начинается все с корня дерева, к которому вы добавляете первых потомков. К этим потомкам в свою очередь добавляются свои потомки, и так продолжается до построения всего дерева.

«Чудесно, но как отобразить эту структуру?» — спросите вы. Мы уже упоминали, что с узлами TreeNode работает стандартная модель дерева DefaultTreeModel. Эта мо-

дель может использовать в качестве узлов любые объекты, реализующие интерфейс TreeNode, но нетрудно догадаться, что чаще всего ей передаются экземпляры класса DefaultMutableTreeNode. Стандартная модель и заботится о том, чтобы организованная нами структура вписывалась в рамки интерфейса TreeModel и выводилась деревом на экран.

## Стандартная модель DefaultTreeModel

Итак, стандартная модель дерева DefaultTreeModel хранит иерархические структуры, образованные узлами TreeNode. Для создания стандартной модели достаточно передать в ее конструктор узел TreeNode, являющийся корнем дерева, а обо всем остальном модель позаботится сама, так что дерево, начинающееся с этого узла, будет правильно выведено на экран. Дополнительных возможностей в стандартной модели практически нет, и это неудивительно: все, что вам может понадобиться при работе с деревом, уже есть в узлах DefaultMutableTreeNode (ну а если вы реализуете интерфейс TreeNode самостоятельно, то, скорее всего, на это есть веская причина, и нужные вам возможности у вас уже есть). Рассмотрим пример использования стандартной модели, который заодно станет первым примером применения узлов DefaultMutableTreeNode:

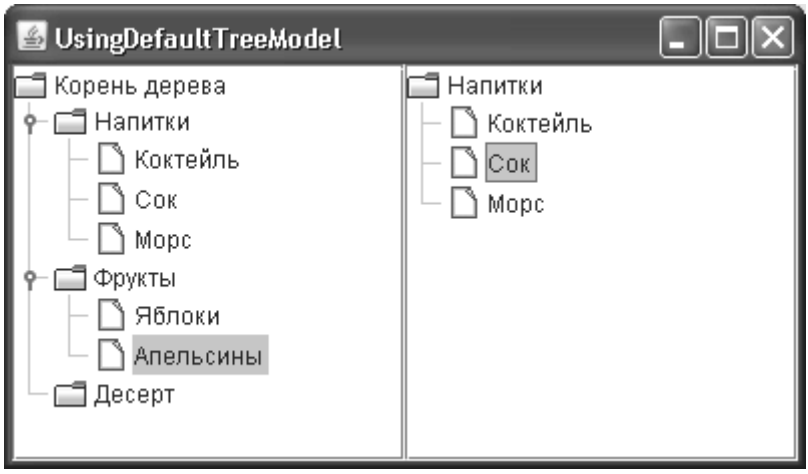
```
// UsingDefaultTreeModel.java
// Использование стандартной модели дерева и
// узлов DefaultMutableTreeNode
import javax.swing.*;
import javax.swing.tree.*;
import java.awt.*;

public class UsingDefaultTreeModel extends JFrame {
  // для удобства листья будем хранить в массивах
  private String[] drinks =  { "Коктейль", "Сок", "Морс" };
  private String[] fruits =  { "Яблоки", "Апельсины" };
  public UsingDefaultTreeModel() {
    super("UsingDefaultTreeModel");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем нашу древовидную структуру
    DefaultMutableTreeNode root =
        new DefaultMutableTreeNode("Корень дерева");
    // основные ветви
    DefaultMutableTreeNode drink =
        new DefaultMutableTreeNode("Напитки");
    DefaultMutableTreeNode fruit =
        new DefaultMutableTreeNode("Фрукты");
    // добавляем ветви
    root.add(drink);
    root.add(fruit);
    // специальный конструктор
    root.add(new DefaultMutableTreeNode("Десерт", true));
```

```
    // добавляем листья
    for (String _drink : drinks) {
      drink.add(
          new DefaultMutableTreeNode(_drink, false));
    }
    for (String _fruit : fruits) {
      fruit.add(
          new DefaultMutableTreeNode(_fruit, false));
    }
    // создаем стандартную модель и дерево
    DefaultTreeModel dtm1 =
        new DefaultTreeModel(root, true);
    JTree tree1 = new JTree(dtm1);
    // модель можно создать, начиная с любого узла
    DefaultTreeModel dtm2 = new DefaultTreeModel(drink);
    JTree tree2 = new JTree(dtm2);
    // добавляем деревья в окно и показываем его
    setLayout(new GridLayout(1, 2));
    add(new JScrollPane(tree1));
    add(new JScrollPane(tree2));
    setSize(400, 300);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new UsingDefaultTreeModel(); } });
  }
}
```

В примере мы создаем древовидную структуру на основе стандартных узлов DefaultMutableTreeNode так, как мы и обсуждали: сначала создается корень нашего дерева (объект root), к которому присоединяются его потомки (узлы с информацией о доступных нам напитках и фруктах, а также десерте). Далее мы присоединяем к этим узлам (первого уровня, корень является узлом нулевого уровня, кстати, уровень любого узла позволяет узнать полезный метод getLevel() класса DefaultMutableTreeNode) их потомков, которые для удобства хранятся в массивах и добавляются к дереву в цикле. Заметьте, что для узла с информацией о десерте мы использовали особенный конструктор: в качестве параметров он принимает не только данные, которые будет хранить узел, но еще и булево значение. Последнее управляет тем, как будет вести себя метод getAllowsChildren(), определенный в интерфейсе TreeNode. Если он возвращает true, это означает, что у данного узла могут быть потомки, даже если их в данный момент нет (такое поведение характерно, например, для пустого каталога без файлов, который, тем не менее, остается каталогом и в будущем может содержать приличное количество файлов). В стандартной модели DefaultTreeModel поддерживаются два способа, позволяющие различать листья дерева (узлы без потомков). Согласно первому способу считается, что узел является листом, если у него в дан-

ный момент нет потомков, а согласно второму узел не является листом, если его метод getAllowsChildren() возвращает true (то есть потомки могут появиться). Для включения второго способа нужно использовать специальный конструктор класса DefaultTreeModel, передав ему значение true в качестве второго параметра (что мы и сделали для первой модели в примере) или вызвав метод модели setAsksAllowsChildren(true). В этом случае узел «с десертом» будет выглядеть как «папка», так оно и должно быть, поскольку обычно предоставляется несколько видов десерта. Подобное поведение может пригодиться в программах довольно часто: в деревьях встречаются «ненастоящие» листья, в которых потомков нет, но они могут появиться, и их нужно отличать от листьев «настоящих».

Вторая модель дерева создается, начиная не с корня, а с узла drink, так что вы свободно можете показывать свое дерево по частям, если в этом возникнет необходимость. Во второй модели у нас нет пустых «папок», поэтому мы не используем второй параметр конструктора, и листами считаем все узлы, у которых нет потомков. Для того чтобы дерево начало отображать данные созданных нами стандартных моделей, достаточно передать их в соответствующий конструктор класса JTree (или вызвать метод setModel(), если вы предпочитаете другие конструкторы). После этого остается разместить деревья в окне (мы разделяем окно на две равные части с помощью менеджера табличного расположения GridLayout и размещаем в них наши деревья, предварительно «обернутые» в панели прокрутки JScrollPane) и вывести последнее на экран.




После того как вы насладитесь напитками и фруктами, представленными с помощью стандартной модели и дерева JTree, может возникнуть вопрос: а как у стандартной модели обстоит дело с динамическим изменением узлов и потомков? И здесь стандартная модель на высоте. У вас есть целый арсенал методов для динамической манипуляции деревом, начиная от полной смены корня (методом setRoot()), вставки (специальным методом insertNodeInto()) узла типа MutableTreeNode в узел такого же типа на произвольную позицию с автоматическим оповещением слушателей и заканчивая простым изменением созданных вами узлов DefaultMutableTreeNode, которые, как мы прекрасно знаем, очень легко изменять и обновлять.

Если вы меняете узлы, не касаясь модели, ей надо об этом сообщить, чтобы она, в свою очередь, сообщила дереву о необходимости перерисовки. Для этого можно воспользоваться методом reload() модели, сообщающим слушателям об изменении всей модели (впрочем, есть перегруженная версия этого метода, сообщающего об изменениях в определенном узле дерева), или обратиться к методам вида nodesXXX(), которых в стандартной модели несколько. Эти методы (например, метод nodesChanged()) позволяют со-

общать об изменениях в потомках некоторого узла и экономят время обновления дерева, сообщая только о действительно измененных узлах. «Точечное» обновление дерева очень важно для высокой производительности перерисовки и мгновенной реакции на действия пользователей, которые всегда в восторге от быстрых приложений.

**Таблица 15.1.** Методы стандартной модели для оповещения об обновлении данных

| Методы | Описание |
|---|---|
| reload() | Говорит модели о том, что полностью поменялся некоторый узел и его необходимо перезагрузить и перерисовать. Метод без параметров перезагружает все дерево начиная с корня. Однако если у вас не меняется структура узлов, лучше применять методы, описанные ниже, потому что после перезагрузки узла он будет полностью закрыт, так что пользователь потеряет из виду все узлы, которые он открыл вручную |
| nodeStructureChanged() | Полный аналог предыдущего метода с другим названием |
| nodesWereInserted(), nodesWereRemoved() | Говорит модели о том, что для указанного в качестве параметра узла-предка были добавлены или удалены узлы-потомки. Это более эффективные сообщения, чем два, описанные выше |
| nodesChanged() | Метод сообщает модели (а та дереву), что поменялось значение указанного узла, так что его необходимо перерисовать. Структура дерева при этом не меняется, что делает данный вызов самым эффективным. Именно его следует предпочитать при желании обновить внешний вид узла после обновления его данных. |

Для того чтобы убедиться в том, что «точечное» обновление модели позволяет получить высокую производительность даже если приходится постоянно обновлять изображение дерева на экране, напишем небольшой пример. В нем мы создадим очень большое, нагруженное узлами дерево, и заставим все узлы постоянно обновляться. Как правило, именно так ситуация и обстоит в приложениях, работающих с большими объемами данных. Посмотрим, что у нас получается:

```
// TreeModelUpdates.java
// Пример эффективного обновления большого
// дерева с переменными данными
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeModel;
import javax.swing.tree.TreePath;
import javax.swing.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.*;

public class TreeModelUpdates extends JFrame {
  // дерево
  private JTree tree;
  // стандартная модель дерева
  private DefaultTreeModel model;
```

```
public TreeModelUpdates() {
  super("TreeModelUpdates");
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  // создаем дерево на основе модели
  tree = new JTree(model = createTreeModel());
  add(new JScrollPane(tree));
  setSize(400, 300);
  setVisible(true);
}
// создание несложной, но большой модели дерева
private DefaultTreeModel createTreeModel() {
  // корень нашего дерева
  DefaultMutableTreeNode root =
      new DefaultMutableTreeNode(
          "Большоооое Дерево");
  // присоединяем листья
  for (int i=0; i<100; i++) {
    DefaultMutableTreeNode node = new ChangeableColorNode();
    root.add(node);
    for (int k=0; k<10; k++) {
      node.add(new ChangeableColorNode());
    }
  }
  // создаем стандартную модель
  return new DefaultTreeModel(root);
}
// узел дерева, динамически обновляющий свое содержимое
class ChangeableColorNode
    extends DefaultMutableTreeNode implements ActionListener {
  // случайная задержка обновления
  private int delay = (int) (Math.random()*3000 + 500);
  // цвета текста, хранимого в узле
  private int r,g,b;
  public ChangeableColorNode() {
    // генерируем цвета и запускаем таймер обновления
    generateColors();
    new Timer(delay, this).start();
  }
  // этот метод вызывается таймером
  public void actionPerformed(ActionEvent e) {
    // обновляем цвета и текст
```

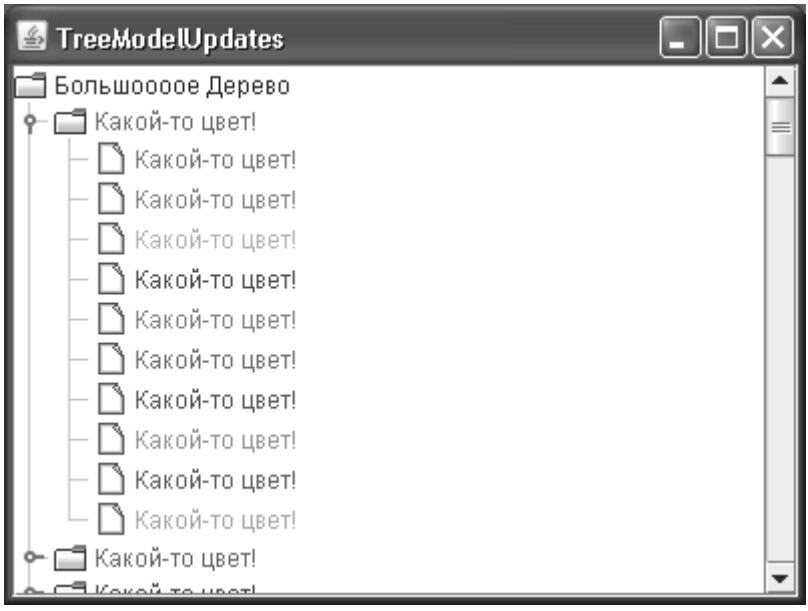
```
        generateColors();
        // обновляем узел если он развернут
        TreePath path = new TreePath(
            model.getPathToRoot(this));
        if ( tree.isVisible(path) ) {
          model.nodeChanged(this);
        }
      }
      // метод генерирует случайные цвета и обновляет текст
      public void generateColors() {
        r = (int) (Math.random()*255);
        g = (int) (Math.random()*255);
        b = (int) (Math.random()*255);
        setUserObject("<html><font color=rgb("
            +r+","+g+","+b+")>Какой-то цвет!");
      }
    }
    public static void main(String[] args) {
      SwingUtilities.invokeLater(
          new Runnable() {
            public void run() { new TreeModelUpdates(); } });
    }
}
```

В примере мы унаследуем от окна JFrame, в центр которого будет добавлена панель прокрутки с нашим деревом. Модель для дерева будет создана простым методом createTreeModel(), на основе уже хорошо нам знакомой стандартной модели дерева. Главным ее отличительным качеством является размер — к корню дерева присоединяется ни много ни мало 100 дочерних узлов, у которых, в свою очередь, будет по 10 узлов-потомков. В качестве узлов применяется класс ChangeableColorNode, унаследованный от класса стандартного узла дерева DefaultMutableTreeNode.

Самое интересное происходим именно в классе этих специальных узлов. Прежде всего, строка, которая хранится в узле, представляет собой текст HTML определенного цвета, заданного с помощью трех стандартных цветов RGB. Цвета эти генерируются случайным образом в методе generateColors(), после чего текст (а точнее, его цвет) обновляется. Чтобы симулировать постоянное обновление узла, мы используем таймер, представленный классом Timer из пакета javax.swing. Мы можем беспрепятственно создавать множество объектов этого класса, так как таймер все равно работает из одного потока и не будет отнимать слишком много ресурсов. Для того чтобы создать таймер, необходимо указать ему период срабатывания в миллисекундах (мы генерируем его случайным образом, но делаем не меньше 500), а также слушателя ActionListener, которому таймер будет сообщать о срабатываниях. Слушателя реализует сам же класс нашего узла. Интересно, что таймер из пакета javax.swing, в отличии от стандартного таймера JDK из пакета java.util, всегда вызывает слушателя из потока рассылки событий, что очень удобно при работе со Swing.

Самое интересное происходит при срабатывании таймера и вызове метода actionPerformed(). Мы обновляем цвет текста узла, и дальше нам необходимо сообщить де-

реву о том, что узел поменялся и его нужно перерисовать. Здесь нам помогает и стандартная модель, и само дерево. С помощью стандартной модели и ее метода getPathToRoot() мы получаем путь до текущего узла ( в самом узле неизвестна вся иерархия его предков) и выясняем с помощью метода дерева isVisible(), виден ли этот путь в данный момент на экране. Если он виден, то мы используем метод стандартной модели nodeChanged(), чтобы сообщить дереву о перемене в узле. Конечно, дерево и само проверяет, видел ли узел на экране, прежде чем его рисовать, однако перед этим оно заново рассчитывает размеры узла и свои собственные. Проверив самостоятельно, виден ли узел на экране, мы делаем его обновление еще быстрее.




Запустив пример, вы сможете развернуть любые узлы и прокручивать дерево, но не заметите никаких проблем с производительностью. Точное обновление данных на экране всегда гарантирует нам прекрасную скорость приложения, написанного с применением Swing.

## Оповещение в нестандартных моделях

Если ваши данные хранятся в иерархии и модель дерева TreeModel прекрасно им подходит, без сомнения стоит написать свою собственную модель на их основе. Однако останется вопрос оповещения дерева об изменениях в вашей модели, если последняя динамична и это подразумевает. Для этого в нестандартной модели придется поддерживать список слушателей и самостоятельно создавать и «рассылать» события TreeModelEvent, так как никакого базового класса с поддержкой слушателей и методами fireXXX() у нас нет (не считая стандартной модели, но она работает с узлами TreeNode). Подобный процесс мы подробно рассматривали в главе 3, посвященной событиям в Swing.

Однако, это довольно низкоуровневая работа, особенно это касается списков слушателей и создания событий со всеми нужными параметрами, которых у деревьев немало. К счастью, в библиотеке расширений SwingX имеется специальный класс с названием TreeModelSupport, который возьмет на себя заботу о списке слушателей, и предоставит

вам удобные методы fireXXX(), помогающие оповещать слушателей о переменах в ваших данных. Чтобы сократить повторяющийся код, всегда применяйте в своих нестандартных моделях этот класс, если включение библиотеки SwingX в ваше приложение не противоречит его архитектуре и не слишком накладно.

### Модели деревьев — краткое резюме

В общем-то, описание модели дерева TreeModel на этом можно закончить. Мы узнали, что прямое использование модели для вывода на экран обычных деревьев не слишком оправдано — все-таки машинное представление иерархических данных довольно специфично. Впрочем, если ваше приложение работает с особыми древовидными структурами данных, например, хранит схему распределения в сети серверов имен, модель дерева может подойти как нельзя лучше и легко упорядочит ваши данные. Но чаще всего для создания деревьев применяются стандартная модель DefaultTreeModel и узлы DefaultMutableTreeNode, которые позволяют без трудностей манипулировать данными и быстро подготавливать их к выводу на экран. Как правило, данные дерева не записываются непосредственно в код (это неудобно и делает программу менее гибкой), а хранятся в файлах ресурсов в подходящем формате (например, в изначально являющимся иерархическим формате XML). При необходимости эти данные из файлов ресурсов считываются и включаются в узлы DefaultMutableTreeNode. Это можно сделать компактно, после чего полученные узлы передаются стандартной модели. И конечно у вас остаются все преимущества модели: данные отделяются от пользовательского интерфейса, а модель может быть разделена между несколькими видами.

## *Выделение*

После появления вашего дерева на экране требуется определить, какие именно узлы или листья выбирает в нем пользователь, поскольку чаще всего именно с выбранной пользователем информацией приложению приходится работать. Информацию о выделенных узлах дерева хранит специальная модель выделения, описанная интерфейсом TreeSelectionModel (из пакета javax.swing.tree). Для выделения элементов дерева имеется не слишком много способов, здесь все довольно просто: выделение можно производить, во-первых, по одному узлу дерева, во-вторых, смежными интервалами по несколько узлов, в-третьих, произвольно, составляя выделение из любого набора узлов, входящих в дерево. Сложность состоит в другом: в дереве с его запутанной иерархической структурой узлы нельзя определить простым числовым индексом. Для определения местоположения узла в дереве используются два средства. Во-первых, это пути TreePath, состоящие из набора узлов, начиная с корня, переходя по которым можно достигнуть требуемого узла. Во-вторых, это номера строк в дереве, которые не слишком удобны, так как для одного и того же выделения номера могут быть разными (при раскрытии или свертывании узлов появляются новые строки и прежние номера строк становятся недействительными).

Таким образом, модель выделения TreeSelectionModel не слишком сложна. Как оказывается, гораздо сложнее потом работать с выделенными путями в дереве, по которым необходимо определять, какую информацию выбрал пользователь. Основные две группы методов модели выделения таковы: методы вида addSelectionPath(s) добавляют к уже имеющемуся выделению новые узлы, заданные путями TreePath, а методы вида setSelectionPath(s) заменяют текущее выделение новым, также заданным путями TreePath. Именно в этих методах и происходит вся работа. К примеру, если модель выделения работает в режиме выделения только одного узла за раз, при вызове метода addSelectionPath() она смотрит, есть ли в модели уже выделенные узлы, и, если таковые имеются, вместо добавления к ним нового узла выделение с узлов снимается, и вместо них выделенным становится новый узел. Если бы в модели поддерживался режим произвольного выделения, в данном методе новые узлы просто добавлялись бы к старым. То же относится

и к методам setSelectionPath(s). Последний выделенный в дереве путь отмечается моделью выделения особым образом и возвращается методом getLeadSelectionPath(). На работу дерева или режима выделения это не оказывает никакого влияния, хранить последний выделенный элемент требуют некоторые внешние виды Swing, которые для удобства пользователя оформляют его особым образом[2].

Модель выделения дерева TreeSelectionModel поддерживает списки слушателей TreeSelectionListener, которые оповещаются при изменении выделенных элементов в дереве. Присоединив к используемой деревом модели выделения своего слушателя, вы будете оперативно узнавать о каждом изменении в выделенных элементах дерева и при необходимости сразу же на это реагировать.

Реализовывать модель выделения «с нуля» вам вряд ли понадобится, гораздо проще задействовать стандартную модель выделения DefaultTreeSelectionModel из пакета javax.swing.tree. Она поддерживает все три возможных режима выделения узлов дерева, списки слушателей, быстрый сброс выделенных в данный момент элементов и дает вам возможность в любой момент времени выяснить все о выделенных элементах. Именно стандартная модель (в режиме выделения произвольного количества узлов) используется по умолчанию деревом JTree. Рассмотрим небольшой пример и увидим, на что она способна:

```
// TreeSelectionModes.java
// Использование стандартной модели выделения и
// всех поддерживаемых ею режимов
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;
import java.awt.*;

public class TreeSelectionModes extends JFrame {
  public TreeSelectionModes() {
    super("TreeSelectionModes");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создадим модель нашего дерева
    TreeModel model = createTreeModel();
    // дерево с одиночным режимом выделения
    JTree tree1 = new JTree(model);
    tree1.getSelectionModel().setSelectionMode(
        TreeSelectionModel.SINGLE_TREE_SELECTION);
    // дерево с выделением непрерывными интервалами
    JTree tree2 = new JTree(model);
    tree2.getSelectionModel().setSelectionMode(
        TreeSelectionModel.CONTIGUOUS_TREE_SELECTION);
    // модель выделения можно хранить и отдельно
    TreeSelectionModel selModel =
```

[2] Если пользователь на мгновение отвлечется от увлекательного процесса выделения узлов дерева, он затем без труда сможет определить, на каком месте остановился (благодаря специально оформленному последнему выделенному узлу).

```
        new DefaultTreeSelectionModel();
    selModel.setSelectionMode(
        TreeSelectionModel.DISCONTIGUOUS_TREE_SELECTION);
    JTree tree3 = new JTree(model);
    tree3.setSelectionModel(selModel);
    // будем следить за выделением в последнем дереве
    tree3.addTreeSelectionListener(new SelectionL());
    // размещаем деревья в панели
    JPanel contents = new JPanel(new GridLayout(1, 3));
    contents.add(new JScrollPane(tree1));
    contents.add(new JScrollPane(tree2));
    contents.add(new JScrollPane(tree3));
    add(contents);
    // добавляем текстовое поле
    add(new JScrollPane(log), "South");
    // выводим окно на экран
    setSize(500, 300);
    setVisible(true);
  }
  // текстовое поле для информации
  private JTextArea log = new JTextArea(5, 20);
  // листья дерева храним в массивах
  private String[] langs = { "Java", "Scala", "Ruby" };
  private String[] ides =
      { "IDEA", "Eclipse", "NetBeans" };
  // создание несложной модели дерева
  private TreeModel createTreeModel() {
    // корень нашего дерева
    DefaultMutableTreeNode root =
        new DefaultMutableTreeNode("Создание кода");
    // основные ветви
    DefaultMutableTreeNode lang =
        new DefaultMutableTreeNode("Языки");
    DefaultMutableTreeNode ide =
        new DefaultMutableTreeNode("Среды");
    root.add(lang);
    root.add(ide);
    // присоединяем листья
    for (int i=0; i<langs.length; i++) {
      lang.add(new DefaultMutableTreeNode(langs[i]));
      ide.add(new DefaultMutableTreeNode(ides[i]));
```

```
    }
    // создаем стандартную модель
    return new DefaultTreeModel(root);
  }
  // этот слушатель следит за изменением выделения
  class SelectionL implements TreeSelectionListener {
    public void valueChanged(TreeSelectionEvent e)  {
      // получаем источник события — дерево
      JTree tree = (JTree)e.getSource();
      // добавленные/удаленные пути
      TreePath[] paths = e.getPaths();
      log.append("Изменено путей: " +
          paths.length + "\n");
      // выделенные элементы и их номера строк
      TreePath[] selected = tree.getSelectionPaths();
      int[] rows = tree.getSelectionRows();
      // последние элементы в пути
      for (int i=0; i<selected.length; i++) {
        log.append("Выделен: " +
            selected[i].getLastPathComponent() +
            " на строке: " + rows[i] +  "\n");
      }
      // полная информация о пути в дереве
      if ( selected.length > 0 ) {
        TreePath path = selected[0];
        Object[] nodes = path.getPath();
        for (int i=0; i<nodes.length; i++) {
          // путь состоит из узлов
          DefaultMutableTreeNode node =
              (DefaultMutableTreeNode)nodes[i];
          log.append("Отрезок пути " + i + " : " +
              node.getUserObject() + " ");
        }
      }
      log.append("\n");
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new TreeSelectionModes(); } });
```

```
    }
}
```

Итак, в примере мы создаем окно, в котором разместится три дерева, каждое со стандартной моделью выделения, но с разным режимом работы этой модели. Для дерева JTree необходимо предоставить данные для отображения, чтобы было что выделять, так что в методе createTreeModel() мы создаем простую модель с корнем и двумя потомками, к которым присоединяются листья, хранящиеся для удобства в массивах. Здесь все просто, фактически повторяется пример для стандартной модели дерева, рассмотренный в предыдущем разделе. Полученную модель мы будем разделять между нашими видами — тремя деревьями, так что в них мы увидим одинаковые данные. В этом примере гораздо интереснее не данные, а то, как эти данные можно выделять. Создав на основе модели первое дерево, мы меняем режим выделения его узлов. Для этого нужно получить используемую деревом модель выделения (методом getSelectionModel()) и сменить режим выделения (методом setSelectionMode()). Три возможных режима выделения узлов дерева описаны константами интерфейса TreeSelectionModel, для первого дерева мы выбираем режим SINGLE_TREE_SELECTION, позволяющий выделить только один узел за один раз.

Точно так же мы поступаем и со вторым деревом, устанавливая для него режим CONTIGUOUS_TREE_SELECTION. Этот режим допускает выделение произвольного количества узлов дерева, но только с одним условием: выделяемые узлы дерева должны следовать один за другим без перерывов. Как только пользователь попытается выделить узел, не входящий в непрерывный интервал выделения, все выделение сбросится, и выбранным останется только один узел.

Никто не запрещает нам создавать стандартную модель выделения напрямую и передавать ее затем в метод setSelectionModel() класса JTree. Так мы и делаем для третьего дерева в примере, устанавливая для него режим выделения DISCONTIGUOUS_TREE_SELECTION. Кстати, мы могли этого и не делать, такой режим выделения является для стандартной модели режимом по умолчанию. Режим этот очень прост: пользователь может выделять любое количество узлов в любом порядке, как ему заблагорассудиться. К третьему дереву мы присоединяем слушателя событий от модели выделения TreeSelectionListener, который позволит нам быть постоянно в курсе того, что выделяет в дереве пользователь.

Но о слушателе событий чуть позже, а пока нам остается только вывести наши деревья на экран и оценить различные режимы выделения. Для размещения трех деревьев используется панель с табличным расположением GridLayout, разделенная на три ячейки одинакового размера. В этих ячейках размещаются панели прокрутки с деревьями. Кроме того, на юг окна помещается многострочное текстовое поле JTextArea с названием log, в него мы будем записывать сообщения о событиях, поступающих слушателю событий от модели выделения. Запустив программу с примером и пробуя выделять различные узлы деревьев (дополнительные режимы выделения доступны при нажатых управляющих клавишах Ctrl и Shift), вы легко увидите разницу в режимах выделения и сможете выбрать для своего дерева тот режим, который вам больше всего подходит.

Ну а теперь о том, как с помощью слушателей TreeSelectionListener следить за всеми изменениями в выделенных элементах дерева JTree. Присоединить слушателя можно как к модели выделения, так и к самому дереву JTree. Это почти одно и то же, разница лишь в том, что именно будет возвращать метод getSource() в ответ на события. В первом случае он вернет нам ссылку на модель выделения, в которой произошло изменение, а во втором — ссылку на само дерево. В классе JTree дублируются все методы модели выделения (они просто делегируют ей свои вызовы), кроме того, в классе JTree есть несколько весьма полезных методов, способных упростить работу с выделенными элементами дерева, так что, как правило, лучше присоединять слушателя непосредственно к дереву. Мы так и поступаем. Наш слушатель описан во внутреннем классе SelectionL. Каждый раз при изменении в выделенных узлах дерева вызывается метод слушателя valueChanged(). Параметр этого метода — объект TreeSelectionEvent. Он позволяет получить, во-первых, как и в случае с любым событием, источник события (у нас это дерево JTree), во-вторых, список путей узлов, которые были *затронуты* при смене выделения (методом getPaths()). Обратите внимание, что этот метод *не позволяет* получить выделенные в данный момент узлы дерева, он всего лишь содержит список изменившихся в последний раз узлов. Например, если у вас в дереве уже выделено с десяток узлов, и пользователь решает прибавить к ним еще один, данный метод вернет только один путь для нового выделенного узла, так как остальные при этом не изменятся. Вы сможете убедиться в этом, запустив программу с примером и при выделении новых узлов отслеживая (в текстовом поле), сколько путей возвращает данный метод.

Гораздо удобнее узнавать обо всех выделенных в данный момент узлах с помощью метода getSelectionPaths(), который есть и в модели выделения, и в дереве (метод дерева просто обращается к модели выделения). Данный метод возвращает массив объектов-путей TreePath ко всем выделенным в данный момент узлам дерева. Как вы помните, мы уже отмечали, что путь TreePath — это список узлов, начиная с корня, по которым надо пройти для достижения определенного узла. В случае стандартной модели (как в нашем примере) путь состоит из списка объектов-узлов TreeNode. Как правило, весь путь к выделенному узлу требуется знать редко, чаще всего достаточно информации о самом выделенном узле. Для получения этой информации служит метод getLastPathComponent(), возвращающий последний узел в пути (это собственно и есть выбранный узел). Кроме того, вы можете работать со строками, а не с путями. Поскольку дерево на экране отображается последовательно (корень, потомки в порядке добавления, их потомки также в порядке добавления), можно подсчитать, на какой строке находится тот или иной узел[3]. Получить номера строк выделенных узлов позволяет метод getSelectionRows(). Однако у строк есть недостаток: при сворачивании или развертывании узлов дерева их номера меняются, даже если не меняются выделенные элементы, и ваша информация устаревает. Все же удобнее пользоваться

[3] Переводом узлов дерева в строки на экране занимается специальный объект, ассоциированный с моделью выделения и реализующий интерфейс RowMapper. Он работает за кулисами деревьев, так что подробно разбираться в нем необязательно.

путями TreePath, но номера строк могут быть достойной альтернативой. Вы также сможете увидеть их в текстовом поле log.

Ну и напоследок мы полностью исследуем путь TreePath первого выделенного элемента. Обратите внимание, что прежде чем получать пути, необходимо удостовериться, что массив, возвращаемый методом getSelectionPaths(), их вообще содержит. В то время как сам массив никогда не станет пустой ссылкой null, элементов в нем может и не быть, что необходимо учитывать (такое событие возникает, когда пользователь сбрасывает выделение в дереве). Анализировать путь проще всего с помощью метода getPath(), возвращающего массив узлов, начиная с корня, ведущих к нужному нам узлу. Так как мы используем стандартную модель и данные узлов у нас хранятся в объектах DefaultMutableTreeNode, мы можем смело преобразовывать узлы, из которых состоит путь, к данному типу. Данные всех узлов, составляющих путь, мы получаем методом getUserObject() и выводим в текстовое поле, так что вы сможете своими глазами убедиться в том, что путь начинается от корня и постепенно приводит нас к выделенному узлу.

Впечатляющая часть методов класса JTree посвящена именно работе с выделенными элементами, так что пользоваться можно не только интерфейсом модели выделения и методом getSelectionPaths(). Составим небольшую таблицу наиболее полезных методов класса JTree для работы с выделением, в случае необходимости вы сможете обращаться к ней как к простому справочнику (табл. 15.2).

**Таблица 15.2.** Методы класса JTree для работы с выделением

| Методы | Описание |
|---|---|
| add(set)SelectionPath(s), add(set)SelectionRow(s) | На самом деле данные методы принадлежат модели выделения TreeSelectionModel дерева и позволяют программно прибавлять к выделенным узлам новые. Но если вам захочется забыть о моделях, Swing позволит это сделать, как будто их вовсе нет: все методы модели выделения есть и в классе самого дерева JTree |
| clearSelection() | Сбрасывает все выделенные в данный момент узлы дерева, так что выделенного в нем ничего не остается |
| getMinSelectionRow(), getMaxSelectionRow() | Удобные методы при работе с узлами дерева по номерам строк: первый метод возвращает номер первой выделенной строки (минимальный), а второй номер последней выделенной строки (максимальный) |
| isPathSelected(), isRowSelected() | Эта пара методов позволяет без труда выяснить, выделен ли некоторый узел в дереве (он может быть задан путем TreePath или номером строки) |

## *Внешний вид деревьев*

Мы узнали, как подготавливать и настраивать модель данных дерева для вывода данных на экран и как работать с выделенными узлами. Теперь было бы неплохо познакомиться с тем, как дерево позволяет настраивать свой внешний вид и внешний вид своих узлов. Нетрудно догадаться, что все полностью находится в ваших руках, как и принято в Swing. Работу по отображению узлов дерево JTree выполняет не само, оно переадресует ее специальному отображающему объекту (renderer), который должен реализовывать несложный интерфейс TreeCellRenderer. Создавая необходимый отображающий объект и учитывая в нем особенности узлов вашего дерева, вы сможете расцветить его всеми возможными цветами и украсить любыми значками.

У вас также остаются общие унаследованные от базового класса JComponent библиотеки Swing свойства, такие как font или foreground, которые мы не раз уже обсуждали при рассмотрении различных компонентов Swing. Они позволят быстро сменить используе-

мые деревом шрифты и цвета для всех узлов дерева разом. Это не самый гибкий подход, но иногда его оказывается достаточно.

Интерфейс отображающего объекта TreeCellRenderer на самом деле чрезвычайно прост. В нем определен всего один метод getTreeCellRendererComponent(), который обязуется возвратить некоторый графический компонент для переданных ему в качестве параметров дерева JTree, узла для отображения и дополнительных свойств узла (эти свойства определяют, выделен ли узел, раскрыт или свернут, лист ли это, владеет ли он фокусом ввода). Этот графический компонент и служит для отображения деревом узла. Используется данный компонент точно так же, как и в случае со списками JList, которые мы изучали в главе 11: он не добавляется на экран и остается незамеченным для графической системы, дерево задействует его как особую «кисть», вызывая метод paint() компонента в том месте, где располагается отображаемый в данный момент узел. Поэтому создавать новый компонент для каждого отображаемого узла не только не нужно, но и крайне расточительно и неэффективно. Компонент требуется один, для каждого узла просто меняются свойства этого компонента[4].

По умолчанию в дереве JTree используется стандартный отображающий объект DefaultTreeCellRenderer из пакета javax.swing.tree. Он унаследован от надписи JLabel и для каждого узла возвращает соответствующим образом настроенный (с подходящими текстом и значком) единственный экземпляр самого себя, то есть надписи. Этот экземпляр и служит для отображения всех узлов дерева. При этом объект DefaultTreeCellRenderer старается максимально оптимизировать прорисовку узлов дерева, в частности, он отключает все стандартные механизмы надписи JLabel, например, оповещение об изменении надписи и значка (это привязанные свойства JavaBeans) и автоматическую проверку корректности надписи. Учитывая, сколько узлов приходится отображать в больших деревьях, подобная оптимизация приходится весьма кстати.

Возникает резонный вопрос: «В дереве могут храниться данные любых типов, как их отображает стандартный объект?». Интересно, что для преобразования узла в строку, которая будет выведена на экран, стандартный отображающий объект не вызывает напрямую метод toString(), как это делает, к примеру, стандартный отображающий объект списка JList, а применяет особый метод класса JTree с названием convertValueToText(). По умолчанию данный метод просто вызывает для узла метод toString(), но его можно переопределить и по-своему обрабатывать преобразование узла в строку. Иногда это проще, чем написание нового отображающего объекта.

Объект DefaultTreeCellRenderer способен не только на отображение текста узла со стандартными значками. Как оказывается, его можно довольно тонко настроить, и во многих ситуациях этого оказывается достаточно, чтобы ваше дерево приняло именно тот внешний вид, который необходим приложению. При этом не стоит забывать, что стандартный отображающий объект унаследован от надписи JLabel, а, значит, вы смело можете использовать в тексте узла язык HTML, начиная его с «волшебной» комбинации символов <html>. Уже одно это дает вам прекрасные возможности для получения текста различных размеров, цветов и начертаний буквально одним мановением руки. Более того, легко можно сменить значки, необходимые для вывода узлов, и еще некоторые свойства (табл. 15.3).

**Таблица 15.3.** Свойства стандартного отображающего объекта дерева

| Свойства | Описание |
|---|---|
| font | Задает шрифт, который будет использоваться отображающим объектом для вывода узлов. Того же эффекта можно добиться сменой свойства font самого дерева JTree |

[4] Это ярко выраженный шаблон проектирования *приспособленец*, упомянутый нами в главе 11, когда мы в первый раз столкнулись с отображающими объектами.

**Таблица 15.3** *(продолжение)*

| Свойства | Описание |
|---|---|
| closedIcon | Позволяет сменить значок, обозначающий свернутый узел с потомками, например, в дереве с файлами это нераскрытая папка |
| openIcon | Значок для раскрытого узла с потомками (в дереве файлов это раскрытая папка) |
| leafIcon | Задает значок для узла, который является листом, то есть не обладает потомками. Как вы помните из обсуждения стандартной модели, есть два способа определения того, какой узел следует считать листом |
| backgroundNonSelectionColor, backgroundSelectionColor | Управляют цветом фона отображаемых на экране узлов. Первое свойство задает цвет фона для узла, когда он не выделен, второе — когда выделен |
| textNonSelectionColor, textSelectionColor | Управляют цветом шрифта узлов. Первое свойство хранит цвет шрифта невыделенного узла, второе — выделенного |

Используя любые поддерживаемые Swing возможности HTML и перечисленные в таблице свойства, можно удивительным образом раскрасить любое дерево. Рассмотрим пример, в котором задействуем самые эффектные возможности стандартного отображающего объекта дерева JTree:

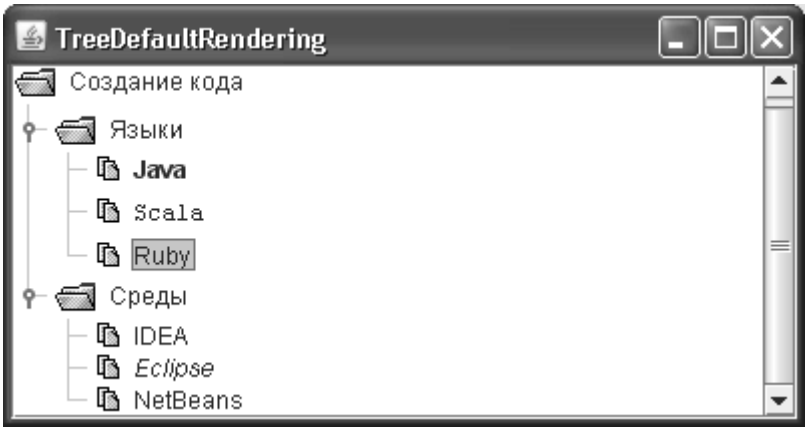
```
// TreeDefaultRendering.java
// Использование возможностей стандартного
// отображающего объекта DefaultTreeCellRenderer
import javax.swing.*;
import javax.swing.tree.*;
import java.awt.*;

public class TreeDefaultRendering extends JFrame {
  public TreeDefaultRendering() {
    super("TreeDefaultRendering");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем дерево на основе незатейливой модели
    JTree tree = new JTree(createTreeModel());
    // создадем и настраиваем отображающий объект
    DefaultTreeCellRenderer renderer =
        new DefaultTreeCellRenderer();
    renderer.setLeafIcon(new ImageIcon("Leaf.gif"));
    renderer.setClosedIcon(
        new ImageIcon("NodeClosed.gif"));
    renderer.setOpenIcon(
        new ImageIcon("NodeOpened.gif"));
    // передаем его дереву
    tree.setCellRenderer(renderer);
```

```
    // добавляем дерево и выводим окно на экран
    add(new JScrollPane(tree));
    setSize(400, 300);
    setVisible(true);
  }
  // листья дерева храним в массивах
  private String[] langs = {
      "<html><b>Java", "<html><pre>Scala", "Ruby" };
  private String[] ides = {
      "IDEA", "<html><i>Eclipse", "NetBeans" };
  // создание несложной модели дерева
  private TreeModel createTreeModel() {
    // корень нашего дерева
    DefaultMutableTreeNode root =
        new DefaultMutableTreeNode(
            "<html><font color=blue>Создание кода");
    // основные ветви
    DefaultMutableTreeNode lang =
        new DefaultMutableTreeNode("Языки");
    DefaultMutableTreeNode ide =
        new DefaultMutableTreeNode("Среды");
    root.add(lang);
    root.add(ide);
    // присоединяем листья
    for (int i=0; i<langs.length; i++) {
      lang.add(new DefaultMutableTreeNode(langs[i]));
      ide.add(new DefaultMutableTreeNode(ides[i]));
    }
    // создаем стандартную модель
    return new DefaultTreeModel(root);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new TreeDefaultRendering(); } });
  }
}
```

Здесь мы создаем небольшое окно с деревом JTree. Дерево строится из стандартной модели (а та, в свою очередь, принимает данные от стандартных узлов DefaultMutableTreeNode). Нетрудно заметить, что создание модели дерева в методе createTreeModel() практически не отличается от предыдущего примера. Единственное отличие в том, что мы стали использовать для данных некоторых узлов не простой текст,

а HTML-разметку, не забыв предупредить об этом отображающий объект с помощью магических символов <html>.

Посмотрите, как одним мановением руки настраивается стандартный отображающий объект дерева DefaultTreeCellRenderer. Мы сменяем три значка, используемые для вывода узлов и листьев, не затрагивая цвета и шрифты (это, как нетрудно заметить, гораздо проще сделать с помощью встроенного языка HTML). Чаще всего именно сменой значков для узлов дерева добиваются его уникального узнаваемого вида, который к тому же в состоянии «подсказать» пользователю, данные какого типа дерево отображает. Значки, показанные в примере, были частично заимствованы из коллекции графики Java Look And Feel Graphics Repository, специально созданной для внешнего вида Metal (вы можете найти эту коллекцию по адресу java.sun.com).




После того как мы задаем наш новый отображающий объект методом setCellRenderer(), остается включить дерево JTree в панель прокрутки и поместить его в окно, после чего вывести окно на экран. Заметьте, насколько мало кода нам понадобилось написать (если не считать кода, создающего модель дерева) и какой совершенно уникальный облик обрело наше дерево. Обратите внимание, как показаны узлы, названия которых написаны с помощью HTML. Одним словом, фантазия ваша в отображении узлов дерева ничем не ограничена, нужно лишь следить за тем, чтобы узлы дерева имели более или менее одинаковую высоту, иначе дерево теряет стройность и начинает запутывать пользователя.

## *Дерево с флажками*

Конечно, вам не обязательно использовать стандартный отображающий объект DefaultTreeCellRenderer, который при всех своих преимуществах все же довольно ограничен в возможностях: значки и стиль текста задаются для всего дерева сразу, вы не сможете задать разные значки для разных узлов, а это требуется не так уж и редко. Поэтому иногда приходится создавать собственные отображающие объекты. Сделать это несложно — нам уже известно, что в интерфейсе отображающего объекта TreeCellRenderer всего один метод, который должен вернуть для узла дерева компонент, призванный отображать данные узла на экране.

Создадим теперь специализированный компонент пользовательского интерфейса — дерево, способное отображать флажки. В некоторых ситуациях применение подобного компонента делает пользовательский интерфейс значительно удобнее (к примеру, там, где имеется некая иерархия множества доступных свойств приложения). Для создания

такого дерева нам понадобится как-то хранить данные узлов (текст и признак выбора флажка) и написать отображающий объект для таких данных. В главе 11 мы написали класс CheckBoxListElement для списка, состоящего из флажков; используем его и здесь (данный класс хранит именно текст и признак выделения элемента). Чтобы новое дерево было удобно задействовать в ваших программах, мы разместим его в библиотечном пакете com.porty.swing. Отображающий объект будет сразу же присоединяться в конструкторе дерева:

```
// com/porty/swing/CheckBoxTree.java
// Дерево, способное отображать в качестве узлов флажки
package com.porty.swing;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.*;

public class CheckBoxTree extends JTree {
  // конструктор на основе модели
  public CheckBoxTree(TreeModel model) {
    super(model);
    // задаем собственный отображающий объект
    setCellRenderer(new CheckBoxRenderer());
    // следим за щелчками мыши
    addMouseListener(new MouseL());
  }
  // стандартный объект для отображения узлов
  private DefaultTreeCellRenderer renderer =
      new DefaultTreeCellRenderer();
  // флажок для отображения узлов дерева
  class CheckBoxRenderer extends JCheckBox
      implements TreeCellRenderer {
    public CheckBoxRenderer() {
      // делаем флажок прозрачным
      setOpaque(false);
    }
    // данный метод должен вернуть компонент для узла
    public Component getTreeCellRendererComponent(
        JTree tree, Object value, boolean selected,
        boolean expanded, boolean leaf, int row,
        boolean hasFocus) {
      // проверяем, что используется стандартная модель
      if (!(value instanceof DefaultMutableTreeNode)) {
```

```
      // если нет, то используем стандартный объект
      return renderer.getTreeCellRendererComponent(
          tree, value, selected, expanded,
          leaf, row, hasFocus);
    }
    Object data = ((DefaultMutableTreeNode)value).
        getUserObject();
    // проверяем, подходит ли нам тип данных узла
    if ( data instanceof CheckBoxListElement ) {
      CheckBoxListElement element =
          (CheckBoxListElement)data;
      // настраиваем флажок
      setSelected(element.isSelected());
      setText(element.getText());
      return this;
    }
    // иначе задействуем стандартный объект
    return renderer.getTreeCellRendererComponent(tree,
        value, selected, expanded, leaf, row, hasFocus);
  }
}
// класс, следящий за щелчками мыши
class MouseL extends MouseAdapter {
  public void mousePressed(MouseEvent e) {
    // получим путь к узлу
    TreePath path = getClosestPathForLocation(
        e.getX(), e.getY());
    if ( path == null ) return;
    // проверим, подходят ли нам данные узла
    Object _node = path.getLastPathComponent();
    if (_node instanceof DefaultMutableTreeNode) {
      DefaultMutableTreeNode node =
          (DefaultMutableTreeNode)_node;
      Object data = node.getUserObject();
      if ( data instanceof CheckBoxListElement ) {
        // меняем состояние флажка
        CheckBoxListElement element =
            (CheckBoxListElement)data;
        element.setSelected(! element.isSelected());
        repaint(getPathBounds(path));
      }
```

```
            }
        }
    }
}
```

Мы наследуем наше дерево от класса JTree и определяем только один конструктор на основе модели TreeModel. Как мы знаем, остальные конструкторы, создающие дерево с помощью простых структур данных, не слишком полезны, так что от их отсутствия мы потеряем немного. В конструкторе сразу же происходит дополнительная настройка дерева: мы задаем для него особый отображающий объект, который будет ответственен за вывод на экран флажков, а также присоединяем к нему слушателя событий от мыши MouseListener. Слушатель событий будет следить за нажатиями кнопок мыши и менять состояние флажков: мы уже знаем, что компоненты, служащие для отображения узлов дерева, не получают никаких событий (так как фактически отсутствуют на экране), поэтому их приходится поддерживать самостоятельно.

Отображающий объект для нового дерева унаследован от флажка JCheckBox (так будет проще настраивать свойства флажка) и реализует необходимый интерфейс TreeCellRenderer. Метод getTreeCellRendererComponent(), определенный в данном интерфейсе, должен вернуть для некоторого узла дерева (известны само дерево, отображаемый узел и его состояние, в том числе, выбран ли он, является ли листом и т. п. — все это передается в качестве параметров) компонент, который данный узел отобразит. Так же как и в случае со списками и их отображающими объектами, компонент, как правило, один, он настраивается для каждого отображаемого элемента. Обратите внимание, что в конструкторе мы меняем свойство непрозрачности, устанавливая его в false. Флажки JCheckBox по умолчанию непрозрачны, а в составе других компонентов непрозрачные компоненты выглядят не слишком привлекательно, закрашивая все вокруг себя цветом фона. Для начала в методе getTreeCellRendererComponent() мы проверяем, что деревом используется стандартная модель и узлы представлены классом DefaultMutableTreeNode. Если это не так, работа передается стандартному отображающему объекту DefaultTreeCellRenderer, который специально для подобных случаев хранится в нашем дереве в виде переменной. Благодаря стандартному объекту наше новое дерево будет способно отображать не только флажки, но и любые другие элементы.

Если в дереве используется стандартная модель, то мы преобразуем узел к соответствующему типу и получаем значение, которое хранится в узле. Если значением является объектом CheckBoxListElement, в котором хранятся название узла и признак его выделения, в действие вступает наш отображающий объект. Он настраивает свойства флажка (флажком является сам отображающий объект), а именно текст и выделение, и возвращает флажок в качестве объекта, отображающего данный узел. Когда в узле хранится не объект CheckBoxListElement, а что-то иное, в действие снова вступает стандартный отображающий объект, который способен отобразить любой объект в виде строки. Обратите внимание, что мы не поддерживаем выделение цветом или рамку, когда флажок обладает фокусом или выбран. В этом примере мы будем считать, что главное — отобразить само состояние флажка, ну а добавить все вышеперечисленное очень просто, в том случае, если эти свойства вам понадобятся, надо будет лишь задать нужный цвет и рамку для флажка[5].

[5] Интересно, что найти цвет для выделенного узла дерева не так-то просто — в дереве JTree нет метода для его определения. Вы можете задействовать для этой цели стандартный отображающий объект, получив от него компонент с нужными настройками и получив его цвет фона методом getBackground().

Отображающий объект отвечает за вывод на экран флажков, слушатель мыши, присоединенный к дереву, обеспечит смену состояния флажков. Компонент, используемый для отображения узлов, в контейнер не добавляется, так что все необходимые ему события необходимо поддерживать вручную. В слушателе мы следим за нажатиями кнопок мыши. Прежде всего необходимо выяснить, в районе какого узла произошло нажатие кнопки. В этом нам помогает метод getClosestPathForLocation(), он возвращает путь TreePath до нужного нам узла. Данный метод может возвращать и пустую (null) ссылку (так может случиться, если в дереве не будет данных), так что нам необходимо проверить это. Если проверка успешна, мы получаем последний элемент пути — как нам уже известно, это тот узел, в районе которого и произошло событие. Подразумевая, что дерево использует стандартную модель, мы преобразуем узел к объекту DefaultMutableTreeNode и получаем хранящиеся в этом объекте данные (проверив предварительно тип узла). Если это данные нужного нам типа (CheckBoxListElement), то состояние флажка меняется на противоположное, после чего дерево перерисовывается. Обратите внимание на метод перерисовки: мы перерисовываем лишь нужный нам узел, а получить его прямоугольник на экране помогает метод дерева под названием getPathBounds(). В небольших деревьях «точечная» перерисовка вряд ли будет заметна, но в больших «лесах» с тысячами листьев вы обязательно оцените производительность.

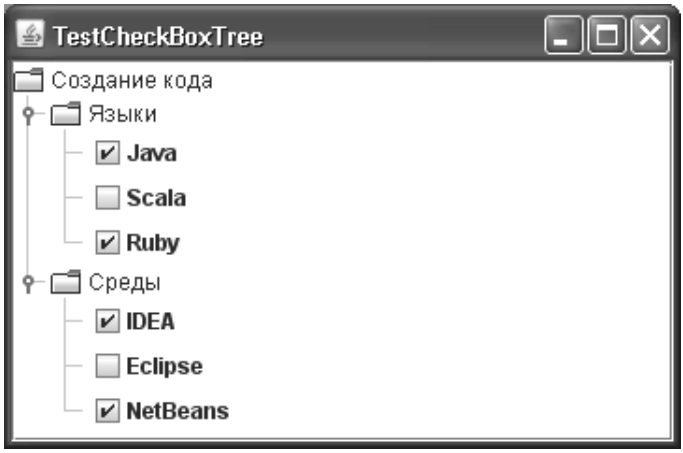
Проверим теперь новое дерево в работе и попробуем отобразить несложную модель, в которой будет совмещены и флажки, и обычные элементы. Приведем пример:

```
// TestCheckBoxTree.java
// Проверка дерева с поддержкой флажков
import com.porty.swing.*;
import javax.swing.*;
import javax.swing.tree.*;
import java.awt.*;

public class TestCheckBoxTree extends JFrame {
  public TestCheckBoxTree() {
    super("TestCheckBoxTree");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем модель и дерево
    TreeModel model = createTreeModel();
    CheckBoxTree tree = new CheckBoxTree(model);
    // добавляем дерево в окно
    add(new JScrollPane(tree));
    // выводим окно на экран
    setSize(400, 300);
    setVisible(true);
  }
  // листья дерева храним в массивах
  private String[] langs = { "Java", "Scala", "Ruby" };
  private String[] ides =
      { "IDEA", "Eclipse", "NetBeans" };
```

```
  // создание несложной модели дерева
  private TreeModel createTreeModel() {
    // корень нашего дерева
    DefaultMutableTreeNode root =
        new DefaultMutableTreeNode("Создание кода");
    // основные ветви
    DefaultMutableTreeNode lang =
        new DefaultMutableTreeNode("Языки");
    DefaultMutableTreeNode ide =
        new DefaultMutableTreeNode("Среды");
    root.add(lang);
    root.add(ide);
    // присоединяем листья с данными для флажков
    for (int i=0; i<langs.length; i++) {
      lang.add(new DefaultMutableTreeNode(
          new CheckBoxListElement(false, langs[i])));
      ide.add(new DefaultMutableTreeNode(
          new CheckBoxListElement(false, ides[i])));
    }
    // создаем стандартную модель
    return new DefaultTreeModel(root);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new TestCheckBoxTree(); } });
  }
}
```

Основную часть программы примера занимает код создания модели (в методе createTreeModel()), которую затем попытается отобразить наше новое дерево с поддержкой флажков. Модель основана на тех же данных, что мы использовали для нескольких предыдущих примеров, только на этот раз данные листьев будут храниться в объектах CheckBoxListElement, так что дерево отобразит их в виде флажков. В модели будет две ветви (они созданы на основе обычных строк), у каждой имеется по три потомка (все эти потомки будут флажками). Для простоты и компактности примера потомки добавляются в цикле. Для хранения данных применяются объекты DefaultMutableTreeNode, которые затем передаются в стандартную модель DefaultTreeModel.

Остальная часть программы примера и вовсе проста: мы создаем небольшое окно с рамкой, в центр его добавляется новое дерево CheckBoxTree, предварительно вставленное в панель прокрутки JScrollPane. Дерево создается на основе подготовленной модели. После создания модели и дерева остается лишь вывести окно на экран.




Запустив программу с примером, вы оцените внешний вид и удобство дерева с флажками, а заодно увидите, как флажки меняют свое состояние при нажатиях кнопок мыши[6]. Конечно, не хватает поддержки смены состояния посредством клавиатуры, но это легко исправить: просто присоедините к дереву CheckBoxTree слушателя событий KeyListener или, что еще лучше, потому что обеспечит дополнительную гибкость, зарегистрируйте в карте входных событий InputMap подходящее клавиатурное сокращение, при срабатывании которого будет меняться состояние выделенного в данный момент флажка.

Подобным образом поддерживается отображение в дереве самых экзотичных данных. Дерево само заботится о том, чтобы они находились в иерархических отношениях, а вы можете хранить их в модели, стандартной или своей собственной, и правильно отображать. Все это обеспечит удивительную гибкость и прозрачность вашего кода, легкость проектирования и позволит максимально удовлетворить пользователя. Кстати, в качестве упражнения вы можете написать отображающий объект, который позволял бы выводить для дерева строки с любыми значками, а не только с заданными в стандартном отображающем объекте.

## *Редактирование узлов*

Узлы, а точнее хранящиеся в них данные, можно не только отображать, но и редактировать. По умолчанию редактирование узлов дерева отключено, но вы всегда можете включить его, присвоив свойству editable значение true. Редактированием узлов, так же как и их отображением, дерево занимается не само, оно поручает эту работу специальному редактору. Для того чтобы некий объект смог стать редактором узлов дерева JTree, ему необходимо реализовать интерфейс TreeCellEditor.

Интерфейс TreeCellEditor унаследован от базового интерфейса CellEditor, который описывает обязанности любого редактора сложных компонентов Swing, а именно — редактора деревьев и таблиц. В интерфейсе CellEditor представлено довольно много методов, предназначенных для выполнения различных вспомогательных функций: для отмены и прекращения редактирования; для регистрации и отсоединения слушателей, получа-

[6] Интересно, что наше дерево при использовании мыши даже удобнее стандартного. В стандартной реализации для выбора узла необходимо щелкнуть на самом тексте, который может быть совсем невелик, в то время как метод getClosestPathForLocation() возвращает узел на одном уровне с щелчком даже если он находится от него на почтительном расстоянии. Неплохая добавка к интерфейсу, если вы чтите *закон Фитса* — чем больше объект на экране, тем проще в него попасть мышью.

ющих уведомления о процессе редактирования; для определения возможности редактирования ячейки в случае произошедшего события и т. п. В интерфейсе TreeCellEditor появляется еще один метод с названием getTreeCellEditorComponent(). Данный метод обязуется возвратить компонент, который и будет использован для редактирования данных узла. В итоге задача написания редактора для дерева «с нуля» оказывается довольно непростой: работы предстоит немало.

Однако, как и всегда в Swing, у вас есть возможность воспользоваться стандартными редакторами, поставляющимися вместе с библиотекой. Зачастую их возможностей вполне хватает для вашего приложения, а настраивать их и затем работать с ними совсем просто. Стандартный редактор для деревьев (кстати, тот же редактор применяется и для таблиц) реализован в классе DefaultCellEditor. Он позволяет организовать редактирование значений с помощью одного из трех компонентов: текстового поля JTextField, раскрывающегося списка JComboBox или флажка JCheckBox. Действие происходит следующим образом: вы создаете компонент для редактирования, настраиваете его (например, заполняете список элементами) и передаете в соответствующий конструктор стандартного редактора. Далее остается включить редактирование дерева и присоединить редактор методом setCellEditor(). Дерево JTree будет готово использовать настроенный вами стандартный редактор.

В пакете javax.swing.tree есть вспомогательный редактор DefaultTreeCellEditor. Он, хотя и реализует все предписанные интерфейсом TreeCellEditor методы, сам ничего не редактирует, а передает свои обязанности «настоящему» редактору, который вы можете задать в конструкторе. В качестве «настоящего» редактора, как правило, применяется только что описанный нами редактор DefaultCellEditor. Главным его достоинством является то, что он производит редактирование узла дерева, не убирая с экрана значок, ассоциированный с этим узлом. Такое поведение более понятно пользователю, чем исчезновение значка при редактировании и восстановление его на экране после окончания редактирования.

В следующем несложном примере попробуем применить стандартный редактор для дерева:

```
// TreeDefaultEditing.java
// Стандартные редакторы для деревьев
import javax.swing.*;
import javax.swing.tree.*;
import java.awt.*;

public class TreeDefaultEditing extends JFrame {
  // листья дерева храним в массивах
  private String[] basics = {
      "Красный", "Зеленый", "Синий" };
  private String[] extendeds = {
      "Желтый", "Голубой", "Розовый" };
  public TreeDefaultEditing() {
    super("TreeDefaultEditing");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем дерево на основе простой модели
    JTree tree = new JTree(createTreeModel());
    // включаем редактирование
```

```
    tree.setEditable(true);
    // "настоящий" редактор
    JComboBox combo = new JComboBox(
        new String[] {"Красный", "Зеленый"});
    DefaultCellEditor editor = new DefaultCellEditor(combo);
    // специальный редактор для дерева
    DefaultTreeCellRenderer renderer =
        new DefaultTreeCellRenderer();
    DefaultTreeCellEditor treeEditor =
        new DefaultTreeCellEditor(tree, renderer, editor);
    // присоединяем редактор к дереву
    tree.setCellEditor(treeEditor);
    // выводим окно на экран
    add(new JScrollPane(tree));
    setSize(400, 300);
    setVisible(true);
  }
  // создание несложной модели дерева
  private TreeModel createTreeModel() {
    // корень нашего дерева
    DefaultMutableTreeNode root =
        new DefaultMutableTreeNode("Цвета");
    // основные ветви
    DefaultMutableTreeNode basic =
        new DefaultMutableTreeNode("Основные");
    DefaultMutableTreeNode extended =
        new DefaultMutableTreeNode("Дополнительные");
    root.add(basic);
    root.add(extended);
    // присоединяем листья
    for (int i=0; i<basics.length; i++) {
      basic.add(new DefaultMutableTreeNode(basics[i]));
      extended.add(
          new DefaultMutableTreeNode(extendeds[i]));
    }
    // создаем стандартную модель
    return new DefaultTreeModel(root);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
```

```
            public void run() { new TreeDefaultEditing(); } });
    }
}
```

В примере мы создаем дерево JTree, основанное на простой модели, описывающей взаимоотношения цветов. Модель создается в методе createTreeModel(). Процесс наполнения модели данными нам уже прекрасно знаком.

Гораздо интереснее посмотреть, как настраиваются стандартные редакторы для нашего дерева. Прежде всего необходимо создать «настоящий» редактор DefaultCellEditor, тот самый, который можно затем будет передать в редактор DefaultTreeCellEditor. В качестве компонента, заведующего процессом редактирования, мы выбрали раскрывающийся список JComboBox, который создали на основе массива из двух строк. Настроенный список затем надо передать в конструктор класса DefaultCellEditor. После этого уже можно приступать ко второму редактору DefaultTreeCellEditor, он позволит нам проводить редактирование ячеек, не отказываясь от значков. Конструктор второго редактора требует сразу три параметра: дерево, стандартный отображающий объект и редактор для дерева. В дереве будет проводиться редактирование, стандартный отображающий объект[7] потребуется для получения значков, ассоциированных с узлами, а редактор, естественно, будет выполнять редактирование. Настроенный редактор DefaultTreeCellEditor можно присоединять к дереву.


Запустив программу с примером, вы увидите, как стандартные редакторы позволяют организовать редактирование на основе раскрывающегося списка. Начать редактирование позволяет тройной щелчок мыши или двойной щелчок, выполненный с паузой. Кстати, с помощью специальных методов стандартных редакторов вы можете задать число щелчков мыши, необходимых для начала редактирования. Обратите внимание, что во время редактирования значок узла остается на экране — это «заслуга» редактора DefaultTreeCellEditor, используй мы сразу редактор DefaultCellEditor, значок во время редактирования с экрана исчезал бы. Помимо списка JComboBox вы также можете задействовать для редактирования текстовое поле JTextField или флажок JCheckBox, хотя последний применяется редко[8]. Однако редактирование с помощью стандартных редакторов все же довольно ограничено:

[7] Редактор DefaultTreeCellEditor может работать только со стандартным отображающим объектом, что, конечно же, ограничивает область его применения.

[8] Лучше обратиться к созданному нами недавно специальному дереву с флажками, позволяющему менять их состояние.

редактировать можно все узлы «без разбора», и для редактирования всех узлов всегда применяется один и тот же компонент. Для деревьев, отображающих сложные данные различных типов, зачастую приходится писать собственные редакторы.

## Создание собственного редактора

Как мы уже упоминали, создание собственного редактора для узлов дерева может вылиться в приличную работу: методов в интерфейсе TreeCellEditor (большая часть которых находится в его базовом интерфейсе CellEditor) немало. Однако библиотека предлагает нам помощника — абстрактный класс AbstractCellEditor реализует большую часть методов интерфейса CellEditor, и в том числе предоставляет нам поддержку слушателей CellEditorListener. Эти слушатели оповещаются при окончании или отмене редактирования, чтобы компонент (дерево) знал, когда работа редактора оканчивается. Для того чтобы наш собственный редактор для дерева смог заработать, нужно будет определить всего два метода: метод getCellEditorValue() должен возвращать значение, в данный момент находящееся в редакторе, а уже знакомый нам метод getTreeCellEditorComponent() — компонент, который и задействуется деревом как редактор. Остальные методы реализованы классом AbstractCellEditor и их требуется переопределять только в том случае, если вам понадобятся дополнительные возможности.

Попробуем написать особый редактор для дерева, которое отображает каталог телефонных номеров некоторой компании. Телефонные номера могут быть отсортированы по отделам и подразделениями, которые, как правило, находятся в иерархических отношениях, так что использование дерева вполне оправдано. Редактировать номера, отображаемые деревом, с помощью стандартных редакторов неудобно: во-первых, они не обеспечивают никаких ограничений на вводимые пользователем данные (а вводить можно будет только телефонные номера), во-вторых, они позволяют редактировать все узлы даже со вспомогательной информацией.

Новый редактор для дерева с телефонами использует для редактирования текстовое поле JFormattedTextField, позволяющее вводить данные в определенном формате. Формат вводимых данных определяется специальной текстовой маской[9], которая в нашем примере задает простой телефонный номер. Кроме того, редактор следит за тем, чтобы редактировать можно было только листья дерева, в которых и хранятся телефонные номера, а не остальные узлы, служащие для упорядочения информации.

```
// CustomTreeEditor.java
// Создание специализированного редактора узлов
import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import java.util.EventObject;

public class CustomTreeEditor extends JFrame {
  public CustomTreeEditor() {
    super("CustomTreeEditor");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
```

[9] Мы обсудим текстовое поле JFormattedTextField и поддерживаемые им форматы данных в главе 16, которая полностью посвящена текстовым компонентам Swing.

```
  // настраиваем дерево
  JTree tree = new JTree(createTreeModel());
  // включаем редактирование узлов
  tree.setEditable(true);
  DefaultTreeCellRenderer renderer =
      new DefaultTreeCellRenderer();
  DefaultTreeCellEditor editor =
      new DefaultTreeCellEditor(tree, renderer,
          new MaskTreeEditor(tree));
  tree.setCellEditor(editor);
  // выводим окно на экран
  add(new JScrollPane(tree));
  setSize(400, 300);
  setVisible(true);
}
// список телефонов
private String[] phoneDirectory = {
    "123-13-13", "444-55-67", "111-23-45"};
// создает модель дерева
private TreeModel createTreeModel() {
  DefaultMutableTreeNode root =
      new DefaultMutableTreeNode("Компания");
  DefaultMutableTreeNode node =
      new DefaultMutableTreeNode("Отдел кадров");
  root.add(node);
  // присоединяем листья
  for (String phone : phoneDirectory) {
    node.add(new DefaultMutableTreeNode(phone));
  }
  return new DefaultTreeModel(root);
}
// специальный редактор узлов дерева
class MaskTreeEditor extends AbstractCellEditor
    implements TreeCellEditor {
  // дерево
  private JTree tree;
  // текстовое поле, применяемое для редактирования
  private JFormattedTextField editor;
  // конструктор редактора
  public MaskTreeEditor(JTree tree) {
    this.tree = tree;
    // создаем форматирующий объект
```

```
        try {
          MaskFormatter phone =
              new MaskFormatter("###-##-##");
          editor = new JFormattedTextField(phone);
        } catch (Exception ex) {
          ex.printStackTrace();
        }
        // присоединяем к полю слушателя
        editor.addActionListener(new ActionListener() {
          // вызывается при окончании редактирования
          public void actionPerformed(ActionEvent e) {
            stopCellEditing();
          }
        });
      }
      // возвращает компонент, используемый как редактор
      public Component getTreeCellEditorComponent(
          JTree tree, Object value, boolean selected,
          boolean expanded, boolean leaf, int row) {
        // устанавливаем новое значение
        editor.setText(value.toString());
        // возвращаем текстовое поле
        return editor;
      }
      // возращает текущее значение в редакторе
      public Object getCellEditorValue() {
        return editor.getText();
      }
      // определяет, можно ли проводить редактирование
      public boolean isCellEditable(EventObject event) {
        MutableTreeNode node = (MutableTreeNode)
            tree.getLastSelectedPathComponent();
        return ( (node != null) && ( node.isLeaf()) );
      }
    }
    public static void main(String[] args) {
      SwingUtilities.invokeLater(
          new Runnable() {
            public void run() { new CustomTreeEditor(); } });
    }
}
```

Мы создаем небольшое окно с рамкой, в центре этого окна разместится дерево JTree (как и всегда, дерево предварительно вставлено в панель прокрутки JScrollPane). Данные дерево черпает из несложной модели, которая создается методом createTreeModel(). Модель состоит из пары узлов (корня и его потомка), к которым присоединены три листа, в листьях и располагаются телефонные номера. Чтобы номера было удобно присоединять к узлам (в цикле), они хранятся в массиве строк с названием phoneDirectory.

Специальный редактор для дерева с телефонными номерами описан во внутреннем классе с названием MaskTreeEditor. Данный класс унаследован от абстрактного базового класса AbstractCellEditor (в нем описана большая часть методов интерфейса CellEditor) и реализует интерфейс TreeCellEditor. Интерфейс TreeCellEditor на самом деле унаследован от интерфейса CellEditor, но методы CellEditor, описанные в базовом классе AbstractCellEditor, «считаются» методами TreeCellEditor, которые нам нужно реализовать, поэтому работы остается сделать не так уж и много. Прежде всего мы настраиваем компонент JFormattedTextField, который и будет применяться для редактирования значений, находящихся в узлах дерева. Это делается в конструкторе класса: в блоке try/catch создается форматирующий объект MaskFormatter, описывающий телефонный номер (подробнее о форматирующих объектах мы узнаем в главе 16), который затем передается в поле JFormattedTextField. Обратите внимание, что конструктору класса MaskFormatter требуется передать ссылку на дерево, с которым он будет работать. Чуть позже мы увидим, зачем это нужно. К созданному текстовому полю тут же присоединяется слушатель ActionListener, который оповещается об окончании редактирования. Как только мы получаем подобное оповещение, то вызываем метод stopCellEditing(), описанный в базовом классе AbstractCellEditor. Реализация базового класса при вызове данного метода сообщает слушателям CellEditorListener о том, что редактирование закончено.

Далее мы реализуем метод getTreeCellEditorComponent(), который должен возвратить компонент, используемый для редактирования. Здесь все просто: мы помещаем в текстовое поле строковое значение узла (подразумевая, что в узле хранится правильный телефонный номер) и возвращаем ссылку на это текстовое поле.

Следующая пара методов относится к интерфейсу CellEditor. Метод getCellEditorValue() возвращает значение, находящееся в данный момент в редакторе. В нем мы передаем текст, находящийся в текстовом поле. Метод isCellEditable() определяет, можно ли проводить редактирование для некоторого события EventObject, переданного ему в качестве параметра. Событие мы не анализируем, а просто получаем выделенный узел дерева (ссылку на дерево, как вы помните, мы сохранили в конструкторе) и разрешаем редактирование только для листьев дерева, так как в нашем случае телефонные номера хранятся только в листьях.

После написания редактора остается присоединить его к дереву. Мы включаем для дерева возможность редактирования, а дальше делаем все то же самое, что и в предыдущем примере: создаем стандартный отображающий объект и стандартный редактор DefaultTreeCellEditor, которому передаем ссылки на дерево, отображающий объект и наш новый редактор.

Запустив программу с примером, вы оцените удобство работы с новым, специализированным редактором узлов дерева. Редактировать можно только листья дерева с телефонными номерами, причем вводить разрешается лишь правильные телефонные номера: в противном случае вы просто не сможете передать новое значение дереву (редактирование, как и в предыдущем примере, включается тройным щелчком мыши или двойным щелчком с задержкой). В деревьях, предназначенных для хранения и отображения сложных данных, специальные редакторы незаменимы, во много раз повышая удобство работы с интерфейсом.

В дальнейшем, если вам понадобятся более сложные условия редактирования, вы сможете настроить их с помощью вспомогательных методов из интерфейса CellEditor, позволяющих анализировать событие в интерфейсе, данные и само дерево, прежде чем передавать дереву компонент для редактирования.

## *Дополнительные события деревьев*

Пока мы увидели только одни события у деревьев, а именно события, сообщающие об изменении выделенных узлов, да и те скорее были у модели выделения дерева. Однако есть и еще кое-что. Мы можем узнать о моменте развертывания и свертывания узлов дерева, причем как уже о свершившемся факте, так и о моменте непосредственно перед тем, как узел будет развернут. Для этого служат слушатели TreeExpansionListener и TreeWillExpandListener, соответственно. Их присоединяют непосредственно к дереву JTree. Если первое событие полностью информационное, то второе позволяет нам предотвратить развертывание (или свертывание) определенного узла, если вдруг это понадобится. Рассмотрим простой пример:

```
// TreeExpansionEvents.java
// События при развертывании узлов дерева
import javax.swing.*;
import javax.swing.event.TreeExpansionListener;
import javax.swing.event.TreeWillExpandListener;
import javax.swing.event.TreeExpansionEvent;
import javax.swing.tree.DefaultTreeModel;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.TreeModel;
import javax.swing.tree.ExpandVetoException;
import java.awt.*;

public class TreeExpansionEvents extends JFrame {
  public TreeExpansionEvents() {
    super("TreeExpansionEvents");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем дерево на основе модели
    JTree tree = new JTree(createTreeModel());
    // добавляем слушателей
    TreeListener listener = new TreeListener();
    tree.addTreeExpansionListener(listener);
    tree.addTreeWillExpandListener(listener);
```

```
  add(new JScrollPane(tree));
  setSize(400, 300);
  setVisible(true);
}
// листья дерева храним в массивах
private String[] langs = {
    "<html><b>Java", "<html><pre>Scala", "Ruby" };
private String[] ides = {
    "IDEA", "<html><i>Eclipse", "NetBeans" };
// создание несложной модели дерева
private TreeModel createTreeModel() {
  // корень нашего дерева
  DefaultMutableTreeNode root =
      new DefaultMutableTreeNode(
          "<html><font color=blue>Создание кода");
  // основные ветви
  DefaultMutableTreeNode lang =
      new DefaultMutableTreeNode("Языки");
  DefaultMutableTreeNode ide =
      new DefaultMutableTreeNode("Среды");
  root.add(lang);
  root.add(ide);
  // присоединяем листья
  for (int i=0; i<langs.length; i++) {
    lang.add(new DefaultMutableTreeNode(langs[i]));
    ide.add(new DefaultMutableTreeNode(ides[i]));
  }
  // создаем стандартную модель
  return new DefaultTreeModel(root);
}
// слушатель событий о развертывании узлов
class TreeListener implements TreeExpansionListener,
    TreeWillExpandListener {
  public void treeExpanded(TreeExpansionEvent event) {
    System.out.println("Узел развернут: " + event.getPath());
  }
  public void treeCollapsed(TreeExpansionEvent event) {
    System.out.println("Узел свернут: " + event.getPath());
  }
  public void treeWillExpand(TreeExpansionEvent event)
      throws ExpandVetoException { }
```

```
    public void treeWillCollapse(TreeExpansionEvent event)
        throws ExpandVetoException {
      if ( event.getPath().getLastPathComponent().
          toString().equals("Языки") )
        throw new ExpandVetoException(event);
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new TreeExpansionEvents(); } });
  }
}
```

Начальная часть примера нам уже хорошо знакома из материала этой главы — мы добавляем дерево, обернутое в панель прокрутки, в центр окна, а для данных применяем стандартную модель дерева и стандартные узлы, данные для которых находятся в массивах. Самое интересное находится в слушателях, которых мы присоединяем к дереву. Для простоты оба они реализованы одним и тем же внутренним классом.

Методы слушателя TreeExpansionListener вызываются уже «постфактум», так что мы можем использовать этот факт только в качестве полезной информации. В примере мы так и поступаем, просто выводя путь к развернутому или свернутому узлу на консоль. Заметьте, что путь развернутого узла можно получить прямо из события с помощью метода getPath().

Методы же слушателя TreeWillExpandListener (их можно определить по слову «will» в названии) вызываются перед развертыванием узла, и что самое интересное, позволяют возбуждать исключение ExpandVetoException. Если вы создадите подобное исключение, для дерева это будет знаком того, что данный узел нельзя разворачивать, или, как в нашем примере, сворачивать. Чтобы определить, что это тот самый узел, мы проверяем его строковое значение, но это не самый гибкий способ, как правило, проще получить данные узла и каким-то образом их проверить. Запустив пример, вы поймете, что развернуть узел с названиями языков можно, а вот свернуть обратно никак не получится. Интересно, что корень дерева при этом все равно можно свернуть, поэтому если вы хотите удостовериться в том, что на экране виден важный для вас узел, имеет смысл следить не только за ним, но и за его узлами-предками.

## *Резюме*

Деревья Swing позволяют выводить на экран любые данные, находящиеся в иерархических отношениях, даже очень сложных. Все аспекты работы дерева (возможность редактирования, внешний вид узлов и листьев, режимы выделения и многое другое) полностью управляемы, и вы сможете работать с деревом именно так, как это нужно вашему приложения и как это удобно вам.

# Глава 16. Текстовые компоненты

Одной из самых впечатляющих частей библиотеки Swing является пакет javax.swing.text, обеспечивающий ее инструментами для работы с текстом. Благодаря этому пакету в вашем арсенале появляются несколько мощных текстовых компонентов, позволяющих реализовать в приложении средства ввода и редактирования текста любой сложности, начиная от однострочного текстового поля, часто применяемого для получения от пользователя простой информации, и заканчивая многофункциональным текстовым редактором с разнообразными возможностями. Учитывая, что текстовые компоненты используются в приложениях очень часто, мощь и гибкость текстовых компонентов библиотеки Swing будут как нельзя кстати. И действительно, трудно представить себе приложение, в котором нет текстовых компонентов, предназначенных для ввода информации пользователем. В том или ином виде, но текстовые компоненты, в том числе и встроенные в другие компоненты, присутствуют в приложении всегда.

Пакет javax.swing.text тщательно спланирован, качественно реализован и без сомнения очень велик и довольно сложен. Его объем и сложность его изучения вполне сравнимы с объемом и сложностью изучения самой библиотеки Swing, если иметь в виду все ее компоненты и возможности, а не только относящиеся к обработке текста. Не погрешив против истины, можно сказать, что текстовый пакет представляет собой небольшое «королевство» внутри Swing. Тем не менее, как и всегда в случае с библиотекой Swing, вы можете добиться очень многого, не прикладывая для этого титанических усилий и не вникая во все тонкости реализации текстового пакета Swing. Как мы вскоре увидим, несколькими строками кода можно создать вполне приличные инструменты для редактирования текста, достаточные для широкого класса приложений.

Основные возможности всех текстовых компонентов Swing и их базовая архитектура описаны в абстрактном классе JTextComponent из пакета javax.swing.text. Именно от этого класса унаследованы все текстовые компоненты Swing, будь то простое текстовое поле или многофункциональный редактор. Помимо того что в данном базовом классе задаются общие для всех текстовых компонентов свойства и действия (такие как цвет выделенного текста, цвет самого выделения, курсор, сам текст, механизмы работы с буфером обмена), в нем описывается взаимодействие практически всех составных частей пакета javax.swing.text, которых в нем насчитывается очень и очень много. Постепенно, по мере чтения этой главы, мы будем узнавать самое важное о механизмах текстового пакета Swing, пока же стоит отметить, что в текстовых компонентах Swing, в отличие от остальных компонентов библиотеки, модель, вид и контроллер практически полностью *разделены*. Как известно, текстовые компоненты довольно сложны и состоят из многих элементов. Оказалось, что раздельные элементы архитектуры MVC подходят для них как нельзя лучше. Модель текстовых компонентов представлена довольно простым интерфейсом Document, который позволяет получать информацию об изменениях в документе и хранящийся в нем текст, а также при необходимости изменять полученный текст. Вид, как нетрудно догадаться, в конечном итоге реализован в UI-представителях текстовых компонентов, но составляется он на основе специальных объектов Element и View, больше отвечающих именно текстовым компонентам. Благодаря этим объектам вы сможете гибко настраивать и расширять внешний вид и структуру текстовых компонентов без вмешательства в сложный процесс их конечной прорисовки. Контроллер частично соединен с видом (особенно это касается событий, не связанных с клавиату-

рой), а частично реализован в виде *карты клавиатуры* (keymap), аналога карты входных событий, известной нам еще с главы 5. Карта клавиатуры позволяет гибко, без смены UI-представителя текстового компонента, менять реакцию текстового компонента на нажатия клавиш (это очень важно, поскольку даже в одном текстовом компоненте могут поддерживаться несколько видов документов, способы редактирования которых возможно кардинально отличаются) и таким образом представляет собой ограниченный вариант контроллера.

Тем не менее, для будничной работы с текстовыми компонентами, за рамки которой большинство приложений не выходит, нам не придется вникать в детали реализации текстового пакета и тонко настраивать его механизмы. Познакомимся теперь с текстовыми компонентами Swing и убедимся, что работать с ними не сложнее, чем с другими компонентами библиотеки.

# Каталог текстовых компонентов

Как мы уже отмечали, Swing предоставляет нам целый арсенал текстовых средств, от небольших полей для ввода простого текста до полнофункциональных редакторов со всеми мыслимыми возможностями. Прежде чем перейти к более детальному изучению их возможностей (большинство из которых для всех текстовых компонентов одинаковы), составим своеобразный каталог текстовых компонентов Swing с небольшими примерами, которые позволят нам быстро познакомиться с ними. Кроме того, данный каталог удобно будет держать «под рукой» в качестве простого справочника.

## Текстовые поля

*Текстовые поля* (text fields) — самые простые и наиболее часто встречающиеся в пользовательских интерфейсах текстовые компоненты. Как правило, такое поле является однострочным и служит для ввода текста. В библиотеке Swing имеется два текстовых поля. Первое, представленное классом JTextField, позволяет вводить некоторый простой однострочный текст (разнообразные атрибуты форматирования, такие как дополнительные цвета и шрифты, не поддерживаются). Второе поле, реализованное классом JPasswordField и унаследованное от простого поля JTextField, дает возможность организовать ввод «секретной» информации (чаще всего паролей), которая не должна напрямую отображаться на экране. Оба текстовых поля очень просты, работа с ними чаще всего сводится к заданию количества отображаемых в поле символов и начального текста (если таковой требуется), после чего остается только поместить поле в контейнер и в нужный момент получить из него набранный пользователем текст. Рассмотрим простой пример использования текстовых полей Swing:

```
// UsingTextFields.java
// Использование текстовых полей Swing
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class UsingTextFields extends JFrame {
  // наши поля
  private JTextField smallField, bigField;
  public UsingTextFields() {
    super("UsingTextFields");
```

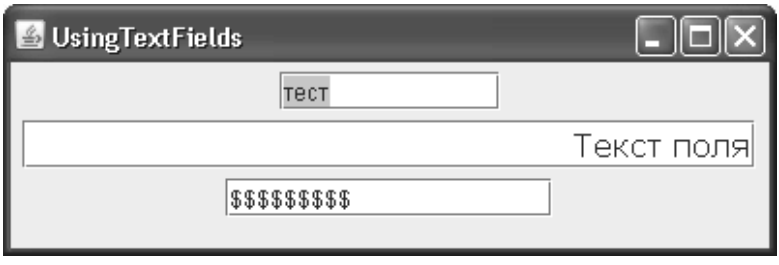
```
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем текстовые поля
    smallField = new JTextField(10);
    bigField = new JTextField("Текст поля", 25);
    // дополнительные настройки
    bigField.setFont(new Font("Verdana", Font.PLAIN, 16));
    bigField.setHorizontalAlignment(JTextField.RIGHT);
    // слушатель окончания ввода
    smallField.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        // показываем введенный текст
        JOptionPane.showMessageDialog(
            UsingTextFields.this,
            "Ваше слово: " +
                smallField.getText());
      }
    });
    // поле с паролем
    JPasswordField password = new JPasswordField(15);
    password.setEchoChar('$');
    // добавляем поля в окно и выводим его на экран
    setLayout(new FlowLayout());
    add(smallField);
    add(bigField);
    add(password);
    setSize(400, 300);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new UsingTextFields(); } });
  }
}
```

Здесь мы создаем окно небольших размеров с несколькими текстовыми полями. Первое поле создается с помощью наиболее распространенного варианта конструктора класса JTextField, которому необходимо передать максимальное количество символов в поле. Для однострочных текстовых полей прокрутка не нужна, и размер поля (в символах) должен примерно соответствовать объему информации, которую пользователь вводит в поле. В противном случае работать с текстовым полем было бы не совсем удобно, причем как при нехватке места для вводимой информации, так и при избыточной длине поля. В нашем примере особенных требований к вводимой информации нет, так что мы вполне можем указать произвольное количество символов. Второе поле созда-

ется более функциональным конструктором: ему необходимо передать текст, который будет записан в поле, и максимальное количество символов (для примера мы делаем это значение достаточно большим). Далее демонстрируются дополнительные возможности текстовых полей, довольно примитивные, но вполне уместные: вы можете с легкостью сменить шрифт и вариант выравнивания текста в поле (по умолчанию текст выравнивается по левому краю, мы выровняли его по правому краю, таким же образом можно задать выравнивание текста по центру).

Обратите внимание на то, что к текстовому полю можно присоединить слушателя событий ActionListener. Такие слушатели оповещаются о нажатии пользователем специальной клавиши, сигнализирующей об окончании ввода (обычно это клавиша Enter). Использовать слушателя особенно удобно в случае текстовых полей, предназначенных для ввода важной информации, такой как имя пользователя в диалоговом окне входа в систему. Присоединение к полю слушателя ActionListener ускоряет процесс работы с интерфейсом, так как избавляет пользователя от необходимости каждый раз по окончании ввода данных щелкать на подтверждающих кнопках (подобных кнопке OK) — вместо этого он сразу же, не покидая текстового поля, нажимает клавишу Enter и быстро переходит к следующему этапу работы с приложением. Помимо прямого присоединения к полю слушателя ActionListener вы можете воспользоваться методом setAction(), присоединяющего к полю объект-команду Action (применение этого метода не удаляет уже присоединенных к полю слушателей, все они также будут оповещаться о завершении ввода).

Пример также иллюстрирует работу с полем для ввода «секретных» данных JPasswordField. Не забывайте, что данное поле унаследовано от обычного поля JTextField, так что все вышесказанное верно и для него. Из собственных методов поля JPasswordField можно упомянуть лишь метод setEchoChar(), служащий для смены символа-заменителя (символа, который отображается вместо всех вводимых пользователем в поле символов во избежание появления на экране конфиденциальной информации). По умолчанию в качестве такого символа используется звездочка (*), которую для примера мы заменили символом доллара ($). Разработчики класса JPasswordField не рекомендуют также применять для получения введенного в поле значения (пароля) обычный метод getText(). Дело в том, что создаваемая данным методом строка String не может меняться все время жизни программы (объекты String в Java максимально оптимизируются компилятором и виртуальной машиной), и вы не сможете «стереть» ее из памяти. Для получения данных предоставляется более безопасный метод getPassword(), возвращающий массив символов char (значения которого после проверки имеет смысл обнулить и при желании вызвать сборщик мусора). Поле JPasswordField также особым образом копирует данные в буфер обмена — оно переопределяет методы cut() и copy() (эти методы определены в базовом классе JTextComponent), запрещая копировать набранный текст в буфер обмена, что также служит для защиты конфиденциальной информации.




Запустив программу с примером, вы сможете увидеть работу текстовых полей воочию. Подвести итог нам поможет табл. 16.1 с описанием основных свойств текстовых полей Swing.

**Таблица 16.1.** Свойства текстовых полей

| Свойства (и методы get/set) | Описание |
|---|---|
| text | Позволяет получить введенный в поле текст или заменить его. Для поля с конфиденциальной информацией лучше использовать метод getPassword() |
| columns | Задает количество столбцов в поле, так что вы можете получить размер поля или в любой момент изменить его |
| font | Определяет используемый в текстовом поле шрифт. Не стоит слишком вольно распоряжаться этим свойством, так как текстовые поля встречаются в интерфейсе часто и должны органично вливаться в общий стиль внешнего вида приложения. Экзотичные шрифты, особенно больших размеров, могут этому помешать |
| horizontalAlignment | Управляет выравниванием текста в поле. По умолчанию текст выравнивается по левой границе поля, вы можете выбрать для текста любой из трех вариантов выравнивания (слева, справа или по центру) |
| echoChar (только для класса JPasswordField) | Задает символ-заменитель для ввода секретной информации. По умолчанию используется символ звездочки (*) |

И еще одно замечание: если вы помните, в главе 7 при изучении блочного расположения BoxLayout мы столкнулись с неприятной проблемой, связанной с текстовым полем JTextField: при использовании менеджера расположения, учитывающего максимальный размер компонентов, поле «раздувается», так как максимальный размер его не ограничен. Свойства неограниченности максимального размера унаследовано классом JTextField от своего предка JTextComponent. Для всех остальных текстовых компонентов оно подходит как нельзя лучше — вы можете вводить в такие компоненты неограниченное количество символов текста (все упирается только в доступную вашему приложению память), но для однострочных полей JTextField и JPasswordField это не так. В рассмотренном нами примере использовалось последовательное расположение FlowLayout (в панелях JPanel оно устанавливается по умолчанию), в котором учитывается только предпочтительный размер компонента, так что с ним вопросов не возникает. Ну а трудности с другими менеджерами расположения мы решили в главе 7, написав для исправления размера текстового поля вспомогательный инструмент.

## Многострочное поле JTextArea

Многострочное текстовое поле JTextArea представляет собой немного более расширенную версию обычного текстового поля JTextField (хотя класс JTextArea не унаследован от класса JTextField, слишком уж различна их реализация). Как и обычное поле JTextField, многострочное поле JTextArea предназначено для ввода простого неразмеченного различными атрибутами текста, но в отличие от обычных полей, позволяющих вводить только одну строку текста, многострочные поля дают пользователю возможность вводить произвольное количество строк текста. Кроме того, поле JTextArea часто используется для вывода разного рода подробной информации о работе приложения, отображать которую в диалоговых окнах или в консоли неудобно. Вывод информации в многострочное текстовое поле позволяет пользователю без труда просмотреть ее, как бы много ее ни было, и при необходимости скопировать или изменить.

Работают с многострочными полями так же, как с обычными, за тем исключением, что для них приходится задавать не только ширину (максимальное количество символов), но и высоту (максимальное количество строк). Многострочные текстовые поля

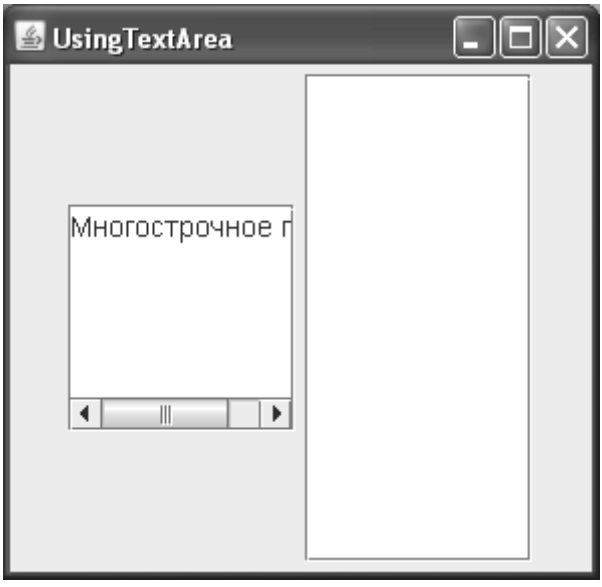
следует размещать в панелях прокрутки JScrollPane, иначе при вводе множества строк текста они «расползутся» по контейнеру и испортят вид остальных компонентов. Рассмотрим несложный пример и познакомимся с основными возможностями класса JTextArea:

```
// UsingTextArea.java
// Использование многострочных полей
import javax.swing.*;
import java.awt.*;

public class UsingTextArea extends JFrame {
  public UsingTextArea() {
    super("UsingTextArea");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем пару многострочных полей
    JTextArea area1 = new JTextArea(
        "Многострочное поле", 5, 10);
    // нестандартный шрифт и табуляция
    area1.setFont(new Font("Dialog", Font.PLAIN, 14));
    area1.setTabSize(10);
    JTextArea area2 = new JTextArea(15, 10);
    // параметры переноса слов
    area2.setLineWrap(true);
    area2.setWrapStyleWord(true);
    // добавим поля в окно
    setLayout(new FlowLayout());
    add(new JScrollPane(area1));
    add(new JScrollPane(area2));
    // выводим окно на экран
    setSize(400, 300);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new UsingTextArea(); } });
  }
}
```

В примере мы размещаем в окне пару многострочных текстовых полей JTextArea, для которых были изменены некоторые наиболее интересные свойства, чаще других использующиеся в приложениях. Первое текстовое поле создается с помощью довольно функционального конструктора, задающего для поля находящийся в нем текст, количество строк и символов. Обратите внимание, что количество строк идет в списке параметров перед количеством символов; здесь часто возникает путаница, так как для

однострочных текстовых полей мы сразу указываем количество символов, и нередко это правило ошибочно переносится на класс JTextArea. Задаваемые нами в конструкторе количества строк и символов поля определяют его первоначальный предпочтительный размер в контейнере, но не накладывают каких-либо ограничений на объем вводимого текста, который может быть произвольным. Для первого поля мы изменяем шрифт (здесь все аналогично однострочным полям) и задаем нестандартное значение для табуляции, вызывая метод setTabSize(). Данный метод позволяет указать, какое количество символов будет замещать символ табуляции (вставляемый нажатием клавиши Tab). По умолчанию это значение равно 8, но видно, что его можно изменить. Это может быть полезно в текстах с большим количеством отступов, например в текстах программ или в журнальных записях.

Второе текстовое поле создается с помощью конструктора, принимающего в качестве параметров только количества строк и символов; поначалу текста в таком поле не будет. Для второго поля мы меняем свойства, управляющие процессом переноса текста на новые строки, именно эти свойства наиболее полезны при работе с многострочными полями. По умолчанию текст в поле JTextArea не переносится на новую строку (после запуска программы с примером вы убедитесь в этом, взглянув на первое поле). Изменить данное поведение позволяют использованные нами в примере методы. Метод setLineWrap() включает автоматический перенос текста на новую строку. Длинные слова будут переноситься на следующие строки, так что в таком многострочном поле никогда не потребуется горизонтальная прокрутка. Метод setWrapStyleWord() изменяет стиль переноса длинных слов на новые строки. Если вы передадите в этот метод значение true, то слова, не умещающиеся в строке, будут целиком переноситься на строку новую. По умолчанию значение этого свойства равно false, это означает, что текст переносится, как только ему перестает хватать места в строке, независимо от того, в каком месте слова приходится делать перенос.




В заключение текстовые поля добавляются в панель содержимого окна, которое затем выводится на экран. Еще раз обратите внимание на то, что многострочные поля всегда нужно размещать в панелях прокрутки JScrollPane, иначе ваш интерфейс станет неконтролируемым (поскольку при вводе каждого нового символа сверх максимально-

го их количества по высоте и ширине размер полей будет увеличиваться). Более того, текстовое поле JTextArea не имеет собственной рамки и без панели прокрутки выглядит довольно бледно. В крайнем случае, если вам не хочется использовать панель прокрутки, вы можете вручную задать для него максимальный размер и рамку, но помните, что в таком случае пользователь не сможет увидеть текст, оказавшийся за пределами определенного вами прямоугольника.

Запустив программу с примером, вы увидите, как функционируют многострочные текстовые поля и как влияют на них сделанные нами изменения в их основных настройках. Обратите внимание на разницу в переносе текста: для переноса текста в первом поле приходится вручную нажимать клавишу Enter, а второе поле, когда вы станете набирать в нем текст (или добавите текст программно, с помощью метода setText()), выполнит перенос автоматически, да еще и по границе слов (хотя в нем также действует нажатие клавиши Enter).

Интересно, что для второго многострочного поля включенный режим переноса фактически отключает прокрутку по горизонтали. На снимке экрана хорошо видно, что при недостатке места полосы прокрутки все равно не меняются. На самом деле, зачем включать прокрутку, если мы сказали, что слова в любом случае будут переноситься на новую строку. Сама же панель прокрутки не станет меньше, чем изначально указанный для текстового поля размер на основании столбцов и строк. Этот факт следует учитывать при расположении компонентов в интерфейсе.

Напоследок сведем изученные нами свойства многострочных полей в табл. 16.2.

**Таблица 16.2.** Свойства многострочных текстовых полей

| Свойства (и методы get/set) | Описание |
|---|---|
| rows, columns | Задают размеры многострочного текстового поля, в строках и символах соответственно. Изменить размер поля можно и прямо во время работы программы, поле JTextArea при этом автоматически проведет проверку корректности и перерисовку контейнера |
| lineWrap, wrapStyleWord | Управляют включением переноса текста по строкам и типом этого переноса. Когда перенос строк отключен, второе свойство не действует. Если перенос строк включен и второе свойство равно false, перенос происходит в том месте, где заканчивается строка, независимо от того, в какой точке слова это случается. В противном случае перенос происходит по словам, которые не разбиваются на части, а переходят на новую строку целиком (если такая возможность есть и слово не слишком длинное) |
| font | Позволяет задать шрифт для многострочного текстового поля. Шрифт по умолчанию задается текущим менеджером внешнего вида и поведения Swing |
| tabSize | Контролирует размер символа табуляции (вставляется нажатием клавиши Tab) в текстовом поле. По умолчанию для символа табуляции используются 8 «обычных» символов, это наиболее часто употребляемое значение |
| lineCount, lineOfOffset, lineStartOffset, lineEndOffset (только методы get) | Данные методы позволяют получить исчерпывающую информацию о распределении текста многострочного поля по строкам. Первый метод get дает общее количество строк текста в поле. Второй метод предоставляет возможность узнать, на какой строке поля находится символ с данным смещением от начала текста. Последние два метода действуют обратным образом: для заданного номера строки они позволяют узнать смещение символа, находящегося в начале строки и в конце строки |

Помимо описанных в таблице свойств и методов многострочное поле JTextArea обладает еще парой весьма полезных методов. Метод append() позволяет присоединить к уже имеющемуся в поле тексту новую часть без удаления прежнего содержимого (в отличие от метода setText()). Метод insert() дает возможность вставить в произвольную область находящегося в поле текста новую строку. Оба этих метода избавляют нас от дополнительной «возни» с методом setText(), при использовании которого пришлось бы вручную манипулировать несколькими строками и преобразовывать их.

## Редактор JEditorPane

Редактор JEditorPane представляет собой мощный инструмент, способный отображать на экране текст любого формата. В данный момент библиотекой Swing поддерживается два широко распространенных формата: HTML и RTF (Rich Text Format – расширенный текстовый формат)[1], но потенциально редактор JEditorPane может отображать текст любого формата, с любыми элементами и любым оформлением. Такую гибкость редактору обеспечивает фабрика классов EditorKit, в обязанности которой входит создание и настройка всех объектов, необходимых для отображения текста некоторого типа[2], в том числе модели документа (объекта Document), фабрики для отображения элементов документа ViewFactory, курсора и списка команд, поддерживаемых данным типом текста. Фабрика EditorKit также отвечает за правильное открытие и сохранение документа поддерживаемого ею формата. Так что возможности JEditorPane ограничены лишь наличием фабрик для различных текстовых форматов. Поддерживаемые стандартно форматы RTF и HTML описываются фабриками RTFEditorKit и HTMLEditorKit соответственно.

Редактор JEditorPane, как правило, требуется именно для отображения текста поддерживаемого им формата, для редактирования текста пользователем с предоставлением последнему всех мыслимых возможностей (изменение стилей, вставка компонентов и значков, и пр.) лучше подходит текстовый компонент JTextPane. Класс JTextPane унаследован от JEditorPane, но гораздо удобнее для редактирования. Работа же с классом JEditorPane обычно происходит следующим образом. Вы методом setContentType() задаете, какой тип документа будет отображать редактор, и указываете местоположение документа, вызвав метод setPage() для перехода по URL-адресу документа (в этом случае считывать текст по указанному адресу будет фабрика EditorKit). Можно также самостоятельным считать текст документа и передать его в метод setText(). Редактор отобразит документ (если он был успешно считан) согласно его типу с помощью соответствующей этому типу фабрики EditorKit.

Для документов, которые поддерживают различного рода ссылки, редактор JEditorPane предоставляет слушателей HyperlinkListener. Эти слушатели оповещаются при активизации пользователем ссылки в документе. Слушателям передается URL-адрес активизированной ссылки, так что они могут заставить редактор немедленно перейти по этому адресу, вызвав все тот же метод setPage(), или использовать полученную информацию по своему усмотрению. Как и когда активизируются ссылки и что считается в документе ссылками, зависит, как нетрудно догадаться, от задействованной редактором фабрики EditorKit. Ссылки поддерживаются стандартной фабрикой HTMLEditorKit, необходимой для отображения HTML-документов, так что вы сможете сразу узнать, когда

[1] Создатели Swing честно реализовали все возможности, описанные в спецификациях форматов HTML 3.2 и RTF. К сожалению, время от времени Swing оказывается не в состоянии прочитать тот или иной документ в этих форматах: слишком уж много «недокументированных» дополнений и нестандартных возможностей встречается в документах, создаваемых наиболее популярными редакторами, не говоря уже об обилии Javascript. Если вам важна возможность считывать любые документы, придется многое доделать своими руками.

[2] Тип текста задается как строка MIME, например, для HTML – это text/html. Этой строке и сопоставляется подходящая фабрика EditorKit.

пользователь активизирует ссылку[3]. Есть только одно маленькое «но»: ссылки активизируются, только когда редактирование текста запрещено (свойство editable равно false).

Попробуем теперь с помощью редактора JEditorPane создать простой браузер для просмотра HTML-документов. С помощью адресной строки или ссылок в текущем документе можно будет переходить с одной страницы на другую :

```
// JEditorPaneBrowser.java
// Простой браузер на основе редактора JEditorPane
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

public class JEditorPaneBrowser extends JFrame {
  // наш редактор
  private JEditorPane editor;
  // текстовое поле с адресом
  private JTextField address;
  public JEditorPaneBrowser() {
    super("JEditorPaneBrowser");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем пользовательский интерфейс
    createGUI();
    // выводим окно на экран
    setSize(500, 400);
    setVisible(true);
  }
  // настройка пользовательского интерфейса
  private void createGUI() {
    // панель с адресной строкой
    JPanel addressPanel = new JPanel();
    addressPanel.setLayout(
        new FlowLayout(FlowLayout.LEFT));
    addressPanel.setBorder(BorderFactory.
        createEmptyBorder(5, 5, 5, 5));
    // поле для адреса
    address = new JTextField(30);
    // слушатель окончания ввода
    address.addActionListener(new NewAddressAction());
```

[3] Слушатели оповещаются о трех событиях, происходящих со ссылками. Самым популярным событием без сомнения является активизация ссылки, но вы также можете узнать о «входе» в область ссылки (например, когда пользователь наводит на ссылку указатель мыши) и «выходе» из этой области. Используя их, вы сможете оперативно обновлять, к примеру, строку состояния своего приложения, так что пользователь будет видеть, по какому адресу предлагает перейти ссылка.

```
    addressPanel.add(new JLabel("Адрес:"));
    addressPanel.add(address);
    // настраиваем редактор
    try {
      // пути к ресурсам нужно записывать
      // полностью, вместе с протоколами
      editor = new JEditorPane("http://java.sun.com");
    } catch (Exception ex) {
      JOptionPane.showMessageDialog(
          this, "Адрес недоступен");
    }
    editor.setContentType("text/html");
    editor.setEditable(false);
    // поддержка ссылок
    editor.addHyperlinkListener(new HyperlinkL());
    // добавляем все в окно
    add(addressPanel, "North");
    add(new JScrollPane(editor));
  }
  // слушатель, получающий уведомления о вводе нового адреса
  class NewAddressAction implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      // переходим по адресу
      String newAddress = address.getText();
      try {
        editor.setPage(newAddress);
      } catch (Exception ex) {
        JOptionPane.showMessageDialog(
            JEditorPaneBrowser.this,"Адрес недоступен");
      }
    }
  }
  // слушатель, обеспечивающий поддержку ссылок
  class HyperlinkL implements HyperlinkListener {
    public void hyperlinkUpdate(HyperlinkEvent he) {
      // нужный ли это тип события
      if ( he.getEventType() ==
          HyperlinkEvent.EventType.ACTIVATED ) {
        // переходим по адресу
        try {
          editor.setPage(he.getURL());
```

```
            } catch (Exception ex) {
              JOptionPane.showMessageDialog(
                  JEditorPaneBrowser.this,"Адрес недоступен");
            }
          }
        }
      }
      public static void main(String[] args) {
        SwingUtilities.invokeLater(
            new Runnable() {
              public void run() { new JEditorPaneBrowser(); } });
      }
    }
```

Наш браузер состоит из текстового поля, в котором пользователь будет набирать адрес для перехода, и редактора JEditorPane, предназначенного для отображения документа по указанному в поле адресу. Для начала необходимо создать нужные компоненты и расположить их в контейнере. Этим занимается метод createGUI(). Первым делом создается панель JPanel, в которой будет располагаться текстовое поле для ввода адреса. В ней мы используем менеджер последовательного расположения FlowLayout с выравниванием по левому краю, а также пустую рамку EmptyBorder, позволяющую визуально отделить содержимое панели от границ окна и улучшить восприятие интерфейса. В панель добавляется надпись, поясняющая функцию текстового поля, и само текстовое поле. Как видно, в нем будет 30 символов. К текстовому полю мы присоединяем слушателя окончания ввода, он реализован во внутреннем классе NewAddressAction (при описании текстовых полей мы отмечали в том числе и то, что в них полезно задействовать слушателей ActionListener.) Когда пользователь заканчивает ввод (нажав специальную клавишу, обычно Enter), вызывается метод слушателя actionPerformed(). В нем мы получаем текст, набранный в поле ввода адреса (это должен быть правильный URL-адрес, например такой: http://www.yahoo.com) и методом setPage() переводим редактор на новую страницу. В случае неудачи будет выведено краткое сообщение.

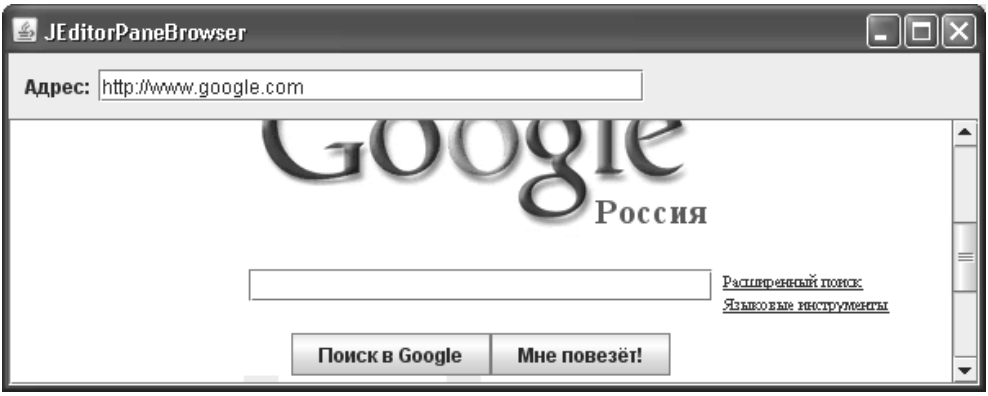
Теперь о настройке редактора JEditorPane. Для его создания мы выбираем конструктор, который позволяет сразу же задать документ для отображения (очень удобно для браузера: ведь в нем всегда есть домашняя страница). В качестве начальной страницы мы выбираем официальный сайт технологии Java, указывая ее полный адрес. Если вы захотите отобразить локальный файл, то это будет не так-то просто: методу setPage() требуется правильный URL-адрес, поэтому вам придется указать протокол доступа (file:)[4] и правильный путь к файлу. Процедура не слишком приятная, но о ней придется вспоминать каждый раз, когда вы будете работать с локальными документами и методом setPage() (на самом деле при работе с локальными документами лучше загрузить текст файла самому и изменять его методом setText(), но будем считать, что наш браузер ориентирован на работу в Web). Далее нужно указать, какие документы должен отображать редактор, так чтобы он выбрал и настроил подходящую фабрику EditorKit. Делается это методом setContentType(), у нас документы имеют тип «text/html». Для поддержки ссылок редактирование должно быть отключено, управляем этим свойство editable. Ну и напоследок мы присоединяем к редактору слушателя HyperlinkListener, который будет получать

[4] Символы «file:» используются для задания относительных путей, таких как images/picture.gif, а символы «file://» — для задания абсолютных.

информацию о событиях, происходящих со ссылками документа. Слушатель реализован во внутреннем классе HyperlinkL.

В метод hyperlinkUpdate() класса HyperlinkL приходит вся информация о происходящих со ссылками событиях. Нас интересует только активизация ссылки пользователем, в самом начале метода мы проверяем, подходит ли нам пришедшее событие. Если событием на самом деле являлась активизация ссылки, то мы получаем адрес активизированной ссылки методом getURL() и переводим редактор на новую страницу методом setPage(), передав ему полученный адрес. В случае ошибки на экран выводится краткое сообщение.

Созданные и настроенные компоненты добавляются в окно: панель с полем для ввода адреса размещается на севере, редактор, предварительно включенный в панель прокрутки, — в центре. Запустив программу с примером, вы сможете оценить, как компонент JEditorPane показывает HTML-документы, и увидеть, на что похожи ваши любимые страницы, отображенные средствами Swing.




Правда, выбирать страницы для просмотра нужно аккуратнее: ни языки сценариев, ни модули расширения, ни возможности HTML 4.0 и выше не поддерживаются, так что не удивляйтесь, если порядочная часть страниц нашему браузеру так и не поддастся. Впрочем, не все так плохо: поддержка HTML позволяет несколькими строками кода выводить правильно составленные документы и, что еще лучше, сохранять в этом формате документы, созданные вашими программами. Браузеры и веб-страницы — это отдельная история, и если уж вам потребуется выводить на экран любые страницы, попробуйте воспользоваться специальными компонентами JavaBeans от сторонних производителей. Еще одним вариантом может быть использование класса Desktop из пакета java.awt, предоставляющего доступ к системному браузеру. Получив активный экземпляр этого класса методом getInstance(), а затем вызвав метод browse(), вы сможете открыть для пользователя окно с системным браузером, используемым по умолчанию. Иногда это именно то, что нужно.

## Редактирование по максимуму — компонент JTextPane

Если компонент JEditorPane используется в основном для статичного отображения уже созданных текстовых документов некоторого формата (заданного фабрикой EditorKit), то унаследованный от него текстовый компонент JTextPane незаменим при создании в приложении многофункционального текстового редактора. Обладая всеми впечатляющими возможностями своего предка JEditorPane, класс JTextPane добавляет к нему то, без чего представить себе современные редакторы практически невозможно — разметку текста

стилями. Для этого в нем используется специальная модель документа StyledDocument и настроенная на поддержку такой модели фабрика классов StyledEditorKit.

Концепция стиля в текстовом редакторе очень проста, тем не менее она многократно повышает эффективность работы пользователя с текстом. *Стиль* (style) — это некоторый набор атрибутов редактируемого текста (такими атрибутами могут быть шрифт текста, его размер и цвет, выравнивание и т. п.), который можно применить к любому фрагменту текста. Стиль позволяет четко разделить внешний вид документа и собственно текст. Пользователь может сосредоточиться на наборе текста, используя при этом несколько стилей (пара заголовков, основной текст, текст сносок), а если у него возникнет необходимость оформить текст по-другому, понадобится лишь поменять атрибуты стилей. Текст, набранный с использованием стилей, изменится автоматически. Вы найдете стили в любом современном редакторе, в том числе они встроены в язык разметки HTML посредством расширений CSS (Cascaded Style Sheets — каскадные таблицы стилей).

Помимо стилей (в компоненте JTextPane стили идентифицируются строковыми именами) поддерживаются и просто неупорядоченные наборы атрибутов текста, заданные объектами AttributeSet. Наборы атрибутов позволяют изменять произвольные фрагменты текста или даже целые абзацы, слегка отступая от комплекта заранее заданных стилей. Это также очень полезная возможность, хотя всегда лучше создавать свой уникальный стиль на каждый отдельный формат текста.

Одной из самых полезных является способность стилей образовывать иерархии. Одни стили могут наследовать атрибуты других стилей, прибавляя при этом что-то свое. К примеру, стиль для основного текста может определять шрифт, его размер и выравнивание, а стиль для заголовка, унаследованный от этого базового стиля, может поменять только размер шрифта, не меняя остальное. Иерархия стилей дает пользователю возможность еще быстрее настраивать внешний вид и формат документа, не затрагивая текста: сменив шрифт в основном стиле, пользователь сменит его и для всех унаследованных от него стилей.

Рассмотрим пример, в котором применим часть впечатляющих возможностей компонента JTextPane, в том числе наборы атрибутов и именованные стили. После разбора примера мы сможем в деталях понять, как работают механизмы класса JTextPane:

```
// StyledText.java
// Богатые возможности редактора JTextPane
import javax.swing.*;
import javax.swing.text.*;
import java.awt.*;

public class StyledText extends JFrame {
  public StyledText() {
    super("StyledText");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создадим редактор
    JTextPane textPane = new JTextPane();
    // создание документа и стилей
    createDocument(textPane);
    // добавим редактор в окно
    add(new JScrollPane(textPane));
    // выводим окно на экран
```

```
    setSize(400, 300);
    setVisible(true);
  }
  private void createDocument(JTextPane tp) {
    // настройка стилей
    // стиль основного текста
    Style normal = tp.addStyle("Normal", null);
    StyleConstants.setFontFamily(normal, "Verdana");
    StyleConstants.setFontSize(normal, 13);
    // заголовок
    Style heading = tp.addStyle("Heading", normal);
    StyleConstants.setFontSize(heading, 20);
    StyleConstants.setBold(heading, true);
    // наполняем документ содержимым, используя стили
    insertString("Незамысловатый Заголовок", tp, heading);
    insertString("Далее идет обычное содержимое,", tp, normal);
    insertString("помеченное стилем Normal.", tp, normal);
    insertString("Еще Один Заголовок", tp, heading);
    // меняем произольную часть текста
    SimpleAttributeSet red = new SimpleAttributeSet();
    StyleConstants.setForeground(red, Color.red);
    StyledDocument doc = tp.getStyledDocument();
    doc.setCharacterAttributes(5, 5, red, false);
    // добавим компонент в конец текста
    tp.setCaretPosition(doc.getLength());
    JCheckBox check = new JCheckBox("Все возможно!");
    check.setOpaque(false);
    tp.insertComponent(check);
  }
  // вставляет строку в конец документа с переносом,
  // используя заданный стиль оформления
  private void insertString(String s, JTextPane tp, Style style) {
    try {
      Document doc = tp.getDocument();
      doc.insertString(doc.getLength(), s + "\n", style);
    } catch (BadLocationException ex) {
      ex.printStackTrace();
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
```

```
            new Runnable() {
              public void run() { new StyledText(); } });
    }
}
```

В примере мы создаем текстовый редактор JTextPane, который разместится в центре нашего окна с рамкой (редактор предварительно размещен в панели прокрутки JScrollPane). Самые захватывающие события происходят в методе createDocument(), в котором мы создаем стили и наполняем документ содержимым.

Сначала создаются два именованных стиля, которые мы затем будем использовать при разметке своего текста. Посмотрите, как это делается: сначала создается базовый для всего документа стиль (с названием Normal). Для добавления стиля в редактор применяется метод с говорящим названием addStyle()[5]. Ему нужно передать два параметра: название нового стиля и стиль, который будет являться родительским для нового стиля, а в ответ он вернет новый стиль Style. Если в качестве второго параметра передать пустую ссылку null, стиль окажется без родителя и будет создаваться «с нуля». Именно таким образом мы создаем базовый стиль своего документа . Второй стиль в нашем документе (у него название Heading) служит для заголовков. Он унаследован от стиля Normal, а значит, перенимает от своего родителя все установленные для того атрибуты, такие как размер и цвет шрифта. Стиль заголовка отличается увеличенным размером шрифта и полужирным начертанием. Обратите внимание, что для установки атрибутов стилей используются удобные статические методы класса StyleConstants.

После создания стилей можно приступать к вставке в документ текста, размеченного только что настроенными стилями. Вставить текст в компонент JTextPane можно только посредством модели документа Document, в которой имеется метод insertString(). Этот метод требует трех параметров: позицию для вставки, строку, которую вы собираетесь вставлять, и стиль, который будет иметь вставляемая строка. У нас вставкой текста в документ занимается собственный вспомогательный метод insertString(). В нем строка добавляется к концу документа и при этом дополняется символами переноса строки (их приходится указывать вручную, автоматически они не добавляются). Для вставки текста в конец документа в качестве первого параметра метода insertString() необходимо указать размер документа, который несложно узнать с помощью метода getLength(). Кстати, приходится следить за исключением, которое не преминет возникнуть в том случае, если позиция в документе будет неверно указана[6]. С помощью вспомогательного метода мы добавляем в документ текст: сначала заголовок, потом несколько строк с обычным стилем и снова заголовок. При этом используются созданные нами стили.

Далее демонстрируется, как можно изменять оформление произвольного фрагмента текста, каким бы стилем он ни был размечен. Для этого предназначен метод setCharacterAttributes(), которому необходимо указать диапазон в тексте, набор атрибутов, а также булево значение. Последнее указывает, нужно ли полностью заменить имеющийся стиль новым набором атрибутов, или надо совместить имеющийся стиль с новым

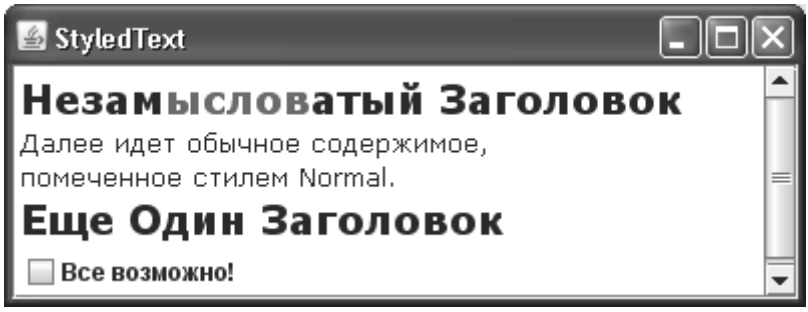
---

[5] Такой же метод имеется и в модели StyledDocument. На самом деле практически все методы, определенные в классе JTextPane и так или иначе манипулирующие текстом, его атрибутами и стилями, определены в модели документа StyledDocument. В больших приложениях лучше работать с моделью напрямую, мы прекрасно знаем, сколько преимуществ это несет.

[6] Случай с текстовыми компонентами и работой с их текстом как раз показывает, как иногда бывают надоедливы проверяемые (checked) исключения. Как правило, манипулируя текстом, мы осознаем его границы, и здесь вполне хватило бы непроверяемого исключения, которое не обязательно включать в блок try.

набором. Для хранения набора атрибутов мы применяем класс SimpleAttributeSet[7], в котором указываем, что текст должен иметь красный цвет. Передав данный набор в метод и отказавшись от полной замены имеющегося стиля (третий параметр равен false), мы окрасим часть заголовка в красный цвет, не меняя остальных его атрибутов, подобных размеру и шрифту.

Напоследок мы задействуем кое-что из экзотических возможностей редактора JTextPane: добавляем в конец документа самый настоящий компонент — флажок JCheckBox. Предварительно флажок делается прозрачным (свойство opaque устанавливается в false), в противном случае он будет закрашивать свою область цветом фона и выпадать из общей картины документа. Также приходится переместить курсор в конец документа методом setCaretPosition(), иначе флажок появится вместо конца документа в его начале. Ну а сама вставка компонента на текущую позицию (указываемую курсором) выполняется методом insertComponent().




Запустив программу с примером, вы увидите, как отображается на экране размеченный разнообразными стилями документ. Легко видеть, что текстовый редактор JTextPane как нельзя лучше подходит там, где нужны многофункциональные мощные текстовые возможности. С его помощью вы сможете управлять стилями, обновлять произвольные фрагменты текста, вставлять в текст компоненты и значки, причем сделать все это будет совсем несложно. Для компонента JTextPane практически нет невозможного: любые ваши запросы он сможет удовлетворить, просто придется провести немного больше исследований и работы.

## Форматированный вывод — компонент JFormattedTextField

Поле для вывода данных в определенном формате JFormattedTextField унаследовано от обычного текстового поля JTextField, оно позволяет выводить данные согласно специальным текстовым форматам и маскам, а также ограничивает ввод пользователя, разрешая тому вводить данные только в соответствии с заданным в поле форматом. Необходимость в таком текстовом компоненте имелась давно, и он мгновенно стал популярным участником интерфейсов. Ничего удивительного в этом нет: ограничения на вводимые данные встречаются очень часто, можно вспомнить хотя бы поле для ввода телефонных номеров. До появления компонента JFormattedTextField реализация даже такого простого поля превращалась в порядочную головную боль.

[7] Набор атрибутов описывается интерфейсом AttributeSet, класс SimpleAttributeSet — самая простая реализация этого интерфейса, позволяющая быстро настроить набор. Кстати, стили по сути также являются наборами атрибутов: класс стилей Style реализует интерфейс AttributeSet, отличие лишь в том, что стиль обладает собственным уникальным именем.

Отличие между полем для форматированного вывода данных и обычным текстовым полем заключается в наличии в поле JFormattedTextField особого форматирующего объекта, который должен быть унаследован от абстрактного внутреннего класса AbstractFormatter. Когда пользователь проводит в текстовом поле какие-либо действия — набирает новый символ, удаляет фрагмент текста, отменяет операцию — вызывается форматирующий объект. В его задачу входит анализ значения, которое оказывается в текстовом поле, и принятие решения о соответствии этого значения некоторому формату. Основными составляющими форматирующего объекта являются фильтр документа (DocumentFilter), который принимает решение, разрешать или нет очередное изменение в документе, а также навигационный фильтр (NavigationFilter). Навигационный фильтр получает исчерпывающую информацию о перемещениях курсора в текстовом поле и способен запрещать курсору появляться в некоторых областях документа (таких как разделители номеров, дат, и других данных, которые не должны редактироваться). Форматирующий объект также ответственен за действие, которое предпринимается в случае ввода пользователем неверного значения (по умолчанию раздается звуковой сигнал).

Написав свой форматирующий объект (унаследовав его от абстрактного класса AbstractFormatter), вы сможете от начала и до конца следить за процессом редактирования в текстовом поле JFormattedTextField, организуя ограниченный ввод самых сложных данных. Но главное преимущество компонента JFormattedTextField заключается не в этом, а в наличии нескольких уже готовых форматирующих объектов, способных помочь в организации эффективного ввода самых разнообразных данных. В табл. 16.3 представлен краткий перечень стандартных объектов (все описываемые форматирующие объекты находятся в пакете javax.swing.text).

**Таблица 16.3.** Стандартные форматирующие объекты для JFormattedTextField

| Форматирующий объект | Описание |
|---|---|
| MaskFormatter | Организует ввод данных на основе простой маски: набора специальных символов, задающих символы, которые допустимы на определенных позициях документа. Подробное описание масок вы найдете в интерактивной документации данного класса, а простой пример мы вскоре увидим. Вместо маски или в дополнение к ней можно указывать набор символов, которые разрешено добавлять в документ |
| DateFormatter | Позволяет редактировать даты в любом удобном вам (или пользователю ваших приложений) формате. Для форматирования дат используется класс DateFormat из пакета java.text. Довольно удобен, так как позволяет быстро локализовать ввод дат для любого поддерживаемого JDK языка (а в современных выпусках JDK поддерживаются практически все языки мира). Формат для отображения даты вы сможете задать в конструкторе форматирующего объекта или поменять прямо во время работы программы. Впрочем, современные интерфейсы предполагают применение для выбора дат всплывающих диалогов, реализацию такого диалога можно найти в библиотеке дополнений SwingX |
| NumberFormatter | Рассматривает вводимые пользователем значения как числа, записанные в определенном формате. Для форматирования чисел использует класс NumberFormat из пакета java.text. Также как и предыдущий класс, позволяет легко локализовать приложение и настроить формат прямо во время работы программы |

Все стандартные форматирующие объекты унаследованы от класса DefaultFormatter, реализующего основные функции базового класса AbstractFormatter. Как правило, ваша работа заключается именно в настройке одного из доступных форматирующих объек-

тов и подключении его к полю JFormattedTextField. Создание собственных форматирующих объектов — довольно кропотливая задача. Зачастую проще создать собственную модель документа Document (мы ее вскоре обсудим).

Кроме того, что вы можете указать форматирующий объект, который следует использовать в текстовом поле JFormattedTextField, имеется еще одна возможность настраивать форматирование. Фабрика AbstractFormatterFactory, описанная, как и форматирующий объект AbstractFormatter, в виде абстрактного внутреннего класса, служит для создания форматирующих объектов для текстовых полей JFormattedTextField. Если вы создадите поле JFormattedTextField, передав в конструктор фабрику форматирующих объектов, то поле при редактировании будет каждый раз запрашивать ее для получения форматирующего объекта. Это дает дополнительную гибкость: вы можете возвращать различные форматирующие объекты, в зависимости от того, что в данный момент находится в текстовом поле. В пакете javax.swing.text имеется стандартная реализация фабрики DefaultFormatterFactory. С ее помощью вы сможете задать различные форматирующие объекты для трех ситуаций: когда поле не пусто и обладает фокусом ввода, когда поле не пусто и не обладает фокусом ввода, и, наконец, когда поле пусто. Хотя фабрика форматирующих объектов и предоставляет дополнительную гибкость, чаще всего достаточно одного форматирующего объекта «на все случаи жизни».

А теперь рассмотрим пример, в котором используем все три стандартных форматирующих объекта, поставляемых вместе со Swing, а заодно применим некоторые наиболее полезные свойства текстового поля JFormattedTextField:

```
// FormattedFields.java
// Применение полей JFormattedTextField
import javax.swing.*;
import javax.swing.text.*;
import java.text.*;
import java.util.Date;
import java.awt.*;

public class FormattedFields extends JFrame {
  // поля для форматированного ввода данных
  private JFormattedTextField
      phoneField, dateField, numberField;
  public FormattedFields() {
    super("FormattedFields");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // ограниченный ввод на основе маски
    // телефонный номер
    try {
      MaskFormatter phone =
          new MaskFormatter("+#-###-###-##-##");
      phone.setPlaceholderCharacter('0');
      phoneField = new JFormattedTextField(phone);
      phoneField.setColumns(15);
    } catch (Exception ex) {
      ex.printStackTrace();
```

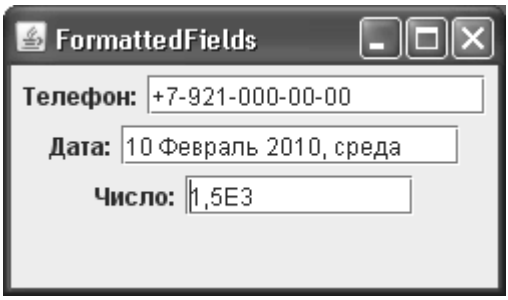
```
    }
    // редактирование даты
    // формат даты
    DateFormat date =
        new SimpleDateFormat("dd MMMM yyyy, EEEE");
    // настройка форматирующего объекта
    DateFormatter formatter = new DateFormatter(date);
    formatter.setAllowsInvalid(false);
    formatter.setOverwriteMode(true);
    // настройка текстового поля
    dateField = new JFormattedTextField(formatter);
    dateField.setColumns(15);
    dateField.setValue(new Date());
    // редактирование чисел
    // формат числа с экспонентой
    NumberFormat number = new DecimalFormat("##0.##E0");
    numberField = new JFormattedTextField(
        new NumberFormatter(number));
    // настройка поля
    numberField.setColumns(10);
    numberField.setValue(1500);
    // добавляем поля в панель содержимого
    setLayout(new FlowLayout());
    add(new JLabel("Телефон:"));
    add(phoneField);
    add(new JLabel("Дата:"));
    add(dateField);
    add(new JLabel("Число:"));
    add(numberField);
    // выводим окно на экран
    setSize(400, 300);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new FormattedFields(); } });
  }
}
```

Мы создаем небольшое окно с рамкой, в нем разместятся три текстовых поля для вывода форматированных данных JFormattedTextField. Каждое из полей продемонстрирует работу одного из стандартных форматирующих объектов.

Первое поле применяет в качестве форматирующего объекта MaskFormatter, данный объект позволяет организовать ввод данных на основе особой маски, составленной из специальных символов. Подробное описание масок, как мы уже отмечали, находится в интерактивной документации. В примере маска составляется для ввода расширенного телефонного номера: с кодом страны и города. Обратите внимание, как в маске задаются десятичные числа — они обозначаются символом #. Остальные символы, использованные при создании объекта MaskFormatter, обычные, а это означает, что редактировать их пользователю будет нельзя, так как они служат разделителями данных. Объект MaskFormatter приходится создавать в блоке try/catch — если при анализе маски будет обнаружена ошибка синтаксиса, возникнет исключение. Помимо собственно создания форматирующего объекта мы настраиваем некоторые дополнительные свойства. Свойство placeholderCharacter отвечает за то, какой символ будет заменять незаполненные пользователем части маски. По умолчанию используется пробел, но для нашей маски он не очень хорошо подходит, потому что размер текстового поля с пробелами намного меньше того же текстового поля, но уже заполненного цифрами. К тому же по маске, заполненной пробелами, трудно определить, что в ней требуется записывать цифры, а не что-либо иное. Поэтому мы заменяем пробел символом нуля. Более того, мы увеличиваем размер текстового поля до 15 символов. Поле с избыточным количеством символом смотрится лучше, а за пределы маски пользователю все равно выходить запрещено (за этим следит форматирующий объект).

Второе поле служит для набора дат, заданных в определенном формате. Формат даты задается объектом DateFormat. В классе DateFormat имеются несколько статических методов, позволяющих создавать стандартные форматы дат, принятых в различных странах, однако мы настраиваем формат даты по своему вкусу, создавая объект SimpleDateFormat. Он позволяет указать формат даты с помощью несложной текстовой маски; полное описание правил создания таких масок вы найдете в документации класса SimpleDateFormat. Наш формат для даты содержит число месяца (за него отвечают символы маски «dd»), полное название месяца («MMMM»), четырехзначные число года и название дня недели («yyyy» и «EEEE», соответственно). Созданный формат даты мы передаем в форматирующий объект DateFormatter, после чего настраиваем для него несколько дополнительных свойств. В отличие от объекта MaskFormatter форматирующий объект для дат по умолчанию разрешает ввод неверных (не соответствующих формату даты) значений. Мы запрещаем ввод таких значений, изменяя свойство allowsInvalid. Для удобства изменения даты мы также включаем режим перезаписи значений (по умолчанию работает режим вставки), применяя свойство overwriteMode. После настройки форматирующий объект передается в текстовое поле, размер которого мы также увеличиваем до 15 символов. Обратите внимание, как задается новое значение, которое будет отображаться текстовым полем: в метод setValue() мы передаем объект Date, инкапсулирующий дату. За преобразование этого объекта в текст отвечает как раз форматирующий объект DateFormatter.

Наконец, третье поле JFormattedTextField настраивается для ввода чисел в определенном формате. Формат чисел определяется объектом NumberFormat. Как и в случае с объектом для формата дат DateFormat, в нем имеется несколько статических методов для получения стандартных форматов чисел различных стран и языков, однако мы применим для создания формата числа объект DecimalFormat. Он позволяет настраивать формат числа с помощью несложной текстовой маски. В примере мы настроили маску для целых чисел в экспоненциальном формате: две десятичных цифры в самом числе, две возможных десятичных цифры в мантиссе. Число и мантисса разделяются точкой и заканчиваются нулями. Подробное описание маски для определения формата числа вы найдете в документации класса DecimalFormat. Созданный формат числа присоединяется к форматирующему объекту NumberFormatter, а тот, в свою очередь, передается текстовому полю. Для текстового поля мы немного увеличиваем размеры (увеличиваем количество столбцов).




После настройки всех трех текстовых полей JFormattedTextField они добавляются в окно с последовательным расположением FlowLayout. Каждое поле снабжается небольшой поясняющей надписью. Запустив программу с примером, вы увидите поле JFormattedTextField в действии и сможете оценить достоинства и недостатки ввода ограниченных данных. Поле, действующее на основе маски, наиболее удобно и интуитивно понятно для пользователя, поле для ввода чисел также довольно удобно, хотя неожиданная смена записи числа, введенного в обычной форме, на запись экспоненциальную, иногда может быть некстати, и об этом пользователя стоит предупреждать. А вот поле для ввода дат действует не лучшим образом: если ввести новую дату или год еще возможно, то название месяца или дня недели поменять одним символом нельзя. К тому же при изменении дат или года любое случайно введенное значение может привести к скачкам на десятки месяцев, а то и лет. Так что для ввода дат гораздо лучше применять уже изученные нами в главе 12 счетчики JSpinner. Как мы помним, для вывода дат они применяют именно поля JFormattedTextField, так что вместо самостоятельной настройки поля JFormattedTextField для вывода даты вы сможете настроить поле счетчика, а для выбора собственно даты применить сам счетчик. Еще лучше для выбора дат использовать отдельные всплывающие окна. Ну а самым удобным и полезным (и к тому же простым) форматирующим объектом без сомнения является MaskFormatter.

## *Модель документа Document*

Как и у всех достаточно сложных компонентов библиотеки Swing, у текстовых компонентов есть *модель*. Именно в модели хранятся данные, отображаемые текстовыми компонентами; нетрудно догадаться, что в качестве данных выступает текст, набранный пользователем или вставленный программно. Модель всех текстовых компонентов описывается не слишком сложным интерфейсом Document из пакета javax.swing.text. Поддержка интерфейса Document встроена в базовый класс всех текстовых компонентов JTextComponent библиотеки Swing, так что во всех текстовых компонентах вы сможете манипулировать данными посредством модели или менять саму модель (модель хранится в свойстве document). Для манипуляции текстом применяются несколько методов интерфейса Document, перечисленных в табл. 16.4.

**Таблица 16.4.** Методы модели Document, служащие для манипуляции текстом

| Метод | Описание |
|---|---|
| getText(позиция, длина) | Позволяет получить фрагмент текста, заданный начальной позицией и длиной. Позиция должна быть не меньше нуля и не больше длины текста, иначе возникнет исключение. Длина получаемого фрагмента также не должна выходить за пределы документа |

**Таблица 16.4** *(продолжение)*

| Метод | Описание |
|---|---|
| insertString(позиция, текст, атрибуты) | Вставляет на произвольную позицию документа текст, который к тому же может быть снабжен набором некоторых атрибутов (о них чуть позже). Позиция должна укладываться в пределы имеющегося текста. Именно этот метод вызывается при любых дополнениях в текстовом документе, так что его можно использовать для анализа и модификации текста перед тем как он попадет в текстовый компонент |
| remove(позиция, длина) | Удаляет из документа фрагмент текста, заданный позицией и длиной. Позиция и длина фрагмента должны укладываться в пределы текста |
| getLength() | Позволяет получить длину текста, хранимого в данный момент в модели документа |

Модель документа Document не только хранит простой текст, но и позволяет сопоставлять ему наборы атрибутов AttributeSet (к атрибутам текста относится, например, шрифт и размер шрифта, цвет текста и т. п.). Каким образом атрибуты текста хранятся и задействуются в модели документа, зависит исключительно от ее реализации. Например, модель, используемая в текстовых полях, не сохраняет эти атрибуты, так что весь текст прорисовывается в едином виде. Модель, применяемая в редакторе JTextPane, напротив, тщательно сохраняет атрибуты текста и позволяет выводить текст в различном начертании.

Помимо текста и назначенных ему атрибутов модель Document позволяет определять, как текст распределен по элементам. *Элемент документа*, который может содержать произвольный фрагмент текста, размеченный различными атрибутами, описывает какую-то единую логическую единицу текста, например, абзац, маркированный список или таблицу. Таким образом, элементы документа, представленные интерфейсом Element, позволяют описывать структуру текста. Элементы могут находиться в иерархии. В простых текстовых полях элементами являются строки текста, в сложных редакторах — абзацы, находящиеся в отношениях «родитель-потомок». Методы, предназначенные для получения корневых элементов модели, перечислены в табл. 16.5.

**Таблица 16.5.** Методы модели Document для получения элементов документа

| Метод | Описание |
|---|---|
| getDefaultRootElement() | Возвращает основной корневой элемент документа. С его помощью можно получить элементы, являющеюся потомками данного элемента, и таким образом полностью раскрыть структуру документа. Основной корневой элемент обычно вмещает в себя весь текст документа |
| getRootElements() | Позволяет получить массив корневых элементов документа. Как правило, корневой элемент в документе один (именно он возвращается предыдущим методом), так что массив будет состоять из одного элемента. Впрочем, иногда документ предполагает наличие нескольких корневых элементов для специализированных целей, для таких случаев и предназначен данный метод |

Для того чтобы вид (текстовый компонент) смог вовремя узнавать об изменениях в тексте или структуре модели документа Document и перерисовать себя для отображения этих изменений, необходим механизм оповещения об изменениях. Данный механизм реализуется событием DocumentEvent и его слушателем DocumentListener, которого вы сможете присоединить к модели Document. Событие DocumentEvent запускается каж-

дый раз при изменении текста документа, именно его следует обрабатывать, если вы заинтересованы в отслеживании каждого изменения текста. В интерфейсе слушателя DocumentListener определены три метода; каждый из них вызывается при конкретном типе события в документе: удалении, обновлении или вставке текста.

Все модели документов, применяемые различными текстовыми компонентами Swing, унаследованы от базового класса AbstractDocument, реализующего механизм обновления текста, безопасный с точки зрения работы нескольких потоков. Достигается это использованием в классе внутренней синхронизации при изменении и чтении текста. Проще говоря, это означает, что вы можете получать текст документа или изменять его из любого потока выполнения, даже если это не поток рассылки событий EventDispatchThread. Тем самым и отличается работа с текстовыми компонентами от работы с остальными компонентами Swing: если для основной массы компонентов Swing действует правило «одного потока» (все действия с компонентом должны выполняться из потока рассылки событий, мы обсудили данное правило в главе 3), то текстовые компоненты (точнее, операции с хранимым в них текстом) освобождены от него. Сделано это не случайно: работа с текстом почти всегда подразумевает, во-первых, загрузку текста из различных источников данных. Загрузка может занимать много времени, особенно если текст велик, а пользователю зачастую хочется сразу видеть, что же он загружает. Благодаря механизму синхронизации класса AbstractDocument загрузку можно проводить из отдельного потока, и пользователь будет видеть текст уже в процессе загрузки. Во-вторых, текст довольно часто приходится анализировать непосредственно во время его набора пользователем, например чтобы вывести подсказки для пользователя или провести автоматическое форматирование. Без поддержки синхронизации в текстовых компонентах проводить такой анализ было бы практически невозможно.

Модели для разнообразных компонентов Swing очень часто приходится создавать и программистам-клиентам (с точки зрения таких программистов мы и рассматриваем библиотеку), реализуя соответствующие интерфейсы и описывая данные в нескольких методах этих интерфейсов. Поставляемые со Swing стандартные модели применяются реже, хотя и требуют меньше усилий. Текстовые компоненты и здесь отличаются от своих собратьев. Модель документа Document не так сложна, чтобы ее нельзя было реализовать самому, однако поддержка этой модели, создание подходящей структуры элементов и ее отображение могут потребовать многих нетривиальных действий. Поэтому обработка текста обычно сводится к работе со стандартными моделями и фабриками классов — они достаточно гибкие, чтобы обеспечить программиста нужной информацией о документе и тексте, и поддерживают большую часть необходимых операций.

## Текстовое поле с автоматическим заполнением

Используем полученные только что знания о модели документа Document для реализации эффектного участника пользовательских интерфейсов — текстового поля с автоматическим заполнением. Такое текстовое поле следит за изменениями в своем тексте и постоянно анализирует их, определяя, когда пользователь набирает некоторую ключевую комбинацию. Когда такая комбинация обнаруживается, поле предлагает завершить ее автоматически, подставляя в документ соответствующий текст и выделяя его. Пользователь решает, подходит ли ему предлагаемый вариант.

В текстовых полях по умолчанию применяется стандартная упрощенная модель документа под названием PlainDocument. Логично унаследовать от нее и переопределить метод, в который приходит любой новый текст, и в нем, прежде чем действительно добавлять текст в модель, анализировать и дополнять его. Для того чтобы было удобно использовать новую модель документа в других программах, мы снабдим его полезным программным интерфейсом и разместим в библиотечном пакете com.porty.swing. Вот что у нас получится:

```
// com/porty/swing/AutoCompleteTextDocument.java
// Модель документа с поддержкой автозаполнения
package com.porty.swing;

import javax.swing.text.*;
import javax.swing.*;
import java.util.*;

public class AutoCompleteTextDocument extends PlainDocument {
  // текстовый компонент в котором работает документ
  private JTextComponent comp;
  // список слов для автозаполнения
  private List<String> words = new ArrayList<String>();
  // конструктор требует текстовый компонент
  public AutoCompleteTextDocument(JTextComponent comp) {
    this.comp = comp;
    comp.setDocument(this);
  }
  // добавляет слово в список
  public void addWord(String word) {
    words.add(word);
  }
  // свойство, управляющее началом автозаполнения
  private int beforeCompletion = 3;
  public void setBeforeCompletion(int value) {
    beforeCompletion = value;
  }

  // вызывается при вставке в документ нового текста
  @Override
  public void insertString(int offs, String str, AttributeSet a)
      throws BadLocationException {
    // текущая позиция в тексте
    int end = offs + str.length();
    // определяем позиции текущего слова
    final int wordStart = Utilities.getWordStart(comp, offs);
    // длина текущего слова
    int wordLength = end — wordStart;
    // проверим, можно ли завершать слово
    if ( wordLength >= beforeCompletion) {
      // получаем текущее слово
```

```
        String word = getText(wordStart, offs - wordStart) + str;
        // пытаемся найти его полный вариант в списке
        String wholeWord = "";
        for (String next : words) {
          if (next.startsWith(word)) {
            // слово найдено
            wholeWord = next;
            break;
          }
        }
        // если слово найдено
        if ( wholeWord.length() > 0) {
          // вырезаем часть для автозаполнения
          final String toComplete =
              wholeWord.substring(wordLength);
          // позиции для выделения этой части
          final int startPos = offs + str.length();
          final int endPos = end + toComplete.length();
          // добавляем добавку к тексту
          str = str + toComplete;
          // отложенная задача для выделения добавки
          SwingUtilities.invokeLater(new Runnable() {
            public void run() {
              try {
                // выделим добавленную часть
                comp.setSelectionStart(startPos);
                comp.setSelectionEnd(endPos);
              } catch (Exception ex) {
                ex.printStackTrace();
              }
            }
          });
        }
      }
      // родительский метод добавит текст
      super.insertString(offs, str, a);
    }
  }
```

Наша новая модель документа унаследована от стандартной модели текстовых полей и отличается парой вспомогательных методов, позволяющих быстро настроить режим работы механизма автоматического заполнения. Конструктор требует передать модели текстовый компонент, в котором будет работать модель — как мы увидим

чуть позже, это необходимо нам для выделения текста. Заодно он сразу же присоединяет модель к этому компоненту методом setDocument(). Новые методы позволяют контролировать автоматическое заполнение: метод addWord() добавляет в список слов новое слово; именно список слов words содержит слова, которые могут быть дополнены автоматически, а свойство beforeCompletion (и метод set) определяет, какое количество букв в слове должен набрать пользователь, прежде чем слово будет анализироваться на предмет автоматического дополнения. По умолчанию свойство beforeCompletion равно трем.

Для автоматического заполнения потребуется метод, который вызывается при вставке в документ нового содержимого (это метод insertString()); именно при вставке (наборе) нового текста пользователь может ожидать подсказки от текстового поля. Удаление содержимого, как правило, не должно сопровождаться автоматическим заполнением, это может привести к путанице и излишне усложнить работу с полем. Замена же текста сводится к поэтапному удалению фрагмента текста и вставке нового текста на место старого. Она вполне подходит нашему алгоритму.

Первым делом в методе insertString() мы получаем необходимую для дальнейшей работы текущую позицию курсора в документе end (получить ее можно, суммируя смещение вставки в документе, переданное в метод как параметр offs, и длину нового текста для вставки). Далее нам необходимо определить, какое слово находится в данный момент на текущей позиции курсора. Здесь нам неоценимую услугу окажут статические методы класса Utilities[8]. Метод getWordStart() позволяет получить позицию начала слова. В качестве параметров им требуется передать текстовый компонент, в котором находится анализируемый текст, и позицию, с которой надо начинать поиск слова. В нашем случае позиция известна: это позиция начала вставки, переданная нам параметром. После получения позиции текущего слова мы проверяем, можно ли начинать его дополнение, то есть достаточна ли длина этого слова (она должна быть не меньше свойства beforeCompletion). Если слово недостаточно длинное, текст вставляется без изменений.

В случае если слово имеют достаточную длину, мы начинаем его поиск в списке слов, которые можно дополнять. Для этого мы перемещаемся по списку со словами words, проверяя, не начинается ли некое слово с набранных пользователем символов. Если такое слово обнаруживается в списке, оно сохраняется в переменной wholeWord, после чего поиск заканчивается. Если слово не обнаруживается, работа заканчивается. После обнаружения подходящего слова остается получить ту его часть, которую необходимо предложить пользователю в качестве подсказки, вставить эту часть в документ, а затем выделить ее.

В нашем примере вставка текста в документ очень проста, поскольку мы и являемся документов — мы просто добавляем часть для дополнения к новому тексту и передаем его родительскому методу, который и будет хранить его. Выделение добавленной части осуществляется из потока рассылки событий EventDispatchThread, с помощью метода invokeLater() класса EventQueue. Выделение позволит пользователю быстро отказаться от нашей подсказки, если она ему не подойдет, и продолжить печатать свой вариант как ни в чем не бывало. Текст необходимо выделять из потока рассылки событий, потому что, во-первых, мы будем проводить манипуляции с графическим компонентом Swing, а во-вторых, нам нужно «подождать», пока новый текст окажется в поле. Это сделает наш предок, когда мы вызовем его родительский метод и выполнение метода insertString() закончится, а затем уже, когда дойдет очередь в потоке событий, вызовется наш метод run(), который выделит новый текст.

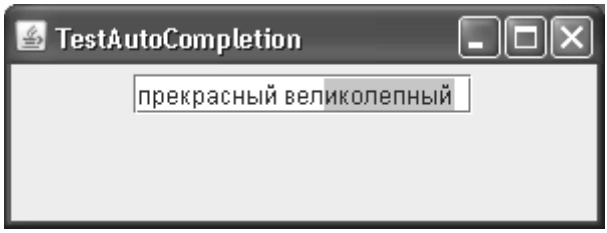
---

[8] Это класс с набором статических методов, он находится в пакете javax.swing.text, и содержит несколько весьма полезных алгоритмов для работы с текстовыми компонентами и их моделями. Если вы собираетесь вплотную заниматься текстовыми возможностями Swing, обратите на данный класс свое внимание. Методы для нахождения слов в тексте к тому же полностью готовы к работе со всеми возможными языками, а не основаны на примитивных алгоритмах.

Теперь, когда мы создали нашу модель документа, необходимо проверить ее в действии. Следующий простой пример показывает, на что она способна:

```
// TestAutoCompletion.java
// Проверка работы текстового поля с
// автоматическим заполнением
import com.porty.swing.*;
import javax.swing.*;
import java.awt.*;

public class TestAutoCompletion extends JFrame {
  public TestAutoCompletion() {
    super("TestAutoCompletion");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем и настраиваем поле
    JTextField field =
        new JTextField();
    field.setColumns(15);
    // слова для автозаполнения
    AutoCompleteTextDocument doc =
        new AutoCompleteTextDocument(field);
    doc.addWord("прекрасный");
    doc.addWord("великолепный");
    // добавляем поле в окно
    setLayout(new FlowLayout());
    add(field);
    // выводим окно на экран
    setSize(300, 200);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new TestAutoCompletion(); } });
  }
}
```

В примере создается небольшое окно с рамкой, в него мы добавляем текстовое поле. После создания текстового поля необходимо создать нашу модель с автоматическим заполнением и передать поле в ее конструктор (в нем модель добавится к полю автоматически), а также провести небольшую настройку: количество столбцов следует задать равным 15 и в список слов для автоматического заполнения добавить два подходящих прилагательных. Запустив программу с примером и набирая добавленные в список слова, вы увидите, как поле услужливо предоставляет вам подсказки при вводе.




Эти подсказки, если они отображают то, что вы хотели набрать, значительно ускоряют ваш труд и легко устраняются, если вы пожелаете набрать что-либо иное. Работать с такими компонентами удобно, и они добавляют вашему пользовательскому интерфейсу немало блеска.

Вы можете задействовать созданное нами текстовое поле не только в качестве отдельного элемента интерфейса, но и в качестве редактора для сложных компонентов Swing, таких как списки или таблицы. Так вы сделаете и эти компоненты значительно более удобными, особенно если правильно подберете список слов для автоматического заполнения.

Рассмотренный нами в этом параграфе подход, когда мы берем стандартную модель документа простых текстовых полей PlainDocument и переопределяем ее метод insertString(), очень полезен, и его определенно стоит взять на заметку. Если с простыми случаями ввода специальных данных может справиться поле с форматированием JFormattedTextField, то в более сложных ситуациях очень ценится возможность полностью контролировать процесс ввода данных и незамедлительно их менять , что мы и увидели. Еще одним вариантом является применение слушателя документа DocumentListener, но в этом случае модифицировать текст гораздо труднее, поскольку слушателю достается информация уже после того, как изменения произошли, и любые изменения текста в нем опять же приводят к вызову методов слушателя, а это не совсем элегантно.

## Отмена и повтор операций

К модели документа Document можно присоединить не только слушателя DocumentListener. Есть событие и соответствующий ему слушатель еще одного типа — это событие UndoableEditEvent и слушатель UndoableEditListener. Данное событие модель документа обеспечивает путем поддержки операций повтора и отмены (undo/redo). Эти операции можно встретить во всех современных текстовых компонентах (и не только в текстовых), и они весьма удобны: пользователь может не опасаться совершить какую-либо ошибку, к примеру, случайно удалить набранный текст. К его услугам всегда имеется возможность отмены операций, позволяющая привести документ к прежнему виду. Повтор операции аналогичным образом дает пользователю возможность снова выполнять ту операцию, от которой он по ошибке отказался. Все текстовые компоненты в ваших программах должны поддерживать по меньшей мере отмену операций, иначе ваш интерфейс рискует стать очень неудобным.

Если уже знакомое нам событие DocumentEvent несет в себе подробную информацию об изменениях в структуре и содержимом документа, то у события UndoableEditEvent совершенно иная цель. Оно должно сообщать, что пользователь совершил в документе какую-то операцию, которую можно отменить или повторить. Такая операция описана в классе UndoableEdit, объект именно этого класса возвращает метод getEdit() события UndoableEditEvent. В данном объекте описано немного: имя операции, если оно есть (у некоторых операций могут быть различные имена для отмены и повтора), возможность отмены или повторения операции, и два основных метода, которые и занимаются отменой и повторением. Больше никакой информации объект UndoableEdit в себе не таит, да

это ему и не нужно: единственная его цель состоит в описании операции, совершенной в документе. Подробности сообщает событие DocumentEvent.

Теперь вы можете наделять свои текстовые компоненты возможностью отмены и повторения операций: достаточно зарегистрировать в модели документа слушателя UndoableEditListener, получить уведомления о произведенных пользователем операциях, хранить историю операций (например, в виде списка объектов UndoableEdit) и при запросе пользователя вызывать для последних метод undo() или redo(). Правда, есть еще один путь: для удобства программистов библиотека Swing предоставляет небольшой пакет javax.swing.undo. Ядром данного пакета является класс UndoManager, вы можете регистрировать его в любом текстовом компоненте как слушателя UndoableEditListener. Этот класс позаботится о хранении истории операций и предоставит удобный программный интерфейс, позволяющий быстро наладить отмену и повторение операций в вашем приложении.

Рассмотрим пример, в котором мы обеспечим многострочное текстовое поле JTextArea специальной панелью инструментов, позволяющей отменять и повторять произведенные в поле операции. Для удобства работы хранить операции и работать с ними мы будем посредством услуг класса UndoManager. Вот что получается:

```
// UndoAndRedo.java
// Поддержка отмены и повтора операций
// в текстовых компонентах Swing
import javax.swing.*;
import javax.swing.undo.*;
import java.awt.event.*;
import java.awt.*;

public class UndoAndRedo extends JFrame {
  // поддержка отмены/повтора операций
  private UndoManager undoManager = new UndoManager();
  public UndoAndRedo() {
    super("UndoAndRedo");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // панель инструментов
    JToolBar toolBar = new JToolBar();
    toolBar.add(new UndoAction());
    toolBar.add(new RedoAction());
    // текстовое поле
    JTextArea textArea = new JTextArea();
    // добавляем слушателя операций
    textArea.getDocument().
        addUndoableEditListener(undoManager);
    // добавляем компоненты в окно
    add(toolBar, "North");
    add(new JScrollPane(textArea));
    // выводим окно на экран
    setSize(400, 300);
```

```
    setVisible(true);
  }
  // команда — отмена операции
  class UndoAction extends AbstractAction {
    public UndoAction() {
      // настройка команды
      putValue(AbstractAction.SMALL_ICON,
          new ImageIcon("undo16.gif"));
    }
    public void actionPerformed(ActionEvent e) {
      if ( undoManager.canUndo() )
        undoManager.undo();
    }
  }
  // команда — повтор операции
  class RedoAction extends AbstractAction {
    public RedoAction() {
      // настройка команды
      putValue(AbstractAction.SMALL_ICON,
          new ImageIcon("redo16.gif"));
    }
    public void actionPerformed(ActionEvent e) {
      if ( undoManager.canRedo() )
        undoManager.redo();
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new UndoAndRedo(); } });
  }
}
```

В центре нашего окна размещается многострочное текстовое поле JTextArea, предварительно «обернутое» в панель прокрутки JScrollPane. На севере окна мы размещаем панель инструментов JToolBar; она состоит из двух кнопок, отвечающих за отмену и повторение операций. Кнопки создаются на основе команд Action, описанных в виде внутренних классов. Настройка команд выполняется в конструкторе: мы меняем свойство команды SMALL_ICON, задавая для кнопок значок[9]. Само же действие, выполняемое кнопкой, происходит в методе actionPerformed().

Для хранения происходящих в текстовом поле операций и их последующей отмены или повторения мы воспользуемся услугами класса UndoManager. Для того чтобы он

[9] Значки для кнопок из этого примера были взяты из коллекции Java Look and Feel Graphics Repository. Вы можете найти ее на сайте java.sun.com.

начал получать информацию об операциях, происходящих в поле, необходимо зарегистрировать его как слушателя UndoableEditListener. Сделать это несложно: надо получить использующийся в текстовом поле документ Document и присоединить к нему объект UndoManager. После этого последний будет получать и обрабатывать всю информацию о происходящих в модели документа операциях.

Команды отмены и повторения операций очень просты, во многом благодаря услугам объекта UndoManager. Они проверяют, возможны ли сейчас отмена или повторение операций, и если да, выполняют отмену (методом undo()) или повторение (redo()). Управлять списком операций UndoableEdit будет класс UndoManager. Запустив программу с примером и работая с текстом, вы увидите, каким образом кнопки панели инструментов позволяют отменять или повторять операции.

Класс UndoManager обладает несколькими дополнительными интересными возможностями и зачастую его хватает для достаточно гибкого управления операциями, происходящими в ваших текстовых компонентах. Если же возникнет необходимость в особом поведении, вы всегда сможете самостоятельно реализовать интерфейс слушателя UndoableEditListener, присоединить его к модели документа и распоряжаться информацией об операциях так, как вам потребуется. Увы, но по умолчанию никакой отмены или повторения операций в текстовых компонентах Swing не поддерживается, так что вам всегда придется добавлять ее самостоятельно — в противном случае работать с вашими текстовыми компонентами будет не слишком удобно. Для поддержки отмены и повтора операций через клавиатуру чрезвычайно хорошо подходят карты клавиатурных сокращений InputMap, которые вы можете настроить на вызов тех же самых команд, что настроены на панели инструментов, в случае нажатия подходящего сокращения (обычно Control-Z).

## *Управление курсором — интерфейс Caret*

Важную роль при работе с текстом играет *текстовый курсор* (caret), который показывает пользователю, в каком месте будут появляться набираемые им символы, позволяет перемещаться по строкам, столбцам и другим элементам текста, а также управляет процессом выделения текста. Курсор текстовых компонентов библиотеки Swing (а к таким компонентам относятся те, что унаследованы от базового класса JTextComponent), должен реализовывать интерфейс Caret из пакета javax.swing.text. Все знакомые нам текстовые компоненты используют стандартную реализацию этого интерфейса DefaultCaret.

Именно курсор позволяет программно изменять текущую позицию в тексте (это в том числе означает и прокрутку текста до нужной позиции, если она в данный момент не видна на экране) и управлять выделенным текстом (узнавать, что в данный момент выделено или выделять текст программно). Правда, для работы с выделенным текстом в базовом классе JTextComponent имеется несколько удобных методов, но изначально выделение текста находится в ведении текстового курсора.

Два основных свойства курсора — его позиция (вы сможете управлять ею посредством свойства dot) и начало отсчета (узнать его позволяет метод getMark()). Позиция курсора показывает, где будут появляться набираемые пользователем или вставляемые вами программно символы. Начало отсчета служит для поддержки выделения текста или позиции курсора, когда выделенного текста нет. Если же выделение текста происходит, начало отсчета указывает на начало выделенного фрагмента, а позиция указывает на конец этого фрагмента (в конце выделенного фрагмента находится курсор). В зависимости от направления выделения (вперед по тексту или назад) начало отсчета может быть больше позиции или меньше ее.

Курсор, описанный интерфейсом Caret, поддерживает событие ChangeEvent, оно запускается каждый раз при изменении позиции курсора. Присоединив к курсору слушателя ChangeListener, вы сможете получать подробную информацию обо всех

перемещениях курсора в вашем текстовом компоненте. Кроме того, базовый класс JTextComponent поддерживает событие CaretEvent. Оно также запускается при перемещениях курсора, но если событие ChangeEvent содержит только источник события (сам курсор), то событие CaretEvent позволяет сразу же узнать текущую позицию курсора и начало отсчета.

Рассмотрим теперь небольшой пример и увидим, как можно управлять курсором, используемым в текстовых компонентах Swing по умолчанию, а также узнавать от него о событиях:

```
// ControllingCaret.java
// Управление текстовым курсором
import javax.swing.*;
import javax.swing.event.CaretListener;
import javax.swing.event.CaretEvent;
import javax.swing.text.*;
import java.awt.*;

public class ControllingCaret extends JFrame {
  public ControllingCaret() {
    super("ControllingCaret");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // многострочное текстовое поле
    final JTextArea textArea = new JTextArea();
    // добавим текст
    textArea.append("Просто какой-то текст");
    // настройка курсора и выделение текста
    Caret caret = textArea.getCaret();
    caret.setBlinkRate(50);
    caret.setDot(5);
    caret.moveDot(10);
    // добавим текстовое поле в окно
    add(new JScrollPane(textArea));
    // надпись для слова у курсора
    final JLabel label = new JLabel();
    add(label, "South");
    // слушатель перемещений курсора
    textArea.addCaretListener(new CaretListener() {
      public void caretUpdate(CaretEvent e) {
        // выведем слово у курсора
        try {
          // начало и конец слова на позиции
          int wordStart =
              Utilities.getWordStart(textArea, e.getDot());
          int wordEnd =
```

```
                Utilities.getWordEnd(textArea, e.getDot());
            label.setText(textArea.getText(
                wordStart, wordEnd - wordStart));
          } catch (BadLocationException ex) {
            ex.printStackTrace();
          }
        }
      });
      // выводим окно на экран
      setSize(400, 300);
      setVisible(true);
    }
    public static void main(String[] args) {
      SwingUtilities.invokeLater(
          new Runnable() {
            public void run() { new ControllingCaret(); } });
    }
}
```

Мы создаем небольшое окно, в центре его разместится многострочное текстовое поле JTextArea, «обернутое» в панель прокрутки. Для того чтобы можно было работать с курсором, необходим какой-либо текст, так что мы добавляем в поле строку текста, вызывая метод append(). Далее мы получаем используемый в текстовом поле курсор (для этого служит метод getCaret()) и настраиваем для него некоторые свойства.

Прежде всего меняется частота мерцания курсора, задаваемая свойством blinkRate. Это промежуток между мерцаниями в миллисекундах. В нашем случае курсор будет мерцать довольно быстро: мы задаем промежуток равным всего 50 миллисекундам. Мерцание можно и отключить: достаточно задать нулевой промежуток. Далее мы устанавливаем позицию курсора равной 5 и выделяем фрагмент текста. Текст удобно выделять с помощью метода moveDot(): он перемещает позицию в то место, что вы ему указываете, а начало отсчета делает равным прежнему значению позиции. В примере мы выделяем фрагмент текста в пять символов с пятой позиции по десятую.

На юге окна мы размещаем надпись, в которой будем писать слово, находящееся на позиции курсора. Чтобы следить за курсором, мы присоединяем слушателя для событий CaretEvent прямо к текстовому полю, и при наступлении события, пользуясь уже знакомыми нам методами класса Utilities, устанавливаем текущее слово в качестве текста для надписи. Как видно, событие CaretEvent позволяет узнать текущую позицию курсора в тексте с помощью метода getDot().

Запустив программу с примером, вы увидите, каким образом проведенные нами манипуляции отразились на текстовом поле и каким образом мы мгновенно узнаем о слове на текущей позиции курсора. Курсор Caret незаменим при управлении текущей позицией в тексте и работе с выделением, он позволяет мгновенно узнавать о перемещениях в тексте, так что при работе с текстом вы не раз будете обращаться к его возможностям.

## *Дополнительное выделение текста*

При работе с текстом, имеющим определенную структуру, таким например, как язык программирования или язык разметки наподобие HTML, возможность дополнительно размечать часть текста приходится весьма кстати. Это может быть, например прорисовка тегов в языке разметки другим цветом, выделение находящейся в данный момент у курсора единицы языка по всему тексту программы, или другие задачи, требующие помечать некий текст.

В качестве решения можно было бы применять редактор с поддержкой стилей и задавать цвет и начертание для текста, однако это не совсем элегантное решение. Стили скорее предназначены для типографских целей, и не служат целям разметки содержимого. Текстовые компоненты Swing предлагают нам отличное решение — встроенную поддержку рисования фона за текстом на определенных позициях.

Именно рисование фона применяется для прорисовки выделенного текста, однако это решение доступно и нам, и не ограничивается выделением. В архитектуру текстовых компонентов встроен рисующий объект, описанный интерфейсом Highlighter, который прорисовывает фон текста, перед тем как будет прорисован сам текст. Вы можете реализовать его самостоятельно и задать для любого текстового компонента, используя свойство с одноименным названием. Однако, как это бывает в Swing, почти всегда есть возможность встать на плечи гигантов и воспользоваться стандартной реализацией. Дело в том, что интерфейс Highlighter предусматривает добавление новых регионов для прорисовки фона и их последующее удаление. Прорисовкой же самого фона для заданного региона текста занимается отдельный объект HighlightPainter. Таким образом, вы можете взять стандартный объект, рисующий фон, из любого текстового компонента и добавить к нему дополнительный регион текста, а прорисовку реализовать в интерфейсе HighlightPainter, или же и здесь использовать стандартную реализацию DefaultHighlightPainter, которая рисует фон в одном цвете.

Чтобы не быть голословными, рассмотрим простой пример и убедимся, насколько просто размечать фон для текста с помощью стандартных средств Swing.

```
// TextHighlights.java
// Дополнительное выделение текста
import javax.swing.*;
import javax.swing.text.*;
import java.awt.*;

public class TextHighlights extends JFrame {
  public TextHighlights() {
    super("TextHighlights");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем и настраиваем поле
    JTextArea area =
        new JTextArea("Привет мир!");
```

```
    try {
      // добавим выделенный фрагмент желтого цвета
      Object reference =
          area.getHighlighter().
              addHighlight(0, 6,
                  new DefaultHighlighter.
                      DefaultHighlightPainter(Color.YELLOW));
    } catch (BadLocationException e) {
      e.printStackTrace();
    }
    // добавляем поле в окно
    add(new JScrollPane(area));
    // выводим окно на экран
    setSize(300, 200);
    setVisible(true);
  }

  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new TextHighlights(); } });
  }
}
```

В примере мы создаем многострочное текстовое поле и размещаем его в центре нашего окна. Далее мы просто получаем используемый в нем по умолчанию рисующий объект фона, и методом addHighlight() добавляем новую выделенную область. Обратите внимание на параметры метода — это начало и конец фрагмента текста, фон которого вы собираетесь рисовать, а также рисующий объект. Мы использовали стандартный объект, попросив его рисовать фон желтым цветом. Метод же возвращает ссылку на добавленный фрагмент прорисовки фона. В дальнейшем эту ссылку можно использовать, чтобы поменять фрагмент или совсем удалить его методами changeHighlight() и removeHighlight(). Запустив программу, вы увидите, как работает прорисовка фона текста.


Пример намеренно сделан очень простым, чтобы продемонстрировать непосредственно прорисовку фона, однако в качестве упражнения вы можете совместить его с предыдущим примером, а именно следить за курсором, получать слово на текущей позиции и выделять такое слово с помощью фонового рисования по всему доступному тексту. Таким образом вы создадите простой прототип мгновенного поиска слова по тексту и примените все возможности фонового рисования, в том числе сможете и удалять уже добавленные фрагменты фона.

Создать свой собственный рисующий объект для прорисовки фона за текстом очень просто — интерфейс HighlightPainter предусматривает всего лишь один метод paint(), в который вам передается объект для рисования Graphics и область, в который вы должны прорисовать фон. Используя механизм рисования фона для текстовых компонентов, вы можете создавать весьма удобные для пользователя режимы работы, подсказывая ему с помощью визуальных эффектов что-то, в чем он, скорее всего, в данный момент наиболее заинтересован, или разделяя на экране информацию различной структуры и предназначения.

## *Резюме*

Текстовые компоненты Swing позволят организовать в своем приложении ввод и редактирование текста самыми разными способами: начиная от редактирования простого текста в одну строку и заканчивая составлением и отображением документов в известных форматах RTF и HTML. В данной главе мы разобрали базовые возможности текстовых компонентов Swing, на их основе вы сможете реализовать большую часть требуемых в приложениях операций. Если же вашей целью является мощный текстовый редактор с огромным спектром возможностей, то к вашим услугам пакет javax.swing.text: он содержит все, чтобы ваши фантазии можно было воплотить в жизнь.

С другой стороны, если речь заходит о мощных текстовых возможностях, вы можете начать с платформы NetBeans, основной целью которой является удобное редактирование текста, в том числе и структурированного. Скорее всего, если ваши цели и способности NetBeans совпадут, вы сэкономите на разработке очень много времени — реализация качественных текстовых компонентов требует усилий.

# Глава 17. Таблицы

Пришло время познакомиться с одним из самых впечатляющих компонентов библиотеки Swing — таблицей JTable. Таблица JTable дает возможность с легкостью выводить двухмерную информацию, расположенную в виде строк и столбцов, достаточно легко настраивать и сортировать данные, выводить их в любом подходящем для вас виде, управлять заголовками таблицы и ее выделенными элементами и с небольшими усилиями делать еще многое другое. Бывают случаи, когда всю необходимую пользователю информацию вы можете разместить в нескольких таблицах, применяя остальные компоненты лишь в качестве вспомогательных, и с помощью тех же таблиц получать любую нужную информацию.

Впрочем, за все приходится платить. Это правило действует и в случае с таблицами. Чтобы полностью овладеть всей мощью класса JTable, придется изучить довольно много вспомогательных классов, узнать способы их взаимодействия, понять, как реализованы модели и почему они так реализованы, в каких ситуациях таблица ведет себя определенным образом, и многое другое. Вспомогательным классам таблицы JTable посвящен целый пакет (впечатляющего размера) javax.swing.table. Но пугаться не стоит, создать таблицу с некоторой информацией, полученной из любого источника, довольно просто и не требует особенного напряжения. Как всегда, Swing подстраивается под уровень пользователя: для получения простой таблицы потребуются простые действия, а все более сложные таблицы будут требовать от вас все больше и больше усердия. Начнем с самого простого.

## *Простые таблицы*

Таблица JTable позволяет выводить двухмерные данные, записанные в виде строк и столбцов. Данные для таблицы поставляет специальная весьма гибкая модель, обязанности которой описаны в классе TableModel (стоит помнить, что практически все вспомогательные классы и интерфейсы для таблиц JTable находятся в пакете javax.swing.table). Она позволяет передать таблице все необходимые данные для верного вывода информации (количество строк и столбцов, названия столбцов, элемент, находящийся в определенном месте таблицы, тип данных, хранящийся в столбце, признак доступности для редактирования некоторого элемента), а также поддерживает слушателей, получающих уведомления об изменениях в данных модели. Всего этого хватает таблице JTable для отображения двухмерных данных любого типа.

Как и всегда в Swing, вам необязательно использовать для таблиц подготовленные и наполненные данными модели, вы можете передать в специальные конструкторы таблицы JTable массивы или векторы с информацией о ячейках таблицы и, при необходимости, о названиях ее столбцов. Довольно часто, особенно при создании простых программ или простых незатейливых таблиц, которые вряд ли будут в будущем расширяться или повторно использоваться, такой подход удобнее и быстрее. Учитывая при этом, что вы с легкостью можете получить значение любой ячейки таблицы методом getValueAt() и изменить значение любой ячейки методом setValueAt(), у вас и без моделей (хотя на самом деле они незримо работают за кулисами) есть возможность полностью управлять данными таблицы.

Таблица JTable, как и большинство сложных компонентов библиотеки Swing, реализует интерфейс Scrollable, так что вы смело можете добавлять ее в панель прокрутки JScrollPane, не заботясь о правильности прокрутки. При прокрутке полностью появля-

ются следующий столбец или следующая строка таблицы, это удобно и интуитивно понятно пользователю. Интересно, что заголовок таблицы (специального вида надписи с названиями столбцов, реализованных классом JTableHeader) будет виден пользователю только при помещении таблицы в панель прокрутки, потому что таблица размещает компонент JTableHeader в виде заголовка панели прокрутки (как мы отмечали в главе 13, заголовок панели прокрутки хранится в ее свойстве rowHeaderView). Так что если вам нужно увидеть заголовок таблицы, придется включить таблицу в панель прокрутки или вручную разместить около верхней границы таблицы компонент JTableHeader. Чуть позже мы обсудим подробнее заголовок таблицы.

Ну а сейчас посмотрим, насколько просто можно создавать вполне работоспособные таблицы стандартного вида с помощью специальных конструкторов:

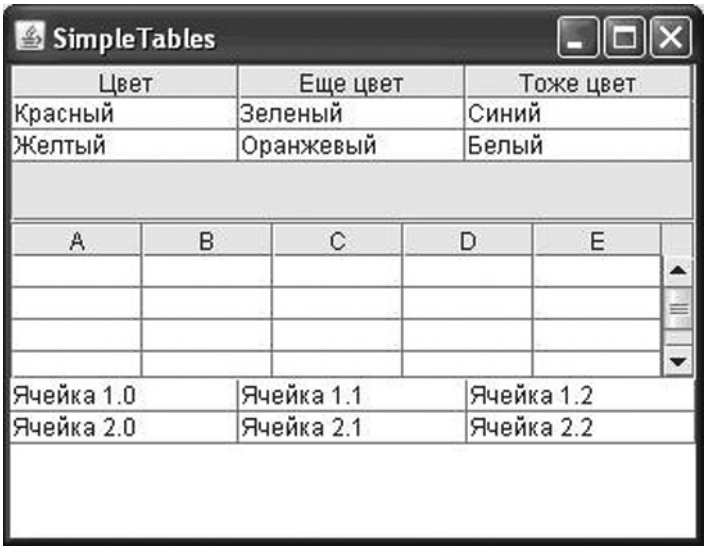
```
// SimpleTables.java
// Простые таблицы с помощью удобных конструкторов
import javax.swing.*;
import java.util.*;
import java.awt.*;

public class SimpleTables extends JFrame {
  // данные для таблиц
  private Object[][] colors = new String[][] {
      { "Красный", "Зеленый", "Синий" },
      { "Желтый", "Оранжевый", "Белый" },
  };
  // названия заголовков столбцов
  private Object[] colorsHeader = new String[] {
      "Цвет", "Еще цвет", "Тоже цвет"
  };
  public SimpleTables() {
    super("SimpleTables");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // несколько простых таблиц
    JTable table1 = new JTable(colors, colorsHeader);
    JTable table2 = new JTable(5, 5);
    // таблица на основе вектора, состоящего из векторов
    Vector<Vector<String>> data =
        new Vector<Vector<String>>();
    Vector<String> row1 = new Vector<String>();
    Vector<String> row2 = new Vector<String>();
    // вектор с заголовками столбцов
    Vector<String> columnNames = new Vector<String>();
    // наполнение данными
    for (int i=0; i<3; i++) {
      row1.add("Ячейка 1." + i);
      row2.add("Ячейка 2." + i);
```

```
        columnNames.add("Столбец #" + i);
      }
      data.add(row1);
      data.add(row2);
      JTable table3 = new JTable(data, columnNames);
      // добавляем таблицы в панель с тремя рядами
      setLayout(new GridLayout(3, 1));
      add(new JScrollPane(table1));
      add(new JScrollPane(table2));
      add(table3);
      // выводим окно на экран
      setSize(350, 400);
      setVisible(true);
    }
    public static void main(String[] args) {
      SwingUtilities.invokeLater(
          new Runnable() {
            public void run() { new SimpleTables(); } });
    }
}
```

В примере мы наследуем от окна JFrame, указываем, что при закрытии окна нужно будет закончить работу приложения, и размещаем в панели содержимого несколько таблиц JTable, созданных с помощью удобных конструкторов. Первая таблица (table1) создается конструктором, принимающим два массива с данными. Первый массив должен быть двухмерным и хранить строки таблицы с данными некоторого типа. Мы использовали в качестве данных самое простое из всего возможного — строки String. Остается только вспомнить, как записываются в Java двухмерные массивы, довольно редкие гости обычных программ. Вы видите, что данные для первой таблицы (две строки с названиями цветов) описаны в двухмерном массиве colors. Второй передаваемый в конструктор массив одномерный и хранит названия столбцов таблицы, с ним все проще. Вторая таблица (table2) создается с помощью, пожалуй, самого простого конструктора, которому требуется указать лишь количество строк и столбцов в таблице. Все ячейки будут пустыми, а в качестве названий столбцов использованы латинские буквы, как во всех электронных таблицах. Такой конструктор требуется редко, но он может быть полезен там, где с помощью таблиц нужно получать информацию от пользователя или имитировать вид электронной таблицы. Ну и, наконец, третий конструктор работает с вектором (динамическим массивом) Vector из пакета java.util. Ему требуется передать два вектора. Первый должен хранить в себе такие же векторы, в которых, в свою очередь, должны храниться данные строк таблицы. Каждый вектор отвечает за свою строку таблицы. Второй вектор хранит обычные объекты — названия столбцов таблицы. В примере мы наполняем векторы строк и названий столбцов в цикле. Затем векторы строк добавляются в вектор, который будет хранить все данные таблицы (data), после чего таблицу table3 можно создавать.

Созданные три таблицы добавляются в окно, в котором мы установили табличное расположение GridLayout из трех рядов и одного столбца. Как нам известно, табличное расположение выделяет своим ячейкам одинаковое пространство. Последовательное расположение FlowLayout и вообще любое расположение, предоставляющее компонен-

там все затребованное ими «предпочтительное» пространство, нам не подойдут, потому что таблицы, помещенные в панель прокрутки JScrollPane, очень требовательны к пространству, даже если в них хранится совсем немного данных. Учитывайте этот факт при создании пользовательского интерфейса и подбирайте такой менеджер расположения, который не позволит таблицам в панели прокрутки «отнять» у контейнера слишком много места. С другой стороны, таблицы без панелей прокрутки занимают ровно столько места, сколько им требуется для отображения своих данных. Заметьте, что мы используем панель прокрутки только для первых двух таблиц, а третью добавляем в панель напрямую. После запуска программы с примером вы увидите, что в такой таблице не отображается заголовок. Для вывода заголовка нужно либо добавить таблицу в панель прокрутки (в заголовке которой и окажется заголовок таблицы), либо ввести его в контейнер вручную. Как работать с заголовками таблиц, мы узнаем немного позже.




На заключительном этапе наше окно выводится на экран. Запустив программу с примером, вы убедитесь в том, что несколькими строками кода мы создали вполне работоспособные таблицы, достаточные для многих простых приложений. Разбирая пример, вы также увидите, что выделение в таблицах JTable по умолчанию работает построчно, выделять можно произвольное количество строк, все элементы считаются редактируемыми (начать редактирование позволяет двойной щелчок мыши, таким образом он действует для всех внешних видов, поставляемых с библиотекой Swing), а столбцы таблицы можно перетаскивать и менять местами (правда, делать это можно только с первыми двумя таблицами, у которых есть заголовки). Все это поддается настройке. Эти вопросы мы будем обсуждать постепенно на протяжении всей главы. Перетаскивание столбцов таблицы, с одной стороны, может быть удобным (те столбцы, с которыми он чаще работает, пользователь сделает первыми), однако, с другой стороны, может запутать и усложнить работу даже с простой таблицей (пользователю придется выяснять, какие столбцы были перемещены, где они теперь расположены и в каких логических отношениях они находятся с другими столбцами). Перетаскивание столбцов смотрится эффектно, но, как правило, во избежание путаницы его следует отключать, вскоре мы узнаем, как это делается. С другой стороны, программисту-клиенту таблицы не придется учитывать перемещения столбцов: их номера в таблице и ее моделях не меняются, как бы изощренно пользователь не тасовал столбцы.

## Простая настройка внешнего вида

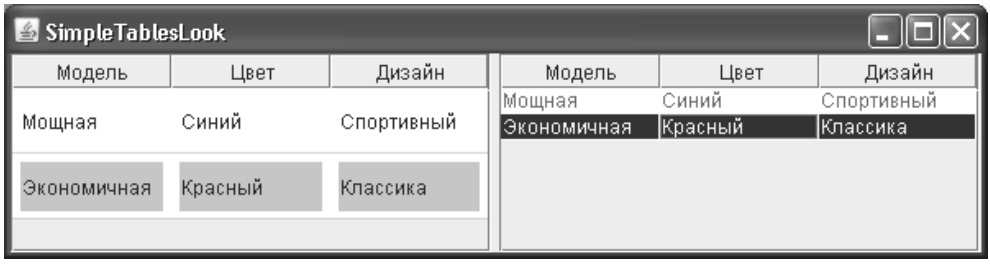
Мы только что видели, как парой строк кода без привлечения особенных мощностей и внутренних ресурсов класса JTable и его «сотрудников» можно создать вполне достойные таблицы. Аналогичные действия можно проделать и при изменении внешнего вида таблицы, причем внешний вид можно поменять полностью, вплоть до переписывания UI-представителя класса JTable, однако есть путь и попроще. С помощью нескольких полезных свойств можно буйно раскрасить таблицу, использовать различные цвета для текста и выделения, настроить сетку таблицы, задать разные расстояния между ячейками. Бывает, что этого хватает для получения той самой эффектной таблицы, которая вам нужна.

Рассмотрим несложный пример с парой таблиц, внешний вид которых мы изменим с помощью некоторых свойств класса JTable. Запустив программу с этим примером, вы воочию убедитесь, как именно влияют те или иные свойства на внешний вид таблиц, а затем мы обсудим их подробнее:

```
// SimpleTablesLook.java
// Небольшое изменение внешнего вида таблиц
import javax.swing.*;
import java.awt.*;

public class SimpleTablesLook extends JFrame {
  // данные и заголовки для таблицы
  private Object[][] data = new String[][] {
      { "Мощная", "Синий", "Спортивный" },
      { "Экономичная", "Красный", "Классика" }
  };
  private Object[] columns = new String[] {
      "Модель", "Цвет", "Дизайн"
  };
  public SimpleTablesLook() {
    super("SimpleTablesLook");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // таблица с разными расстояниями между ячейками
    JTable table1 = new JTable(data, columns);
    // настройка расстояний и цветов
    table1.setRowHeight(40);
    table1.setIntercellSpacing(new Dimension(10, 10));
    table1.setGridColor(Color.green);
    table1.setShowVerticalLines(false);
    // таблица с разными цветами
    JTable table2 = new JTable(data, columns);
    table2.setForeground(Color.red);
    table2.setSelectionForeground(Color.yellow);
    table2.setSelectionBackground(Color.blue);
    table2.setShowGrid(false);
```

```
        // добавляем таблицы в панель из двух ячеек
        setLayout(new GridLayout(1, 2, 5, 5));
        add(new JScrollPane(table1));
        add(new JScrollPane(table2));
        // выводим окно на экран
        setSize(600, 200);
        setVisible(true);
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(
            new Runnable() {
              public void run() { new SimpleTablesLook(); } });
    }
}
```

В примере мы создаем небольшое окно, в котором разместятся две таблицы. Данные для обеих таблиц одинаковы, они описаны в виде двух массивов: двухмерного data, в нем хранится информация о ячейках таблицы, и одномерного columns, в котором мы разместили названия столбцов таблицы. Первая таблица table1 демонстрирует некоторые свойства класса JTable, позволяющие управлять расстоянием между ячейками, высотой строк таблицы, а также цветом и стилем сетки таблицы. Вторая таблица table2 демонстрирует, как можно изменить цвета, используемые для вывода текста и выделенных элементов, и как отключить прорисовку сетки таблицы. Обе таблицы размещаются в панелях прокрутки и добавляются в окно с установленным табличным расположением, при котором панель разбивается на две ячейки одинакового размера с небольшим (в 5 пикселов) расстоянием между ними. После этого остается только вывести наше окно на экран.




Запустив программу с примером, вы сможете увидеть изменения во внешнем виде таблиц и оценить, в каких ситуациях подобное изменение внешнего вида может вам понадобится. Давайте опишем использованные в примере свойства немного подробнее, так чтобы они всегда были у вас «под рукой» (табл. 17.1).

**Таблица 17.1.** Свойства, меняющие внешний вид таблицы JTable

| Свойства | Краткое описание |
|---|---|
| selectionBackground, selectionForeground | Управляют цветами прорисовки фона выделенной ячейки (первое свойство) и текста выделенной ячейки (второе свойство). Вкупе со стандартными свойствами background и foreground, отвечающими за фон и цвет любого компонента Swing, позволяют полно настроить цветовую гамму таблицы |

**Таблица 17.1** *(продолжение)*

| Свойства | Краткое описание |
|---|---|
| rowHeight, intercellSpacing | Первое свойство позволяет задать высоту сразу всех строк таблицы или (с помощью перегруженного метода set) отдельно некоторой строки таблицы. Второе свойство управляет расстоянием между ячейками как по оси X, так и по оси Y (в примере вы можете видеть, что задается это расстояние в виде объекта Dimension) |
| showVerticalLines, showHorizontalLines, showGrid | С помощью этих свойств вы можете указывать таблице, как и какие линии сетки следует рисовать. Первые два свойства по отдельности управляют прорисовкой вертикальных и горизонтальных линий сетки таблицы, а третье свойство позволяет «одним махом» показать или скрыть всю сетку таблицы |
| gridColor | Позволяет задать цвет сетки таблицы JTable (вы могли видеть, как использовать это свойство, в рассмотренном нами примере) |

С помощью перечисленных свойств вы сможете быстро и без особых усилий настроить некоторые аспекты внешнего вида таблицы JTable, иногда именно это нужно интерфейсу программы. Далее, читая эту главу, мы узнаем множество других немного более сложных, но и более тонких способов настройки внешнего вида таблицы, а точнее, ее ячеек.

# *Модели таблицы JTable*

Для правильной работы таблица JTable нуждается сразу в трех моделях, поставляющих ей данные и при изменении этих данных сохраняющих изменения. Первая и самая важная модель, о которой мы уже упоминали в начале главы, хранит непосредственно данные ячеек таблицы, а также дополнительную служебную информацию об этих ячейках. Обязанности этой модели описываются интерфейсом TableModel, который довольно прост, и реализовать его самому не представляет особого труда. Вторая модель управляет столбцами таблицы и описана в интерфейсе TableColumnModel. Столбцы очень важны для любой таблицы: вспомним, что, как правило, строки таблицы представляют собой различные наборы однотипных данных, а столбцы описывают тип хранимых в строках данных и порядок их следования. Именно столбцы определяют, какой набор данных отображает таблица, данные в таблице сортируются по столбцам, так что отдельная модель для столбцов не является не роскошью. Модель столбцов таблицы TableColumnModel позволяет добавлять, перемещать, находить столбцы, узнавать об изменениях в их данных или расположении, с ее помощью также можно изменить расстояние между столбцами и находящимися в них ячейками, изменить модель выделения столбцов. Наконец, третья и последняя модель таблицы нам хорошо знакома еще с главы 11, в которой мы знакомились со списками библиотеки Swing – это модель выделения списка ListSelectionModel. Она управляет выделением *строк* таблицы и не затрагивает столбцы, в модели которых TableColumnModel есть собственная модель выделения (в качестве последней используется все та же модель выделения списка ListSelectionModel). Так что формально можно говорить о четырех моделях таблицы JTable, просто одна из этих моделей вызывается неявно, через другую модель. Помимо моделей таблица JTable предоставляет еще несколько удобных механизмов выделения строк, столбцов и ячеек; вскоре мы о них узнаем.

Нет ничего удивительного в том, что у вас имеется широкий выбор альтернатив при работе с моделями, в этом заключается сама суть Swing. Вы можете использовать для каждой из них соответствующий абстрактный класс, начинающийся со слова Abstract, в котором тщательно реализованы поддержка слушателей и рассылка событий. Наследуя от такого класса, вам остается только наполнить модель данными, применяя удобные способы и инструменты, и в нужные моменты оповещать присоединенных слушателей

об изменениях в данных. Для каждой из моделей таблицы JTable имеется стандартная реализация (класс, название которого начинается со слова Default), хорошо подходящая в тех случаях, когда вам требуется модель, но нет причин или желания писать собственную . Впрочем, никто не отбирает у вас право полностью, «с нуля», написать свою модель и реализовать ее нужным образом. Перейдем теперь непосредственно к работе с моделями и изучим возможности каждой из них.

## Модель данных TableModel

Модель TableModel позволяет полно описать каждую ячейку таблицы JTable. Определенные в ней методы дают возможность таблице получить значение произвольной ячейки и изменить его (модель данных таблицы считается изменяемой, для этого не нужно проводить дополнительных действий или реализовывать новых интерфейсов), узнать о том, можно ли пользователю редактировать ячейки, получить исчерпывающую информацию о столбцах (их количестве, названиях и типе) и строках. Нам, со своей стороны, нужно определить эти методы и с их помощью передать таблице все подготовленные данные.

Прежде чем начать разработку собственной модели, можно рассмотреть вариант использования стандартной модели DefaultTableModel. Если вы вспомните первый пример данной главы, где мы создавали простые таблицы с помощью удобных конструкторов, то там таблица JTable неявно создавала и наполняла переданными в конструктор данными именно стандартную модель DefaultTableModel. Модель DefaultTableModel хранит переданные ей данные в двух векторах Vector, примерно так, как мы это делали в первом примере для одного из конструкторов таблицы: в одном из векторов хранятся такие же вектора, в которых в свою очередь хранятся данные строк, а во втором векторе хранятся названия столбцов таблицы. Если вам безразличны структура и способ получения отображаемых в таблице данных, лучше ограничиться стандартной моделью. С другой стороны, у вас остается не так много рычагов управления данными таблицы: все ячейки окажутся редактируемыми, у вас не будет способа тонко настроить тип данных для определенного столбца, то есть таблица будет иметь довольно безликий вид. К недостаткам стандартной модели можно отнести также некоторый архаизм: класс DefaultTableModel появился еще в самом первом выпуске Swing для JDK 1.1 и использует для хранения данных класс Vector, устаревший и уступающий по многим показателям новым контейнерам данных (таким как список ArrayList). Так что во многих ситуациях лучше написать собственную модель.

Впрочем, применение стандартной модели все равно остается самым быстрым способом создать таблицу и наполнить ее данными (и вы можете разделять ее между видами и отдельно от пользовательского интерфейса обрабатывать хранящиеся в ней данные). Рассмотрим небольшой пример, который продемонстрирует некоторые возможности стандартной модели:

```
// UsingDefaultTableModel.java
// Использование стандартной модели при создании таблицы
import javax.swing.*;
import javax.swing.table.*;
import java.awt.event.*;
import java.awt.*;

public class UsingDefaultTableModel extends JFrame {
  // наша модель
  private DefaultTableModel dtm;
  public UsingDefaultTableModel() {
```

```
    super("UsingDefaultTableModel");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем стандартную модель
    dtm = new DefaultTableModel();
    // задаем названия стоблцов
    dtm.setColumnIdentifiers(
        new String[] {"Номер", "Товар", "Цена"});
    // наполняем модель данными
    dtm.addRow(new String[] {"№1", "Блокнот", "5.5"});
    dtm.addRow(new String[] {"№2", "Телефон", "175"});
    dtm.addRow(new String[] {"№3", "Карандаш", "1.2"});
    // передаем модель в таблицу
    JTable table = new JTable(dtm);
    // данные могут меняться динамически
    JButton add = new JButton("Добавить");
    add.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        // добавляем новые данные
        dtm.addRow(
            new String[] {"?", "Новинка!", "Супер Цена!"});
      }
    });
    JButton remove = new JButton("Удалить");
    remove.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        // удаляем последнюю строку (отсчет с нуля)
        dtm.removeRow(dtm.getRowCount() - 1);
      }
    });
    // добавляем кнопки и таблицу
    add(new JScrollPane(table));
    JPanel buttons = new JPanel();
    buttons.add(add);
    buttons.add(remove);
    add(buttons, "South");
    // выводим окно на экран
    setSize(300, 300);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
```

```
            public void run() { new UsingDefaultTableModel(); } });
    }
}
```

В примере мы наследуем наш класс от окна с рамкой JFrame и размещаем в панели содержимого окна таблицу JTable, которая получает данные из созданной нами стандартной модели, а также две кнопки, позволяющие динамически, непосредственно во время работы программы, изменять данные модели (именно модели, код в теле слушателей не знает о существовании пользовательского интерфейса программы и о деталях таблицы JTable, он просто работает с данными, хранящимися в стандартной модели). Прежде всего создается сама стандартная модель DefaultTableModel. Мы использовали конструктор без параметров, а это означает, что после создания модель хранит нулевое количество строк и столбцов. Далее мы добавляем в модель значения строк методом addRow(), в который передаем массив со значениями, хранящимися в строке. Задействовать можно не только массив; перегруженная версия метода addRow() обеспечивает добавление строки, данные которой хранятся в векторе Vector. Заметьте, что *перед* добавлением в модель новых строк мы вызвали метод setColumnIdentifiers(), позволяющий задать названия (а значит и количество) столбцов таблицы. Если вызвать этот метод *после* добавления в модель DefaultTableModel новых строк, он посчитает, что в модели хранится нулевое количество столбцов (добавление строк не меняет хранимое в модели количество столбцов) и просто удалит все уже добавленные нами строки. Правда, можно задать количество столбцов с помощью метода setColumnCount(), не указывая их названий. Если вы вызовете этот метод перед добавлением в модель строк, последующий вызов метода setColumnIdentifiers() не удалит добавленные строки. Это общее правило работы со стандартной моделью таблицы DefaultTableModel: при вызове методов, изменяющих количество строк или столбцов, данные могут теряться, так что нужно следить за тем, чтобы новое количество строк или столбцов было достаточным для хранимых в модели данных, и заранее заботиться о правильности структуры таблицы.

После добавления в модель всех нужных данных мы передаем ее таблице JTable, которая затем разместится в панели прокрутки JScrollPane в центре панели содержимого (вы помните, что по умолчанию расположение в панели содержимого полярное). Далее создается пара кнопок add и remove, позволяющих динамически манипулировать данными модели. Мы присоединяем к ним слушателей и при щелчках на кнопках соответственно добавляем в модель новую строку и удаляем из нее последнюю строку. О таблице (или нескольких таблицах, или других видах) думать нам не придется — модель сама оповестит всех зарегистрированных в ней слушателей об изменениях в своих данных, а правильно отобразить эти изменения или провести какие-либо другие действия обязаны будут эти слушатели, заинтересованные в данных нашей модели.

Наконец, таблица и кнопки (помещенные в дополнительную панель на юге окна) размещаются в окне, и оно выводится на экран. Запустив программу с примером, вы убедитесь, что стандартная модель исправно снабжает таблицу данными и заботится о том, чтобы информация обо всех изменениях тут же поступала в таблицу. Щелкая на кнопках, нетрудно убедиться, что данными модели прекрасно можно манипулировать и динамически. Единственное «но» — никогда не следует добавлять, удалять или каким-либо другим образом манипулировать большими объемами данных, которые вы храните в стандартной модели, уже после присоединения ее к виду (таблице). Таблица будет отображать каждое, даже самое незначительное изменение в данных, и это может привести к значительному снижению производительности. В таких случаях лучше написать собственную модель, в которой оповещать об изменениях в данных только после завершения изменений. Основной же объем данных, так же как это делалось для моделей списков или деревьев, следует добавлять перед присоединением модели к виду.

Стандартная модель довольно удобна, когда вам не хочется думать о представлении своих данных в виде модели TableModel; она позволяет просто добавить строки и задать информацию о столбцах так, как будто вы работаете с очередным контейнером данных, подобным списку или очереди. Но чаще гораздо лучше создать собственную модель, в которой можно сполна учесть специфику источника данных, получить данные, при необходимости подготовить и отсортировать их, сообщить таблице дополнительную информацию об этих данных (в том числе и о том, можно ли их изменять, и каков их тип). Собственная модель позволяет также учесть, когда удобнее всего сообщать присоединенным к модели слушателям об изменениях в данных (вы помните, что слишком частые оповещения об изменениях приводят к снижению производительности). Благодаря модели все действия, связанные с данными, будут выполняться в одном месте, что упрощает обновление и расширение программы.

Есть два способа создать свою модель данных для таблицы. Во-первых, можно «с нуля» реализовать описывающий модель данных таблицы интерфейс TableModel, обеспечив всю функциональность модели самостоятельно (в том числе придется реализовать поддержку слушателей). Во-вторых, можно унаследовать класс своей модели от абстрактного класса AbstractTableModel, в котором уже реализована поддержка слушателей и есть удобные методы для инициирования событий нужного вам вида. Второй способ проще и быстрее, так что мы выбираем его. Для того чтобы наша модель заработала, нужно совсем немного — написать три метода: методы getRowCount() и getColumnCount() позволяют таблице определять количество строк и столбцов, а основной метод модели getValueAt() возвращает значение ячейки с указанными номерами строки и столбца. Рассмотрим пример, в котором создадим простую модель на основе класса AbstractTableModel:

```
// SimpleTableModel.java
// Создание простой модели для таблицы
import javax.swing.*;
import javax.swing.table.*;
import java.awt.*;

public class SimpleTableModel extends JFrame {
  public SimpleTableModel() {
    super("SimpleTableModel");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем таблицу на основе нашей модели
```

```
    JTable table = new JTable(new SimpleModel());
    table.setRowHeight(32);
    add(new JScrollPane(table));
    // выводим окно на экран
    setSize(400, 300);
    setVisible(true);
  }
  // наша модель
  class SimpleModel extends AbstractTableModel {
    // количество строк
    public int getRowCount() {
      return 100000;
    }
    // количество столбцов
    public int getColumnCount() {
      return 3;
    }
    // тип данных, хранимых в столбце
    public Class getColumnClass(int column) {
      switch (column) {
        case 1: return Boolean.class;
        case 2: return Icon.class;
        default: return Object.class;
      }
    }
    // данные в ячейке
    public Object getValueAt(int row, int column) {
      boolean isEven = (row % 2 == 0);
      // разные данные для разных стобцов
      switch (column) {
        case 0: return "" + row;
        case 1: return isEven;
        case 2: return new ImageIcon("Table.gif");
      }
      return "Пусто";
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new SimpleTableModel(); } });
```

```
    }
}
```

Мы создаем небольшое окно, в панель содержимого которого добавляем таблицу, включенную в панель прокрутки JScrollPane. Данные для таблицы будет поставлять наша собственная модель SimpleModel, которую мы унаследовали от абстрактного класса AbstractTableModel. Модель мы передаем прямо в конструктор класса JTable (хотя вы можете использовать для присоединения модели и метод setModel()), этого достаточно, чтобы таблица начала отображать данные, хранимые моделью. Дополнительно мы немного увеличиваем высоту строк таблицы методом setRowHeight(). Как вы сможете увидеть при запуске программы с примером, это позволяет таблице лучше справиться с отображением значков (в модели есть столбец, данные которого представляют собой значки Icon).

Самое интересное скрыто в модели. Для того чтобы определить структуру и тип данных, а также передать таблице эти данные, нам приходится переопределять несколько описанных в интерфейсе TableModel методов, в которых и выполняется вся работа. Давайте кратко опишем эти методы, а затем обсудим, как мы использовали их в нашем примере. При создании таблиц к этим методам приходится обращаться так часто, что табл. 17.2 пригодится нам в качестве удобного справочника (звездочкой отмечены те методы, которые *обязательно* необходимо описать при наследовании от класса AbstractTableModel, чтобы создать простейшую модель для таблицы).

**Таблица 17.2.** Основные методы модели данных таблицы TableModel

| Название метода | Описание |
|---|---|
| getRowCount()* | Метод должен возвращать целое число — количество строк в таблице. Если вы изменяете количество строк, не забудьте оповестить об этом слушателей модели (чаще всего это таблицы или их UI-представители), чтобы они смогли сразу же отобразить изменения. Это верно и для остальных данных модели — при их изменении не забывайте оповещать слушателей |
| getColumnCount()* | Метод возвращает количество столбцов в вашей таблице в виде целого (int) числа. Количество столбцов, как и количество строк, также может изменяться по мере работы программы, и об этом необходимо сообщать слушателям модели чтобы увидеть изменения на экране |
| getValueAt(строка, столбец)* | Основной метод модели данных таблицы. Позволяет указать, какие данные находятся в ячейке, определяемой по указанным строке и столбцу. Данные могут иметь любой тип (возвращается ссылка на базовый тип Object). Отсчет строк и столбцов, как нетрудно догадаться, ведется с нуля |
| getColumnName(столбец) | Метод позволяет задать имя для столбца, которое отображается в заголовке таблицы JTableHeader (как вы помните, заголовок таблицы появляется, только если последняя помещается в панель прокрутки). Реализация в абстрактном классе AbstractTableModel именует столбцы как в электронных таблицах: латинскими буквами, сначала одиночными, затем сдвоенными, и т. д. |
| isCellEditable(строка, столбец) | Метод позволяет указать, можно ли редактировать ячейку в таблице с указанным местоположением. Если в вашей таблице есть редактируемые ячейки, не забудьте определить метод setValueAt(), иначе как бы старательно пользователь не редактировал данные, в модели (а значит, и на экране) изменения отражаться не будут |

**Таблица 17.2** *(продолжение)*

| Название метода | Описание |
|---|---|
| setValueAt(значение, строка, столбец) | Метод используется для изменения значения некоторой ячейки таблицы. Реализуйте этот метод, если в вашей таблице есть редактируемые ячейки, иначе их значение невозможно будет поменять |
| getColumnClass(столбец) | Один из самых интересных методов модели данных таблицы. Позволяет задать тип данных, хранимых в столбце (тип задается в виде объекта Class). На основе типа данных столбца таблица определяет, как следует отображать и редактировать эти данные. Таблица JTable стандартно поддерживает несколько типов данных для столбцов, подробнее мы их вскоре обсудим |

Как видно из кода примера, мы в нашей модели SimpleModel сразу же задаем количество строк и столбцов. Обратите внимание, какое гигантское количество строк в нашей модели, и как просто его задать. Используй мы построчное добавление в стандартную модель или двухмерные массивы, написание даже простой программы рисковало превратиться в сущий ад. В методе getValueAt() возвращается значение каждой ячейки таблицы: для первого столбца мы возвращаем номер строки, для второго – логическую переменную isEven, которая автоматически преобразуется языком в объект Boolean (она равна true для четных строк и false для нечетных), для третьего – значок ImageIcon, созданный на основе файла в формате GIF с небольшим изображением. Тут же возникает вопрос: «Как же таблица сможет это все отобразить?» Мы уже привыкли, что для правильного отображения нестандартных данных (отличных от обычных строк) приходится писать вспомогательные объекты, но как оказывается, таблица JTable способна справиться со многими объектами сама, нужно лишь правильно вызывать метод модели getColumnClass().

Как мы уже отметили в табл. 17.2, метод getColumnClass() позволяет указать тип (а это в Java эквивалентно классу, так что метод возвращает объект Class) данных столбца, так чтобы таблица соответствующим образом отображала и позволяла редактировать эти данные. Типы данных для столбцов, которые поддерживает таблица JTable, перечислены в табл. 17.3.

**Таблица 17.3.** Типы данных для столбцов таблиц

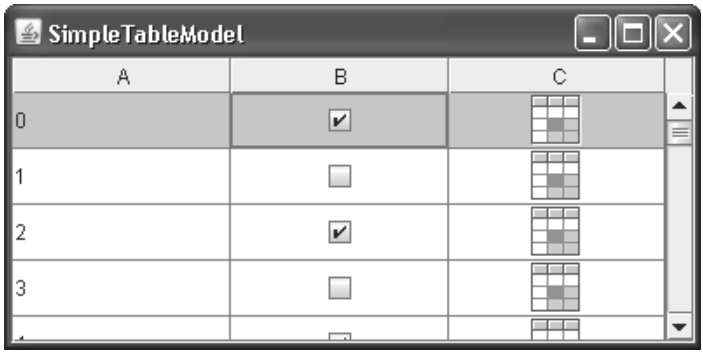
| Тип данных | Описание |
|---|---|
| Number.class | Столбец с таким типом данных должен хранить данные в виде объектов Integer, Long, Byte и т. д., то есть в виде стандартных классов-оболочек для целых чисел, унаследованных от базового класса Number. Таблица позаботится о том, чтобы числа отображались в подходящем виде (к примеру, числа в ячейке выравниваются по правому краю, хотя это зависит от выбранного языка) и чтобы при редактировании пользователь мог вводить в качестве нового значения ячейки только целые числа в подходящем диапазоне (в противном случае пользователь не сможет закончить редактирование и вернуться к работе с таблицей) |
| Float.class или Double.class | Если столбец имеет такой тип данных, то он должен хранить числа с плавающей запятой, «обернутые» в объекты Float или Double. Таблица будет отображать эти объекты как числа с плавающей запятой и соответствующим образом проводить редактирование, запрещая пользователю ввод неподходящих значений |

**Таблица 17.2** *(продолжение)*

| Тип данных | Описание |
|---|---|
| Boolean.class | В столбце такого типа хранятся данные о логических переменных (boolean), «обернутых» в стандартный класс Boolean. Данные такого столбца таблица отображает, помещая в ячейку экземпляр флажка JCheckBox, который может быть и редактируемым (если ячейка с таким типом данных редактируема, состояния флажка можно изменить, используя мышь или клавиатуру) |
| Date.class | Удобный тип столбца, особенно полезный в таблицах, отображающих результаты запросов к базам данных, где даты встречаются очень часто. Столбец с таким типом должен хранить данные в виде дат Date из пакета java.util. Таблица будет отображать даты согласно локальным настройкам операционной системы (в разных странах стандарты отображения дат различны), используя для форматирования объект DateFormat. Правда, в Swing нет компонентов или стандартных диалоговых окон для ввода дат, так что их редактирование проводится так же, как редактирование обычных строк, что неудобно и запутывает пользователя. Поэтому, если в вашей программе нужно редактировать даты, проще написать собственный редактор ячеек с таким типом. Как это сделать, мы вскоре увидим |
| Icon.class | Очень полезный тип столбца, указывающий, что в нем хранятся значки Icon, к которым в том числе относятся и значки на основе изображений ImageIcon. Как правило, ячейки со значками не редактируются, так что особого редактора для значков в таблице нет, а если он вам понадобится, придется написать его самостоятельно |

По умолчанию считается, что данные в столбце могут иметь любой тип, таким данным соответствует тип Object.class. Данные этого типа отображаются как обычные строки: для любого объекта вызывается метод toString(), полученная строка и выводится таблицей в соответствующей ячейке. При редактировании столбцов с таким типом данных на вводимые пользователем данные не накладывается никаких ограничений.

Используя метод getColumnClass(), мы с легкостью можем указать таблице, что в наших столбцах хранятся нестандартные данные, так что она сможет соответствующим образом вывести их на экран. Для второго столбца нашей таблицы мы будем использовать тип Boolean.class (так что логические переменные отобразятся флажками), а для третьего — Icon.class, после чего таблица будет выводить в соответствующих ячейках не текст, а значки. Для первого столбца нашей таблицы можно было бы указать тип Integer.class, но мы для простоты ограничились простыми строками String, автоматически преобразуя любые объекты и примитивные типы в строки.



Запустив программу с примером, вы сможете оценить, как описанная в примере простая модель поставляет данные для таблицы JTable. Заметьте, что программа работает быстро, хотя таблица отображает огромное количество записей. Собственная модель, в отличие от стандартной, позволяет учитывать внутреннюю структуру данных (в нашем примере она очень проста, поскольку мы не храним ни одного из значений ячеек нашей модели, а создаем их по мере необходимости), так что при возможности «поставка » в таблицу может быть максимально оптимизирована.

Далее мы увидим, что внешний вид столбцов и способ редактирования хранящихся в них ячеек может быть настроен гибко. Это можно сделать с помощью типа столбца, который вы сообщаете методом модели getColumnClass(). При этом вы не ограничены стандартными типами данных для столбцов и можете регистрировать новые типы, указывая таблице JTable, как их нужно отображать и редактировать. Можно также действовать «в обход» модели, указав, как следует отображать и редактировать данные для некоторого столбца, представленного объектом TableColumn. Пока же нам важно то, как модель позволяет гибко описать все ячейки таблицы независимо от того, где и как эти ячейки будут отображаться.

## Модель таблицы для работы с базами данных

Большая часть современных приложений так или иначе взаимодействует с базами данных, либо напрямую сотрудничая с системами управления базами данных (СУБД), либо посредством вспомогательных компонентов, выполняющих дополнительные действия, таких как EJB-компоненты (аббревиатура EJB означает Enterprise JavaBeans). Идея разделения данных и места, где эти данные (или с их часть) обрабатываются, давно признана удачной и активно эксплуатируется, позволяя эффективно разделять, обновлять и защищать данные (в меньшем масштабе эта идея прекрасно прижилась в библиотеке Swing, в которой разделены модели, представляющие собой данные, и виды, эти данные отображающие).

Не составляет большого труда заметить, что в Swing лучше всего для отображения результатов запросов к базам данных подходит именно таблица JTable. Она с легкостью отобразит любое количество столбцов и строк, позволит задать различные типы данных для столбцов, соответствующим образом редактировать и отображать их (а в современных базах данных типы хранимой информации задаются именно для столбцов), при необходимости внести в данные изменения и передать модифицированные результаты обратно в хранилище. Самое приятное, что благодаря интерфейсу TableModel мы можем один раз создать модель, получающую информацию из базы данных, а затем использовать ее для вывода в таблицах JTable результатов запросов к разнообразным базам данных.

Работать с базами данных мы будем с помощью интерфейса JDBC, в котором результат запроса к базе данных возвращается в виде объекта ResultSet. С помощью этого объекта несложно получить данные любого типа, считывая их по строкам и столбцам. Для каждого объекта ResultSet существует дополнительная информация в виде объекта ResultSetMetaData, позволяющая выяснить количество, типы и названия столбцов, полученных в результате запроса. Мы используем эту информацию для настройки нашей модели. Итак, приступим:

```
// com/porty/swing/DatabaseTableModel.java
// Модель данных таблицы, работающая
// с запросами к базам данных
package com.porty.swing;

import javax.swing.table.*;
import java.sql.*;
```

```
import java.util.*;

public class DatabaseTableModel
    extends AbstractTableModel {
  // здесь мы будем хранить названия столбцов
  private ArrayList<String> columnNames = new
ArrayList<String>();
  // список типов столбцов
  private ArrayList<Class> columnTypes = new ArrayList<Class>();
  // хранилище для полученных данных из базы данных
  private ArrayList<ArrayList<Object>> data
      = new ArrayList<ArrayList<Object>>();
  // признак редактирования таблицы
  private boolean editable;
  // конструктор позволяет задать возможность редактирования
  public DatabaseTableModel(boolean editable) {
    this.editable = editable;
  }
  // количество строк
  public int getRowCount() {
    return data.size();
  }
  // количество столбцов
  public int getColumnCount() {
    return columnNames.size();
  }
  // тип данных столбца
  public Class getColumnClass(int column) {
    return columnTypes.get(column);
  }
  // название столбца
  public String getColumnName(int column) {
    return columnNames.get(column);
  }
  // данные в ячейке
  public Object getValueAt(int row, int column) {
    return (data.get(row)).get(column);
  }
  // возможность редактирования
  public boolean isCellEditable(int row, int column) {
    return editable;
  }
```

```
    // замена значения ячейки
    public void setValueAt(
        Object value, int row, int column){
      (data.get(row)).set(column, value);
    }
    // получение данных из объекта ResultSet
    public void setDataSource(
        ResultSet rs) throws Exception {
      // удаляем прежние данные
      data.clear();
      columnNames.clear();
      columnTypes.clear();
      // получаем вспомогательную информацию о столбцах
      ResultSetMetaData rsmd = rs.getMetaData();
      int columnCount = rsmd.getColumnCount();
      for ( int i=0; i<columnCount; i++) {
        // название столбца
        columnNames.add(rsmd.getColumnName(i+1));
        // тип столбца
        Class type =
            Class.forName(rsmd.getColumnClassName(i+1));
        columnTypes.add(type);
      }
      // получаем данные
      while ( rs.next() ) {
        // здесь будем хранить ячейки одной строки
        ArrayList<Object> row = new ArrayList<Object>();
        for ( int i=0; i<columnCount; i++) {
          row.add(rs.getObject(i+1));
        }
        data.add(row);
      }
      // сообщаем об изменениях в структуре данных
      fireTableStructureChanged();
    }
}
```

Мы назвали нашу модель DatabaseTableModel и разместили ее в пакете com.porty.swing, так что при необходимости вы сможете включать ее в программы и без особых трудностей наглядно выводить результаты запросов к базам данных. Основной метод нашей модели — setDataSource(). Он принимает результат запроса к базе данных ResultSet, удаляет хранящиеся в модели данные, полученные после обработки предыдущих запросов (если таковые были), и заново описывает структуру столбцов таблицы на основе типов

данных полученного запроса. Все данные, включая названия и типы данных (объекты Class) столбцов, хранятся в списках ArrayList. При получении в метод setDataSource() нового результата запроса к базе данных мы удаляем из списков всю прежнюю информацию и приступаем к обработке объекта ResultSet.

Как уже было упомянуто, с каждым объектом ResultSet ассоциирован другой объект ResultSetMetaData, содержащий вспомогательную информацию о результатах запроса. Он позволит нам гибко, не вдаваясь в подробности того, к какой базе данных мы подключены, узнавать все необходимое для построения подходящей модели. Прежде всего мы получаем количество столбцов в полученном результате запроса (с помощью метода getColumnCount()), и для каждого столбца выясняем его название (это название столбца в базе данных, чаще всего оно не слишком удобно для прочтения человеком, но в реальном приложении вы сможете легко сменить идентификаторы столбцов на что-либо более подходящее) и тип (тип, как всегда в Java, аналогичен классу, так что мы получаем название класса и создаем его экземпляр статическим методом forName()). Обратите внимание на то, что при получении информации из объекта ResultSetMetaData (а затем и из объекта ResultSet) отсчет столбцов ведется *с единицы*, а не с нуля, как в остальных случаях. Так уж устроен интерфейс JDBC, это правило стоит помнить, иначе при попытке получения нулевого столбца запроса вы вместо данных «словите» исключение.

Далее можно приступать непосредственно к работе с данными. Для каждой строки с данными создается свой отдельный список ArrayList, в него будут поочередно добавляться значения данных всех столбцов текущей строки запроса. Данные можно просто получать методом getObject(), пригодным на все случаи жизни, класс ResultSet обещает вернуть в нем подходящий типу столбца объект. После заполнения данными очередной строки она добавляется в список всех данных data.

Как только мы выясняем «всю подноготную» новых столбцов таблицы и наполняем списки строками таблицы, вызывается метод fireTableStructureChanged(), который оповестит слушателей о том, что структура столбцов таблицы поменялась. Такое оповещение приведет к отображению на экране (в том случае если модель присоединена к виду, то есть к таблице JTable), и пользователь увидит, какие столбцы присутствуют в новом наборе данных. При смене структуры столбцов все данные также будут считаны заново и отображены на экране.

Остальные методы нам прекрасно знакомы и служат для правильной работы модели. Большая часть из них получает информацию (название столбца, количество строк и столбцов, тип столбца) из списков ArrayList, которые заполняются данными в уже рассмотренном нами методе setDataSource(). Конструктор нашей модели позволяет указать, будет ли она разрешать редактирование. Новые данные записываются в списке в методе модели setValueAt(), который вызывает таблица при окончании редактирования.

А теперь проверим работу нашей новой модели. Приведем простой пример:

```
// DatabaseTable.java
// Таблица, работающая с базой данных
// посредством специальной модели
import java.sql.*;
import java.awt.*;
import javax.swing.*;
import com.porty.swing.*;

public class DatabaseTable {
  // параметры подключения
  private static String
```

```
      dsn = "jdbc:odbc:Library",
      uid = "",
      pwd = "";
  public static void main(String[] args) throws Exception {
    // инициализация JDBC
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    // объект-соединение с БД
    Connection conn = DriverManager.getConnection(dsn, uid, pwd);
    Statement st = conn.createStatement();
    // выполняем запрос
    ResultSet rs = st.executeQuery(
        "select * from readers.csv");
    // наша модель
    final DatabaseTableModel dbm =
        new DatabaseTableModel(true);
    // считываем данные в таблицу
    dbm.setDataSource(rs);
    // таблица и окно
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() {
            JTable table = new JTable(dbm);
            JFrame frame = new JFrame("DatabaseTable");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setSize(400, 300);
            frame.add(new JScrollPane(table));
            frame.setVisible(true);
          }});
    rs.close();
    conn.close();
  }
}
```

В программе мы непосредственно в методе main() создаем небольшое окно JFrame и размещаем в нем таблицу, которая будет получать данные из только что созданной модели для работы с базами данных. Конечно же, мы работаем с компонентами из потока рассылки событий, так как хорошо знаем правила Swing. Для правильной работы модели нам нужно настроить и открыть подключение к базе данных JDBC. Здесь используется все то же подключение к базе данных некой гипотетической библиотеки, мы уже применяли его в главе 11, когда создавали модели списков JList и JComboBox для работы с базами данных. В качестве драйвера задействован стандартный мост JDBC-ODBC, годящийся в большинстве случаев жизни. Для работы программы с вашей базой данных вам нужно сменить лишь идентификатор базы данных, имя пользователя и пароль, а также, если это необходимо, драйвер JDBC, в том случае если вы используете специальную базу данных с собственным драйвером. Открыв соединение с базой данных, программа вы-

полняет запрос (в примере запрос получает список всех читателей библиотеки), и если последний успешен, то результат запроса возвращается в виде объекта ResultSet. Заметьте, что мы передаем объект ResultSet нашей модели еще до того, как таблица присоединяется к ней, добавляется в окно, и последнее выводится на экран. Это позволяет нам загрузить данные из отдельного потока выполнения метода main(), так как таблица пока не обращается к модели. В противном случае, если модель уже присоединена к таблице, необходимо выполнять все действия из потока рассылки событий, иначе конфликты неизбежны.

Запустив программу с примером, предварительно настроив любую вашу базу данных и выполнив запрос по получению любых, даже самых сложных данных, вы увидите, что они правильно появляются в таблице, и структура таблицы имеет нужный вид (название и тип столбцов), несмотря на всю простоту написанной нами модели. Действительно, JDBC и таблица JTable словно созданы друг для друга.


Конечно, рассмотренная нами модель на самом деле простовата для работы с реальными базами данных, в которых, как правило, хранится информация впечатляющего размера. Она будет занимать слишком много места в памяти, так как вычитывает всю таблицу в списки и будет вызывать замедление в интерфейсе, поскольку процесс считывания данных необходимо выполнять в потоке рассылки событий одним вызовом, что на время остановит прорисовку и отзывчивость интерфейса. Можно развить наш пример в различных направлениях:– например, вычитывать ограниченный набор данных, видимый в данный момент на экране, а при прокрутке таблицы читать недостающие строки, либо применять объект ResultSet с поддержкой прокрутки по записям таблицы в произвольном направлении, что позволит вообще не хранить данных в модели, но может быть не слишком производительным. Эта увлекательная тема, однако, относится скорее к работе с базами данных, чем к библиотеке Swing. Способы добиться максимальной производительности мы уже неоднократно обсуждали при работе с другими сложными компонентами Swing, списками JList и деревьями JTree — точечное обновление данных, работа с блоками данных и взаимодействие с потоком рассылки событий вы можете найти в главах 11 и 15. Все эти техники в одинаковой мере относятся ко всем ситуациям с большим количеством данных в модели.

Мы можем подвести предварительный итог: основная модель таблицы TableModel удивительно изящна и гибка; она позволяет быстро описать даже самые сложные двухмерные данные с учетом всей их специфики. Удачной следует признать и концепцию типа столбца (представленного объектами Class) — вы можете быстро указать, как следует отображать (и при необходимости редактировать) данные этого столбца, потому что при наличии встроенной поддержки таких фундаментальных типов, как значки, булевы значения и числа, таблица даже без специализированной настройки становится мощным инструментом в руках создателя пользовательского интерфейса.

## Модель столбцов таблицы

Столбцы таблицы JTable обладают по-настоящему гибким поведением: вы можете менять их размеры, менять их местами друг с другом, настраивать объекты для отображения в столбце и для редактирования ячеек столбца, тонко управлять заголовком столбца и делать многое другое — все аспекты поведения столбцов таблицы настраиваются. В тоже время в модели данных таблицы TableModel столбцы представлены только своим названием и типом хранящихся в них данных. Остальная информация о столбцах, а ее немало, хранится в специальной предназначенной специально для них модели TableColumnModel.

Модель TableColumnModel отвечает за то, какие столбцы, в каком порядке и с какими параметрами выводит на экран таблица JTable. Грубо говоря, это специализированный контейнер данных, который хранит список столбцов таблицы в том порядке, в каком они представлены на экране (в отличие от модели данных TableModel, в которой порядок следования столбцов всегда один и тот же независимо от того, как столбцы располагаются на экране). Каждый столбец представлен объектом TableColumn. С помощью этого объекта вы можете указать размеры столбца (у столбца, как и у обычного компонента, может быть предпочтительный, максимальный и минимальный размеры), индекс данного столбца в модели данных TableModel (по этому индексу таблица определяет, какой набор данных отображает столбец), объекты для отображения и редактирования данных, хранящихся в столбце (об этих объектах мы вскоре поговорим подробнее, когда доберемся до детального обсуждения внешнего вида ячеек таблицы), и настроить многие другие параметры, в том числе и модель выделения столбцов.

По умолчанию в таблице для хранения информации о столбцах используется стандартная модель DefaultTableColumnModel, и ее возможностей хватает в подавляющем большинстве ситуаций. Работа модели столбцов таблицы не слишком сложна: она хранит список столбцов TableColumn и при смене какого-либо свойства столбцов или их расположения оповещает об этом событии слушателей TableColumnModelListener. Стандартная модель прекрасно справляется с оповещением слушателей, а столбцы хранит в динамическом массиве (векторе) Vector. Как правило, писать собственную модель столбцов таблицы «с нуля» или наследуя от стандартной модели не имеет особого смысла — стандартная модель позволяет динамически манипулировать столбцами и проводить с ними все необходимые операции. Также несложно с помощью слушателей TableColumnModelListener постоянно быть в курсе всех изменений в структуре столбцов таблицы.

А сейчас в несложном примере рассмотрим основные возможности стандартной модели столбцов таблицы и объектов TableColumn:

```
// UsingTableColumnModel.java
// Использование стандартной модели столбцов
// таблицы и объектов TableColumn
import javax.swing.*;
import javax.swing.table.*;
import java.awt.event.*;
import java.awt.*;
import java.util.Enumeration;

public class UsingTableColumnModel extends JFrame {
  // модель столбцов
  private TableColumnModel columnModel;
  // названия столбцов таблицы
```

```
private String[] columnNames = {
    "Имя", "Любимый цвет", "Напиток"
};
// данные для таблицы
private String[][] data = {
    { "Иван", "Зеленый", "Апельсиновый сок"},
    { "Александр", "Бежевый", "Зеленый чай"}
};
public UsingTableColumnModel() {
  super("UsingTableColumnModel");
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  // наша таблица
  JTable table = new JTable(data, columnNames);
  // получаем стандартную модель
  columnModel = table.getColumnModel();
  // меняем размеры столбцов
  Enumeration e = columnModel.getColumns();
  while ( e.hasMoreElements() ) {
    TableColumn column =
        (TableColumn)e.nextElement();
    column.setMinWidth(30);
    column.setMaxWidth(90);
  }
  // создадим пару кнопок
  JPanel buttons = new JPanel();
  JButton move = new JButton("Поменять местами");
  move.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      // меняем местами первые два столбца
      columnModel.moveColumn(0, 1);
    }
  });
  buttons.add(move);
  JButton add = new JButton("Добавить");
  add.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      // добавляем столбец
      TableColumn newColumn =
          new TableColumn(1, 100);
      newColumn.setHeaderValue("<html><b>Новый!");
      columnModel.addColumn(newColumn);
```

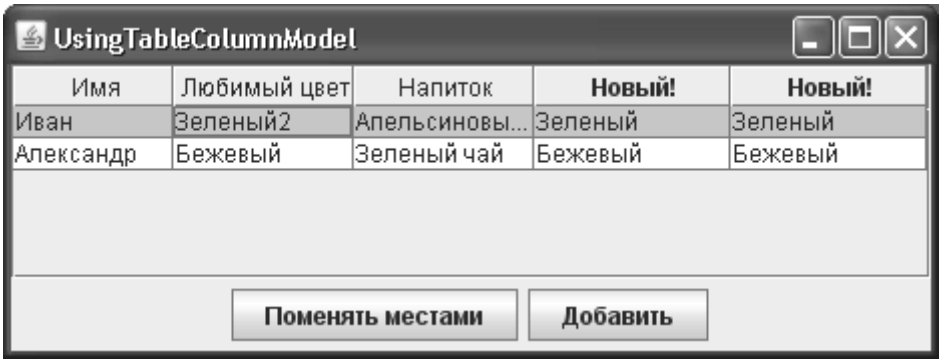
```
        }
      });
      buttons.add(add);
      // выводим окно на экран
      add(new JScrollPane(table));
      add(buttons, "South");
      setSize(400, 300);
      setVisible(true);
    }
    public static void main(String[] args) {
      SwingUtilities.invokeLater(
          new Runnable() {
            public void run() { new UsingTableColumnModel(); } });
    }
}
```

Мы создаем небольшое окно, в центре которого будет располагаться небольшая таблица JTable. Таблицу мы создаем на основе двухмерного массива с данными и массива с названиями столбцов. Все это нам уже прекрасно знакомо. Дальше интереснее: мы получаем стандартную модель столбцов таблицы методом getColumnModel() и можем приступать к работе с ней. Прежде всего, обратите внимание на то, как модель столбцов позволяет получить перечисление (Enumeration) имеющихся в данный момент в таблице столбцов и изменить их размеры. По умолчанию минимальным размером для любого столбца является 15 пикселов, предпочтительным — 75 пикселов, а максимальный размер его не ограничен. Однако столбцы у таблиц бывают разные, и эти значения на все случаи жизни не подходят, так что нет ничего удивительного в том, что чаще всего работа со столбцами сводится к настройке их размеров. В примере у нас нет особых требований к столбцам, так что мы можем сделать их любого понравившегося нам размера, видно, как меняются минимальный и максимальный размеры столбцов. Запустив программу с примером, вы сразу оцените разницу: если раньше свободное пространство в таблице было бы равномерно распределено между столбцами, то теперь, благодаря ограниченному максимальному размеру, свободное место просто ничем не занято — столбцы никогда не становятся длиннее своего максимального размера (и соответственно, короче минимального).

Далее пример показывает, каким образом модель выделения столбцов (в лице ее стандартной реализации) позволяет динамически манипулировать столбцами таблицы. В южную часть окна добавляется панель с двумя кнопками, к каждой из которых присоединен слушатель ActionListener. Первая кнопка меняет местами первый и второй столбцы таблицы, специально для этого в модели имеется удобный метод move(). Еще в самом первом примере данной главы мы отметили, что по умолчанию таблица позволяет нам мышью «тасовать» столбцы, меняя их местами. Подобное поведение обеспечивает именно модель столбцов TableColumnModel, и мы видим, что совсем нетрудно переставлять столбцы программно. Вторая кнопка показывает, как можно добавить в таблицу еще один столбец, даже если в модели основных данных TableModel этот столбец и не присутствует. В слушателе создается объект TableColumn, инкапсулирующий информацию о столбце таблицы. Мы использовали конструктор, который требует указать индекс столбца в модели данных TableModel (это обязательно, и такой столбец в модели данных должен присутствовать, иначе неизбежны исключения — столбцу будет нечего показывать) и предпочтительный размер столбца (остальные размеры задаются по

умолчанию, мы уже знаем их значения). Есть и более полные конструкторы, которые вы сможете отыскать в интерактивной документации, но всегда для любого столбца TableColumn должен иметься соответствующий столбец в модели данных TableModel. Полученный столбец добавляется в модель с помощью метода addColumn(). Правда, приходится вручную задавать для него текст заголовка (методом setHeaderValue(), и здесь поддерживается HTML): объект TableColumn ничего не знает о модели данных и о том, какой у него должен быть заголовок.

Таким образом, мы видим, что один столбец, описанный в модели данных, может отображаться на экране несколькими столбцами благодаря отдельной модели столбцов и ее гибкости. Впрочем, все же лучше, если изменения в количестве столбцов идут из одного источника, где их легко обнаружить и поддерживать: таким источником должна быть модель данных TableModel. Добавление столбцов посредством модели TableColumnModel имеет смысл, только если вы «копируете» некоторые столбцы, несколько раз воспроизводя их на экране. К тому же после запуска примера возникнет небольшая неприятность: если вы меняете данные в ячейке в одном столбце, дополнительный столбец не перерисовывается автоматически, он будет перерисован лишь тогда, когда поменяется его размер или выделение строки таблицы. Столбец таблицы — довольно низкоуровневая ее часть, и при работе с ним стоит это учитывать. В нашем случае необходимо вручную попросить все столбцы, отображающие изменившийся столбец, перерисовать себя, что конечно не совсем удобно.




Запустив программу с примером, вы сможете увидеть, как модель столбцов таблицы позволяет менять их размеры и динамически манипулировать местоположением и количеством столбцов, даже если в модели данных TableModel (бесспорно, главной модели таблицы JTable) новые столбцы не имеют своего представительства. Сведем наиболее полезные методы и свойства модели TableColumnModel в табл. 17.4.

**Таблица 17.4.** Самые полезные методы и свойства модели TableColumnModel

| Методы и свойства | Описание |
|---|---|
| addColumn(), removeColumn() | Методы дают возможность динамически управлять количеством хранимых в модели TableColumnModel столбцов. Первый метод присоединяет к модели новый столбец (заданный объектом TableColumn), второй удаляет из модели заданный столбец |
| addColumnModelListener(), removeColumnModelListener() | Эти два метода поддерживают присоединение и отсоединение слушателей TableColumnModelListener. Присоединив к модели такого слушателя, вы будете получать полную информацию обо всех происходящих со столбцами таблицы действиях |

**Таблица 17.4** *(продолжение)*

| Методы и свойства | Описание |
|---|---|
| getColumnCount(), getColumn(), getColumns() | Позволяют получить исчерпывающую информацию о содержащихся в модели столбцах. Первый метод сообщает количество столбцов, второй возвращает столбец по его порядковому номеру, и наконец, третий метод возвращает перечисление (Enumeration) всех содержащихся в модели столбцов |
| moveColumn() | Меняет местами два столбца таблицы, заданных (своими индексами в модели) в виде параметров данного метода. Не забывайте, что отсчет столбцов ведется с нуля |
| columnMargin (методы get/set) | Это свойство управляет расстоянием между столбцами таблицы (в пикселах) |

Это основные методы модели столбцов таблицы, но не забывайте также о том, что львиная доля свойств отдельного столбца задается в объекте TableColumn, который мы кратко рассмотрели. Основными свойствами любого столбца являются его размеры (и мы видели, как они меняются), а также собственные объекты для отображения и редактирования ячеек, если они заданы (до них мы вскоре доберемся и обсудим подробно). Чтобы при случае основные свойства столбца таблицы были у вас «под рукой», соорудим еще одну полезную таблицу (табл. 17.5).

**Таблица 17.5.** Важнейшие свойства столбца TableColumn

| Свойства | Описание |
|---|---|
| modelIndex | Индекс столбца в модели данных таблицы TableModel. Данные именно этого столбца отображает объект TableColumn |
| preferredWidth, minWidth, maxWidth | Предпочтительный, минимальный и максимальный размеры столбца таблицы (длина столбца в пикселах). Таблица всегда соблюдает минимальный и максимальный размеры столбца, так что эти свойства довольно «сильного» действия |
| resizable | Позволяет указать, можно ли будет динамически менять размеры столбца, или он всегда будет иметь предпочтительный размер |
| cellRenderer, cellEditor, headerRenderer | Объекты для отображения и редактирования ячеек данного столбца, а также объект, отображающий заголовок столбца. Мы обсудим их в разделе, посвященном внешнему виду ячеек таблицы |
| headerValue, identifier | Первое свойство хранит значение, которое будет отображаться в заголовке столбца, а второе, используемое довольно редко, позволяет связать со столбцом некоторый уникальный идентификатор. Применяя этот идентификатор, вы впоследствии всегда сможете обнаружить данный столбец в модели TableColumnModel (с помощью метода getColumnIndex()), как бы сильно не изменилась структура последней |

Как правило, модель столбцов таблицы и объекты TableColumn чаще всего требуются для настройки размеров столбцов. Таблица JTable разрешает пользователю менять размеры столбцов прямо во время работы программы (если только вы не установили свойство столбца resizable равным false), поэтому в дополнение к трем размерам столбца иногда необходимо управлять тем, как таблица будет перераспределять

пространство при изменении размеров того или иного столбца. Режим перераспределения пространства таблицы управляется специальным свойством autoResizeMode. Класс JTable поддерживает несколько режимов перераспределения пространства таблицы, давайте посмотрим, как работает каждый из них, с помощью следующего примера:

```
// TableResizeModes.java
// Режимы перераспределения пространства таблицы
// при изменении размеров столбцов
import javax.swing.*;
import java.awt.*;

public class TableResizeModes extends JFrame {
  // названия столбцов таблицы
  private String[] columnNames = {
      "Название", "Вкус", "Цвет"
  };
  // данные для таблицы
  private String[][] data = {
      { "Апельсин", "Кисло-сладкий", "Оранжевый"},
      { "Лимон", "Кислый", "Желтый"}
  };
  // массив таблиц
  private JTable[] tables = new JTable[5];
  public TableResizeModes() {
    super("TableResizeModes");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // окно делим на пять ячеек
    setLayout(new GridLayout(5, 1));
    // создаем массив таблиц
    for (int i=0; i<tables.length; i++) {
      tables[i] =
          new JTable(data, columnNames);
      add(new JScrollPane(tables[i]));
    }
    // меняем режимы распределения пространства
    tables[1].setAutoResizeMode(
        JTable.AUTO_RESIZE_OFF);
    tables[2].setAutoResizeMode(
        JTable.AUTO_RESIZE_NEXT_COLUMN);
    tables[3].setAutoResizeMode(
        JTable.AUTO_RESIZE_LAST_COLUMN);
    tables[4].setAutoResizeMode(
```

```
            JTable.AUTO_RESIZE_ALL_COLUMNS);
        // придаем окну оптимальный размер и
        // выводим его на экран
        pack();
        setVisible(true);

    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(
            new Runnable() {
              public void run() { new TableResizeModes(); } });
    }
}
```

В примере создается пять таблиц, отображающих одни и те же данные, полученные из массивов data (данные ячеек) и columnNames (названия столбцов). Таблицы размещаются в окне с менеджером расположения GridLayout, который разбивает панель на пять ячеек одинакового размера, располагающихся вертикально. Единственное различие в созданных таблицах заключается в том, что они задействуют различные режимы распределения пространства таблицы. Таблица JTable предоставляет нам пять таких режимов, пять таблиц мы и создали в примере. Обратите внимание, что для первой таблицы в массиве (tables[0]) режим не меняется – он остается используемым по умолчанию режимом AUTO_RESIZE_SUBSEQUENT_COLUMNS.

Запустив программу с примером, вы сможете своими руками «пощупать», чем характерен тот или иной режим перераспределения пространства, при этом вам наверняка пригодится краткое описание этих режимов, представленное в табл. 17.6.

**Таблица 17.6.** Режимы распределения пространства таблицы JTable

| Режим | Краткое описание |
|---|---|
| AUTO_RESIZE_OFF | В данном режиме пространство не перераспределяется — размеры столбцов, если вы не меняете их вручную, таблицей не меняются. По умолчанию размер столбца равен его предпочтительному размеру, если вы меняете размер столбца, то меняется только он, остальные столбцы не затрагиваются. Если после изменения размеров столбцов места в таблице начинает не хватать, появляется горизонтальная полоса прокрутки (в том случае если таблица была размещена в панели прокрутки JScrollPane, в противном случае крайние столбцы просто исчезают с экрана) |
| AUTO_RESIZE_NEXT_COLUMN | При изменении размеров столбца пространство перераспределяется только между ним и следующим за ним столбцом — если вы делаете столбец больше, следующий столбец становится меньше, и наоборот, при уменьшении столбца следующий за ним сосед забирает освободившееся пространство |
| AUTO_RESIZE_LAST_COLUMN | Столбцы можно увеличивать или уменьшать только за счет последнего столбца таблицы, все остальные столбцы (не считая того, размер которого вы меняете) остаются неизменными |

**Таблица 17.6** *(продолжение)*

| Режим | Краткое описание |
|---|---|
| AUTO_RESIZE_SUBSEQUENT_COLUMNS | Данный режим используется таблицей по умолчанию. При изменении размеров какого-либо столбца перераспределение пространства происходит равномерно между всеми следующими за ним столбцами. Последний столбец нельзя отдельно уменьшить или увеличить, разве что меняя размеры предпоследнего столбца |
| AUTO_RESIZE_ALL_COLUMNS | Перераспределение пространства в этом режиме происходит между всеми столбцами таблицы. Если вы, к примеру, увеличиваете размер некоторого столбца, то требуемое для этого пространство «отнимается» у всех остальных столбцов в равном соотношении (правда, если столбец достигает минимального размера или изначально не допускает изменения своих размеров, он перестает участвовать в «жертвовании» пространства) |

Вкупе с регулированием минимального, предпочтительного и максимального размеров столбца, которое предоставляет нам класс TableColumn, различные режимы перераспределения пространства таблицы дают нам возможность сделать размеры таблицы и ее ячеек именно такими, какими вы хотите их видеть. Не забывайте и о простых свойствах rowHeight и intercellSpacing, рассмотренных нами в начале главы. Они позволяют парой строк кода дополнить только что изученные возможности по управлению размерами столбцов — первое свойство отвечает за высоту строк таблицы, а второе задает расстояние между соседними ячейками как по вертикали, так и по горизонтали. Любые размеры таблицы теперь у вас в руках.

Напоследок об еще одном свойстве таблицы JTable, которое имеет непосредственное отношение к модели столбцов TableColumnModel. Это свойство autoCreateColumnsFromModel булева типа. Когда данное свойство равно true (а по умолчанию оно равно true), то происходит следующее: при присоединении модели данных TableModel таблица получает из этой модели информацию обо всех столбцах и на ее основе создает объекты TableColumn, которые затем передает в модель столбцов TableColumnModel. Если вы хотите получить больше контроля над тем, как, когда и какие столбцы будет выводить таблица на экран, сделайте это свойство равным false и добавляйте столбцы в модель вручную, когда настанет подходящий момент — в этом случае столбцы не будут создаваться автоматически. Правда, обращаться с данным свойством нужно аккуратнее и не забывать добавлять все нужные вам столбцы на экран, в противном случае от таблицы останется только рамка.

## Модели выделения

Ячейки таблицы можно выделять любыми способами, которые вы только в состоянии себе представить: строками, столбцами, интервалами, единичными ячейками и многими другими. Нетрудно догадаться, что такую гибкость в выделении содержимого таблицы обеспечивают очередные модели, верно служащие классу JTable. На этот раз это модели, хранящие выделенные строки и столбцы таблицы. И для столбцов, и для строк таблицы имеется собственная модель выделения, но не опасайтесь сложностей — это прекрасно знакомая нам еще с главы 11 (в которой мы изучали списки Swing) модель выделения элементов списка ListSelectionModel. Многое, что мы узнали об этой модели из главы 11, мы здесь повторять не будем, так что если вы не читали про модель выделения списка или просто немного забыли ее, самое время ненадолго вернуться назад.

Итак, модели выделения две. Первая модель хранится непосредственно в таблице JTable и отвечает за выделение строк таблицы. Вторая модель (представленная все тем

же интерфейсом ListSelectionModel) отвечает за выделение столбцов таблицы, но хранится она немного в другом месте: в модели столбцов таблицы TableColumnModel, которую мы недавно изучали. Как вы помните, модель выделения списка позволяет проводить выделение в одном из трех режимов: в режиме выделения одиночных элементов (SINGLE_SELECTION), в режиме выделения непрерывными интервалами (SINGLE_INTERVAL_SELECTION), а также в наиболее гибком режиме выделения произвольного набора элементов MULTIPLE_INTERVAL_SELECTION (в случае таблицы элементами для выделения являются строки или столбцы). Таким образом, для строк и для столбцов имеется по три режима выделения, так что всего таблица предоставляет нам девять комбинаций — впечатляющая гибкость, способная удовлетворить даже самым взыскательным нуждам.

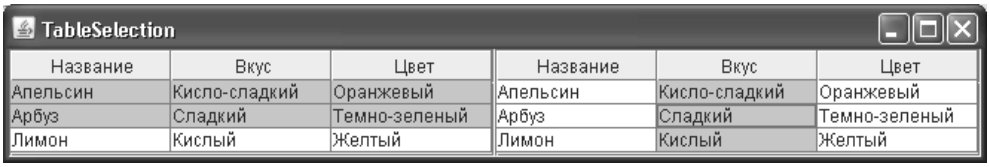
Нетрудно догадаться, что по умолчанию в таблице и для строк, и для столбцов используется стандартная реализация интерфейса ListSelectionModel — класс DefaultListSelectionModel. Когда мы обсуждали выделение в списках, то пришли к выводу, что чаще всего писать собственную модель выделения нет особого смысла, стандартная реализация прекрасно подходит в подавляющем большинстве ситуаций. Она поддерживает списки слушателей ListSelectionListener, которые оповещаются об изменениях в выделенных строках и столбцах таблицы, и все три режима выделения. Только при реализации нетривиального поведения таблицы вам может понадобиться создание собственной модели выделения (как правило, путем наследования ее от класса DefaultListSelectionModel, как это делается, рассказано в главе 11).

Разберем теперь небольшой пример, в котором взглянем на некоторые способы управления выделением строк и столбцов в таблице JTable:

```
// TableSelection.java
// Режимы выделения ячеек таблицы
import javax.swing.*;
import javax.swing.table.*;
import java.awt.*;

public class TableSelection extends JFrame {
  // названия столбцов таблицы
  private String[] columnNames = {
      "Название", "Вкус", "Цвет"
  };
  // данные для таблицы
  private String[][] data = {
      { "Апельсин", "Кисло-сладкий", "Оранжевый"},
      { "Арбуз", "Сладкий", "Темно-зеленый"},
      { "Лимон", "Кислый", "Желтый"}
  };
  public TableSelection() {
    super("TableSelection");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем пару таблиц с одинаковыми данными
    // первая таблица — выделение по строкам
    JTable table1 = new JTable(data, columnNames);
    table1.getSelectionModel().setSelectionMode(
        ListSelectionModel.SINGLE_INTERVAL_SELECTION);
```

```
        // вторая таблица — выделение по столбцам
        JTable table2 = new JTable(data, columnNames);
        table2.setRowSelectionAllowed(false);
        TableColumnModel cm = table2.getColumnModel();
        cm.setColumnSelectionAllowed(true);
        cm.getSelectionModel().setSelectionMode(
            ListSelectionModel.SINGLE_SELECTION);
        // добавляем таблицы и выводим окно на экран
        setLayout(new GridLayout(1, 2));
        add(new JScrollPane(table1));
        add(new JScrollPane(table2));
        pack();
        setVisible(true);
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(
            new Runnable() {
              public void run() { new TableSelection(); } });
    }
}
```

Здесь мы создаем две таблицы с одинаковыми данными, но с разными режимами выделения. Первая таблица будет разрешать выделение только по строкам (это поведение по умолчанию, столбцы таблицы не выделяются, пока вы явно не укажете, что вам это необходимо). Строки таблицы будут выделяться непрерывными интервалами, посмотрите, как выбирается этот режим: сначала мы получаем модель выделения строк методом getSelectionModel() класса JTable, а затем меняем режим работы полученной модели методом setSelectionMode(). Во второй таблице работает выделение по столбцам. Чтобы добиться этого, понадобилось несколько шагов: сначала мы отключили выделение по строкам с помощью свойства rowSelectionAllowed класса JTable, затем получили модель столбцов таблицы TableColumnModel, именно она отвечает за выделение столбцов, и с ее помощью настроили нужное нам поведение. Свойство columnSelectionAllowed позволяет включить выделение столбцов, а модель выделения столбцов, возвращаемая все тем же методом getSelectionModel(), определенным на этот раз в классе TableColumnModel, позволяет выбрать для столбцов любой из трех режимов выделения (в примере мы выбираем режим выделения одного столбца). Напоследок в окно добавляются обе таблицы. Запустив программу и выделяя строки первой таблицы и столбцы второй, вы увидите, как подействовали на таблицу сделанные нами изменения.




Изначально в таблице задействован режим выделения произвольной комбинации строк, а столбцы выделять нельзя, как это изменить, мы только что увидели. Ес-

ли вы включаете одновременно выделение и строк, и столбцов, то по щелчку мыши начинают выделяться отдельные ячейки, и в зависимости от комбинации режимов выделения для строк и столбцов вы сможете выделять смежные блоки ячеек таблицы или блоки из произвольных ячеек. Все рычаги управления режимами выделения таблицы находятся во власти моделей выделения, и мы знаем, как с ними работать. Впрочем, как и всегда в Swing, вас никто не обязывает работать с выделением только посредством специальных моделей. Специально для тех, кого смущают модели, таблица JTable предоставляет богатый выбор методов и свойств, позволяющий полностью управлять режимами выделения таблицы и самими выделенными элементами, не задумываясь о том, где и как используются модели. Составим справочную таблицу с основными методами и свойствами, предназначенными для выделения (опишем и класс JTable, и модели выделения). К этой информации вы всегда сможете обратиться, если у вас возникнет вопрос, связанный с выделением элементов в таблице (табл. 17.7).

**Таблица 17.7.** Основные методы и свойства для работы с выделением в таблице

| Методы или свойства | Описание |
|---|---|
| selectionModel (класс JTable и модель TableColumnModel) | Данное свойство хранит модель выделения, в классе JTable — это модель выделения строк, в модели столбцов TableColumnModel — модель выделения столбцов. По умолчанию используется стандартная реализация DefaultListSelectionModel |
| rowSelectionAllowed (JTable), columnSelectionAllowed (модель TableColumnModel и класс JTable) | Первое свойство позволяет включить или отключить выделение строк (определено только в классе JTable), второе включает или отключает выделение столбцов (имеется и в модели выделения столбцов и в классе JTable). По умолчанию выделение по столбцам отключено |
| selectionMode (свойство моделей выделения и таблицы JTable) | Данное свойство позволяет задать режим выделения, используемый моделью выделения строк или столбцов. Свойство с тем же названием, определенное в классе JTable, задает одинаковый режим выделения сразу для обеих моделей |
| selectAll(), clearSelection() (класс JTable) | Методы класса JTable, первый из них позволяет «одним махом» выбрать все ячейки таблицы, второй дает возможность быстро снять все имеющееся на данный момент выделение |
| cellSelectionAllowed (класс JTable) | Удобное свойство класса JTable, если вы приравняете его значению true, то пользователь сможет выделять отдельные ячейки таблицы (тот же эффект получается, когда разрешено выделение и по строкам, и по столбцам) |
| rowSelectionInterval, columnSelectionInterval, (методы add, set и remove, класс JTable) | Весьма полезные свойства таблицы JTable, задают выделенные строки или столбцы (четыре метода для установки новых значений). Методы, начинающиеся с префикса «add», присоединяют к уже имеющемуся выделению новые строки и столбцы, методы с префиксом «set» заменяют имеющееся выделение новым, методы с префиксом «remove» удаляют из выделения указанные строки или столбцы |

Теперь, когда мы умеем управлять режимами выделения таблицы и настраивать выделенные ячейки, неплохо посмотреть на все с другой стороны и узнать, как таблица

и ее модели выделения позволяют узнавать, какие строки и столбцы выделяет пользователь. В табл. 17.8 собраны воедино основные методы, служащие этой цели.

**Таблица 17.8.** Методы для получения информации о выделенных ячейках

| Методы | Описание |
|---|---|
| isSelectedIndex() (метод модели выделения) | Метод позволяет узнать, выделен ли элемент на некоторой позиции. Именно данным методом модели выделения пользуются все остальные, более удобные в работе, методы |
| isRowSelected(), isColumnSelected(), isCellSelected() | Методы класса JTable, позволяющие быстро узнать, выделена ли строка (или столбец) с некоторым индексом. Третий метод дает возможность определить, выделена ли ячейка, заданная строкой или столбцом |
| getSelectedRows(), getSelectedColumns() | Также методы класса JTable, позволяющие получить массивы индексов выделенных на данный момент строк и столбцов |

Модели выделения и изученные нами методы и свойства таблицы JTable позволяют полностью управлять работой механизмов выделения ячеек таблицы и получать исчерпывающую информацию о выделенных элементах таблицы. Не забывайте только, что модели выделения строк и столбцов действуют *независимо* друг от друга, как бы это не выглядело на экране. К примеру, если в вашей таблице действует режим выделения строк, и вы щелкаете на некоторой ячейке, выделяется вся строка, которой принадлежит выделенная ячейка, то есть все ячейки в данной строке выглядят как выделенные. Однако выделенными будут считаться только та строка и тот столбец, которым принадлежит ячейка, на которой вы щелкнули. Работу с выделением лучше всего проводить посредством моделей, и только изредка прибегать к методам класса JTable. Это сделает код более элегантным, упростит его поддержку и развитие, а модели вы при случае сможете разделить между видами, во многом сократив объем требуемой работы.

## Сортировка и фильтрация

Таблица JTable позволяет не только отображать те данные, которые вы ей предоставляете посредством модели TableModel, но и обладает встроенными функциями их сортировки и фильтрации с минимальными усилиями с вашей стороны. Учитывая, что табличные данные очень часты в приложениях и, как правило, требуют сортировки (особенно если их много и их не охватить взглядом), наличие такой встроенной возможности, позволяющей не беспокоиться об этом в своей модели данных, приходится очень кстати.

Создатели Swing подошли к проблеме сортировки данных со всей возможной серьезностью. В пакет javax.swing был добавлен абстрактный класс RowSorter, способный проводить сортировку строк в любой модели, коль скоро она способна предоставить свои данные в виде строк и дает узнать общее количество этих строк. В принципе, данный класс можно было бы встроить во все компоненты библиотеки Swing, предоставляющие свои данные в виде строк, хотя бы в хорошо известные нам списки JList, однако на данный момент поддержка класса RowSorter имеется только в таблицах. Как и всегда в Swing, библиотека любезно предоставляет некую реализацию по умолчанию класса RowSorter под названием DefaultRowSorter. Однако, несмотря на название класса, он все еще абстрактный, так как все еще не знает точный тип модели, данные которой надо сортировать. Ну а конкретный класс для сортировки таблиц называется TableRowSorter.

Вникать в тонкие взаимоотношения классов для сортировки совсем не обязательно, если ваша задача состоит в простой сортировке таблицы. Пары методов класса JTable и знания некоторых деталей классов для сортировки, как правило, достаточно для подавляющего большинства приложений. Посмотрим, как добавить стандартную сортировку в таблицу:

```
// SimpleSorting.java
// Сортировка таблицы по умолчанию
import javax.swing.*;
import javax.swing.table.DefaultTableModel;
import java.awt.*;

public class SimpleSorting extends JFrame {
  public SimpleSorting() {
    super("SimpleSorting");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем таблицу на основе модели по умолчанию
    SortModel sm = new SortModel();
    sm.setColumnCount(4);
    // добавляем сто строк случайных чисел
    for ( int i = 0; i < 100; i++ ) {
      sm.addRow(new Integer[] { i,
          (int)(5*Math.random()),
          (int)(5*Math.random()),
          (int)(5*Math.random())} );
    }
    JTable table = new JTable();
    // автоматическое включение сортировки
    table.setAutoCreateRowSorter(true);
    // ограничение по количеству столбцов
    ((TableRowSorter)table.getRowSorter()).setMaxSortKeys(2);
    table.setModel(sm);
    add(new JScrollPane(table));
    // выводим окно на экран
    setSize(400, 300);
    setVisible(true);
  }
  // модель для демонстрации сортировки и фильтрации
  static class SortModel extends DefaultTableModel {
    // тип данных, хранимых в столбце
    public Class getColumnClass(int column) {
      return Integer.class;
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
```

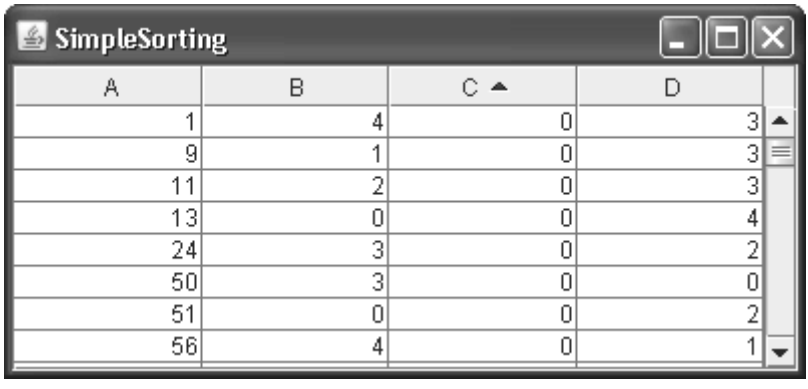
```
            public void run() { new SimpleSorting(); } });

    }
}
```

В примере мы создаем окно, в центре которого разместится таблица, отображающая специальную модель, которая как нельзя лучше подходит для сортировки. Модель, дабы не реализовывать хранение данных самостоятельно, унаследована от модели по умолчанию DefaultTableModel, и отличается лишь методом getColumnClass(), который говорит таблице, что в ячейках хранятся целые числа (по умолчанию все ячейки модели по умолчанию имеют тип Object). Правильный тип нам понадобится для более эффективной сортировки, так как именно по типу столбца выбирается способ его сортировки. Чуть позже мы обсудим это подробнее.

Модель заполняется ста рядами случайных чисел, причем случайные числа размещаются в трех столбцах, а первый столбец просто хранит порядковый номер строки таблицы. Далее нам остается лишь включить сортировку таблицы, доступную нам по умолчанию. Это делает метод setAutoCreateRowSorter(), передав в него значение true, мы просим таблицу автоматически создавать объект для сортировки RowSorter и присоединять его к таблице. Мы можем быть уверены в том, что это объект типа TableRowSorter, потому что именно этот класс таблица использует по умолчанию.

Класс TableRowSorter обладает несколькими интересными возможностями, и одну из них мы задействуем в примере. Изначально сортировка данных в модели таблицы ведется по *всем* столбцам (в сортировке это называется *ключами*). Это значит, что если значения в столбце равны, берется следующий столбец и по его значениям определяется, какая строка будет стоять выше, и так далее по всем столбцам, если необходимо. С помощью свойства maxSortKeys мы можем указать, какое количество столбцов будет участвовать в определении того, какая строка должна стоять выше. Иногда это важно, если таблица велика, и сортировка всех столбцов может сказаться на производительности. В нашем примере будут задействованы два столбца.

Сама сортировка по умолчанию включается через заголовок столбца, на котором пользователь должен щелкнуть мышкой, после чего там появится стрелка направления сортировки, и данные будут отсортированы. Конечно же, отсортировать данные можно и программно, получив объект RowSorter, и вызвав метод toggleSortOrder(), в качестве параметра передав ему номер столбца. Запустив пример и щелкая на заголовках столбцов, вы увидите, как они сортируются, и что сортировка ограничивается двумя столбцами, оставляя данные остальных столбцов в беспорядке.



Как мы видим, стандартная сортировка включается в мановение ока и работает безукоризненно, однако при любом сравнении данных прежде всего интересен вопрос, как именно эти данные сравнивают, тем более, что это происходит внутри объекта RowSorter. Как мы знаем, в Java, как правило, применяется отдельный интерфейс Comparable, в котором объект сравнивает себя с другими объектами, либо интерфейс Comparator, который уже сам сравнивает другие объекты, а не самого себя. Все это поддерживается классом RowSorter, и в следующей последовательности:

1. Вы можете установить для любого столбца таблицы объект Comparator, который будет заниматься сравнением значений из этого столбца, с помощью метода setComparator(). Это позволяет нам полностью контролировать процесс сортировки данных в столбце.

2. Если объект Comparator не установлен, анализируется тип столбца, который, как нам известно, задается классом (именно поэтому мы переопределили его в первом примере). Если класс столбца «умеет» сравнивать себя, то есть реализует интерфейс Comparable, именно он используется для сравнения значений.

3. Если первые два условия не соблюдены, сравнение выполняется на строках, полученных из ячеек. Вопрос лишь в том, как получить эти строки. Простейший способ — вызвать для всех ячеек метод toString(), но это может привести к излишним затратам памяти и потере производительности, если в объекте и без того есть строка, по которой нужно сравнивать, нет необходимости еще раз ее генерировать[1]. Поэтому класс TableRowSorter предоставляет возможность задать объект TableStringConverter, который преобразует объекты из ячейки в строки, используя известные ему данные о структуре данных. Ну а сами строки сортируются согласно текущим настройкам языка и культуры (locale) виртуальной машины JVM, с помощью класса Collator, что гарантирует нам безошибочную работу со всеми языками, какие бы их них не хранились в таблице.

Таким образом, если вы храните в таблице строки, то заботиться особенно не о чем, она прекрасно позаботиться о сортировке, учитывая при этом все особенности национальных языков. Если же в таблице хранятся некие объекты с данными, имеет смысл либо реализовать в них интерфейс Comparable, а если это невозможно или сложно, установить для столбца сортирующий объект Comparator, или, на крайний случай, преобразовывать данные в строки с помощью объекта TableStringConverter, так чтобы таблица сортировала уже строки.

Интересно, что сортирующий объект RowSorter, а точнее его реализация по умолчанию DefaultRowSorter позволяет нам фильтровать строки, то есть удалять их из таблицы и экрана, хотя в модели данных эти строки, конечно же, остаются. Сама таблица JTable при этом ничего не знает про фильтрацию и не имеет никаких посвященных ей управляющих методов. Управляет фильтрацией класс RowFilter из пакета javax.swing, в котором есть всего лишь один метод include(), который возвращает true, в том случае если строку данных следует показывать на экране. Чтобы упростить нам жизнь, в классе RowFilter есть набор заранее определенных фильтров для таблиц, весьма гибких и подходящих для многих ситуаций, например фильтры по числовым значениям или фильтры, проверяющее соответствие ячейки регулярному выражению. Эти предопределенные фильтры возвращаются статическими методами класса RowFilter.

Прежде чем фильтрацию строк таблицы на практике, необходимо вспомнить об одном очень важном вопросе. Мы недавно обсудили способы выделения строк таблицы и различ-

[1] Конечно, вы можете возвращать подходящую строку в методе toString(), что решит проблему. Однако это не всегда возможно, метод toString() всего один и может возвращать не совсем то, что вы хотите показывать в ячейке таблицы.

ные возможности получения выделенных в данный момент строк и столбцов. Что происходит с ними, если пользователь, или сама программа, сортирует данные и таким образом меняет оригинальный порядок данных из модели? Оказывается, когда мы узнаем о номерах выделенных в данный момент строк, эти номера соответствуют строкам *на экране*. Если на экране порядок строк другой, чем в модели, номера строк необходимо *преобразовывать*.

## ВНИМАНИЕ

> Если в таблице включена сортировка или фильтрация строк, вы не можете полагаться на то, что номера выделенных строк представляют собой строки из модели данных TableModel. Их необходимо преобразовывать методом таблицы JTable convertRowIndexToModel(), если вы хотите получить соответствующие этим номерам строки в модели данных.

Рассмотрим пример, в котором мы воспользуемся фильтрацией строк таблицы, и проверим, как преобразовывать номера выделенных строк и столбцов к их «оригинальным» номерам в модели данных:

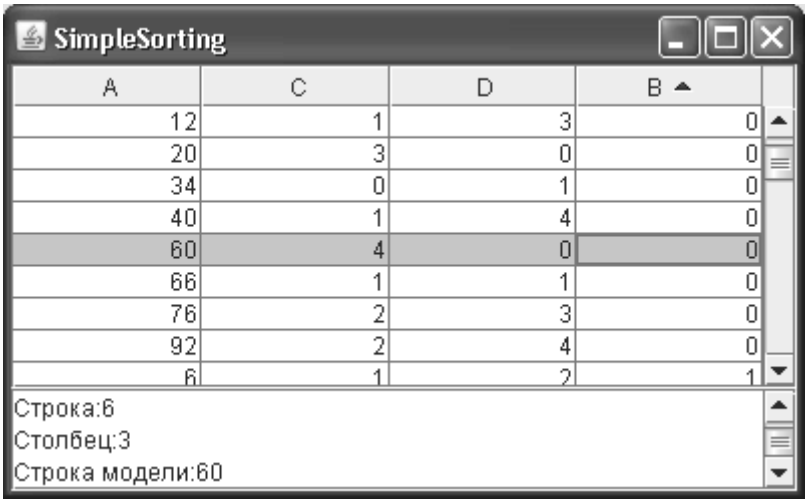
```
// FilterAndSelection.java
// Настройка фильтрации таблицы и выделенные строки
import javax.swing.*;
import javax.swing.event.ListSelectionListener;
import javax.swing.event.ListSelectionEvent;
import javax.swing.table.TableRowSorter;
import java.awt.*;

public class FilterAndSelection extends JFrame {
  public FilterAndSelection() {
    super("FilterAndSelection");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем таблицу на основе модели по умолчанию
    SimpleSorting.SortModel sm = new SimpleSorting.SortModel();
    sm.setColumnCount(4);
    // добавляем сто строк случайных чисел
    for ( int i = 0; i < 100; i++ ) {
      sm.addRow(new Integer[] { i,
          (int)(5*Math.random()),
          (int)(5*Math.random()),
          (int)(5*Math.random())} );
    }
    final JTable table = new JTable(sm);
    // автоматическое включение сортировки
    table.setAutoCreateRowSorter(true);
    // присоединим фильтрующий объект
    ((TableRowSorter)table.getRowSorter()).
```

```
        setRowFilter(new RowFilter() {
          public boolean include(Entry entry) {
            // включаем только четные строки
            return ((Integer)entry.getValue(0)) % 2 == 0;
          }
        });
    add(new JScrollPane(table));
    // поле для вывода номеров выбранных строк
    final JTextArea out = new JTextArea(3, 10);
    add(new JScrollPane(out), "South");
    // следим за выделением в таблице
    table.getSelectionModel().addListSelectionListener(
        new ListSelectionListener() {
          public void valueChanged(ListSelectionEvent e) {
            if ( table.getSelectedRow() != -1) {
             out.append("Строка:" + table.getSelectedRow() + "\n");
              out.append("Столбец:" + table.getSelectedColumn() +
"\n");
              out.append("Строка модели:" +
                  table.convertRowIndexToModel(
                      table.getSelectedRow()) + "\n");
            }
          }
        });
    // выводим окно на экран
    setSize(400, 300);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new FilterAndSelection(); } });
  }
}
```

В примере мы создадим таблицу на основе такой же модели, что и в предыдущем примере, и заполним ее случайными числами. Таблицу можно будет сортировать, а новинкой является фильтрующий объект RowFilter, который мы присоединяем к сортирующему объекту RowSorter, который мы получаем из таблицы (он там есть, так как мы попросили таблицу включить поддержку сортировки). В единственном методе include() необходимо определить, нужно ли показывать строку таблицы на экране, а объект Entry позволяет определить всю информацию о текущей строке, в том числе метод getValue() возвращает значение определенного столбца. Мы будем показывать только четные строки, определяя это по первому столбцу, в котором хранится номер строки.

Далее на юг окна добавляется текстовая область JTextArea с прокруткой, в которую мы будем записывать номера выделенных строк, когда они будут меняться. Для того чтобы узнавать об изменениях, нам необходимо добавить к модели выделения слушателя, что и мы и сделали. В слушателе в текстовую область выводятся номера выделенной строки и столбца «как есть», а также преобразованный номер строки на тот случай, если порядок строки на экране поменялся.

Запустив пример, вы сразу же убедитесь в том, что нечетные строки на экране не появляются, это заслуга нашего фильтрующего объекта. Затем, выполняя сортировку по одному из столбцов, и выделив любую строку, вы сможете увидеть, что номера строк, которые возвращает нам модель выделения, и номера строк в реальных данных отличаются. Это обязательное правило при работе с таблицами со включенной сортировкой или фильтрацией — иначе получение правильных данных на выделенной позиции будет невозможно.




Сортировка и фильтрация данных таблицы не слишком сложны, особенно если дело сводится к стандартным ситуациям, стоит лишь помнить о необходимости преобразовывать номера выделенных строк. Примеры с таблицами в этой книге никогда не преобразовывают номера строк для простоты понимания, и не включают сортировку, но это не значит, что об этом нюансе можно забыть.

Можно упомянуть еще то, что объекты таблицы для сортировки и фильтрации *типизированы*, то есть позволяют указать при создании конкретные типы модели и вспомогательных данных, с которыми им придется работать. В наших примерах мы обошлись без этого, но если логика этих объектов у вас сложна, и они часто обращаются к модели данных, указать конкретный тип модели будет весьма кстати.

## *Внешний вид ячеек таблицы*

Таблица JTable, как и любой другой сложный компонент Swing, такой как список JList или дерево JTree, дает возможность от начала и до конца управлять процессом отображения содержащихся в нем элементов. Каждая ячейка прорисовывается с помощью отдельного отображающего объекта, который должен реализовывать особый интерфейс TreeCellRenderer. Определив собственный отображающий объект или вмешавшись в работу стандартного объекта, вы получаете неограниченную власть над тем, как будут выглядеть ячейки вашей таблицы.

Мы привыкли, что обычно для сложного компонента задается единственный отображающий объект, который и отвечает за то, как выглядят составные элементы этого компонента. Однако таблица — самый мощный и самый сложный компонент библиотеки Swing, позволяющий выводить гигантские объемы разнородной информации, предоставляет нам более гибкие решения. На самом деле, представьте себе таблицу с десятком-другим столбцов и сотней строк — написание для нее специализированного отображающего объекта рискует превратиться в адский труд, особенно если каждая ячейка должна выглядеть по-своему. Поэтому таблица обладает более гибким поведением.

Таблица позволяет задавать два типа отображающих объектов для ячеек. *Отображающие объекты по умолчанию* (default renderers) задаются посредством специальных методов таблицы JTable и отвечают за прорисовку данных определенного типа, который, как и всегда в Java, задается объектом Class. Вы помните, что разбирая модель данных таблицы TableModel, мы обратили особое внимание на метод getColumnClass(), позволяющий указать, данные какого типа будет содержать тот или иной столбец. Тип столбца и определяет, какой отображающий объект будет прорисовывать ячейки этого столбца. Если вы помните, таблица JTable изначально поддерживает несколько типов данных, для которых предоставляет стандартные отображающие объекты — это простые строки, значки, булевы значения, а также числа и даты. Все поддерживаемые таблицей типы данных прорисовываются с помощью стандартного объекта DefaultTreeCellRenderer или одного из его подклассов. Вы с помощью предназначенных для этого методов класса JTable можете регистрировать свои отображающие объекты для новых типов данных или же заменять стандартные отображающие объекты собственными. Отображающие объекты ячеек могут также задаваться и для отдельных столбцов, какой бы тип не имела хранящаяся в них информация, иногда такой подход удобнее.

Все стандартные отображающие объекты унаследованы от надписи JLabel, так что вы без зазрения совести можете применять для записи ячеек таблицы все возможности HTML и даже загружать изображения. Впрочем, загрузка изображений посредством встроенного языка HTML не позволяет затем успешно распространять приложения в виде единого JAR-файла, поэтому давайте применим, как мы не раз это уже делали для других компонентов Swing, отображающий элемент, способный прорисовывать одновременно и заранее имеющийся значок Icon, и текст. Прежде всего нам понадобится специальный объект для хранения текста и значка. Мы не станем изобретать велосипед, а возьмем уже имеющийся у нас со времен работы со списками класс ImageListElement:

```
// com/porty/swing/ImageListElement.java
// Класс, хранящий значок и текст элемента
package com.porty.swing;

import javax.swing.*;

public class ImageListElement {
  // значок и текст
  private Icon icon;
  private String text;
 // удобный конструктор
 public ImageListElement(Icon icon, String text) {
     this.icon = icon;
     this.text = text;
 }
```

```
  // методы для доступа к значку и тексту
  public Icon getIcon() {  return icon; }
  public String getText() { return text; }
  public void setIcon(Icon icon) { this.icon = icon; }
  public void setText(String text) { this.text = text; }
}
```

В объекте ImageListElement будет храниться текст и значок ячейки. Для удобства имеется конструктор, позволяющий сразу же задать то и другое, и методы для смены данных. На очереди отображающий объект, который займется с прорисовкой текста и значка:

```
// com/porty/swing/ImageTableCellRenderer.java
// Объект для прорисовки значка и текста в таблице
package com.porty.swing;
import javax.swing.*;
import javax.swing.table.*;
import java.awt.*;

public class ImageTableCellRenderer
    extends DefaultTableCellRenderer {
  // метод возвращает компонент для прорисовки
  public Component getTableCellRendererComponent(
      JTable table, Object value, boolean isSelected,
      boolean hasFocus, int row, int column) {
    // получаем объект нужного типа
    if ( value instanceof ImageListElement ) {
      ImageListElement imageCell = (ImageListElement)value;
      // получаем настроенную надпись от базового класса
      JLabel label = (JLabel)super.
          getTableCellRendererComponent(table,
              imageCell.getText(), isSelected, hasFocus,
              row, column);
      // устанавливаем значок и подсказку
      label.setIcon(imageCell.getIcon());
      label.setToolTipText(imageCell.getText());
      return label;
    } else {
      return super.getTableCellRendererComponent(
          table, value, isSelected, hasFocus, row, column);
    }
  }
}
```

Наш новый отображающий объект унаследован от стандартного объекта DefaultTableCellRenderer (а тот, в свою очередь, — от надписи JLabel). Мы переопределяем

метод getTableCellRendererComponent(), возвращающий компонент, который и будет прорисовывать данные, хранящиеся в ячейках таблицы. Здесь, кстати, применимо правило, которое мы не раз уже обсуждали при изучении отображающих объектов: компоненты, возвращаемые этими объектами, не появляются на экране, они служат лишь «кистями», которые таблица использует при прорисовке определенных ячеек, поэтому компонент задействуется в единственном числе, а для каждого значения он заново настраивается. Это позволяет сэкономить уйму памяти и ресурсов системы графического интерфейса. Сначала данные преобразуются в тип ImageTableCell; во избежание ошибок преобразования типов мы сначала проверяем, что это подходящий нам тип. Смотрите, что происходит дальше: мы получаем от базового класса настроенную надпись (с подходящими цветами и шрифтами) с текстом, устанавливаем для нее значок, который также хранится в объекте ImageListElement, добавляем подсказку, и возвращаем таблице полученный результат. Именно отображающие объекты запрашиваются таблицей с целью получить подсказку для конкретной ячейки. Класс ImageListElement и новый отображающий объект разместились в пакете com.porty.swing, откуда их удобнее будет загружать в собственные программы.

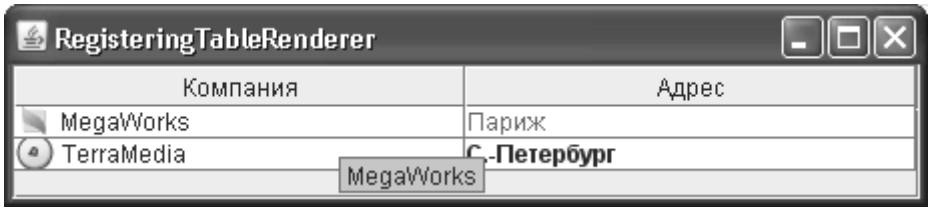
Теперь остается указать таблице, что в некотором столбце будут храниться данные нового типа и зарегистрировать для нового типа подходящий отображающий объект. Тип данных столбца позволяет указать модель данных TableModel, а регистрацию новых отображающих объектов проводит метод setDefaultRenderer() класса JTable. У нас получается следующая программа:

```
// RegisteringTableRenderer.java
// Регистрация в таблице собственного
// объекта для отображения
import javax.swing.*;
import javax.swing.table.*;
import com.porty.swing.*;
import java.awt.*;

public class RegisteringTableRenderer extends JFrame {
  public RegisteringTableRenderer() {
    super("RegisteringTableRenderer");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем таблицы на основе нашей модели
    JTable table = new JTable(new SpecialModel());
    // регистрируем объект для прорисовки новых данных
    table.setDefaultRenderer(ImageListElement.class,
        new ImageTableCellRenderer());
    // выводим окно на экран
    add(new JScrollPane(table));
    pack();
    setVisible(true);
  }
  // модель таблицы
  class SpecialModel extends AbstractTableModel {
    // значки
```

```
    private Icon
        image1 = new ImageIcon("clip1.gif"),
        image2 = new ImageIcon("clip2.gif");
    // названия столбцов
    private String[] columnNames = {
        "Компания", "Адрес"
    };
    // данные таблицы
    private Object[][] data = {
        { new ImageListElement(image1, "MegaWorks"),
            "<html><font color=\"red\">Париж" },
        { new ImageListElement(image2, "TerraMedia"),
            "<html><b>С.-Петербург" }
    };
    // количество столбцов
    public int getColumnCount() {
      return columnNames.length;
    }
    // названия столбцов
    public String getColumnName(int column) {
      return columnNames[column];
    }
    // количество строк
    public int getRowCount() {
      return data.length;
    }
    // тип данных столбца
    public Class getColumnClass(int column) {
      return data[0][column].getClass();
    }
    // значение в ячейке
    public Object getValueAt(int row, int column) {
      return data[row][column];
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new RegisteringTableRenderer(); } });
  }
}
```

В примере мы создаем окно с таблицей в его центре. Данные для таблицы будет поставлять специальная модель, унаследованная от класса AbstractTableModel, благодаря отдельной модели мы сможем указать, какие типы данных (представленные объектами Class) хранятся в каждом столбце. Собственно данные хранятся в двух массивах: в массиве columnNames мы храним названия столбцов, а в массиве data — данные ячеек. Заметьте, что в первом столбце данные хранятся в виде только что созданных нами объектов ImageListElement, позволяющих совместить изображение и текст. Методы модели TableModel нам хорошо знакомы, пристального внимания заслуживает лишь один из этих методов, а именно getColumnClass() — для первого столбца модели он возвращает тип (класс) ImageListElement, и данный тип нужно правильным образом отобразить.

В конструкторе окна создается таблица JTable, которой передается описанная во внутреннем классе модель. Чтобы объекты ImageListElement, хранимые первым столбцом таблицы, правильно отображались, необходимо зарегистрировать для них подходящий отображающий объект. Это и делает метод setDefaultRenderer(), сопоставляющий тип данных и объект, который обязуется эти данные отобразить. Запустив программу с примером, вы увидите наш новый отображающий объект в действии, а также еще раз убедитесь во всемогуществе встроенного в Swing языка HTML, и появлении у каждой ячейки своей подсказки. Стандартный отображающий объект унаследован от надписи JLabel, и все возможности последней остаются в силе.




Отображающие объекты по умолчанию, сопоставляемые с типом данных, очень удобны и позволяют быстро настроить внешний вид таблицы, причем все логично — для каждого типа данных, если он не вписывается в стандартные рамки и ему не хватает возможностей встроенного языка HTML, предоставляется собственный отображающий объект. Благодаря тому, что для необычных типов данных можно предоставить подходящий отображающий объект, таблица является универсальным компонентом, способным хранить и показывать любые данные, даже самые фантастичные.

Однако таблица JTable предоставляет еще одну возможность управления отображением данных, хранимых в ее ячейках. Иногда единое отображение всех данных одного типа (а тип сразу же задается для всего столбца посредством модели) оказывается недостаточно гибким. Вы можете захотеть отображать ячейки в одном и том же столбце поразному. В таких ситуациях таблица позволяет регистрировать разные отображающие объекты для одного столбца независимо от того, данные какого типа там хранятся. Если говорить точнее, отображающий объект регистрируется в объекте с информацией о столбце TableColumn, который нам уже известен.

Для отображения ячеек, как и прежде, используется объект, реализующий интерфейс TreeCellRenderer, только регистрируется он по-иному. Вы получаете нужный столбец из модели TableColumnModel (используя, например метод модели getColumn()) и задаете для его ячеек отображающий объект с помощью метода setCellRenderer(). Стоит отметить, что отображающий объект для вывода ячеек определенного столбца, имеет *приоритет* перед объектом по умолчанию, работающим в зависимости от типа данных, которые хранятся в столбце. Вы всегда сможете вернуться к объекту по умолчанию, передав в метод setCellRenderer() столбца TableColumn пустую (null) ссылку.

Задание отображающих объектов для столбцов позволяет проводить интересные эксперименты с процессом прорисовки таблицей содержимого своих столбцов. После изучения модели столбцов TableColumnModel нам известно, что одни и те же столбцы, описанные данными модели TableModel, могут быть несколько раз выведены на экран различными столбцами TableColumn. Задавая для таких столбцов разные отображающие объекты, вы сможете придать столбцам с одинаковыми данными уникальный внешний вид.

В остальном работа отображающих объектов одинакова независимо от того, используются по умолчанию или задаются для конкретных столбцов. Как работать с такими объектами в своей программе, мы только что выяснили. В том случае, если вам понадобится отображать каждую ячейку по-своему, в отображающем объекте придется возвращать различные компоненты в зависимости от выбранной строки таблицы.

## *Редактирование ячеек таблицы*

Таблица JTable разрешает пользователю изменять значения, хранящиеся в ее ячейках. Ячейки, значения которых можно менять, определяются с помощью модели данных TableModel: в ней есть метод isCellEditable(), говорящий, можно ли редактировать ячейку, расположенную в определенной строке и определенном столбце таблицы. Заведует редактированием ячеек таблицы не сама таблица и не ее UI-представитель, а отдельный объект, реализующий интерфейс TableCellEditor. Вы можете настроить стандартный объект для редактирования, поставляемый вместе с библиотекой Swing, или написать собственный объект и таким образом управлять процессом редактирования ячейки от начала до конца. Только не забывайте определять метод setValueAt() в модели данных TableModel: без него ни один редактор не сможет изменить значения в ячейках.

Объекты для редактирования в таблице JTable очень схожи с объектами для отображения — это также отдельные объекты, позволяющие легко наделить таблицу любыми возможностями редактирования. Так же как и отображающие объекты, объекты для редактирования разделяются на два типа: объекты по умолчанию и объекты, используемые для конкретного столбца таблицы (вторые имеют приоритет перед первыми). Объекты по умолчанию регистрируются посредством специальных методов класса JTable и предназначены для редактирования данных определенного типа (тип данных, хранимых в столбце, возвращает метод getColumnClass() модели TableModel). Изначально таблица поставляется с редакторами для строковых данных (строки редактируются так же, как все данные неизвестного типа, если для них не зарегистрировано специальных редакторов), булевых переменных (в качестве редактора используется флажок JCheckBox) и чисел.

Все редакторы, изначально встроенные в таблицу JTable, реализованы в стандартном объекте для редактирования DefaultCellEditor. Стандартный объект реализует интерфейс TableCellEditor и позволяет использовать для редактирования данных текстовое поле JTextField, флажок JCheckBox или раскрывающийся список JComboBox. Таким образом, стандартный объект довольно гибок и может помочь вам в редактировании ячейки без написания своего объекта для редактирования «с нуля»: текстовое поле позволит ввести любые строковые данные, а раскрывающийся список будет к месту, если вам понадобится организовать редактирование ячейки как выбор одного варианта из нескольких.

Интересно, что нам уже доводилось иметь дело со стандартным редактором ячеек DefaultCellEditor. В главе 15 мы использовали его для редактирования узлов дерева: класс DefaultCellEditor одновременно поддерживает интерфейсы TreeCellEditor (для деревьев) и TableCellEditor (для таблиц). Оба этих интерфейса унаследованы от базового интерфейса CellEditor, который описывает основные обязанности редактора ячеек таблицы или узлов дерева. Интерфейс TableCellEditor прибавляет к базовым методам CellEditor лишь один новый метод getTableCellEditorComponent(), который должен возвращать компонент,

применяемый для редактирования ячеек. Так что все, что мы узнали о стандартном редакторе DefaultCellEditor в главе 15, применимо и для таблиц.

А теперь применим некоторые возможности стандартного редактора в следующем примере:

```
// TableDefaultEditing.java
// Применение стандартного редактора для таблиц
import javax.swing.*;
import java.awt.*;

public class TableDefaultEditing extends JFrame {
  // название столбцов
  private String[] columns = {
      "Имя", "Любимый Цвет" };
  // данные для таблицы
  private String[][] data = {
      { "Иван", "Зеленый" },
      { "Александр", "Красный"},
      { "Петр", "Синий"}
  };
  public TableDefaultEditing() {
    super("TableDefaultEditing");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем таблицу
    JTable table = new JTable(data, columns);
    // настраиваем стандартный редактор
    JComboBox combo = new JComboBox(
        new String[] { "Зеленый", "Желтый", "Белый"});
    DefaultCellEditor editor =
        new DefaultCellEditor(combo);
    table.getColumnModel().getColumn(1).
        setCellEditor(editor);
    // выводим окно на экран
    add(new JScrollPane(table));
    setSize(400, 300);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new TableDefaultEditing(); } });
  }
}
```

Здесь в небольшое окно добавляется таблица JTable, созданная на основе двух массивов: одномерного с названиями столбцов и двухмерного с данными таблицы. Мы передаем эти массивы в конструктор таблицы, на их основе будет создана стандартная модель таблицы. Как мы знаем, в стандартной модели редактирование ячеек по умолчанию включено, так что беспокоиться об этом не нужно. Далее идет код настройки стандартного редактора ячеек таблицы. Как мы уже упомянули, стандартный редактор может выполнять редактирование на основе трех компонентов: текстового поля JTextField, флажка JCheckBox и раскрывающегося списка JComboBox. Для нашего примера мы выбираем раскрывающийся список JComboBox. Прежде чем передать его в конструктор стандартного редактора, необходимо наполнить список элементами, среди которых пользователь и будет делать выбор. Для простоты мы создаем список на основе массива из трех строк. Настроенный список передается стандартному редактору, после чего остается указать таблице, для каких ячеек этот редактор потребуется. В нашем примере мы указываем, что полученный редактор необходимо задействовать для второго столбца таблицы (редактор присоединяется к объекту TableColumn, который позволяет получить модель столбцов таблицы TableColumnModel).


Запустив программу с примером, вы увидите, как второй столбец таблицы редактируется с помощью настроенного нами раскрывающегося списка JComboBox. Процесс редактирования можно тонко настроить, применив все те знания о раскрывающихся списках, которые мы получили в главе 11. В нашем случае видно, что списку не хватает места в ячейке таблицы, мы можем попробовать изменить внешний вид списка или размер используемого в нем шрифта, или же применить хорошо известное нам свойство таблицы rowHeight, чтобы немного увеличить высоту ячеек. Первый столбец таблицы также редактируется, но для него по умолчанию применяется редактор, который задействует для редактирования обычное текстовое поле. Для несложных данных, вроде обычных строк или булевых значений, возможностей стандартного редактора может хватать, однако при появлении в таблице более экзотичных данных приходится задумываться о написании собственного редактора.

## Редактор дат

Когда мы обсуждали встроенные в таблицу JTable типы данных, то упомянули о том, что она способна отображать даты (представленные объектом Date), но вот встроенного редактора для этих дат нет, точнее, редактировать даты с помощью обычного текстового поля неудобно. Теперь, когда мы знаем обязанности редактора таблицы JTable, мы можем написать собственный редактор для дат, позволяющий комфортно редактировать их. В качестве компонента для редактирования мы применим счетчик JSpinner со специальной моделью — в наборе стандартных компонентов Swing это самый удобный способ менять даты.

Реализация «с нуля» интерфейса TableCellEditor — задача довольно трудоемкая, поэтому мы наследуем класс своего нового редактора от абстрактного класса AbstractCellEditor, который реализует большую часть методов базового интерфейса CellEditor. Точно также мы поступили и в главе 15, когда писали собственный редактор для узлов дерева JTree. Вообще, написание редакторов для таблиц и деревьев практически не отличается, и вы сможете легко перенести редакторы, созданные для одного компонента, в другой компонент. После наследования от класса AbstractCellEditor и реализации интерфейса TableCellEditor нам фактически останется определить два метода: метод getTableCellEditorComponent() должен будет возвращать компонент, который применяется при редактировании, а метод getCellEditorValue() сообщать, какое значение находится в данный момент в редакторе. Новый редактор мы поместим в пакет com.porty.swing для того, чтобы было легко применять его в ваших программах.

```
// com/porty/swing/DateCellEditor.java
// Редактор для ячеек таблицы, отображающих даты
package com.porty.swing;

import javax.swing.*;
import javax.swing.table.*;
import java.util.*;

public class DateCellEditor extends AbstractCellEditor
    implements TableCellEditor {
  // редактор — прокручивающийся список
  private JSpinner editor;
  // конструктор
  public DateCellEditor() {
    // настраиваем прокручивающийся список
    SpinnerDateModel model = new SpinnerDateModel(
        new Date(), null, null, Calendar.DAY_OF_MONTH);
    editor = new JSpinner(model);
  }
  // возвращает компонент, применяемый для редактирования
  public java.awt.Component getTableCellEditorComponent(
      JTable table, Object value, boolean isSelected,
      int row, int column) {
    // меняем значение и возвращаем список
    editor.setValue(value);
    return editor;
  }
  // возвращает текущее значение в редакторе
  public Object getCellEditorValue() {
    return editor.getValue();
  }
}
```

Новый редактор получился совсем простым. В конструкторе мы создаем и настраиваем счетчик JSpinner, который и будет выступать в роли редактора дат. Как вы помните из главы 12, для того чтобы счетчик JSpinner можно было применять в качестве редактора дат, ему необходимо передать специальную модель SpinnerDateModel. Конструктор этой модели требует четыре параметра. Первый — это текущее значение (это дата, которая будет изначально отображаться в поле счетчика), в качестве которого мы просто передаем новый объект Date. Ограничители дат определяют диапазон изменения даты — для простоты в качестве ограничителей мы передали пустые (null) ссылки, а это означает, что выбор дат будет неограничен. Последним параметром является поле класса Calendar, определяющее, по какой части даты будет осуществляться «прокрутка» — в редакторе это будет день месяца.

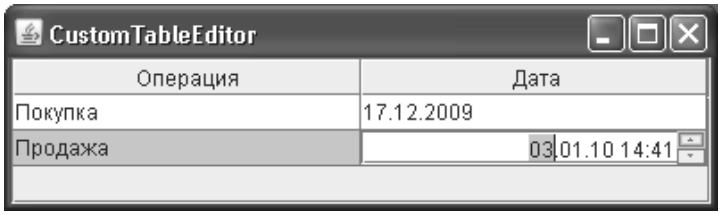
Далее следует пара методов, определяющих работу редактора ячеек таблицы. Метод getTableCellEditorComponent() должен возвращать компонент, который и будет редактором таблицы. Здесь все просто: мы, не мудрствуя лукаво, передаем в счетчик значение текущей ячейки (и подразумеваем, что наш редактор будет применяться лишь для дат, представленных объектами Date, в противном случае возникнет исключение) и возвращаем счетчик. Метод getCellEditorValue(), предназначенный для получения текущего значения в редакторе, еще проще: он просто возвращает текущее значение счетчика. Кстати, текущим значением также будет объект Date, но здесь не должно возникать разногласий: счетчик знает, как такие объекты редактировать, а таблица знает, как их отображать (ей нужно лишь сообщить, что некоторый столбец имеет именно такой тип данных). Все остальные методы, определенные в интерфейсе CellEditor, любезно реализованы для нас базовым классом AbstractCellEditor. Вы всегда сможете переопределить их, если вашему редактору понадобится дополнительная функциональность.

Остается проверить в работе новый редактор для ячеек таблицы. Заодно мы увидим, как в таблице JTable регистрируются редакторы по умолчанию — как вы помните, в отличие от редакторов для отдельных столбцов, редакторы по умолчанию регистрируются для типов данных и автоматически применяются для всех столбцов, имеющих указанный тип.

```
// CustomTableEditor.java
// Применение специализированного редактора
// для ячеек таблицы
import javax.swing.*;
import javax.swing.table.*;
import java.util.*;
import java.awt.*;
import com.porty.swing.*;

public class CustomTableEditor extends JFrame {
  // заголовок столбцов таблицы
  private String[] columns = {
      "Операция", "Дата" };
  // данные таблицы
  private Object[][] data = {
      { "Покупка", new Date() },
      { "Продажа", new Date() }
  };
  public CustomTableEditor() {
```

```
    super("CustomTableEditor");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем таблицу на основе стандартной модели
    DefaultTableModel model =
        new DefaultTableModel(data, columns) {
          // необходимо указать тип столбца
          public Class getColumnClass(int column) {
            return data[0][column].getClass();
          }
        };
    JTable table = new JTable(model);
    table.setRowHeight(20);
    // указываем редактор для дат
    table.setDefaultEditor(
        Date.class, new DateCellEditor());
    // выводим окно на экран
    getContentPane().add(new JScrollPane(table));
    setSize(400, 300);
    setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new CustomTableEditor(); } });
  }
}
```

Здесь мы создаем таблицу JTable на основе стандартной модели, эта таблица затем разместится в панели прокрутки, которая, в свою очередь, разместится в центре панели содержимого окна. Данные для таблицы, как это уже не раз бывало, находятся в двух массивах: одномерный массив columns содержит названия столбцов, а двухмерный data — непосредственно данные ячеек. Обратите внимание, что второй столбец содержит даты, представленные объектами Date. Оба массива мы передаем в специальный конструктор стандартной модели DefaultTableModel и здесь же переопределяем метод модели getColumnClass(). Это необходимо для того, чтобы указать, что второй столбец нашей таблицы хранит объекты типа Date. Если мы этого не сделаем, то наш новый редактор не будет задействован для их редактирования. Переопределять метод isCellEditable() нам не потребовалось: стандартная модель по умолчанию считает, что редактировать можно все ячейки.



Далее мы регистрируем в таблице созданный нами новый редактор по умолчанию в качестве редактора для столбцов с данными типа Date. Видно, что для этого предназначен метод setDefaultEditor(). В дополнение ко всему высота ячеек таблицы немного увеличивается — это позволит счетчику JSpinner чувствовать себя в них более комфортно, поэтому им будет удобнее пользоваться. Запустив программу с примером, вы увидите, что первый столбец таблицы редактируется посредством обычного текстового поля, а вот для редактирования второго применяется наш новый редактор на основе счетчика JSpinner. Без всякого сомнения, это во много раз повышает удобство работы с датами в таблицах. Аналогичным образом вы сможете написать редакторы и для остальных нестандартных типов данных, редактировать которые средствами стандартных редакторов по тем или иным причинам неудобно.

## *Заголовок таблицы JTableHeader*

Заголовок таблицы, реализованный в классе JTableHeader из пакета javax.swing.table, отображает названия столбцов, а также позволяет перетаскивать столбцы, меняя их местами, и менять их размеры. Мы уже знаем, что таблица JTable автоматически добавляет заголовок таблицы в качестве верхнего заголовка панели прокрутки, если вы добавляете таблицу в панель прокрутки. Если таблица будет размещена в любом другом контейнере, места для заголовка таблицы не найдется, поэтому подыскивать его придется самостоятельно.

Заголовок таблицы представляет собой самый обыкновенный компонент со своим UI-представителем, отвечающим за его внешний вид. Данные для работы заголовок таблицы черпает из модели столбцов таблицы TableColumnModel, которую необходимо передавать в конструктор класса JTableHeader или присоединять к заголовку позже. Фактически, заголовок таблицы выступает как дополнительный вид модели столбцов (наряду с таблицей JTable) — он позволяет перетаскивать столбцы и менять их размеры, меняя данные в модели. Никаких данных в самом заголовке не хранится — все находится в модели TableColumnModel (рассматривая модели столбцов TableColumnModel и объектов TableColumn мы уже выяснили, что именно в них хранится исчерпывающая информация о столбцах, в том числе и об их размерах и порядке следования на экране).

Со стандартным заголовком таблицы можно сделать немногое, но все предоставляемые нам возможности очень полезны и весьма часто востребованы. Для начала, вы можете включить или отключить перетаскивание столбцов и смену их размеров. Далее, заголовок таблицы обладает собственным отображающим объектом, который и отвечает за то, как выглядит название столбца и его оформление. По умолчанию отображающий объект для заголовка таблицы предоставляет UI-представитель класса JTableHeader, но это легко изменить, придав тем самым заголовку вашей таблицы уникальный узнаваемый вид (тем более, что мы хорошо знакомы с процессом написания отображающих объектов). Рассмотрим небольшой пример, в котором присоединим к заголовку таблицы собственный отображающий объект, а заодно настроим несколько дополнительных свойств заголовка:

```
// UsingTableHeader.java
// Настройка заголовка таблицы JTableHeader
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.border.*;
import java.awt.*;

public class UsingTableHeader extends JFrame {
  // данные для таблицы
```

```
private String[][] data = {
    { "Июнь", "+18 C" },
    { "Июль", "+22 C" },
    { "Август", "+19 C" }
};
// названия столбцов
private String[] columnNames = {
    "Месяц", "Средняя температура"
};
public UsingTableHeader() {
  super("UsingTableHeader");
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  // создаем таблицу
  JTable table = new JTable(data, columnNames);
  // настраиваем заголовок таблицы
  JTableHeader header = table.getTableHeader();
  header.setReorderingAllowed(false);
  header.setResizingAllowed(false);
  // присоединяем отображающий объект
  header.setDefaultRenderer(new HeaderRenderer());
  // добавляем таблицы в панель прокрутки
  add(new JScrollPane(table));
  setSize(400, 300);
  setVisible(true);
}
// специальный отображающий объект для заголовка
class HeaderRenderer
    extends DefaultTableCellRenderer {
  // метод возвращает компонент для прорисовки
  public Component getTableCellRendererComponent(
      JTable table, Object value, boolean isSelected,
      boolean hasFocus, int row, int column) {
    // получаем настроенную надпись от базового класса
    JLabel label = (JLabel)super.
        getTableCellRendererComponent(table,
            value, isSelected, hasFocus,
            row, column);
    // настраиваем особую рамку и цвет фона
    label.setBackground(Color.gray);
    label.setBorder(border);
    return label;
```

```
    }
    private Border border = BorderFactory.
        createMatteBorder(16, 16, 16, 16,
            new ImageIcon("bullet.png"));
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new UsingTableHeader(); } });
  }
}
```

В примере мы создаем небольшое окно, в центре которого разместится панель прокрутки с расположенной в ней таблицей JTable. Таблицу мы создаем на основе двух массивов: двумерный массив data хранит данные ячеек, а одномерный columnNames — названия столбцов. После создания таблицы мы получаем ее заголовок (для этого предназначен метод getTableHeader()) и принимаемся за его настройку. Для начала меняются два наиболее полезных свойства заголовка: свойство reorderingAllowed контролирует, будет ли доступна возможность перетаскивания столбцов, а свойство resizingAllowed позволяет включить (или, как в нашем примере, отключить) возможность смены размеров столбцов. Напоследок мы устанавливаем для заголовка свой отображающий объект (присоединить его позволяет метод setDefaultRenderer()). Собственно отображающий объект реализован во внутреннем классе HeaderRenderer и не представляет собой ничего хитрого: он унаследован от стандартного отображающего объекта DefaultTableCellRenderer, на который и перекладывается большая часть работы. К полученной в результате работы стандартного объекта надписи (а вы помните, что стандартный отображающий объект представляет собой именно надпись JLabel) прибавляется необычная рамка, составленная на основе небольшого значка.


После запуска программы с примером вы оцените настройку заголовка таблицы: помимо его необычного и красочного внешнего вида вы не сможете поменять столбцы местами или изменить их размеры. Только не забудьте, что стандартный заголовок таблицы применяется для операций сортировки и прорисовывает направление сортировки. Если в таблице включена сортировка и вам необходимо использовать нестандартный заголовок, придется позаботиться о прорисовке значка сортировки самостоятельно, узнавая об изменениях в сортировке с помощью слушателя объекта RowSorter.

Очень часто заголовок таблицы JTableHeader применяется для добавления к таблице операций для конкретного столбца. Для этого к заголовку таблицы присоединяется слушатель событий от мыши, следящий за щелчками на заголовках столбцов. После щелчка

легко определить, на каком именно столбце он произошел (вызвав метод columnAtPoint() из класса JTableHeader), получить данные этого столбца (в этом поможет модель данных TableModel), как-либо применить их и дать сигнал таблице о том, что данные изменились и вид пора обновить, если это необходимо.

## *Резюме*

Разнообразная двухмерная информация, особенно полученная из баз данных, их отображений на объекты средствами, подобными iBatis и Hibernate, или из компонентов Enterprise JavaBeans, — частый гость в приложениях. Благодаря таблице JTable, ее без преувеличения фантастической гибкости и безграничным возможностям настройки, вывод этой информации не станет для вас проблемой. Как мы убедились, читая материал этой главы, с таблицей JTable можно делать практически все. Чем больше знакомишься с компонентом JTable, тем более поражаешься глубинам его возможностей и убеждаешься в том, что таблица JTable иллюстрирует саму суть Swing, позволяя с элегантной простотой и мощью выводить на экран любые данные.

# Глава 18. Круговорот данных

Эта глава будет посвящена самому драгоценному сокровищу пользователей, которое они с трепетом доверяют нашему приложению, — данным, результатам их, без сомнения, трудной и кропотливой работы. Способность приложения не только помогать в обработке данных, но и обеспечивать их обмен с другими приложениями и системой, а также отказ от случайных изменений многократно повышает удобство программы и привязанность к ней пользователей. Все эти возможности библиотека Swing любезно предоставляет сама или с помощью AWT, а нам остается только правильно их использовать.

## *Обмен данными*

Самое неприятное, что вы можете сделать в своем приложении, это заставить пользователя выполнять одну и ту же работу снова и снова, не предоставляя в этом никакой помощи. Что может быть тоскливее и неприятнее, чем набирать вновь один и тот же текст и нажимать те же кнопки. Поэтому обмен данными настолько популярен, особенно в форме копирования и вставки, — пользователь получает возможность быстро переместить данные туда, где они ему требуются, и вам для этого не нужно продумывать каждый его шаг и вставлять повсюду собственные кнопки и меню для копирования и импорта похожих данных. Достаточно предоставить универсальный механизм перемещения данных в приложении.

Благодаря моделям, данные в Swing отделены от визуального представления, и их копирование и особенно совместное использование очень просты. Однако было бы слишком наивно полагать, что пользователи применяют только ваше приложение, и им не захочется скопировать текст из внешней электронной почты в Java-приложение. Необходимо обеспечить взаимодействие со всеми возможностями базовой операционной системы.

На данный момент все операционные системы позволяют своим пользователям перемещать данные между своими приложениями. В основном применяются универсальные операции копирования данных (copy), вставки их в приложение (paste), или «вырезание» (cut) данных, попросту говоря, перемещение. По первым буквам этих трех слов операции эти еще называют CCP. Данные, полученные в результате таких операций, хранятся в *буфере обмена* (clipboard), где они доступны любым приложениям.

Существует и более приятный глазу и интуитивно понятный процесс перетаскивания (drag), когда объект, видимый на экране, перетаскивается в другое место экрана и волшебным образом оказывается в новом месте , когда объект этот там бросают (drop). Эти операции еще называют DnD (drag and drop). Суть операций CCP и DnD одинакова, различается лишь метод. Когда пользователь перетаскивает данные, они не попадают в буфер обмена и доступны только непосредственно в момент перетаскивания.

Все операции обмена данными: и CCP, и DnD, доступны нашим Java-приложениям. Начнем с того, что посмотрим, как получить доступ к системному буферу обмена.

### Буфер обмена Clipboard

Итак, как мы уже упомянули, операции по обмену данных используют общее системное хранилище, так называемый буфер обмена, который позволяет приложениям обмениваться данными друг с другом. Для Java-приложений буфер обмена представ-

лен классом с очевидным названием Clipboard, класс этот находится в пакете java.awt.datatransfer, который, как легко судить по названию, хранит все инструменты для работы с обменом данных.

Напрямую буфер обмена Clipboard обычно не создается (хотя это возможно), да в этом и нет смысла, как правило, нас интересует системный буфер обмена данными. Чтобы получить к нему доступ, используется фабрика классов AWT Toolkit, метод которой getSystemClipboard() возвращает системный буфер. С его помощью можно получать данные из буфера или передавать их туда. Однако данные у каждого приложения свои, и необходимо предоставить способ их распознавания. Для этого служат типы данных для обмена.

## Типы данных для обмена

Современные системы позволяют хранить в буфере обмена все что угодно, будь то простой текст или изображения огромных размеров. Прежде чем что-то получить из буфера, особенно если это что-то получено из другого приложения, нам необходимо понять, как именно представлены данные и каким образом их можно обработать. Существует много разных способов описать данные, но есть и стандарт под названием MIME (MultiPurpose Internet Mail Extenstions), возникший после того, как к сообщениям электронной почты стало возможным присоединять данные разных типов. Для того чтобы как-то определить, что за данные присоединены к почте и как с ними обращаться, была введена классификация типов MIME. Грубо говоря, тип описывается как совокупность базового типа (это может быть текст, видео или что-то еще) и его конкретного типа. В итоге описание типа выглядит как строка вида «text/html». Именно такое описание типов используют классы для обмена данными в Java.

Нам не придется углубляться в детали спецификации MIME, потому что основную часть работы выполняет вспомогательный класс DataFlavor из все того же пакета java.awt.datatransfer. Он сопоставляет описание типа MIME и Java-класс, который может представить этот тип. Данный класс позволяет создавать и свои уникальные типы данных, которые вы можете описать в формате MIME, но, как правило, хватает и стандартных типов. Они описаны как поля-константы класса DataFlavor и представляют собой конкретные объекты этого класса:

**Таблица 18.1.** Основные типы данных для обмена

| Стандартный тип | Предназначение |
|---|---|
| DataFlavor.stringFlavor | Основа из основ, тип данных для строки языка Java String, которая, как известно, представляет собой набор символов Unicode. Как правило, большая часть данных при обмене может быть представлена в виде строки |
| DataFlavor. javaJVMLocal-ObjectMimeType | Тип данных для любого объекта в вашем Java-приложении, но именно в вашем приложении, между разными виртуальными машинами объекты этого типа не передать. Этот тип данных хорош для обмена данными между компонентами внутри Java-приложения, однако данные такого типа не сможет понять ни одно системное или любое другое приложение, так что в дополнение к ним можно представить, к примеру, строковое представление объекта |
| DataFlavor.imageFlavor | Описывает всевозможные изображения, данные при этом представлены как класс изображений AWT Image. |

**Таблица 18.1** *(продолжение)*

| Стандартный тип | Предназначение |
|---|---|
| DataFlavor.javaFileListFlavor | Представляет собой список файлов, выбранных в файловой системе. Такие данные, к примеру, могут быть переданы в буфер обмена различными файловыми приложениями, когда пользователь копирует файл в буфер или «вырезает» его из текущей директории. Данные представлены как список List, в котором находятся объекты, представляющие файлы (или директории) File. |

В абсолютном большинстве случаев вполне достаточно стандартных типов данных, более того, из других приложений вы можете получить только описанные данные описанных выше типов. В своем же приложении вы вольны заполнять буфер локальными объектами Java и работать с ними как обычно, мы уже видели, что для этого есть свой тип данных. Никто не запрещает данным иметь и два, и даже больше типов. Пример этого мы вскоре рассмотрим.

Теперь мы, наконец, можем начать работу с буфером обмена. Для того чтобы выяснить, что же хранится в буфере, класс Clipboard предоставляет метод getAvailableDataFlavors(), который возвращает массив всех типов для данных, находящихся в буфере в текущий момент. Обнаружив в типах то, что мы умеем обрабатывать или показывать, мы можем получить данные. Посмотрим, как это происходит:

```
// ClipboardContents.java
// Доступ к содержимому буфера обмена
import java.awt.*;
import java.awt.datatransfer.DataFlavor;
import java.awt.datatransfer.UnsupportedFlavorException;
import java.awt.datatransfer.Clipboard;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.io.IOException;
import javax.swing.*;

public class ClipboardContents extends JFrame {
  public ClipboardContents() {
    super("ClipboardContents");
    // выход при закрытии окна
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // добавим кнопку получения содержимого
    JButton getButton = new JButton("Получить");
    // текстовое поле для вывода данных
    final JTextArea textArea = new JTextArea();
    getButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        try {
          // буфер обмена
```

```
            Clipboard clipboard = Toolkit.getDefaultToolkit().
                getSystemClipboard();
            // выведем все типы данных
            for (DataFlavor next:
                clipboard.getAvailableDataFlavors()) {
              textArea.append(next.toString() + "\n");
              // попытка получить сами данные
              textArea.append(clipboard.
                  getData(next).toString() + "\n");
            }
          } catch (UnsupportedFlavorException ex) {
            ex.printStackTrace();
          } catch (IOException ex) {
            ex.printStackTrace();
          }
        }
      });
      add(new JScrollPane(textArea));
      add(getButton, "South");
      // выведем окно на экран
      setSize(400, 300);
      setVisible(true);
    }
    public static void main(String[] args) {
      SwingUtilities.invokeLater(
          new Runnable() {
            public void run() { new ClipboardContents(); } });
    }
}
```

Как и всегда, пример максимально прост, так что мы сможем сконцентрироваться на том, что происходит с данными внутри буфера обмена. Мы создаем окно, в центре которого располагается текстовая область в панели прокрутки, так что она сможет показать нам текст любого размера. Внизу окна мы добавили кнопку, при нажатии которой происходит анализ содержимого системного буфера обмена.

При нажатии кнопки мы получаем ссылку на системный буфер обмена, а также все доступные на текущий момент типы данных, которые находятся в буфере. Чтобы получить их, вызываем уже известный нам метод getAvailableDataFlavors(). Для каждого поддерживаемого данными типа мы выводим строковое представление типа на экран, а также с помощью метода getData() пытаемся получить объект, которые представляет эти данные в Java-приложении.Мы не рассчитываем на конкретный тип данных, поэтому не можем преобразовать полученный объект во что-то более ценное, и просто выводим в текстовую область его строковое представление. Обратите внимание, что нам приходится учитывать исключения, которые могут возникнуть при невозможности получить данные запрошенного типа (в нашем примере это невозможно, так как мы

получаем данные типа, который точно поддерживается ими) или ошибке при считывании этих данных (IOException). Запустив приложение, скопируйте что-нибудь в буфер обмена системы, например файлы или просто текст, и нажмите кнопку, чтобы изучить содержимое и типы объектов, которыми оно представлено.


На указанном снимке экрана приложения видно, как сначала в буфер попал список файлов, описанный как прекрасно всем нам знакомый список элементов List. Список выводится на экран в виде распечатки всех своих элементов, объектов File с их полными путями в файловой системе.

Далее ситуация становится еще более интересной. В буфер был скопирован текст из внешнего редактора («привет из блокнота!»), казалось бы, он будет представлен только как объект String, но не тут-то было. Если для нас в Java-приложении действительно хватает строки String, то для системных приложений этого мало, и более того, они могут воспринимать строки по-разному. Здесь система обмена данными выполняет за нас всю нудную работу — она представляет строку в виде множества комбинаций, в различных кодировках, в виде набора байтов, в виде потока данных, и много другого, вы все сможете увидеть, запустив приложение. Ну а в Java-приложении остается лишь получить строку String, в универсальном формате Unicode, и оценить свои преимущества — в таких ситуациях универсальность Java окупает себя как никогда.

Итак, задача получения данных, хоть и влечет за собой анализ типа этих данных и преобразование к соответствующему объекту, который представляет данные, все же несложна, и встроить в приложение получение из системного буфера строк, изображений или файлов можно всего несколькими строками кода. Однако это лишь одна сторона медали, приложение должно также позволять пользователям копировать свои данные и переносить их в другие приложения. Сделать это просто, требуется один лишь метод setContents() класса Clipboard, но вот сами данные при этом придется подготовить для выхода «на люди». Посмотрим, как это делается.

## Адаптация данных

Платформа Java создана работать одинаково на всех операционных системах и аппаратных комбинациях, однако при обмене данными с приложениями «родной» системы встает вопрос понимания друг друга. В предыдущем разделе мы узнали, как нам понять, что же лежит в системном буфере обмена и как эти данные перевести в понятные нам объекты Java. С другой стороны, очевидно, что наше Java-приложение имеет свои форматы данных, даже простых строк, и их необходимо представить так, чтобы

их правильно поняли и в других приложениях, будь они системными или другими Java-приложениями.

Задачей адаптации данных Java-приложения для обмена ими с системой занимается интерфейс Transferable из хорошо известного нам пакета java.awt.datatransfer. Именно реализацию этого интерфейса необходимо передать в метод setContents() класса Clipboard, чтобы разместить свои данные в системном буфере. По сути, интерфейс просто связывает данные Java-приложения, представленные некоторым объектом, с описанием типа данных DataFlavor, так что реализовать его самостоятельно совсем несложно[1].

**Таблица 18.2.** Методы интерфейса Transferable

| Метод | Предназначение |
|---|---|
| getTransferDataFlavors() | Возвращает массив всех типов, поддерживаемых данными, которые описываются интерфейсом. Подразумевается, что самым первым элементов массива идет самый удобный тип, последним — соответственно самый мало подходящий тип данных |
| isDataFlavorSupported() | Позволяет определить, поддерживается ли данными переданный в качестве параметра тип |
| getTransferData() | Основной метод, возвращает объект, который будет представлять данные для указанного типа. К примеру, для типа-изображения необходимо будет вернуть изображение Image |

Видно, что для адаптации данных необходимо всего лишь определиться с тем, какие типы будут поддерживаться и для каждого их них возвращать соответствующий объект. Объекты, сопоставленные типам класса DataFlavor, мы обсуждали в предыдущем разделе, анализируя содержимое буфера обмена.

Как всегда, не обязательно для каждого случая с нуля писать свою реализацию интерфейса Transferable, есть и реализация по умолчанию, которая называется StringSelection. В конструктор этого класса можно передать строку String, и затем передать полученный объект в буфер обмена. Это прекрасный и быстрый способ вставить в буфер обмера строку с текстом, однако вы редко будете использовать этот класс напрямую, так как все текстовые компоненты Swing и без того обладают встроенной поддержкой операций копирования и вставки текста. Гораздо чаще встречается ситуация когда вам нужно хранить в буфере обмена не только текстовое представление данных, но и другое представление, к примеру, Java-объект или изображение. В этом случае реализовать интерфейс Transferable придется самостоятельно. Сделать это несложно, попробуем вставить в буфер обмена данные нескольких типов:

```
// PastingClipboard.java
// Вставка данных в системный буфер обмена
import javax.swing.*;
import java.awt.*;
import java.awt.image.BufferedImage;
import java.awt.datatransfer.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

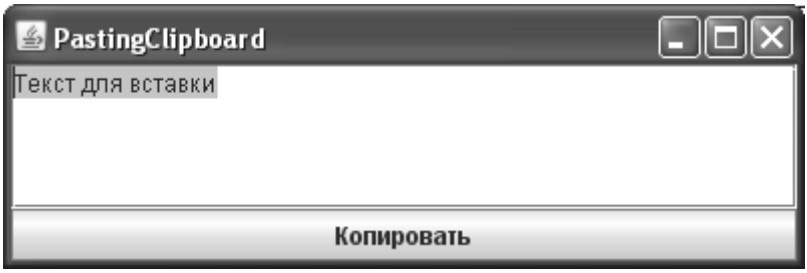
[1] Мы недаром используем в этом разделе термин «адаптация», ведь интерфейс Transferable по сути и является представителем известного шаблона проектирования «адаптер»

```
import java.io.IOException;
import java.util.Arrays;

public class PastingClipboard extends JFrame {
  private JTextArea textArea;
  public PastingClipboard() {
    super("PastingClipboard");
    // выход при закрытии окна
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // добавим кнопку для вставки данных
    JButton copyButton = new JButton("Копировать");
    // текстовое поле для получения текста
    textArea = new JTextArea();
    copyButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        // буфер обмена
        Clipboard clipboard = Toolkit.getDefaultToolkit().
            getSystemClipboard();
        // вставим в буфер наши адаптированные данные
        clipboard.setContents(new ComplexTransferable(), null);
      }
    });
    add(new JScrollPane(textArea));
    add(copyButton, "South");
    // выведем окно на экран
    setSize(400, 300);
    setVisible(true);
  }
  // объект-адаптер для наших данных
  class ComplexTransferable implements Transferable {
    // список поддерживаемых типов данных
    private DataFlavor[] supportedTypes = new DataFlavor[]
        { DataFlavor.stringFlavor, DataFlavor.imageFlavor };
    public DataFlavor[] getTransferDataFlavors() {
      return supportedTypes;
    }
    public boolean isDataFlavorSupported(DataFlavor flavor) {
      return Arrays.asList(supportedTypes).contains(flavor);
    }
    public Object getTransferData(DataFlavor flavor)
        throws UnsupportedFlavorException, IOException {
```

```
        if ( flavor == DataFlavor.imageFlavor ) {
          // если запрошено изображение, вернем красный квадрат
          BufferedImage bi = new
              BufferedImage(100, 100, BufferedImage.TYPE_INT_RGB);
          Graphics g = bi.getGraphics();
          g.setColor(Color.RED);
          g.fillRect(0, 0, 100, 100);
          return bi;
        } else if ( flavor == DataFlavor.stringFlavor ) {
          // если запрошена строка, вернем выделенный текст
          return textArea.getSelectedText();
        }
        // исключение в случае неизвестного типа данных
        throw new UnsupportedFlavorException(flavor);
      }
    }
    public static void main(String[] args) {
      SwingUtilities.invokeLater(
          new Runnable() {
            public void run() { new PastingClipboard(); } });
    }
}
```

В примере мы создадим окно JFrame и разместим в его центре многострочное текстовое поле JTextArea, текст, который в нем набирается, потом можно будет скопировать в буфер обмена. На юге окна разместится кнопка, по нажатию которой в буфере обмена будут размещаться наши данные. При нажатии кнопки мы получаем системный буфер обмена уже знакомым нам методом класса Toolkit, и вызываем его метод setContents(), который размещает переданный ему объект Transferable в буфере. Вторым параметром этого метода можно передать слушатель событий, который будет оповещаться в случае утраты «владения» буфером обмена, то есть если другое приложение разместит в нем свои данные и перезапишет ваши. Эта возможность, хоть и может быть полезной, редко востребуется, и вместо слушателя мы передаем в метод пустую ссылку null.

Самое интересное скрыто в классе ComplexTransferable, который адаптирует несколько видов данных для размещения в системном буфере. Класс реализует интерфейс Transferable и все три его метода, которые мы чуть ранее обсудили. Их очень несложно реализовать . Мы поддерживаем два типа данных — строки и изображения, оба эти типа стандартны и описаны как константы класса DataFlavor. В методе getTransferData() нам остается вернуть строку или изображение, в зависимости от переданного типа данных. Строку мы берем из текстового поля, это выделенный в данный момент текст, а вот изображение строим в памяти динамически с помощью класса BufferedImage. Для этого требуется задать размер и тип изображения в конструкторе (тип изображения у нас стандартный RGB), а затем можно получить для изображения объект для рисования Graphics и нарисовать на нем все что угодно. Мы для простоты просто закрашиваем все красным цветом. После этого изображение передается для хранения в буфере.




Запустив приложение, вы сможете набрать в текстовом поле любой текст, выделить его и нажатием на кнопку вставить в буфер обмена не только текст, но и дополнительное изображение. Для проверки того, что все работает, останется найти приложение с поддержкой вставки изображений из буфера обмена, к примеру, любой графический редактора, и убедиться в том, что в буфере обмена находится красный квадрат. Одновременно с этим текстовый редактор сможет получить из буфера строку с нашим текстом.

Как правило, работа со сложными данными и буфером обмена в большом приложении сводится примерно к такому решению, что мы описали в примере. Для обмена с остальными приложениями системы вы размещаете в буфере строки, списки файлов или изображения, а для удобства своей работы заодно храните в буфере ссылку на объект приложения внутреннего типа, так что вам не нужно будет дополнительно обрабатывать его при вставке в другом месте своего приложения.

## Буфер обмена и стандартные компоненты Swing

Несмотря на то, что мы разобрались с буфером обмена и подготовкой для него данных и можем все сделать своими руками, многое для нас уже сделано по умолчанию в стандартных компонентах Swing. Прежде всего, это относится ко всем текстовым компонентам, унаследованным от базового класса JTextComponent. В нем определены методы cut(), copy() и paste(), которые простым вызовом позволяют манипулировать текстом в буфере обмена, а также зарегистрированы стандартные клавиатурные сокращения (Control-C и так далее), которые предоставят пользователям привычные операции с помощью клавиатуры. К сожалению, контекстное меню для текстовых компонентов по умолчанию не предусмотрено, и его придется создавать самостоятельно, впрочем, сделать это просто — слушатели пунктов меню будут просто вызывать упомянутые чуть выше методы.

Дело не ограничивается текстовыми компонентами, компоненты посложнее, вроде списков JList, деревьев JTree или таблиц JTable также обладают некоторым уровнем поддержки буфера обмена. В них тоже зарегистрированы стандартные клавиатурные сокращения, нажав которые, пользователь может получить текстовое представление выбранных в данный момент данных (для таблицы это будет даже текстовое представление всех столбцов выбранной строки). Хорошо, что по умолчанию есть и это, но как правило, это всего лишь приятная мелочь. Если сложные компоненты призваны обмениваться данными друг с другом или другими приложениями, обычно приходится писать собственную реализацию этого процесса, но, как мы поняли из изучения обмена данных, сделать это несложно.

## Операции перетаскивания

Операции визуального перетаскивания и вставки данных (drag and drop, DnD) выглядят для пользователей чрезвычайно заманчиво, потому что они очень естественны — даже пользуясь мышью, вы, по сути, перетаскиваете ее с места на место. Увидеть такой же процесс на экране приятно и очень понятно — вы берете нечто важное для вас, та-

щите это в другое место, и оно там оказывается. Конечно, остаются вопросы эффективности подобного интерфейса — перетаскивание обычно удобно с экранными объектами большого размера, например схемами и изображениями, а вот с более мелкими элементами, вроде слов в тексте или очень точными размерами, удобнее пользоваться клавиатурой или меню. Однако для начинающих пользователей, в качестве более удобного интерфейса, перетаскивание довольно удобно.

Реализовать перетаскивание не совсем просто. Мало следить за всевозможными комбинациями движения мыши и клавиатуры, надо еще узнавать, готов ли компонент под курсором мыши принять данные, и понять, в каком месте компонента они окажутся. Поэтому, как правило, операционные системы предоставляют способы управления перетаскиванием более высокого уровня. В AWT перетаскиванию данных посвящен немаленький пакет java.awt.dnd, и пользоваться им возможно, но требуются усилия. К счастью, библиотека Swing предоставляет инструменты более высокого уровня, и чаще всего перетаскивание встроить в приложение довольно просто.

Важнейшую роль играет объект TransferHandler, присоединенный к любому компоненту Swing, унаследованному от класса JComponent. Объект этот хранится в свойстве transferHandler, и вы можете получить его или задать новый, собственный. Задачей этого объекта является предоставление данных в готовом виде из компонента для обмена данными (то есть в виде объекта Transferable, который нам уже знаком по разделу о буфере обмена) и обработка различных ситуаций при перетаскивании и вставке данных в компонент, который, как правило, пользуется моделью и не знает, как добавить в нее новые данные. Все эти спорные моменты улаживает объект TransferHandler, либо стандартными путями, либо с вашей помощью, когда вы переопределяете его методы.

Перетаскивание и вставка данных встроена не во все компоненты, а лишь в те, в которых это, по мнению создателей Swing, имеет смысл. Прежде всего, это текстовые компоненты, унаследованные от класса JTextComponent, там и вставка текста перетаскиванием не только поддерживается, но и включена по умолчанию, в чем вы легко можете убедиться, запустив пример с текстовыми компонентами Swing и перетаскивая в них текст. Не включено по умолчанию перетаскивание данных в списках JList, деревьях JTree и таблицах JTable, но оно там есть и настраивается очень легко. Достаточно установить свойство этих классов dragEnabled в true, как сразу же вы сможете перетаскивать данные с этих компонентов.

Насколько просто включить перетаскивание для упомянутых выше трех компонентов, настолько же требует усилий процесс подготовки данных для обмена данных с другими компонентами или даже приложениями и приема данных в своем компоненте и вставки их в нужное место. Все это обеспечивается методами класса TransferHandler, которые нам необходимо переопределять. К сожалению, данные в каждом случае индивидуальны, и способа отправить их в «путешествие» простым вызовом метода или двух нет. Впрочем, не все так сложно. Давайте составим таблицу с методами класса TransferHandler, которые нам необходимо определить, чтобы данные компонента были доступны не только ему.

**Таблица 18.3.** Методы класса TransferHandler, необходимые для описания данных и их перетаскивания

| Метод | Предназначение |
|---|---|
| createTransferable() | В этом методе необходимо создать и вернуть прекрасно нам знакомый объект-адаптер Transferable, который описывает все поддерживаемые типы данных и возвращает их по запросу извне. К примеру, для списка имеет смысл вернуть строку или объект, который выделен в списке в данный момент. Этот метод вызывается когда пользователь начинает «тащить» что-то с вашего компонента |

**Таблица 18.3** *(продолжение)*

| Метод | Предназначение |
|---|---|
| getSourceActions () | Здесь нужно вернуть те действия, которые поддерживает компонент при перетаскивании. По умолчанию никаких действий не поддерживается. Действия определены как константы в класса TransferHandler, чаще всего это COPY (копирование перетаскиванием), MOVE (перемещение данных), или оба этих действия. Если метод не переопределить, включение перетаскивания ни к чему не приведет |
| canImport () | Если компонент сможет принимать данные, этот метод необходимо переопределить и возвращать true, когда данные подходящие. Этот метод вызывается когда пользователь пытается что-то «бросить» на компонент. Как правило, вы анализируете тип данных и решаете, подходит ли он вам. Доступна также текущая позиция курсора. |
| importData() | Если предыдущий метод «одобрил» данные, они передаются в этот метод, если пользователь решил вставить их в ваш компонент. Здесь данные необходимо каким-то образом обработать, вставить в модель или куда-то еще, и желательно показать пользователю на экране, что данные добавлены. Доступна также текущая позиция курсора, так что данные можно вставлять очень точно. |

Мы уже выполняли основные задачи для обмена данных, когда работали с буфером обмена, новым является лишь использование класса TransferHandler. А теперь перейдем к делу и начнем с простых списков JList, между которыми можно будет перетаскивать строки, из которых они состоят, и вставлять любые другие строки, перетаскивая их из других приложений.

```
// ListDrag.java
// Перетаскивание и вставка данных и TransferHandler
import javax.swing.*;
import java.awt.*;
import java.awt.datatransfer.DataFlavor;
import java.awt.datatransfer.StringSelection;
import java.awt.datatransfer.Transferable;

public class ListDrag extends JFrame {
  // начальные данные для списков
  private String[] listData =
      new String[] { "Раз", "Два", "Три" };
  public ListDrag() {
    super("ListDrag");
    // выход при закрытии окна
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем модели
    DefaultListModel model1 = new DefaultListModel(),
        model2 = new DefaultListModel();
    for (String element: listData) {
```

```
      model1.addElement(element);
      model2.addElement(element);
    }
    // создаем списки
    JList list1 = new JList(model1);
    list1.setTransferHandler(new ListTransferHandler(list1));
    list1.setDragEnabled(true);
    JList list2 = new JList(model2);
    list2.setTransferHandler(new ListTransferHandler(list2));
    list2.setDragEnabled(true);
    // добавим списки на экран
    setLayout(new GridLayout(1, 2));
    add(new JScrollPane(list1));
    add(new JScrollPane(list2));
    // выведем окно на экран
    setSize(400, 300);
    setVisible(true);
  }
  // объект-адаптер для списков и их данных
  class ListTransferHandler extends TransferHandler {
    private JList list;
    public ListTransferHandler(JList list) {
      this.list = list;
    }
    @Override
    public int getSourceActions(JComponent c) {
      // данные могут копироваться или перемещаться
      return TransferHandler.COPY_OR_MOVE;
    }
    @Override
    public boolean canImport(TransferSupport support) {
      // поддерживается импорт только строковых данных
      return support.isDataFlavorSupported(
          DataFlavor.stringFlavor);
    }
    @Override
    protected Transferable createTransferable(JComponent c) {
      // стандартный способ адаптировать строку для обмена
      return new StringSelection(
          list.getSelectedValue().toString());
    }
```

```
    @Override
    public boolean importData(TransferSupport support) {
      if ( support.isDataFlavorSupported(DataFlavor.stringFlavor) ) {
        try {
          // добавляем в модель новую строку на выбранную позицию
          ((DefaultListModel)list.getModel()).
              add(list.locationToIndex(
                  support.getDropLocation().getDropPoint()),
                  // получение строки из Transferable
                  support.getTransferable().
                      getTransferData(DataFlavor.stringFlavor));
          return true;
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
      return false;
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new ListDrag(); } });
  }
}
```

Итак, приступим. В примере создаются два списка JList, которые будут добавлены в окно, разделенное на две одинаковые части с помощью табличного расположения. Для того чтобы мы смогли что-то добавлять в списки, нам необходима модель, которая допускает добавление элементов, здесь как нельзя лучше пригодится стандартная модель списков DefaultListModel. Мы создаем по модели для каждого списка и добавляем в них для начала по три простых строки из массива данных. Затем модели передаются в конструкторы списков.

Создание списков и моделей очень просто и нам прекрасно известно, теперь нужно заставить их уметь отдавать и принимать данные при перетаскивании. Само перетаскивание включается для списков свойством dragEnabled. Этого вполне достаточно. Далее вся работа переносится на объект TransferHandler, который будет отдавать данные при перетаскивании и принимать их при вставке[2]. В примере мы написали универсальный объект ListTransferHandler, годный для любого списка, работающего со стандартной моделью. Он требует список, данные которого требуется «адаптировать», в качестве параметра своего конструктора. Вспоминая таблицу 18.3 с описанием основных методов, требуемых для обмена данными, реализация нашего класса становится довольно очевидной. Сначала в методе getSourceActions() необходимо указать, какие действия поддерживает

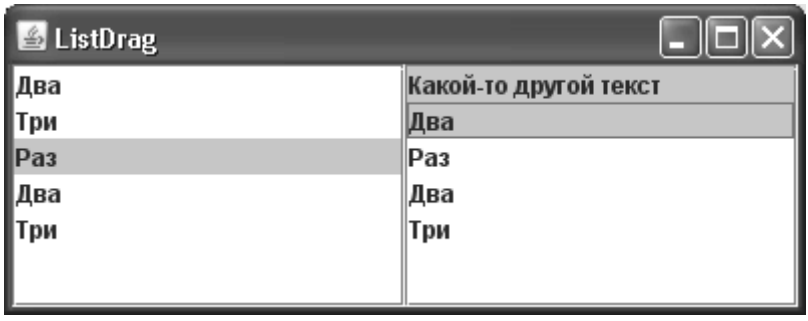
[2] Как нетрудно видеть, объект TransferHandler представляет собой пример классического шаблона «стратегия», когда сторонний объект определяет, как следует обмениваться данными и их описывать, и объект этот легко поменять.

компонент при перетаскивании. У нас это копирование и перемещение данных, хотя, строго говоря, мы лишь копируем данные из списка, а перемещение добавлено только для удобства (копирование требует нажатия клавиши Control при перетаскивании).

Метод createTransferable() создает данные для обмена в виде объекта Transferable. Нам этот процесс уже хорошо знаком, так как именно в таком виде мы вставляли данные в буфер обмена. Так как в списках у нас данные в виде строк, мы вполне можем ограничиться стандартным адаптером для строк StringSelection, ну а как реализовать более сложные случаи, мы уже знаем.

Самое сложное — это вставка данных в список. Прежде всего мы указываем в методе canImport(), что будем поддерживать лишь строковые данные. Вся необходимая информация о данных, которые пользователь пытается «бросить» на компонент, в том числе и их тип DataFlavor, доступна через параметр метода TransferSupport. Ну а метод importData() уже непосредственно отвечает за вставку данных в компонент. На всякий случай мы проверяем, представляют ли собой данные строку, получаем модель списка, и, подразумевая, что это стандартная модель списка DefaultListModel, добавляем в него новый элемент. Посмотрите, как вычисляется позиция для вставки — метод getDropPoint() позволяет узнать точку, в которой пользователь «бросил» данные, а метод списка locationToIndex() преобразует точку в своих координатах в позицию списка.

Запустив пример, вы сможете вдоволь насладиться «бросанием» текста из списка в список, из списка в самого себя (да, формально это не отличается от перетаскивания из других компонентов), или же вообще из других приложений. Самое замечательное в поддержке перетаскивания то, что мы, по сути, им не занимаемся, а сконцентрированы на задаче преобразования данных. Кстати, написанный нами класс ListTransferHandler весьма полезен для передачи строковых данных списка и может многократно использоваться и в других ваших программах, если ваш список основан на стандартной модели.



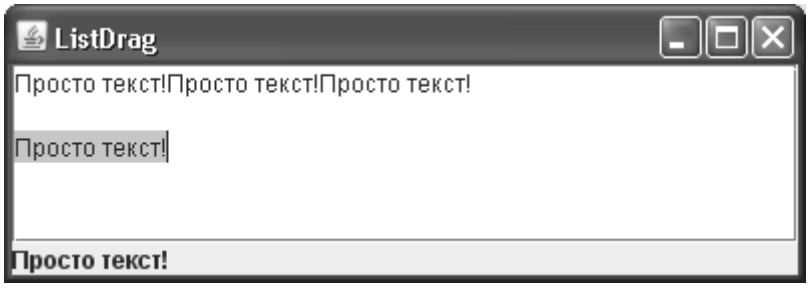

Аналогичным образом работает перетаскивание для деревьев JTree и таблиц JTable — достаточно включить свойство dragEnabled, и описать, как будет организован круговорот данных из компонента и в него. У этих трех компонентов (включая списки), есть и еще одно свойство, которое позволяет указать, каким образом будет прорисовываться место «бросания» данных. По умолчанию в нашем примере используется режим выделения данных особым цветом (элементы списка под курсором выделяются), что не всегда верно, в этом случае скорее стоило показать, что вставка данных будет происходить перед указанным элементом или после него. Изменить графическое представление вставки позволяет свойство dropMode, в которое надо передать одно из значений класса-перечисления DropMode. По умолчанию оно равно USE_SELECTION, а в случае с нашим списком скорее стоило использовать INSERT, что лучше отображает вставку на показанную позицию. Остальные возможности класса DropMode вы можете узнать в интерактивной документации JDK.

А как же остальные компоненты Swing? Можно ли перетаскивать данные с панелей или кнопок? Можно, однако в этом случае придется самостоятельно определять момент начала перетаскивания. Все так же необходимо задать для компонента объект преобразования данных TransferHandler, а затем, в подходящий момент, вызвать его метод exportAsDrag(), который начнет системный процесс перетаскивания данных. Как правило, вы просто присоединяете к компоненту слушатель событий от мыши, следите за перетаскиванием мыши на нем, и при событии перетаскивания вызываете данный метод. Здесь как нельзя лучше пригодится дополнительный конструктор класса TransferHandler, который принимает название свойства компонента для обмена. Это свойство должно иметь методы get и set (если подразумевается вставка данных). К сожалению, такой объект TransferHandler не в состоянии обмениваться данными с остальной операционной системой, даже если свойство представляет собой строку, потому что данные представлены как внутренний объект JVM, однако для простых визуальных операций это может пригодиться. Рассмотрим на простом примере:

```
// LabelManualDrag.java
// Перетаскивание и вставка данных "вручную"
import javax.swing.*;
import java.awt.event.MouseMotionAdapter;
import java.awt.event.MouseEvent;

public class LabelManualDrag extends JFrame {
  public LabelManualDrag() {
    super("ListDrag");
    // выход при закрытии окна
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    final JLabel label = new JLabel("Просто текст!");
    // стандартный способ обмена свойством "text"
    label.setTransferHandler(new TransferHandler("text"));
    // момент начала перетаскивания
    label.addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        TransferHandler handler = label.getTransferHandler();
        // вызов системного перетаскивания
        handler.exportAsDrag(label, e, TransferHandler.COPY);
      }
    });
    // добавим текстовое поле и надпись на экран
    add(new JScrollPane(new JTextArea()));
    add(label, "South");
    // выведем окно на экран
    setSize(400, 300);
    setVisible(true);
  }
```

```
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new LabelManualDrag(); } });
  }
}
```

В примере мы создаем надпись, объект TransferHandler которой будет передавать при перетаскивании значение свойства text, то есть собственно текст надписи. В центр окна мы добавляем текстовую область JTextArea. У нее тоже есть свойство text, так что мы сможем перетаскивать значение свойства с надписи на текстовое поле. Главное же — определить момент перетаскивания. Мы присоединяем к надписи слушатель движений мыши и при любом перетаскивании мыши над надписью указываем объекту TransferHandler начать системную процедуру передачи данных. Запустив пример, вы с легкостью перетащите текст надписи в текстовое поле, однако не сможете «утащить» его за пределы своего приложения. Для сложных деловых данных больших приложений такое поведение вряд ли подходит, необходима интеграция со всеми приложениями системы, однако для приложения с визуальными редакторами, например с перетаскиванием текста на компоненты, представляющие собой элементы некой схемы, может быть очень кстати.




Обсуждение операций перетаскивания в Swing на этом можно закончить. Мы ни разу не упомянули классы из пакета java.awt.dnd, и это лишний раз показывает, что система перетаскивания реализована в Swing на высоком уровне и не требует от нас лишних усилий, и с любыми ситуациями, возникающими на практике, справляется с блеском. На практике почти все задачи , как правило , сводятся к адаптации данных к обмену.

## TransferHandler и буфер обмена

Нетрудно видеть, что при перетаскивании и приеме данных мы фактически делаем все то же самое, что и при работе с буфером обмена: адаптируем данные для обмена и отдаем их «наружу» или же принимаем данные и думаем, куда их «пристроить». Так и просится встроить в класс TransferHandler поддержку буфера обмена, и что неудивительно, эта поддержка там есть (в Swing наши желания, как правило, сбываются). Если в случае с перетаскиванием пользователь визуально «тащит» или «бросает» данные с компонента, то в случае с буфером обмена он обычно использует какую-то кнопку или пункт меню для копирования в буфер или вставки его содержимого.

Класс TransferHandler предоставляет нам все три действия (CCP, cut-copy-paste) в виде команд Action, и мы можем их присоединить к своим компонентами или меню. Когда вызывается команда копирования (getCopyAction()), она вызывает метод createTransferable() и полученные данные вставляет в буфер обмена, команда же вставки из буфера (getPasteAction()) вызывает знакомый нам метод importData(). Таким образом, благодаря

объекту TransferHandler разница между перетаскиванием данных и получением их из буфера обмена практически неотличима. Есть лишь одно «но»: для всех команд из объекта необходимо правильно задавать источник события ActionEvent. Это должен быть компонент, данные которого нужно импортировать или экспортировать.

Вспоминая наш достаточно универсальный объект TransferHandler для списков JList на основе стандартной модели со строковыми элементами, мы можем быстро дополнить список операциями копирования и вставки в буфер обмена, благо адаптация данных уже готова. Давайте попробуем:

```
// TransferHandlerClipboard.java
// Копирование в буфер с помощью TransferHandler
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class TransferHandlerClipboard extends JFrame {
  public TransferHandlerClipboard() {
    super("TransferHandlerClipboard");
    // выход при закрытии окна
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // создаем модель
    DefaultListModel model = new DefaultListModel();
    model.addElement("Раз");
    model.addElement("Два");
    // список со строками
    final JList list = new JList(model);
    list.setTransferHandler(new ListDrag.
ListTransferHandler(list));
    list.setDragEnabled(true);
    list.setDropMode(DropMode.INSERT);
    add(new JScrollPane(list));
    JButton copy = new JButton("Копировать в буфер");
    copy.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        // необходимо для работы команды
        e.setSource(list);
        // вызываем стандартную команду копирования
        TransferHandler.getCopyAction().actionPerformed(e);
      }
    });
    add(copy, "South");
    // выведем окно на экран
    setSize(400, 300);
    setVisible(true);
```

```
    }

    public static void main(String[] args) {
      SwingUtilities.invokeLater(
          new Runnable() {
            public void run() { new TransferHandlerClipboard(); }
});
    }
}
```

В примере нам уже все хорошо знакомо — список на основе стандартной модели в центре окна, а на юге в этот раз кнопка, копирующая выделенный элемент в буфер обмена. Вручную ничего делать не пришлось — мы использовали уже имеющийся у нас объект ListTransferHandler из другого примера, и вызвали стандартное действие из класса TransferHandler по копированию данных в буфер. Запустив пример, вы сможете как перетаскивать данные из списка, так и копировать их в буфер обмена. Обратите внимание, что перед вызовом стандартного действия необходимо задать правильный источник события — компонент с данными, в примере это наш список. Есть лишь одна маленькая неприятность: вставка данных работать не будет, так как в классе ListTransferHandler мы зависим от позиции на экране, где данные «бросаются» на компонент, а при вставке из буфера ее не существует и в методе getDropLocation() возникнет исключение. Для вставки из буфера надо доработать метод importData() и методом isDrop() проверять, перетаскиваются ли данные визуально, и только в этом случае вставлять их на определенную позицию, в противном случае методом getDropLocation() получать позицию нельзя. Пусть доработка примера и класса ListTransferHandler станет для вас простым упражнением.

Как мы видим, правильно написанный объект TransferHandler верно служит и при перетаскивании данных, и при работе с буфером обмена. Его дополнительные возможности избавляют от низкоуровневых подробностей, и в большинстве ситуаций вы включаете в свое приложение поддержку обмена данных именно с помощью собственных объектов TransferHandler. Пользователи обязательно оценят ваши усилия.

## *Отмена и повтор операций*

В процессе работы с разнообразными данными людям свойственно ошибаться, и невозможность исправить ошибку может привести к необходимости все делать заново, что, конечно же, является недопустимой тратой времени. Именно поэтому в приложениях появился механизм отмены операций (undo) — приложение записывает историю всех действий пользователя и в любой момент может отменить их, одно за другим. Пользователи знают, что ошибка не страшна, и комфортно работают со своими данными. Если отмена тоже была вызвана по ошибке, ее саму можно отменить — то есть операцию повторить (redo).

Любое приличное приложение, в котором пользователи тратят хоть какие-то усилия по созданию и изменению данных, должно позволять им избавляться от случайных ошибок, сохраняя историю их изменений и позволяя отказываться от них. Поддержке отказа от изменений посвящен отдельный пакет библиотеки Swing javax.swing.undo, однако этот пакет довольно абстрактный и большую часть работы приходится делать своими руками.

Основой пакета служит класс UndoManager, который, по сути, представляет собой список изменений, описанных интерфейсом UndoableEdit. Изменение в данных, описанное как UndoableEdit, должно знать, как отменить или повторить себя, возможна ли в данный момент отмена или повтор, и предоставлять пользователям удобное описание изме-

нения. По сути, объект UndoableEdit — не что иное как «*команда*» (command), известный шаблон проектирования, объединяющий всю информацию об изменении и возможности его отмены или повтора. Добавить новое изменение к списку UndoManager можно методом addEdit(), а отменить последнее изменение в списке методом undo(). Таким образом нам предоставляется просто удобный список операций, а что это за операция и как она должна отменяться, мы решаем самостоятельно.

Чуть проще обстоят дела с текстовыми компонентами, которые автоматически генерируют команды для отмены и повтора операций. Мы уже кратко обсуждали эту возможность в главе 16, где обсуждали такие компоненты. Интересно, что класс UndoManager может работать как «*наблюдатель*» (observer), наблюдая за объектами, которые генерируют изменения в виде события UndoableEditEvent. Это позволяет отделиться от конкретного объекта UndoManager, именно так с ним работают текстовые компоненты, к которым его нужно присоединить в виде слушателя событий. В собственных компонентах мы вольны выбирать любой способ: добавлять команды для отмены напрямую методом addEdit() либо оповещать неизвестный заранее объект UndoManager обо всех изменениях с помощью событий UndoableEditEvent.

Очевидно, что система отмены изменений полностью зависит от того, как вы будете описывать эти изменения. Как правило, лучше всего забыть о графической составляющей компонентов Swing и описывать изменения в терминах данных, а значит, как изменения моделей компонентов. Этим путем достигается лучшая гибкость при смене компонентов или способе работы с ними, так как изменение данных остается тем же самым. Привязка же к визуальному изменению данных на экране делает систему отмены весьма ограниченной и мало приспособленной к изменениям.

Описывать изменения в терминах моделей весьма просто. Например, добавление элемента в список элементов можно описать как объект UndoableEdit, внутри него хранить список с данными и добавленный элемент и при вызове отмены (undo()) просто удалять его из списка. При вызове же повтора операции (redo()) элемент вновь должен попасть в список. Какой из компонентов отображает список, и как, нас при этом не волнует. Останется лишь обновить экран, чтобы компоненты заново получили данные из моделей и отобразили новое состояние дел. Таким образом, отмена и повтор операций в идеале производятся только над данными приложения и с графическими компонентами никак не связаны.

А теперь рассмотрим пример, в котором мы будем добавлять элементы в список JList и включим поддержку отмены и повтора этих изменений, основываясь на стандартных средствах Swing:

```
// UndoListAdd.java
// Отмена операций в списках и UndoManager
import javax.swing.*;
import javax.swing.undo.*;
import java.awt.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class UndoListAdd extends JFrame {
  // объект для отмены операций
  private UndoManager undoManager = new UndoManager();
  public UndoListAdd() {
    super("ListDrag");
```

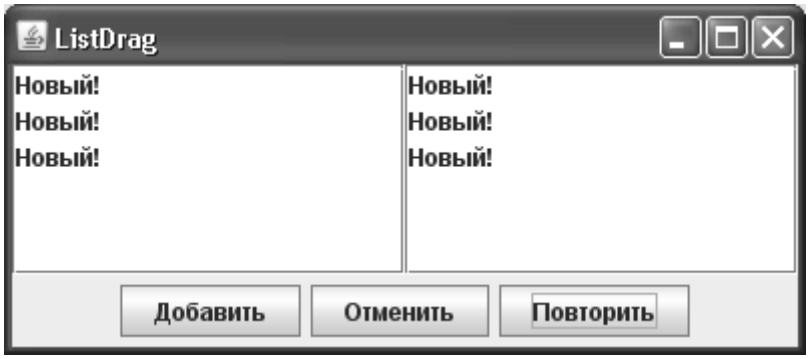
```
// выход при закрытии окна
setDefaultCloseOperation(EXIT_ON_CLOSE);
// создаем модель списка
final DefaultListModel model = new DefaultListModel();
// создаем списки
JList list1 = new JList(model);
JList list2 = new JList(model);
// добавим списки в окно
JPanel listPanel = new JPanel(new GridLayout(1, 2));
listPanel.add(new JScrollPane(list1));
listPanel.add(new JScrollPane(list2));
add(listPanel);
// кнопка добавление элемента
JButton add = new JButton("Добавить");
add.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    String newElement = "Новый!";
    model.addElement(newElement);
    // регистрация новой операции для отмены
    undoManager.addEdit(new
        ListAddUndoableEdit(model, newElement));
  }
});
// кнопки отмены и повтора
final JButton undo = new JButton("Отменить");
final JButton redo = new JButton("Повторить");
// обработчик нажатий кнопок
ActionListener al = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    // отмена и повтор при возможности
    if ( e.getSource() == undo
        && undoManager.canUndo() ) {
      undoManager.undo();
    } else if ( undoManager.canRedo() ) {
      undoManager.redo();
    }
  }
};
undo.addActionListener(al);
redo.addActionListener(al);
// добавим кнопки на юг окна
```

```
    JPanel buttons = new JPanel();
    buttons.add(add);
    buttons.add(undo);
    buttons.add(redo);
    add(buttons, "South");
    // выведем окно на экран
    setSize(400, 300);
    setVisible(true);
  }
  // класс, описывающий добавление в список
  class ListAddUndoableEdit extends AbstractUndoableEdit {
    // модель и новый элемент
    private DefaultListModel model;
    private Object element;
    public ListAddUndoableEdit(DefaultListModel model, Object
element) {
      this.model = model;
      this.element = element;
    }
    @Override
    public void undo() throws CannotUndoException {
      super.undo();
      model.removeElement(element);
    }
    @Override
    public void redo() throws CannotRedoException {
      super.redo();
      model.addElement(element);
    }
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
        new Runnable() {
          public void run() { new UndoListAdd(); } });
  }
}
```

В примере в центре окна разместятся два списка одинакового размера. У них будет одна стандартная модель DefaultListModel, поначалу пустая. На юге окна у нас разместится панель с последовательным расположением, с тремя кнопками. Первая кнопка будет добавлять к модели новый элемент, вторая отменять последнюю операцию, а третья повторять отмененную операцию. Для того чтобы класс UndoManager смог работать с нашей операцией, необходимо описать ее с помощью интерфейса UndoableEdit. Он довольно громоздкий, поэтому проще воспользоваться абстрактным классом AbstractUndoableEdit,

в котором уже многое реализовано, что мы и делаем. В конструктор операции следует передать стандартную модель списка и новый элемент. В общем же требуется лишь описать методы undo() и redo(), в которых мы удаляем новый элемент или заново добавляем его. Все довольно очевидно, есть лишь одна деталь — необходимо вызвать метод базового класса, так как он должен знать, когда мы отменили или повторили операцию, чтобы затем класс UndoManager смог узнать, доступна ли в данный момент отмена или повтор.

Когда изменение данных описано в требуемом виде, остается регистрировать его в объекте UndoManager в момент изменения, что мы и делаем методом addEdit(). Оставшееся является делом техники — кнопки отмены и повтора операций просто вызывают методы undo() и redo() для объекта UndoManager, предварительно удостоверившись в том, что это возможно, методами canUndo() и canRedo(). Запустив пример, вы убедитесь в том, что отмена и повтор операций работает на обоих списках, и при этом стандартная модель сама оповещает их об изменениях, а непосредственно изменения работают только с данными, ничего не зная о том, какие графические компоненты и где их отображают.




Это лишь начальный пример, и отталкиваясь от данных и моделей, мы сможем легко встроить в приложение отмену и повтор любых операций, причем на гибкости приложения и легкости изменения его графического интерфейса это никак не должно отразиться. Отделение данных от их отображения и деталей прорисовки на экране всегда платит сторицей в виде надежных и практичных приложений.

## *Резюме*

Хотя обмен данными с другими приложениями системы или внутри единственного приложения, а также поддержка отмены случайных изменений в этих данных уже косвенно относятся к Swing и графическим компонентам, библиотека предоставляет весь арсенал, необходимый для реализации этих возможностей. Если в случае обмена данными за нас многое сделано и встроено на самом базовом уровне компонента JComponent, то при отмене и повторе изменений у нас есть лишь легкий фундамент UndoManager, который, тем не менее, является хорошей отправной точкой.