

Основы основ — *проще простого!*



# Swing

руководство для начинающих

БЕСПЛАТНЫЙ  
КОД  
В ИНТЕРНЕТЕ

**БОНУС!**  
[WWW.OSBORNE.COM](http://WWW.OSBORNE.COM)

Мощный инструмент для  
разработки графических  
интерфейсов

**Герберт Шилдт**

Автор бестселлеров по программированию — во всем мире продано более 3,5 млн. экземпляров его книг!

# Swing

## **A Beginner's Guide**

***HERBERT SCHILDT***

**McGraw-Hill/Osborne**

New York Chicago San Francisco  
Lisbon London Madrid Mexico City Milan  
New Delhi San Juan Seoul Singapore Sydney Toronto

# Swing

**руководство для начинающих**

***ГЕРБЕРТ ШИЛДТ***



**Москва • Санкт-Петербург • Киев  
2007**

ББК 32.973.26-018.2.75

Ш57

УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *В.В. Вейтмана*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресам:

info@williamspublishing.com, <http://www.williamspublishing.com>

115419, Москва, а/я 783; 03150, Киев, а/я 152

**Шилдт, Герберт.**

**Ш57 SWING: руководство для начинающих: Пер. с англ. — М.: ООО "И.Д. Вильямс", 2007. — 704 с.: ил. — Парал. тит. англ.**

**ISBN 978-5-8459-1162-9 (рус.)**

Автор данного руководства, известный специалист в области программирования, Герберт Шилдт, рассказывает читателю о базовых средствах библиотеки Swing, используемой для создания графических пользовательских интерфейсов Java-программ. Книга разделена на 10 модулей, каждый из которых посвящен группе сходных между собой управляющих элементов, а завершается она обсуждением технологий, используемых для обеспечения нормальной работы компонентов в реальных приложениях. Данная книга ориентирована на программистов-практиков, поэтому уже в первом модуле рассматриваются коды реальных программ. Материал остальных модулей также сопровождается большим количеством примеров. Освоив материал данной книги, читатель получит знания, которые позволят ему приступить к изучению более сложных вопросов.

**ББК 32.973.26-018.2.75**

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Osborne Media.

Authorized translation from the English language edition published by Osborne Publishing, Copyright © 2007 by The McGraw-Hill Companies.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2007

ISBN 978-5-8459-1162-9 (рус.)

ISBN 0-07-226314-8 (англ.)

© Издательский дом "Вильямс", 2007

© The McGraw-Hill Companies, 2007



# Оглавление

|                                                                               |     |
|-------------------------------------------------------------------------------|-----|
| Введение                                                                      | 15  |
| Модуль 1. Общие сведения о Swing                                              | 21  |
| Модуль 2. Метки, кнопки и обрамление                                          | 73  |
| Модуль 3. Полосы прокрутки, линейные регуляторы<br>и индикаторы хода процесса | 151 |
| Модуль 4. Управление компонентами.<br>Панели и строка подсказки               | 211 |
| Модуль 5. Списки                                                              | 263 |
| Модуль 6. Текстовые компоненты                                                | 319 |
| Модуль 7. Меню                                                                | 373 |
| Модуль 8. Таблицы и деревья                                                   | 433 |
| Модуль 9. Диалоговые окна                                                     | 503 |
| Модуль 10. Потоки, апплеты рисование и компоновка                             | 571 |
| Приложение. Ответы на вопросы для самоконтроля                                | 627 |
| Предметный указатель                                                          | 689 |

# Содержание

|                                                                   |           |
|-------------------------------------------------------------------|-----------|
| <b>Введение</b>                                                   | <b>15</b> |
| <b>Модуль 1. Общие сведения о Swing</b>                           | <b>21</b> |
| 1.1. Причины появления Swing                                      | 23        |
| 1.2. Основные свойства Swing                                      | 24        |
| Легковесные компоненты                                            | 24        |
| Настраиваемые стили                                               | 24        |
| AWT как основа Swing                                              | 25        |
| 1.3. Архитектура MVC                                              | 25        |
| 1.4. Компоненты и контейнеры                                      | 27        |
| Компоненты                                                        | 27        |
| Контейнеры                                                        | 28        |
| Панели контейнеров верхнего уровня                                | 29        |
| 1.5. Простая программа, использующая средства Swing               | 30        |
| Подробное обсуждение примера                                      | 32        |
| 1.6. Поддержка событий                                            | 37        |
| События                                                           | 38        |
| Источники событий                                                 | 38        |
| Обработчики событий                                               | 39        |
| Классы событий и интерфейсы обработчиков                          | 39        |
| 1.7. Использование кнопок                                         | 42        |
| 1.8. Компонент JPasswordField                                     | 54        |
| 1.9. Краткие сведения о диспетчерах компоновки                    | 69        |
| 1.10. Пакеты в составе Swing                                      | 70        |
| <b>Модуль 2. Метки, кнопки и обрамление</b>                       | <b>73</b> |
| 2.1. Общие сведения об обрамлении                                 | 74        |
| 2.3. Включение графического изображения в состав метки            | 85        |
| 2.4. Деактивизация метки                                          | 89        |
| 2.5. Отображение мнемонических обозначений клавиш в составе меток | 92        |
| 2.6. Общие сведения о кнопках                                     | 97        |
| Обработка событий действий                                        | 100       |
| Обработка событий элемента                                        | 101       |
| Обработка событий изменения состояния                             | 102       |
| 2.7. Размещение изображения на кнопке                             | 104       |

|                                                                               |     |
|-------------------------------------------------------------------------------|-----|
| 2.8. Определение кнопки по умолчанию                                          | 109 |
| Дополнительные средства, предоставляемые классом JButton                      | 109 |
| 2.9. Использование неименованного внутреннего класса<br>для обработки событий | 110 |
| 2.10. Использование класса JToggleButton                                      | 119 |
| Использование метода getItem() для определения<br>источника события элемента  | 123 |
| Дополнительные средства, предоставляемые<br>классом JToggleButton             | 126 |
| 2.11. Создание флажков опций с помощью класса JCheckBox                       | 127 |
| Дополнительные возможности при использовании<br>флажков опций                 | 133 |
| 2.12. Создание переключателей опций с помощью класса JRadioButton             | 134 |

### **Модуль 3. Полосы прокрутки, линейные регуляторы и индикаторы хода процесса 151**

|                                                                            |     |
|----------------------------------------------------------------------------|-----|
| 3.1. Интерфейс BoundedRangeModel                                           | 152 |
| 3.2. Полосы прокрутки                                                      | 155 |
| 3.3. Свойства объекта JScrollBar                                           | 157 |
| 3.4. Обработка событий регулировки объекта JScrollBar                      | 159 |
| Использование полос прокрутки                                              | 160 |
| 3.5. Дополнительные возможности, предоставляемые<br>полосами прокрутки     | 164 |
| 3.6. Линейные регуляторы                                                   | 170 |
| 3.7. Установка маркеров и надписей                                         | 173 |
| Пример, демонстрирующий работу с линейными регуляторами                    | 175 |
| 3.8. Дополнительные возможности, предоставляемые<br>линейными регуляторами | 179 |
| 3.9. Индикаторы хода процесса                                              | 202 |
| Использование индикатора хода процесса                                     | 204 |
| 3.10. Дополнительные возможности компонента JProgressBar                   | 207 |

### **Модуль 4. Управление компонентами. Панели и строка подсказки 211**

|                                                                    |     |
|--------------------------------------------------------------------|-----|
| 4.1. Использование объектов JPanel                                 | 212 |
| 4.2. Использование панелей для организации компонентов             | 215 |
| 4.3. Использование компонента JPanel в качестве панели содержимого | 219 |
| 4.4. Использование объекта JScrollPane                             | 224 |
| Простой пример использования класса JScrollPane                    | 226 |

|                                                  |     |
|--------------------------------------------------|-----|
| 4.5. Добавление заголовков                       | 228 |
| 4.6. Дополнительные возможности панели прокрутки | 231 |
| 4.7. Использование класса JTabbedPane            | 239 |
| 4.8. Дополнительные возможности JTabbedPane      | 242 |
| 4.9. Использование JSplitPane                    | 252 |
| 4.10. Дополнительные возможности JSplitPane      | 256 |
| 4.11. Формирование подсказки                     | 258 |

## **Модуль 5. Списки** **263**

|                                                                   |     |
|-------------------------------------------------------------------|-----|
| 5.1. Использование компонента JList                               | 264 |
| 5.2. Выбор нескольких пунктов                                     | 270 |
| 5.3. Дополнительные возможности компонента JList                  | 274 |
| 5.4. Использование JComboBox                                      | 288 |
| 5.5. Создание раскрывающегося списка, допускающего редактирование | 293 |
| 5.6. Дополнительные возможности JComboBox                         | 294 |
| 5.6. Использование компонента JSpinner                            | 301 |
| 5.7. Использование модели SpinnerListModel                        | 303 |
| 5.8. Использование модели SpinnerNumberModel                      | 307 |
| 5.10. Использование модели SpinnerDateModel                       | 311 |

## **Модуль 6. Текстовые компоненты** **319**

|                                                                              |     |
|------------------------------------------------------------------------------|-----|
| 6.1. Класс JTextComponent                                                    | 320 |
| 6.2. Компонент JTextField                                                    | 324 |
| 6.3. Компонент JPasswordField                                                | 332 |
| 6.4. Компонент JFormattedTextField                                           | 336 |
| 6.5. Создание компонента JFormattedTextField на основе типа данных           | 338 |
| 6.6. Создание компонента JFormattedTextField<br>на основе сведений о формате | 338 |
| 6.8. Компонент JTextArea                                                     | 348 |
| 6.9. Дополнительные возможности JTextArea                                    | 354 |

## **Модуль 7. Меню** **373**

|                                                           |     |
|-----------------------------------------------------------|-----|
| 7.1. Общие сведения о меню                                | 374 |
| 7.2. Классы JMenuBar, JMenu и JMenuItem                   | 377 |
| Класс JMenuBar                                            | 377 |
| Класс JMenu                                               | 379 |
| Класс JMenuItem                                           | 381 |
| 7.3. Создание главного меню                               | 382 |
| 7.4. Мнемонические обозначения и клавиши быстрого доступа | 388 |

## 10 Содержание

---

|                                                                         |     |
|-------------------------------------------------------------------------|-----|
| 7.5. Связывание изображений и подсказок с пунктами меню                 | 392 |
| 7.6. Использование объектов JRadioButtonMenuItem<br>и JCheckBoxMenuItem | 400 |
| 7.7. Создание контекстного меню                                         | 405 |
| 7.8. Создание панели инструментов                                       | 410 |
| Дополнительные возможности панелей инструментов                         | 413 |
| 7.9. Действия                                                           | 414 |
| 7.10. Окончательный вариант программы MenuDemo                          | 422 |

### **Модуль 8. Таблицы и деревья**

**433**

|                                                     |     |
|-----------------------------------------------------|-----|
| 8.13. Общие сведения о компоненте JTable            | 434 |
| 8.2. Режимы выбора                                  | 443 |
| 8.3. Выбор столбцов и ячеек                         | 443 |
| Выбор столбцов                                      | 444 |
| Выбор ячеек                                         | 444 |
| 8.4. Обработка событий выбора                       | 446 |
| Общие сведения о событиях выбора                    | 446 |
| Обработка событий выбора строк                      | 447 |
| Обработка событий выбора столбцов                   | 450 |
| Обработка событий выбора ячеек                      | 455 |
| 8.5. Обработка событий модели таблицы               | 456 |
| 8.6. Изменение размеров таблицы                     | 462 |
| 8.7. Модель таблицы, определяемая разработчиком     | 466 |
| 8.8. Дополнительные возможности таблиц              | 472 |
| 8.9. Общие сведения о компоненте JTree              | 481 |
| 8.10. Обработка событий компонента JTree            | 489 |
| Событие TreeSelectionEvent                          | 489 |
| Событие TreeExpansionEvent                          | 490 |
| Событие TreeModelEvent                              | 490 |
| Программа, демонстрирующая обработку событий дерева | 491 |
| 8.12. Дополнительные возможности компонента JTree   | 496 |

### **Модуль 9. Диалоговые окна**

**503**

|                                |     |
|--------------------------------|-----|
| 9.1. Класс JOptionPane         | 504 |
| 9.2. Метод showMessageDialog() | 507 |
| 9.3. Метод showConfirmDialog() | 513 |
| 9.4. Метод showInputDialog()   | 519 |
| 9.5. Метод showOptionDialog()  | 524 |
| 9.6. Класс JDialog             | 530 |

|                                                      |     |
|------------------------------------------------------|-----|
| 9.7. Создание немодального диалогового окна          | 538 |
| 9.8. Выбор файлов с помощью компонента JFileChooser  | 540 |
| 9.9. Использование фильтров                          | 546 |
| 9.10. Дополнительные возможности класса JFileChooser | 550 |
| 9.11. Выбор цвета с помощью окна JColorChooser       | 564 |

## **Модуль 10. Поток, апплеты рисование и компоновка 571**

|                                                                          |     |
|--------------------------------------------------------------------------|-----|
| 10.1. Многопоточность и Swing                                            | 572 |
| 10.2. Использование класса Timer                                         | 580 |
| 10.3. Общие сведения об апплетах                                         | 586 |
| 10.4. Основные элементы апплетов                                         | 586 |
| 10.5. Формирование графического пользовательского интерфейса для апплета | 589 |
| Простой апплет, использующий средства Swing                              | 590 |
| 10.6. Общие сведения о рисовании                                         | 599 |
| 10.7. Графический контекст                                               | 600 |
| 10.8. Определение области отображения                                    | 600 |
| 10.9. Запрос на рисование                                                | 601 |
| Пример программы, осуществляющей рисование                               | 602 |
| 10.10. Диспетчер компоновки GridBagLayout                                | 608 |
| 10.11. Диспетчер компоновки BoxLayout                                    | 617 |

## **Приложение. Ответы на вопросы для самоконтроля 627**

|                                                                            |     |
|----------------------------------------------------------------------------|-----|
| Модуль 2. Метки, кнопки и оформление                                       | 634 |
| Модуль 3. Полосы прокрутки, линейные регуляторы и индикаторы хода процесса | 639 |
| Модуль 4. Управление компонентами. Панели и строка подсказки               | 646 |
| Модуль 5. Списки                                                           | 652 |
| Модуль 6. Текстовые компоненты                                             | 658 |
| Модуль 7. Меню                                                             | 667 |
| Модуль 8. Таблицы и деревья                                                | 669 |
| Модуль 9. Диалоговые окна                                                  | 676 |
| Модуль 10. Поток, апплеты рисование и компоновка                           | 679 |

## **Предметный указатель 689**

## Об авторе

Герберт Шилдт — признанный авторитет в вопросах использования языков C, C++, Java и C#, а также программирования в системе Windows. Общий тираж его книг превышает 3,5 миллиона экземпляров, они переведены на многие языки мира. Шилдт — автор многочисленных бестселлеров, среди которых *Java: A Beginner's Guide*, *Java: The Complete Reference*, *C++: A Beginner's Guide*, *C++: The Complete Reference*, *C: The Complete Reference* и *C#: The Complete Reference* (переводы этих книг на русский язык: *Java: руководство для начинающих*, *Полный справочник по Java*, *C++: руководство для начинающих*, *Полный справочник по C++*, *Полный справочник по C*, *Полный справочник по C#*). Он также является одним из авторов книги *The Art of Java (Искусство программирования на Java)*. Герберт Шилдт закончил университет штата Иллинойс. Адрес его Web-сайта [www.HerbSchildt.com](http://www.HerbSchildt.com).

## Благодарности

Я хочу поблагодарить Хейла Прингла (Hale Pringle) за отлично выполненную работу по редактированию текста, а также за его ценные замечания. Хейл — высококвалифицированный специалист в вопросах использования Swing; его вклад в данную книгу невозможно переоценить.

Я также благодарен Джеймсу Холмсу (James Holmes). Прочитав несколько глав, он щедро поделился со мной своим опытом. Джеймс — талантливый программист, автор книги *Struts: The Complete Reference* и мой соавтор по книге *The Art of Java*.

И наконец, я хочу выразить благодарность Венди Риналди (Wendy Rinaldi), редактору McGraw-Hill, советы которой всегда помогают мне и поддержку которой я чувствую уже в течение многих лет.



# Введение

Как известно, большинство потенциальных покупателей судят о продукте по первому впечатлению. Программные продукты — не исключение из этого правила. Внешний вид окна и расположение управляющих элементов — основные критерии, которые многие пользователи принимают во внимание, оценивая программу, потому важно, чтобы интерфейс был красив и прост в работе. Этого позволяет добиться Swing.

Swing — это набор инструментов, предназначенный для создания графических пользовательских интерфейсов (Graphical User Interface — GUI) современных программ. Предоставляя набор тщательно разработанных компонентов — кнопок, таблиц, деревьев, полей редактирования, панелей с прокруткой, — Swing дает разработчику возможность создавать привлекательные интерфейсы, удовлетворяющие требованиям конкретных приложений и сред, в которых они выполняются. Поддержка переключаемых стилей, архитектура “модель–представление–контроллер” (Model-View-Controller — MVC), простота в настройке — все это создает условия для создания высококачественных продуктов. Средства Swing завоевали себе место среди инструментов программирования, и игнорировать их невозможно.

Цель данной книги — дать читателю возможность, затратив минимальные усилия, получить основные сведения о Swing и программировании с помощью данной библиотеки. Книга ориентирована на программистов-практиков; примеры реальных программ, использующих средства Swing, приводятся уже в конце модуля 1. Начинается книга описанием архитектуры Swing и общих принципов разработки. Затем вниманию читателя предлагаются базовые элементы Swing: набор компонентов и основные технологии, необходимые для их использования. Прочитав данную книгу, вы узнаете достаточно для того, чтобы создавать свои приложения, интерфейс которых будет выглядеть вполне профессионально.

Необходимо, однако, помнить, что эта книга — лишь первый шаг на пути освоения Swing. Данная библиотека таит в себе огромные возможности. Для их реализации разработано большое количество классов, интерфейсов и технологических приемов, но подробное изучение средств, делающих Swing столь мощным инструментом, — задача не для новичка. Поэтому детальное описание всех нюансов библиотеки Swing выходит за рамки рассмотрения данной книги. Настоящая книга призвана лишь открыть вам двери в мир Swing-программирования и детально объяснить те базовые средства, без которых невозможно обойтись и которые вы будете постоянно использовать в рабо-

те над каждой Java-программой. Освоив материал книги, вы получите знания, которые позволят вам приступить к изучению более сложных вопросов.

Заметьте, что Swing не является самым простым в изучении инструментом для создания графических интерфейсов, однако он несомненно самый важный из них. Если вы собираетесь разрабатывать программы на языке Java, вам следует освоить Swing. Затраченное время и усилия многократно окупятся.

## Структура книги

Данная книга представляет собой руководство, разделенное на модули приблизительно одинакового объема. В каждом модуле приведена информация, касающаяся использования Swing. Настоящая книга — уникальна в своем роде, поскольку в ней использован ряд приемов, повышающих эффективность обучения.

## Основные вопросы

Каждый модуль начинается с перечня основных вопросов, рассматриваемых в нем.

## Тест для самоконтроля



В конце каждого модуля приведен тест для самоконтроля, позволяющий читателю проверить качество усвоения материала. Ответы на вопросы приведены в приложении.

## Вопросы для текущего контроля

В конце основных разделов приведены вопросы для текущего контроля, которые позволят вам убедиться, что вы правильно поняли основную мысль раздела. Ответы на вопросы оформлены как сноска и расположены внизу страницы.

## Спросим у опытного программиста

Время от времени вам будут встречаться врезки “Спросим у опытного программиста”. В них содержится дополнительная информация или комментарии по вопросам, рассматриваемым в текущем разделе. Предлагаемые сведения оформлены в виде вопросов и ответов на них.

## Проекты

Каждый модуль включает один или несколько проектов, демонстрирующих практическое применение изложенного материала. Это примеры реальных программ, которые можно использовать как заготовки для ваших приложений.

## Необходимые навыки программирования на Java

При написании данной книги предполагалось, что читатель обладает некоторыми знаниями о языке Java. Вам не обязательно быть признанным специалистом Java-программирования, но основы этого языка вы должны знать. Если вы хотите повысить свою квалификацию в этой области, я могу порекомендовать вам последние редакции моих книг *Java: A Beginner's Guide* и *Java: The Complete Reference*, которые вышли в издательстве McGraw-Hill (*Java: руководство для начинающих* и *Полный справочник по Java*; Издательский дом "Вильямс").

## Необходимое программное обеспечение

Для компиляции и запуска программ, рассмотренных в данной книге, вам потребуется пакет Java Developers Kit (JDK). На момент написания книги самой новой доступной версией была JDK 5 (J2SE 5). Именно с помощью этой реализации JDK тестировался код, приведенный в данной книге в качестве примеров. Тем не менее большая часть кода совместима с версией 1.4 и даже с версиями 1.3 и 1.2. Однако я рекомендовал бы вам использовать последние реализации JDK.

## Код опубликован в Web

Не забывайте, что исходный код всех примеров и проектов, рассмотренных в этой книге, свободно доступен в Web по адресу [www.osborne.com](http://www.osborne.com).

## Другие источники информации

У вас в руках одна из многих книг по программированию, написанных Гербертом Шилдтом. Возможно, вас заинтересуют и другие книги этого автора.

Тем, кто хочет больше узнать о программировании на языке Java, рекомендуем следующие книги:

- *Java: The Complete Reference (Полный справочник по Java)*
- *Java: A Beginner's Guide (Java: руководство для начинающих)*
- *The Art of Java (Искусство программирования на Java)*

Если вы хотите изучить язык C++, вам помогут книги, перечисленные ниже.

- *C++: The Complete Reference (Полный справочник по C++)*
- *Teach Yourself C++*
- *C++ From the Ground Up (C++: базовый курс)*
- *STL Programming From the Ground Up*
- *The Art of C++*

Если вас интересует язык C#, обратите внимание на следующие книги Г. Шилдта:

- *C#: A Beginner's Guide*
- *C#: The Complete Reference (Полный справочник по C#)*

И наконец, тем, кто хочет лучше познакомиться с языком C, можно порекомендовать книги, приведенные ниже.

- *C: The Complete Reference (Полный справочник по C)*
- *Teach Yourself C*

За дополнительными рекомендациями обращайтесь непосредственно к Герберту Шилдту — признанному авторитету в области программирования.

## От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: [info@williamspublishing.com](mailto:info@williamspublishing.com)  
WWW: <http://www.williamspublishing.com>

Адреса для писем из:

России: 115419, Москва, а/я 783  
Украины: 03150, Киев, а/я 152

# Модуль 1

## Общие сведения о Swing

- 1.1.** Причины появления Swing
- 1.2.** Легковесные компоненты и настраиваемые стили
- 1.3.** Архитектура MVC
- 1.4.** Архитектура “модель–представление–контроллер”
- 1.5.** Компоненты и контейнеры Swing
- 1.6.** Написание, компиляция и выполнение простого приложения Swing
- 1.7.** Принципы обработки событий
- 1.8.** Использование компонента JButton
- 1.9.** Использование компонента JTextField
- 1.10.** Общие сведения о диспетчерах компоновки
- 1.11.** Пакеты Swing

Практически каждое приложение можно разделить на две части. Первая — это код, выполняющий те действия, для которых и была написана программа, например копирование файла, формирование запроса к базе данных, оформление заказа или финансовые вычисления. Вторая часть — это интерфейс, который определяет порядок взаимодействия пользователя с программой. Действия, выполняемые приложением, безусловно важны. Но нельзя недооценивать также значение пользовательского интерфейса, который не только задает внешний вид программы, но часто также определяет, насколько успешным будет данный продукт на рынке. Таким образом, создание привлекательных, согласованных и эффективных пользовательских интерфейсов — неотъемлемая часть процесса разработки программ. Для программистов, применяющих язык Java, путь к созданию высококачественных интерфейсов — это использование Swing.

Swing представляет собой набор классов, применяемых для создания графических пользовательских интерфейсов (Graphical User Interface — GUI) современных приложений, в том числе Web-программ. Swing предоставляет богатый набор визуальных компонентов, например, кнопок, полей редактирования, полос прокрутки, флажков опций и таблиц, разработанных так, чтобы их можно было успешно применять в самых различных приложениях. Посредством Swing можно разработать интерфейс приложения на профессиональном уровне, наилучшим образом удовлетворяющий потребностям пользователей. Более того, трудно представить профессионального Java-программиста, не владеющего Swing. Цель данного модуля — дать читателю общее представление о Swing, познакомить его с историей, основными понятиями и базовыми средствами данного инструмента. Как вы увидите, в набор Swing входит большое число взаимозависимых элементов, способных совместно решать одну задачу. Такая взаимозависимость существенно повышает мощность Swing, но в то же время данный инструмент остается простым в использовании. Однако для начинающих зависимость элементов Swing друг от друга поначалу может создать некоторые неудобства, так как для того, чтобы понять один аспект работы Swing, необходимо представлять себе и другие. Поэтому прежде чем приступить к детальному изучению отдельных возможностей Swing, полезно сначала получить общее представление о Swing в целом. Данный модуль предоставляет читателю обзор Swing. В нем также затронуты некоторые часто используемые компоненты и показано их использование в простой программе. В последующих модулях эти же компоненты будут рассмотрены подробнее. Советую внимательно прочитать данный модуль. Это в особенности важно потому, что материал, приведенный в некоторых его разделах, будет подробнее рассмотрен далее в этой книге.

**ВАЖНО!**

## **1.1. Причины появления Swing**

В ранних версиях Java средства Swing еще отсутствовали. Можно считать, что они стали попыткой преодолеть трудности, связанные с использованием первой оконной подсистемы — AWT (Abstract Window Toolkit). В AWT был определен базовый набор управляющих элементов и окон, позволяющий создавать графические интерфейсы, правда, с ограниченными возможностями. Одним из ограничений AWT была платформенно-ориентированная поддержка визуальных компонентов. В результате внешний вид интерфейсных элементов определялся не средствами Java, а используемой платформой. Элементы, создаваемые средствами AWT, назывались тяжеловесными.

Платформенно-ориентированная поддержка интерфейсных элементов стала источником ряда проблем. Во-первых, из-за различия в операционных системах компоненты по-разному выглядели и даже по-разному вели себя на различных платформах. Это было прямым нарушением главного принципа Java: код, единожды написанный, должен работать везде. Во-вторых, внешний вид каждого компонента был фиксированным, и изменить его было достаточно трудно (причина та же — платформенно-ориентированные средства поддержки). В-третьих, использование тяжеловесных компонентов порождало ряд проблем, например, такие компоненты всегда имели прямоугольную форму и были непрозрачны.

### **Спросим у опытного программиста**

**Вопрос.** Вы говорите, что Swing — часть Java Foundation Classes. Что это такое?

**Ответ.** JFC — это набор классов, обеспечивающих базовую поддержку корпоративных приложений с пользовательским интерфейсом. Другими компонентами JFC являются AWT (Abstract Window Toolkit), Java 2D, Drag и Drop и Accessibility API.

Потребовалось не слишком много времени, чтобы осознать, что ограничения AWT слишком серьезны и нужен новый подход. В результате в 1997 г. появился набор Swing. Он был включен в состав JFC (Java Foundation Classes). Первоначально Swing использовался в Java 1.1 как отдельная библиотека. Однако в Java 1.2 средства Swing (как и остальные элементы JFC) были полностью интегрированы в Java. В настоящее время Swing — неотъемлемая часть данного языка.



**ВАЖНО!**

### **1.2. Основные свойства Swing**

Как уже было сказано, набор Swing был создан для того, чтобы преодолеть ограничения, связанные с AWT. Для достижения этой цели разработчики реализовали два подхода: легковесные компоненты и настраиваемые стили. Совместно эти решения позволили создать элегантный и простой в использовании инструмент, свободный от недостатков AWT. Легковесные компоненты и настраиваемые стили считаются основными свойствами Swing. Рассмотрим их подробнее.

#### **Легковесные компоненты**

Все компоненты Swing, за небольшим исключением, являются легковесными. Это означает, что они написаны полностью на Java и не зависят от средств той платформы, на которой выполняется программа. Поскольку при воспроизведении легковесных компонентов используются графические примитивы, они могут иметь форму, отличную от прямоугольной. Следовательно, такие компоненты являются более эффективными и гибкими. Поскольку легковесные компоненты не преобразуются в платформенно-ориентированные элементы, их внешний вид определяет Swing, а не операционная система. Следовательно, интерфейсные элементы, созданные с помощью Swing, выглядят одинаково на разных платформах.

#### **Настраиваемые стили**

Swing поддерживает настраиваемые стили. Поскольку каждый компонент воспроизводится с помощью Java-кода, его внешний вид полностью контролируется средствами Swing. Это означает, что внешний вид компонента можно отделить от логики его работы; эта возможность реализована в Swing. Такое разделение позволяет изменить внешний вид компонента, не затрагивая других его характеристик. Другими словами, появляется возможность перенастроить внешний вид, не создавая побочных эффектов в коде, использующем компонент. Более того, можно также создавать глобальные стили, определяющие, как будет выглядеть интерфейс в целом. При переключении стиля внешний вид всех элементов изменяется автоматически.

Настраиваемые стили создают ряд преимуществ. Становится возможным обеспечить один и тот же внешний вид окна программы на разных платформах. С другой стороны, можно также настроить программу таким образом, что ее интерфейс будет соответствовать той или иной конкретной платформе. Например, если вы знаете, что ваша программа будет выполняться только в среде Windows, вы можете задать для нее внешний вид, соответствующий остальным

программам, работающим в данной системе. Можно также сформировать свой стиль, отличающийся от стилей известных платформ. И наконец, есть возможность динамически изменять внешний вид программы в процессе ее работы.

Каждому пользователю Swing изначально доступны три стиля: *metal*, *Windows* и *Motif*. Стиль *metal* также называют стилем Java. Это платформенно-независимый стиль, доступный во всех средах выполнения Java-программ. Он принимается по умолчанию. Существует также стиль *Mac*, доступный в системе *Mac*, и стиль *GTK+*, который можно использовать на платформах *Mac* и *Solaris*. В данной книге используется стиль *Java*. Такой выбор сделан потому, что данный стиль доступен на всех платформах. Однако никто не мешает вам при желании экспериментировать с другими стилями и даже создавать свои.

## AWT как основа Swing

Перед тем как продолжить наш разговор, необходимо отметить одну очень важную деталь. Несмотря на то что Swing устраняет ряд ограничений, присущих AWT, он не заменяет данный инструмент. Более того, в основу Swing положены некоторые основные решения, принятые в AWT. В частности, в Swing используется такой же механизм обработки событий, как и в AWT. И хотя от читателей данной книги не требуется знание AWT, желательно понимать структуру этой библиотеки и ее возможности. Этим вы упростите для себя восприятие Swing.

На заметку



Подробное описание AWT вы найдете в моей книге *Java: The Complete Reference*, выпущенной издательством McGraw-Hill/Osborne (2007) (русский перевод *Полный справочник по Java*, Издательский дом "Вильямс").

ВАЖНО!

## 1.3 Архитектура MVC

Визуальный компонент можно представить себе как элемент, сочетающий следующие характеристики.

- Способ отображения на экране.
- Реакция на действия пользователя.
- Информация, связанная с данным компонентом.

Этими тремя характеристиками компонент обладает независимо от того, какая архитектура использовалась при его создании, тем не менее одна из архитектур с честью выдержала испытания временем и доказала свою эффектив-

ность. Это архитектура "*модель–представление–контроллер*" (MVC — Model-View-Controller).

Преимущество данной архитектуры состоит в том, что каждая ее составляющая в точности соответствует одной из характеристик компонента. *Модель* задает состояние компонента. Например, для флажка опций модель содержит признак того, установлен или сброшен флажок. *Представление* определяет, как компонент будет отображаться на экране, в том числе как будет представлено текущее состояние модели. *Контроллер* обеспечивает реакцию компонента на действия пользователя. Например, когда пользователь щелкает на флажке опций, реакцией контроллера является изменение модели (установка сброшенного флажка или сброс установленного). Кроме того, в ответ на действия пользователя обновляется представление. Разделяя компонент на модель, представление и контроллер, можно добиться того, что реализация одной части не будет оказывать влияния на остальные две. Например, разные представления могут отображать один и тот же компонент разными способами; при этом модель и контроллер остаются неизменными.

Несмотря на то что принципы архитектуры MVC выглядят очень привлекательно, в случае компонентов Swing разделение функций представления и контроллера не дает ощутимых преимуществ. Поэтому в Swing используется модифицированный вариант MVC, в котором представление и контроллер объединены в единый элемент, называемый *представителем пользовательского интерфейса*. Подход, используемый в Swing, известен как архитектура "*модель–представитель*" или архитектура с *выделенной моделью*. Таким образом, несмотря на то, что компоненты Swing базируются на MVC, их нельзя рассматривать как классическую реализацию данной архитектуры.

Настраиваемые стили Swing стали возможными благодаря использованию архитектуры "*модель–представитель*". Поскольку представление и контроллер отделены от модели, их можно заменить, не затрагивая механизмов взаимодействия компонентов с программой. С другой стороны, модель можно модифицировать, не изменяя при этом внешний вид элемента и его реакцию на действия пользователя.

Поддержка архитектуры "*модель–представитель*" обуславливает наличие в компонентах Swing двух объектов. Первый из них представляет модель, второй соответствует представителю пользовательского интерфейса. Модели определяются посредством интерфейсов, например, модель для кнопки определена с помощью интерфейса `ButtonModel`. Представителями пользовательского интерфейса являются подклассы `ComponentUI`. Например, в роли представителя для кнопки выступает объект `ButtonUI`. В обычных условиях программа не взаимодействует с представителем интерфейса.



## Вопросы для текущего контроля .....

1. Назовите два основных свойства Swing.
2. Заменяет ли Swing AWT?
3. Что означает аббревиатура MVC? Как называется модификация архитектуры MVC, используемая Swing?

**ВАЖНО!**

## 1.4. Компоненты и контейнеры

В состав графического пользовательского интерфейса, созданного средствами Swing, входят элементы двух типов: компоненты и контейнеры. Такое разделение во многом условно, так как контейнеры являются в то же время и компонентами. Различие между ними в их назначении. Компоненты — это независимые элементы, в качестве примеров которых можно привести кнопки или линейные регуляторы. Контейнер может содержать в себе несколько компонентов и представляет собой специальный тип компонента. Чтобы компонент отобразился на экране, его надо поместить в контейнер. Таким образом, в составе графического пользовательского интерфейса должен присутствовать хотя бы один контейнер. Поскольку контейнеры являются компонентами, один контейнер может находиться в составе другого. Это позволяет формировать так называемую *иерархию контейнеров*, на вершине которой должен находиться *контейнер верхнего уровня*.

Рассмотрим компоненты и контейнеры более подробно.

## Компоненты

Подавляющее большинство компонентов Swing создается с помощью классов, являющихся потомками класса `JComponent`. (Единственным исключением из этого правила являются четыре контейнера верхнего уровня, описанных в следующем разделе.) Класс `JComponent` реализует функциональные

1. Двумя основными свойствами Swing считаются легковесные компоненты и настраиваемые стили.
2. Нет. Некоторые решения, типичные для AWT, положены в основу Swing.
3. MVC означает "модель—представление—контроллер" (Model-View-Controller). В Swing используется модификация данной модели, называемая архитектурой "модель—представитель" или архитектурой с выделенной моделью.

возможности, общие для всех компонентов, например, поддерживает настраиваемые стили. `JComponent` наследует свойства классов `AWT Container` и `Component`. Таким образом, компоненты Swing строятся на базе AWT-компонентов и совместимы с ними.

Классы, представляющие все компоненты Swing, содержатся в пакете `javax.swing`. В следующей таблице приведены имена классов для компонентов Swing (включая также компоненты, являющиеся контейнерами).

|                            |                             |                                   |                                  |
|----------------------------|-----------------------------|-----------------------------------|----------------------------------|
| <code>JApplet</code>       | <code>JButton</code>        | <code>JCheckBox</code>            | <code>JCheckBoxMenuItem</code>   |
| <code>JColorChooser</code> | <code>JComboBox</code>      | <code>JComponent</code>           | <code>JDesktopPane</code>        |
| <code>JDialog</code>       | <code>EditorPane</code>     | <code>JFileChooser</code>         | <code>JFormattedTextField</code> |
| <code>JFrame</code>        | <code>JInternalFrame</code> | <code>JLabel</code>               | <code>JLayeredPane</code>        |
| <code>JList</code>         | <code>JMenu</code>          | <code>JMenuBar</code>             | <code>JMenuItem</code>           |
| <code>JOptionPane</code>   | <code>JPanel</code>         | <code>JPasswordField</code>       | <code>JPopupMenu</code>          |
| <code>JProgressBar</code>  | <code>JRadioButton</code>   | <code>JRadioButtonMenuItem</code> | <code>JRootPane</code>           |
| <code>JScrollBar</code>    | <code>JScrollPane</code>    | <code>JSeparator</code>           | <code>JSlider</code>             |
| <code>JSpinner</code>      | <code>JSplitPane</code>     | <code>JTabbedPane</code>          | <code>JTable</code>              |
| <code>JTextArea</code>     | <code>TextField</code>      | <code>JTextPane</code>            | <code>JToggleButton</code>       |
| <code>JToolBar</code>      | <code>JToolTip</code>       | <code>JTree</code>                | <code>JViewport</code>           |
| <code>JWindow</code>       |                             |                                   |                                  |

Заметьте, что имена всех классов начинаются с буквы `J`. Например, метке соответствует класс `JLabel`, кнопке — класс `JButton`, а для поддержки полосы прокрутки используется класс `JScrollBar`. Использованию этих компонентов и посвящена основная часть данной книги.

## Контейнеры

В Swing определены два типа контейнеров. К первому типу относятся контейнеры верхнего уровня: `JFrame`, `JApplet`, `JWindow` и `JDialog`. Они не принадлежат к числу подклассов `JComponent`, но тем не менее являются потомками `Component` и `Container`. В отличие от других компонентов Swing, контейнеры верхнего уровня являются тяжеловесными. По этим причинам контейнеры верхнего уровня составляют специальную группу в наборе Swing.

Как следует из названия, контейнер верхнего уровня должен находиться на вершине иерархии. Он не может включаться в состав других контейнеров. Более того, любая иерархия должна начинаться именно с контейнера верхнего уровня. В приложениях для этой цели чаще всего используется объект `JFrame`, в апплетах — `JApplet`.

Контейнеры второго типа — это легковесные контейнеры, являющиеся потомками `JComponent`. В качестве примеров легковесных контейнеров можно привести `JPanel` и `JRootPane`. Легковесные контейнеры могут включаться в другие контейнеры, поэтому они часто используются для объединения группы связанных друг с другом компонентов.

## Панели контейнеров верхнего уровня

В каждом контейнере верхнего уровня реализован набор панелей. На вершине иерархии находится *корневая панель*, т.е. экземпляр класса `JRootPane`. Это легковесный контейнер, цель которого — управление другими панелями и, при необходимости, полосой меню. Корневая панель включает в себя *"стеклянную" панель* (`glass pane`), *панель содержимого* (`content pane`) и *панель слоя* (`layered pane`).

"Стеклянная" панель — это панель верхнего уровня, которая расположена поверх остальных панелей. По умолчанию "стеклянная" панель представляет собой "прозрачный экземпляр" `JPanel`. "Стеклянная" панель позволяет управлять событиями мыши, относящимися ко всему контейнеру, а не к содержащимся в нем компонентам. Она также обеспечивает рисование компонентов. В большинстве случаев необходимость непосредственного взаимодействия со "стеклянной" панелью не возникает.

Панель слоя представляет собой экземпляр класса `JLayeredPane`. Она поддерживает "третье измерение" для компонентов, т.е. определяет правила перекрытия одних компонентов другими. (Панель слоя позволяет задавать *Z-положение* компонента, однако эта возможность используется сравнительно редко.) В составе панели слоя находится панель содержимого и может также находиться строка меню.

Несмотря на то что "стеклянная" панель и панель слоя — неотъемлемые части контейнера верхнего уровня и выполняют важные функции, их действия большей частью скрыты не только от пользователей, но и от разработчиков. Приложение в основном взаимодействует с панелью содержимого, так как именно в нее включаются визуальные компоненты. Другими словами, добавляя компонент, например кнопку, к контейнеру верхнего уровня, вы на самом деле добавляете его к панели содержимого. По умолчанию панель содержимого представляет собой "непрозрачный экземпляр" `JPanel`.

На заметку



Для обращения к "стеклянной" панели, панели содержимого и панели слоя в `JRootPane` предусмотрены переменные `glassPane`, `contentPane` и `layeredPane`.



## Вопросы для текущего контроля

1. Все компоненты Swing должны располагаться в \_\_\_\_\_.
  2. Все компоненты Swing, за редким исключением, являются подклассами класса `JComponent`. Да или нет?
  3. Какие панели, кроме корневой, содержит каждый контейнер верхнего уровня?
  4. Контейнеры верхнего уровня — это контейнеры специального типа, так как они являются тяжеловесными. Да или нет?
- 

**ВАЖНО!****1.5.**

## Простая программа, использующая средства Swing

Перед тем как продолжить обсуждение теоретических вопросов, желательно рассмотреть в качестве примера простую программу, созданную с использованием средств Swing. Несмотря на простоту, данная программа демонстрирует ключевые свойства Swing и дает представление о контейнере `JFrame` и компоненте `JLabel`. Как было сказано ранее, `JFrame` — это контейнер верхнего уровня, обычно используемый в приложениях Swing. `JLabel` — компонент Swing, с помощью которого создается метка, используемая для представления информации. Метка — самый простой компонент Swing, поскольку он не реагирует на действия пользователя, а лишь отображает данные. В данной программе создается контейнер `JFrame`, в который помещается экземпляр компонента `JLabel`, отображающий текстовое сообщение.

```
// Простая программа, созданная с использованием средств Swing
// Основные классы Swing содержатся в пакете javax.swing.
import javax.swing.*;
```

```
class SwingDemo {

    SwingDemo() {
```

- 
1. Контейнере.
  2. Да.
  3. "Стеклянная" панель, панель содержимого и панель слоя.
  4. Да.

```
// Создание контейнера верхнего уровня (JFrame).
JFrame jfrm = new JFrame("A Simple Swing Program");

// Установка начальных размеров фрейма.
jfrm.setSize(275, 100);

// Завершение программы при закрытии пользователем
// окна приложения.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Создание метки.
JLabel jlab = new JLabel("Swing powers the modern Java GUI.");

// Включение метки в состав панели содержимого.
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Создание фрейма в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new SwingDemo();
        }
    });
}
```

Данная программа компилируется и запускается так же, как и любое Java-приложение. Для компиляции надо ввести в командной строке приведенное ниже выражение.

```
javac SwingDemo.java
```

Команда запуска программы имеет следующий вид:

```
java SwingDemo
```

В процессе работы программа отображает окно, показанное на рис. 1.1.



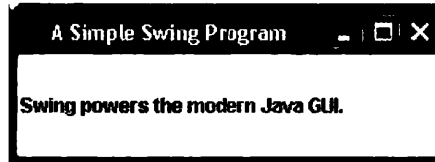


*Рис. 1.1. Окно, отображаемое в процессе работы программы, при создании которой использовались средства Swing*

### Подробное обсуждение примера

Поскольку программа `SwingDemo` иллюстрирует основные понятия Swing, имеет смысл обсудить ее подробно, уделив внимание каждой строке кода. В начале программы осуществляется импорт пакета.

```
import javax.swing.*;
```

Данный пакет содержит компоненты и модели Swing. В частности, в нем определены классы, реализующие метки, кнопки, поля редактирования и меню. Этот пакет должен быть включен в каждую программу, использующую Swing.

Далее в программе объявляется класс `SwingDemo` и его конструктор. Именно в конструкторе выполняется большая часть действий программы. Код конструктора начинается с создания объекта `JFrame`. Для этого используется следующая строка кода:

```
JFrame jfrm = new JFrame("A Simple Swing Program.");
```

Так создается контейнер `jfrm`, который определяет прямоугольное окно, содержащее строку заголовка, кнопки, предназначенные для закрытия, минимизации, максимизации и восстановления размеров окна, а также системное меню. Другими словами, с помощью приведенного выше выражения создается стандартное окно верхнего уровня. Строка заголовка передается конструктору в качестве параметра.

Затем с помощью следующего выражения устанавливаются размеры окна.

```
jfrm.setSize(275, 100);
```

Метод `setSize()`, унаследованный классом `JFrame` от AWT-класса `Component`, устанавливает размеры окна, заданные в пикселях. Заголовок этого метода имеет следующий вид:

```
void setSize(int width, int height)
```

В данном примере задается ширина окна, равная 275 пикселям, и высота, равная 100 пикселям.

По умолчанию при закрытии окна верхнего уровня (для этого предназначена кнопка в верхнем правом углу) оно удаляется с экрана, но приложение не завершает свою работу. В большинстве случаев такое поведение не устраивает разработчиков, хотя для некоторых приложений оно вполне уместно. Обычно желательно, чтобы при закрытии окна верхнего уровня приложение завершало работу. Сделать это можно несколькими способами. Самый простой из них — вызов метода `setDefaultCloseOperation()`. Именно такой способ и применен в данной программе.

```
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

После выполнения указанного метода закрытие окна приведет к завершению всего приложения. Заголовок метода `setDefaultCloseOperation()` приведен ниже.

```
void setDefaultCloseOperation(int what)
```

Значение `value`, передаваемое в качестве параметра, определяет, что произойдет при закрытии окна.

Помимо `JFrame.EXIT_ON_CLOSE`, методу можно передавать и другие значения, описанные ниже.

```
JFrame.DISPOSE_ON_CLOSE  
JFrame.HIDE_ON_CLOSE  
JFrame.DO_NOTHING_ON_CLOSE
```

Имена констант отражают их назначение. Все они определены в составе интерфейса `WindowConstants`, содержащегося в пакете `javax.swing`. Этот интерфейс реализуется классом `JFrame`.

Следующая строка кода создает Swing-компонент `JLabel`:

```
JLabel jlab = new JLabel(" Swing powers the modern Java GUI.");
```

`JLabel` — самый простой в использовании компонент Swing, поскольку он не предполагает обработку действий пользователя. Он лишь отображает информацию: текст, изображение или оба типа данных. Метка, созданная в данной программе, содержит только текст. Строка текста передается конструктору компонента.

Следующая строка кода включает метку в составе панели содержимого фрейма:

```
jfrm.getContentPane().add(jlab);
```

Как было сказано ранее, каждый контейнер верхнего уровня включает панель содержимого, в которую помещаются компоненты. Таким образом, чтобы включить компонент в состав фрейма, вам надо добавить его к панели содер-

жимого. Ссылку на данную панель возвращает метод `getContentPane()`. С помощью этой ссылки можно вызвать метод `add()`, предназначенный для добавления компонентов. Заголовки методов `getContentPane()` и `add()` приведены ниже.

```
Container getContentPane()  
Component add(Component comp)
```

Метод `add()` унаследован `JFrame` от AWT-класса `Container`. С появлением JDK 5 исчезла необходимость в вызове метода `getContentPane()` при включении компонента в `JFrame`. Вместо этого вы можете вызвать метод `add()`, непосредственно обращаясь к объекту `JFrame`. Компонент будет автоматически добавлен к панели содержимого. В программе `SwingDemo` присутствует строка

```
jfrm.getContentPane().add(jlab);
```

В JDK 5 и более поздних версиях ее можно заменить строкой, приведенной ниже.

```
jfrm.add(jlab);
```

Поскольку на момент написания данной книги все еще широко использовались ранние версии Java, в приводимых в ней примерах будет производиться явное обращение к панели содержимого. Сделано это для того, чтобы все читатели могли использовать коды программ, не изменяя их. Кроме того, благодаря вызову `getContentPane()` становится ясно, с какой панелью производятся действия. Если же в вашем распоряжении имеется JDK 5 или более новые версии данного пакета, вы можете несколько сократить коды программ.

Последнее выражение из конструктора `SwingDemo` обеспечивает отображение окна.

```
jfrm.setVisible(true);
```

Метод `setVisible()` унаследован от AWT-класса `Component`. Его заголовок имеет следующий вид:

```
void setVisible(boolean flag)
```

Если значение параметра `flag` равно `true`, окно отображается на экране. В противном случае оно остается скрытым. По умолчанию фрейм невидим, поэтому для его отображения надо вызвать метод `setVisible(true)`.

В методе `main()` создается объект `SwingDemo`, в результате чего окно и метка отображаются на экране. Заметьте, что конструктор `SwingDemo` вызывается с помощью следующего фрагмента кода:

```
SwingUtilities.invokeLater(new Runnable() {  
    public void run() {  
        new SwingDemo();  
    }  
});
```

Этот фрагмент кода создает объект `SwingDemo` не в основном потоке приложения, а в *потоке обработки событий*. Такое решение принимается по следующим причинам. Подавляющее большинство Swing-программ управляется событиями. В частности, события возникают при взаимодействии пользователя с компонентом. Событие передается приложению путем вызова обработчика, определенного в программе, однако этот обработчик выполняется в специальном потоке обработки событий, формируемом Swing, а не в главном потоке приложения. Таким образом, несмотря на то, что обработчик определен в вашей программе, поток, в котором он выполняется, создан другими средствами. Для того чтобы избежать проблем (связанных, например, с попытками двух потоков одновременно обновить один и тот же компонент), все элементы пользовательского интерфейса Swing должны создаваться не в основном потоке приложения, а в потоке обработки событий. Однако метод `main()` выполняется в основном потоке. Следовательно, в нем нельзя непосредственно создавать объект `SwingDemo`. Следует сначала создать объект `Runnable`, выполняемый в потоке обработки событий, а затем предоставить данному объекту возможность создать окно интерфейса программы.

Для того чтобы код, поддерживающий интерфейс, мог быть создан в потоке обработки событий, надо использовать один из двух методов, определенных в классе `SwingUtilities`: `invokeLater()` и `invokeAndWait()`. Их загрузки имеют следующий вид:

```
static void invokeLater(Runnable obj)  
static void invokeAndWait(Runnable obj)  
    throws InterruptedException, InvocationTargetException
```

где `obj` — это объект `Runnable`, метод `run()` которого вызывается в потоке обработки событий. Различие между двумя методами, представленными выше, в том, что `invokeLater()` сразу же возвращает управление вызывающему методу, а `invokeAndWait()` ожидает завершения метода `obj.run()`. Эти методы можно использовать для вызова метода, создающего интерфейс Swing-приложения, а также в тех случаях, когда вам надо изменить состояние интерфейса из метода, выполняющегося за пределами потока обработки событий. Обычно для этой цели используется метод `invokeLater()`. Такое решение было принято в предыдущем примере. Однако при создании интерфейса апплета лучше

использовать `invokeAndWait()`. (Вопросы написания апплетов, использующих средства Swing, будут рассмотрены в модуле 10.)

Необходимо отметить еще одну особенность рассмотренной выше программы: она не реагирует на действия пользователя, поскольку в компоненте `JLabel` не предусмотрены соответствующие средства. Другими словами, `JLabel` не генерирует события. По этой причине в программу не были включены обработчики событий. Однако другие компоненты генерируют события, на которые программа должна выполнять ответные действия. С примером обработки событий вы вскоре познакомитесь.



### Вопросы для текущего контроля

1. С помощью какого класса Swing создается метка?
2. Какой пакет должен быть включен во все Swing-программы?
3. Любой код, создающий или модифицирующий элементы пользовательского интерфейса, должен выполняться в потоке \_\_\_\_\_.
4. Предположим, что вы добавляете компонент к `JFrame`. Необходимо ли вызывать метод `getContentPane()`, если вы используете JDK 5 или более позднюю версию?

### Спросим у опытного программиста

**Вопрос.** Я видел код Swing-программы, в которой не использовался ни метод `invokeLater()`, ни `invokeAndWait()`. Действительно ли эти методы необходимы?

**Ответ.** На ранних этапах развития Swing считалось допустимым отображать элементы интерфейса в основном потоке приложения. Поэтому Swing-приложения, созданные достаточно давно, при формировании интерфейса не вызывали `invokeLater()` или `invokeAndWait()`. Однако со временем позиция Sun изменилась.

- 
1. `JLabel`.
  2. `javax.swing`.
  3. Обработки событий.
  4. Нет.

Теперь специалисты данной компании утверждают, что некорректно инициализировать интерфейс в основном потоке обработки событий, поскольку это может иногда приводить к возникновению проблем. Существует много Swing-приложений, которые инициализируют элементы интерфейса в основном потоке и, тем не менее, работают вполне приемлемо. (Такое решение можно было бы принять и в нашей программе ввиду ее исключительной простоты.) Однако, поскольку рекомендуется создавать интерфейс в потоке обработки событий, именно такое решение реализовано в программах, вошедших в данную книгу в качестве примеров.

**ВАЖНО!**

## **1.6. Поддержка событий**

Предыдущий пример продемонстрировал основы Swing-программы, но в ней не нашел отражение важнейший вопрос: поддержка событий. Поскольку `JLabel` не принимает данных от пользователя, он не генерирует событий, поэтому при работе с данным компонентом обработчик событий не нужен. Однако другие управляющие элементы Swing реагируют на действия пользователя, и генерируемые ими события должны быть обработаны. Например, события генерируются по щелчку мышью на кнопке, нажатию клавиши на клавиатуре или при выборе элемента списка. Существуют события, непосредственно не связанные с действиями пользователей. Например, событие генерируется по истечении интервала времени, установленного для таймера. Независимо от причины того или иного события, средства, обрабатывающие их, являются важной частью любого Swing-приложения.

В Swing используется тот же механизм обработки событий, что и в AWT. Он носит название *модель делегирования событий*. Этот механизм достаточно прост. *Источник* генерирует событие, которое передается одному или нескольким *обработчикам*. В рамках данной схемы обработчики лишь ожидают возникновения события. При возникновении события они обрабатывают его и возвращают управление. Преимущество такого подхода в том, что логика обработки событий отделена от логики пользовательского интерфейса, генерирующего эти события. Элемент интерфейса "делегирует" обработку события отдельному фрагменту кода. В модели делегирования событий обработчик, чтобы получать оповещения о событиях, должен быть зарегистрирован в источнике.

Рассмотрим подробнее события, их источники и обработчики.

## События

Согласно модели делегирования, событие является объектом, описывающим изменения состояния источника. Событие может быть следствием действий пользователя с элементом графического интерфейса или сгенерировано программными средствами. Супреклассом всех событий является `java.util.EventObject`. Многие события объявлены в пакете `java.awt.event`, но некоторые содержатся в `javax.swing.event`.

## Источники событий

Источник события — это объект, сгенерировавший его. Сгенерировав событие, источник должен передать его всем зарегистрированным обработчикам. Следовательно, чтобы обработчик получил событие, он должен быть зарегистрирован в источнике. Регистрация осуществляется путем вызова метода `addТипListener()`, принадлежащего источнику. Для каждого типа события определен собственный метод регистрации. Заголовок метода имеет вид, подобный представленному ниже.

```
public void addТипListener(ТипListener el)
```

где *тип* — это имя события, а параметр *el* представляет собой ссылку на обработчик события. Например, метод, регистрирующий обработчик событий клавиатуры, называется `addKeyListener()`. Для регистрации событий, связанных с перемещением мыши, используется метод `addMouseMotionListener()`. О возникшем событии оповещаются все обработчики.

Источник должен также предоставлять метод, позволяющий отменить регистрацию обработчика событий определенного типа. Этот метод имеет заголовок, представленный в следующей форме:

```
public void removeТипListener(ТипListener el)
```

где *тип* — это имя события, а параметр *el* — ссылка на обработчик. Например, для того, чтобы удалить обработчик событий клавиатуры, надо вызвать метод `removeKeyListener()`.

Методы, добавляющие или удаляющие обработчики, принадлежат объектам-источникам событий. Например, класс `JButton` содержит методы для регистрации отмены обработчика `ActionListener`. Будучи зарегистрированным, этот обработчик получает оповещение о действиях с кнопкой.

## Обработчики событий

Обработчик — это объект, оповещаемый о возникновении события. К нему предъявляются два основных требования. Во-первых, чтобы получать оповещение о конкретном типе событий, он должен быть зарегистрирован в одном или нескольких источниках. Во-вторых, он должен реализовывать метод, предназначенный для обработки события.

Методы, позволяющие получать и обрабатывать события, определены в интерфейсах, содержащихся в пакетах `java.awt.event`, `javax.swing.event` и `java.beans`. Например, в интерфейсе `ActionListener` объявлен метод, который вызывается тогда, когда пользователь щелкает на кнопке или выполняет другое действие, затрагивающее компонент. Это событие может быть обработано любым объектом, при условии, что он реализует интерфейс `ActionListener`.

Общие правила, которые следует учитывать при обработке событий, таковы. Обработчик должен выполнять свою задачу быстро и возвращать управление. По возможности он не должен осуществлять сложные операции, поскольку это может замедлить работу всего приложения. Если же необходимы действия, занимающие длительное время, для их выполнения следует создавать отдельный поток.

## Классы событий и интерфейсы обработчиков

Классы, представляющие события, лежат в основе механизма обработки события Java. Эти классы формируют иерархическую структуру, на вершине которой находится класс `EventObject`, принадлежащий пакету `java.util`. Он является суперклассом для всех событий. Класс `AWTEvent`, объявленный в пакете `java.awt`, представляет собой подкласс `EventObject`. Он, в свою очередь, является родительским классом для всех событий AWT, используемых в модели делегирования событий. Swing использует события AWT и, кроме того, определяет несколько дополнительных событий. Как упоминалось ранее, соответствующие классы определены в пакете `javax.swing.event`.

В табл. 1.1 описаны некоторые классы событий и соответствующие им интерфейсы обработчиков, определенные в пакете `java.awt.event`.

В табл. 1.2 приведено несколько классов событий и интерфейсов обработчиков из пакета `javax.swing.event`. Конкретные классы и интерфейсы будут подробно описываться по мере того, как они будут встречаться в данной книге.



Таблица 1.1. Некоторые классы событий из пакета `java.awt.event`

| Класс события                | Описание                                                                                                                                                                    | Обработчик                                                    |
|------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------|
| <code>ActionEvent</code>     | Генерируется при выполнении действий с интерфейсным элементом, например по щелчку на кнопке                                                                                 | <code>ActionListener</code>                                   |
| <code>AdjustmentEvent</code> | Генерируется при выполнении действий с полосой прокрутки                                                                                                                    | <code>AdjustmentListener</code>                               |
| <code>FocusEvent</code>      | Генерируется тогда, когда компонент получает или теряет фокус ввода                                                                                                         | <code>FocusListener</code>                                    |
| <code>ItemEvent</code>       | Генерируется при выборе или отмене выбора элемента, например, по щелчку на флажке опций                                                                                     | <code>ItemListener</code>                                     |
| <code>KeyEvent</code>        | Генерируется при вводе данных с клавиатуры                                                                                                                                  | <code>KeyListener</code>                                      |
| <code>MouseEvent</code>      | Генерируется при перемещении или перетаскивании мыши, нажатии или отпуске клавиши, а также при помещении курсора мыши на компонент или выводе курсора за пределы компонента | <code>MouseListener</code> и <code>MouseMotionListener</code> |
| <code>MouseWheelEvent</code> | Генерируется при движении колесика мыши                                                                                                                                     | <code>MouseWheelListener</code>                               |
| <code>WindowEvent</code>     | Генерируется при активизации, деактивизации, закрытии, окна, сворачивании его в пиктограмму и разворачивании из пиктограммы                                                 | <code>WindowListener</code>                                   |

Таблица 1.2. Некоторые классы событий из пакета `javax.swing.event`

| Класс события               | Описание                                                                | Обработчик                     |
|-----------------------------|-------------------------------------------------------------------------|--------------------------------|
| <code>AncestorEvent</code>  | Генерируется при добавлении, перемещении или удалении предка компонента | <code>AncestorListener</code>  |
| <code>CaretEvent</code>     | Генерируется при изменении позиции курсора в текстовом компоненте       | <code>CaretListener</code>     |
| <code>ChangeEvent</code>    | Генерируется при изменении состояния компонента                         | <code>ChangeListener</code>    |
| <code>HyperlinkEvent</code> | Генерируется при действиях с гипертекстовой ссылкой                     | <code>HyperlinkListener</code> |
| <code>ListDataEvent</code>  | Генерируется при изменении содержимого списка                           | <code>ListDataListener</code>  |

Окончание табл. 1.2.

| Класс события      | Описание                                                 | Обработчик            |
|--------------------|----------------------------------------------------------|-----------------------|
| ListSelectionEvent | Генерируется при выборе или отмене выбора пунктов списка | ListSelectionListener |
| MenuEvent          | Генерируется при выборе или отмене выбора пунктов меню   | MenuListener          |
| TableModelEvent    | Генерируется при изменении модели таблицы                | TableModelListener    |
| TreeExpansionEvent | Генерируется при разворачивании или сворачивании дерева  | TreeExpansionListener |
| TreeModelEvent     | Генерируется при изменении модели дерева                 | TreeModelListener     |
| TreeSelectionEvent | Генерируется при выборе узла дерева                      | TreeSelectionListener |

## Классы адаптеров

Несмотря на то что большинство интерфейсов обработчиков событий реализовать нетрудно, Java предоставляет набор классов адаптеров, в которых уже определены методы, объявленные в интерфейсах обработчиков. Классы адаптеров удобны тогда, когда вы собираетесь использовать лишь некоторые из событий, предусмотренных в интерфейсе. Новый класс, выполняющий функции обработчика, можно создать как подкласс класса адаптера, переопределив в нем интересующие вас методы. Поскольку при использовании адаптера отпадает необходимость определять все методы, объявленные в интерфейсе обработчика, уменьшается объем кода и трудозатраты по его написанию. Адаптеры часто применяются при реализации неименованных внутренних классов, что также упрощает код программы.

Для некоторых интерфейсов обработчиков адаптеры отсутствуют. Например, для `ActionEvent` адаптер не создан, поскольку в данном интерфейсе объявлен лишь один метод. Адаптеры создаются только для тех интерфейсов, в которых предусмотрено более одного метода. Например, в составе `MouseMotionListener` присутствуют два метода: `mouseDragged()` и `mouseMoved()`. Реализации этих методов, не выполняющие никаких действий, определены в классе `MouseMotionAdapter`. Если вас интересуют только события, связанные с перетаскиванием мыши, вам следует создать обработчик как подкласс `MouseMotionAdapter` и реализовать в нем метод `mouseDragged()`. Пустой метод `mouseMoved()`, унаследованный от суперкласса, будет обрабатывать события перемещения мыши.

Ниже перечислено несколько классов адаптеров. Большинство из них определено в пакете `java.awt.event`, однако класс `MouseInputAdapter` содержится в пакете `javax.swing.event`.

| Класс адаптера                  | Реализуемый интерфейс                                         |
|---------------------------------|---------------------------------------------------------------|
| <code>FocusAdapter</code>       | <code>FocusListener</code>                                    |
| <code>KeyAdapter</code>         | <code>KeyListener</code>                                      |
| <code>MouseAdapter</code>       | <code>MouseListener</code>                                    |
| <code>MouseMotionAdapter</code> | <code>MouseMotionListener</code>                              |
| <code>MouseInputAdapter</code>  | <code>MouseListener</code> и <code>MouseMotionListener</code> |
| <code>WindowAdapter</code>      | <code>WindowListener</code>                                   |

В данном модуле адаптеры использоваться не будут. С примерами их практического применения вы познакомитесь далее в этой книге.



### Вопросы для текущего контроля

1. В модели делегирования определены источники событий и \_\_\_\_\_.
2. Какой интерфейс обработчика ориентирован на события, связанные с действиями пользователя с интерфейсными элементами?
3. Зачем нужны классы адаптеров?

**ВАЖНО!**

## 1.7. Использование кнопок

Кнопка является одним из самых простых управляющих элементов Swing. В то же время она используется чаще других компонентов. Кнопка представляет собой экземпляр класса `JButton`. Этот класс является потомком абстрактного класса `AbstractButton`, в котором определены функции, общие для всех кнопок. На кнопке может отображаться текст, изображение или информация обоих типов. В данном модуле используются только кнопки с надписями в виде строки текста. С другими типами кнопок вы познакомитесь в модуле 2.

1. Обработчики событий.
2. `ActionListener`.
3. Классы адаптеров упрощают реализацию обработчиков событий, предоставляя пустые методы, объявленные в интерфейсе. Реализовывать приходится только те методы, которые необходимы для решения конкретной задачи.

Класс `JButton` содержит три конструктора. Один из них имеет следующий вид:

```
JButton(String msg)
```

Параметр `msg` определяет строку, которая должна отображаться на кнопке.

По щелчку на кнопке генерируется событие `ActionEvent`. Для регистрации и отключения обработчиков данного события `JButton` предоставляет следующие методы (они унаследованы от класса `AbstractButton`):

```
void addActionListener(ActionListener al)
void removeActionListener(ActionListener al)
```

Здесь параметр `al` задает объект, который будет оповещаться о возникновении событий. Объект должен представлять собой экземпляр класса, реализующего интерфейс `ActionListener`.

В интерфейсе `ActionListener` определен только один метод: `actionPerformed()`. Объявление этого метода выглядит следующим образом:

```
void actionPerformed(ActionEvent ae)
```

Данный метод вызывается по щелчку на кнопке, т.е. он занимается обработкой событий, связанных с действиями пользователя с кнопкой. Реализуя метод `actionPerformed()`, необходимо позаботиться о том, чтобы он быстро выполнял свою задачу и возвращал управление. Как было сказано ранее, обработчики не должны выполнять длительных операций, поскольку это может привести к замедлению работы всего приложения. Если же обработка события предполагает действия, требующие длительного времени, их надо выполнять в потоке, специально создаваемом для этой цели.

Объект `ActionEvent`, передаваемый методу `actionPerformed()`, позволяет получить важную информацию, связанную с событием данного типа. В данном модуле мы будем использовать строку команды действия, связанной с кнопкой. По умолчанию в качестве команды действия принимается строка, отображаемая на кнопке. Для получения команды действия надо вызвать метод `getActionCommand()`, принадлежащий объекту события. Он объявляется следующим образом:

```
String getActionCommand()
```

Команда действия идентифицирует кнопку. Таким образом, если в приложении имеется несколько кнопок, команда действия позволяет достаточно просто определить, какая из них явилась источником события.

Ниже приведен текст программы, демонстрирующий использование кнопки, реагирующей на действия пользователя. На рис. 1.2 показан внешний вид окна, отображаемого данной программой.

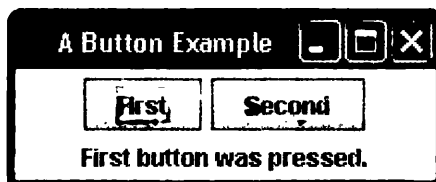


*Рис. 1.2. Выходные данные, которые генерируются программой, демонстрирующей использование компонента JButton*

```
// Пример, демонстрирующий работу с кнопками
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class ButtonDemo implements ActionListener {

    JLabel jlab;

    ButtonDemo() {

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("A Button Example");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout);

        // Установка исходного размера фрейма.
        jfrm.setSize(220, 90);

        // Завершение программы при закрытии пользователем окна.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание двух кнопок.
        JButton jbtnFirst = new JButton("First");
        JButton jbtnSecond = new JButton("Second");

        // Связывание с кнопками обработчиков событий действия.
        jbtnFirst.addActionListener(this);
    }
}
```

```
jbtnSecond.addActionListener(this);

// Добавление кнопок к панели содержимого.
jfrm.getContentPane().add(jbtnFirst);
jfrm.getContentPane().add(jbtnSecond);

// Создание текстовой метки.
jlab = new JLabel("Press a button.");

// Включение метки в состав фрейма.
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

// Обработка событий ActionEvent, связанных с кнопками.
public void actionPerformed(ActionEvent ae) {

    // Использование команды действия для идентификации кнопки
    if(ae.getActionCommand().equals("First"))
        jlab.setText("First button was pressed.");
    else
        jlab.setText("Second button was pressed. ");
}

public static void main(String args[]) {
    // Создание фрейма в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new ButtonDemo();
        }
    });
}
```

Рассмотрим текст программы и обратим внимание на новые для нас решения. В первую очередь заметьте, что в этой программе импортируются пакеты `java.awt` и `java.awt.event`. Пакет `java.awt` необходим, потому что в нем содержится класс диспетчера компоновки `FlowLayout`, используемый

для размещения компонентов во фрейме. (Более подробно диспетчеры компоновки будут рассмотрены далее в этой главе.) Пакет `java.awt.event` также необходим, поскольку в нем определены интерфейс `ActionListener` и класс `ActionEvent`.

Далее в программе объявляется класс `ButtonDemo`. Заметьте, что он реализует интерфейс `ActionListener`. Это означает, что объекты `ButtonDemo` могут быть использованы для обработки событий, соответствующих действиям пользователей с компонентами. Далее объявляется ссылка на объект `JLabel`. Она будет использована в методе `actionPerformed()` для отображения информации о том, на какой кнопке щелкнул пользователь.

Конструктор `ButtonDemo` начинается с создания элемента `JFrame`, на который ссылается переменная `jfrm`. Затем в конструкторе устанавливается диспетчер компоновки `FlowLayout`, управляющий размещением элементов в панели содержимого. По умолчанию с панелью содержимого связан диспетчер `BorderLayout`. Он располагает компоненты в пяти областях: по центру и по краям окна. (Следовательно, в первом примере данного модуля диспетчер `BorderLayout`, принимаемый по умолчанию, помещает метку в центральную область.) Для многих примеров из первой части данной книги больше подходит диспетчер `FlowLayout`. Он размещает компоненты "в строку", располагая каждый следующий элемент справа от предыдущего. Когда очередная строка заполняется, диспетчер компоновки формирует следующую строку. Несмотря на то что данная схема обеспечивает низкий уровень контроля за размещением элементов, использовать ее очень просто. Обратите внимание, что при изменении размеров фрейма размещение компонентов может измениться.

Для связывания диспетчера компоновки с панелью содержимого используется следующее выражение:

```
jfrm.getContentPane().setLayout(new FlowLayout());
```

Начиная с JDK 5, необходимости в явном вызове метода `getContentPane()` уже нет. Обращаться к методу `setLayout()` позволяет ссылка на элемент `JFrame`. Таким образом, в JDK 5 и более поздних версиях предыдущую строку можно записать несколько короче:

```
jfrm.setLayout(new FlowLayout());
```

В примерах, используемых в данной книге, будет явным образом вызываться метод `getContentPane()`. Причины такого решения были описаны ранее при обсуждении механизма добавления компонентов к панели содержимого. Однако, работая с JDK 5 или с одной из последующих версий, вы можете при желании несколько сократить код программы.

После установки размера окна и определения операции завершения программы в конструкторе `ButtonDemo()` создаются две кнопки.

```
JButton jbtnFirst = new JButton("First");  
JButton jbtnSecond = new JButton("Second");
```

На первой кнопке отображается надпись `First`, а на второй — `Second`.

Далее с кнопками связывается обработчик событий, в роли которого выступает экземпляр класса `ButtonDemo`; ссылка на него передается с помощью ключевого слова `this`. Соответствующие строки кода приведены ниже.

```
jbtnFirst.addActionListener(this);  
jbtnSecond.addActionListener(this);
```

В результате выполнения данных выражений объект, создавший кнопки, будет получать оповещение о действиях с ними.

Кнопки включаются в состав панели содержимого.

```
jfrm.getContentPane().add(jbtnFirst);  
jfrm.getContentPane().add(jbtnSecond);
```

Как вы, наверное, помните, вызов `getContentPane()` необходим только при использовании версий Java, предшествующих 1.5 (JDK 5). Здесь он используется для того, чтобы дать более полное представление о выполняемых действиях.

По каждому щелчку на кнопке она генерирует событие, о котором оповещаются зарегистрированные обработчики. Оповещение происходит посредством вызова метода `actionPerformed()`. Объект `ActionEvent`, представляющий событие мыши, передается данному методу в качестве параметра. В программе `ButtonDemo` метод `actionPerformed()` имеет следующий вид:

```
public void actionPerformed(ActionEvent ae) {  
    if(ae.getActionCommand().equals("First"))  
        jlab.setText("First button was pressed.");  
    else  
        jlab.setText("Second button was pressed. ");  
}
```

Для передачи события используется параметр `ae`. В теле метода извлекается команда действия, которая соответствует кнопке, сгенерировавшей событие. Для получения команды действия вызывается метод `getActionCommand()`. (По умолчанию команда действия совпадает с текстом, отображаемым на кнопке.) В зависимости от содержимого строки, представляющей команду действия, устанавливается текст метки.



Необходимо также учесть, что метод `actionPerformed()` вызывается в потоке обработки событий. Он должен как можно быстрее завершаться, чтобы не замедлять работу приложения.



### Вопросы для текущего контроля

1. С помощью какого класса создается кнопка Swing?
2. По умолчанию командная строка, связанная с кнопкой, принимается равной строке текста, отображаемой на этой кнопке. Да или нет?

### Проект 1.1. Простой секундомер

`StopWatch.java` Несмотря на то что вам знакомы только два управляющих элемента Swing, `JLabel` и  `JButton`, этого достаточно, чтобы создать приложение, выполняющее полезные функции, а именно секундомер. В окне секундомера отображаются две кнопки и одна метка. На кнопках, используемых для запуска и остановки секундомера, выводятся надписи `Start` и `Stop`. Метка отображает время, прошедшее с запуска секундомера до остановки. Несмотря на предельную простоту, данный проект демонстрирует, насколько удобны средства Swing для создания пользовательских интерфейсов.

### Последовательность действий

1. Создайте файл `StopWatch.java` и введите приведенные ниже строки комментариев и выражения `import`.

```
// Проект 1.1. Простой секундомер.
```

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

Заметьте, что в числе пакетов импортируется `java.util`. Он необходим, поскольку в нем содержится класс `Calendar`, используемый для получения текущего системного времени.

- 
1. `JButton`.
  2. Да.

**2. Создайте класс Stopwatch, начав его следующим кодом:**

```
class Stopwatch implements ActionListener {  
  
    JLabel jlab;  
    long start; // Содержит время запуска в миллисекундах.
```

Как сказано в комментариях, переменная `start` используется для хранения времени запуска секундомера, выраженного в миллисекундах. Это значение будет вычтено из времени остановки, чтобы получить значение времени, в течение которого производился отсчет.

**3. Начните конструктор Stopwatch() следующими строками кода:**

```
StopWatch() {  
  
    // Создание нового контейнера JFrame.  
    JFrame jfrm = new JFrame("A Simple Stopwatch");  
  
    // Установка диспетчера компоновки FlowLayout.  
    jfrm.getContentPane().setLayout(new FlowLayout());  
  
    // Установка исходного размера фрейма.  
    jfrm.setSize(230, 90);  
  
    // Завершение программы при закрытии пользователем окна.  
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Данные выражения аналогичны тем, которые приводились в предыдущих примерах, поэтому знакомы вам.

**4. Введите приведенный ниже код. Он создает кнопки Start и Stop, связывает с кнопками обработчики событий, с затем добавляет кнопки к панели содержимого.**

```
// Создание двух кнопок  
JButton jbtnStart = new JButton("Start");  
JButton jbtnStop = new JButton("Stop");  
  
// Связывание с кнопками обработчиков событий.  
jbtnStart.addActionListener(this);  
jbtnStop.addActionListener(this);  
  
// Включение кнопок в состав панели содержимого.  
jfrm.getContentPane().add(jbtnStart);  
jfrm.getContentPane().add(jbtnStop);
```

5. Создайте метку и добавьте ее к панели содержимого. Эти действия выполняются с помощью приведенных ниже выражений.1

```
// Создание текстовой метки.  
jlab = new JLabel("Press Start to begin timing.");  
  
// Добавление метки к фрейму.  
jfrm.getContentPane().add(jlab);
```

Эта метка отображает состояние секундомера и измеренное время.

6. Перед тем как завершить код конструктора `StopWatch()`, разрешите отображение фрейма.

```
// Отображение фрейма.  
jfrm.setVisible(true);  
}
```

7. Добавьте метод `actionPerformed()`, код которого показан ниже.

```
// Поддержка событий, связанных с кнопкой  
public void actionPerformed(ActionEvent ae) {  
    Calendar cal = Calendar.getInstance(); // Получение текущего  
                                           // системного времени.  
  
    if (ae.getActionCommand().equals("Start")) {  
        // Сохранение времени запуска.  
        start = cal.getTimeInMillis();  
        jlab.setText("Stopwatch is Running...");  
    }  
    else  
        // Вычисление времени, прошедшего от запуска до остановки.  
        jlab.setText("Elapsed time is"  
            + (double) (cal.getTimeInMillis() - start) / 1000);  
}
```

Заметьте, что объект `Calendar`, ссылка на который хранится в переменной `cal`, создается и инициализируется текущим системным временем посредством вызова статического метода `getInstance()`, определенного в классе `Calendar`. Таким образом, при каждом вызове `actionPerformed()` переменная `cal` будет содержать текущее время.

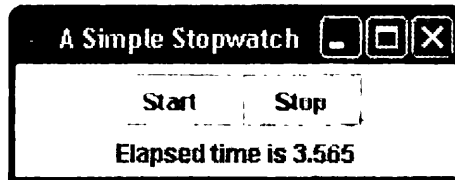
Как вы знаете, по умолчанию команда действия совпадает с текстом, отображаемым на кнопке. Таким образом, для кнопки `Start` команда действия будет представлять собой строку `"Start"`. По щелчку на кнопке `Start` определяется текущее время в миллисекундах (для этого вызывается метод `getTimeInMillis()`) и сохраняется в переменной `start`.

При активизации кнопки **Stop** снова определяется текущее время, и из него вычитается время запуска. Полученное значение приводится к типу `double` и делится на 1000. Таким образом, измеренное время выражается в секундах.

**8.** Завершите код программы методом `main()`.

```
public static void main(String args[]) {
    // Создание фрейма в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new Stopwatch();
        }
    });
}
```

**9.** Полностью код программы приведен ниже. Окно, отображаемое в процессе ее работы, выглядит следующим образом:



// Проект 1.1. Простой секундомер.

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

class Stopwatch implements ActionListener {

    JLabel jlab;
    long start; // Содержит время запуска в миллисекундах.

    Stopwatch() {

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("A Simple Stopwatch");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());
    }
}
```

```
// Установка исходного размера фрейма.
jfrm.setSize(230, 90);

// Завершение программы при закрытии пользователем окна.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Создание двух кнопок.
JButton jbtnStart = new JButton("Start");
JButton jbtnStop = new JButton("Stop");

// Связывание с кнопками обработчиков событий.
jbtnStart.addActionListener(this);
jbtnStop.addActionListener(this);

// Включение кнопок в состав панели содержимого.
jfrm.getContentPane().add(jbtnStart);
jfrm.getContentPane().add(jbtnStop);

// Создание текстовой метки.
JLabel jlab = new JLabel("Press Start to begin timing.");

// Добавление метки к фрейму.
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

// Поддержка событий, связанных с кнопкой.
public void actionPerformed(ActionEvent ae) {
    Calendar cal = Calendar.getInstance(); // Получение текущего
                                           // системного времени.

    if(ae.getActionCommand().equals("Start")) {
        // Сохранение времени запуска.
        start = cal.getTimeInMillis();
        jlab.setText("Stopwatch is Running...");
    }
    else

        // Вычисление времени, прошедшего от запуска до остановки.
```

```
jlab.setText("Elapsed time is "
    + (double) (cal.getTimeInMillis() - start)/1000);
}

public static void main(String args[]) {
    // Создание фрейма в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new Stopwatch();
        }
    });
}
```

## Спросим у опытного программиста

**Вопрос.** При использовании JDK 5 или более поздних версий вызывать метод `getContentPane()` не нужно. В каких случаях действует это правило?

**Ответ.** Как вы знаете, при включении компонента в контейнер верхнего уровня, например `JFrame`, он реально помещается на панель содержимого. До появления JDK 5 необходимо было получать ссылку на панель содержимого посредством метода `getContentPane()` и лишь затем вызывать метод `add()`. Начиная с JDK 5, вызывать `getContentPane()` явным образом нет необходимости, поскольку обращение к методу `add()` контейнера верхнего уровня, например `JFrame`, вызывает автоматическое перенаправление к панели содержимого.

Эти изменения затронули еще два метода: `remove()`, удаляющий компонент из контейнера, и `setLayout()`, используемый для связывания с контейнером диспетчера компоновки. В версиях, предшествовавших JDK 5, при работе с контейнерами верхнего уровня обращение к этим методам также осуществлялось посредством `getContentPane()`. Теперь их можно непосредственно вызывать из контейнера верхнего уровня, и обращение будет переадресовано к панели содержимого.

Как уже говорилось ранее, в данной книге метод `getContentPane()` вызывается явно по двум причинам. Во-первых, для того, чтобы предоставить каждому пользователю рабочий код, независимо от того, какая система установлена на его компьютере. Во-вторых, такой подход делает более понятными выполняемые действия. Имея в своем распоряжении JDK 5 или более поздние версии данного пакета, можно отказаться от вызова метода `getContentPane()`.

**ВАЖНО!****1.8. Компонент JTextField**

К числу широко используемых компонентов относится также `JTextField`. Он позволяет пользователю вводить строку текста. `JTextField` является подклассом абстрактного класса `JTextComponent`, который выступает в роли суперкласса не только для `JTextField`, но и для всех текстовых компонентов. Подробно о таких компонентах речь пойдет в модуле 7; здесь мы приведем лишь самые общие сведения о `JTextField`, так как он удобен для ввода текста.

В классе `JTextField` определено несколько конструкторов. Один из них выглядит следующим образом:

```
JTextField(int cols)
```

где параметр `cols` определяет ширину текстового поля, выраженную в столбцах. Длина вводимой строки не ограничивается шириной поля, отображаемого на экране.

Закончив ввод текста в поле, пользователь нажимает клавишу `<Enter>`, в результате чего генерируется событие `ActionEvent`. Класс `JTextField` предоставляет разработчику методы `addActionListener()` и `removeActionListener()`. Для обработки событий необходимо реализовать метод `actionPerformed()`, объявленный в интерфейсе `ActionListener`. Обработка событий поля редактирования осуществляется так же, как и обработка событий кнопки, о которой шла речь ранее.

Чтобы получить строку, отображаемую в поле редактирования, надо обратиться к экземпляру `JTextField` и вызвать метод `getText()`. Объявление этого метода приведено ниже.

```
String getText()
```

Задать текст для компонента `JTextField` позволяет метод `setText()`, который имеет следующий вид:

```
void setText(String text)
```

Строка передается компоненту посредством параметра `text`.

Ниже приведен код программы, демонстрирующей использование компонента `JTextField`. В результате работы программы создается поле редактирования, ширина которого позволяет отобразить в нем до десяти символов. После нажатия клавиши `<Enter>` текущее содержимое поля редактирования отображается в метке `JLabel`. Окно, создаваемое при работе программы, показано на рис. 1.3.

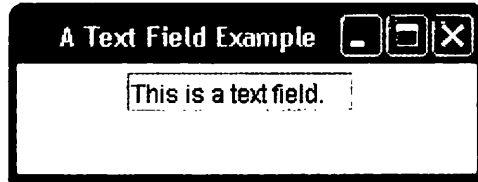


*Рис. 13. Данные, отображаемые программой, демонстрирующей работу компонента `JTextField`*

```
// Использование поля редактирования.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class JTextFieldDemo implements ActionListener {

    JTextField jtf;
    JLabel jlab;

    JTextFieldDemo() {

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("A Text Field Example");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(240, 90);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание поля редактирования, отображающего
        // до десяти символов.
        jtf = new JTextField(10);

        // Связывание с полем редактирования обработчика
        // событий действия.
```



```
jtf.addActionListener(this);

// Включение поля редактирования в состав панели содержимого.
jfrm.getContentPane().add(jtf);

// Создание пустой метки.
jlab = new JLabel("");

// Добавление метки к фрейму.
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

// Поддержка событий действий.
// Событие действия генерируется, если пользователь,
// работая с полем редактирования, нажимает клавишу <Enter>.
public void actionPerformed(ActionEvent ae) {
    // Получение текста и отображение его с помощью метки.
    jlab.setText("Current contents: " + jtf.getText());
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new JTextFieldDemo();
        }
    });
}
```

Большая часть кода программы уже знакома вам, однако обратите внимание на следующую строку (она присутствует в методе `actionPerformed()`):

```
jlab.setText("Current contents: " + jtf.getText());
```

Как было сказано ранее, по нажатию пользователем клавиши <Enter> генерируется событие `ActionEvent` и посредством метода `actionPerformed()` передается всем зарегистрированным обработчикам. В программе `TextFieldDemo` этот метод лишь вызывает метод `getText()`, извлекая текст, содержащийся

в поле редактирования. После этого текст отображается посредством метки, на которую ссылается переменная `jlab`.

Подобно `JButton`, с компонентом `JTextField` связана команда действия. По умолчанию в качестве такой команды принимается текущее содержимое поля редактирования. Однако вы можете установить команду явным образом, вызвав метод `setActionCommand()`.

```
void setActionCommand(String cmd)
```

Строка, переданная посредством параметра `cmd`, становится новой командой действия; текст в поле не изменяется. Установленная строка команды действия остается постоянной, независимо от того, какой текст вводится в поле редактирования. Как правило, разработчики прибегают к явной установке команды действия для того, чтобы обеспечить распознавание компонента, сгенерировавшего событие. Так приходится поступать в том случае, если во фрейме находится несколько управляющих элементов, для которых установлен общий обработчик событий. Установив команду действия, вы получаете удобное средство идентификации компонента. Ниже приведен пример программы, в которой используются два поля редактирования, идентифицируемые командами действия.

```
// Использование двух полей редактирования.
```

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

class TwoTFDemo implements ActionListener {

    JTextField jtf1;
    JTextField jtf2;
    JLabel jlab;

    TwoTFDemo() {

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Use Two Text Fields");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(240, 120);
```

```
// Завершение программы при закрытии окна пользователем.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Создание двух полей редактирования.
jtf1 = new JTextField(10);
jtf2 = new JTextField(10);

// Установка команд действия.
jtf1.setActionCommand("One");
jtf2.setActionCommand("Two");

// Связывание обработчиков событий действия
// с полями редактирования.
jtf1.addActionListener(this);
jtf2.addActionListener(this);

// Включение полей редактирования в состав панели содержимого.
jfrm.getContentPane().add(jtf1);
jfrm.getContentPane().add(jtf2);

// Создание пустой метки.
jlab = new JLabel("");

// Добавление метки к панели содержимого..
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

// Обработка событий действия.
public void actionPerformed(ActionEvent ae) {

    // Для идентификации компонента-источника события
    // используется команда действия.
    if(ae.getActionCommand().equals("One"))
        jlab.setText("ENTER pressed in tf1: "
                     + jtf1.getText());
    else
        jlab.setText("ENTER pressed in jtf2: "
                     + jtf2.getText());
}
```

```
public static void main(String args[]) {  
  
    // Фрейм создается в потоке обработки событий.  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            new TwoTFDemo();  
        }  
    });  
}
```

Окно, создаваемое при работе программы, показано на рис. 1.4.

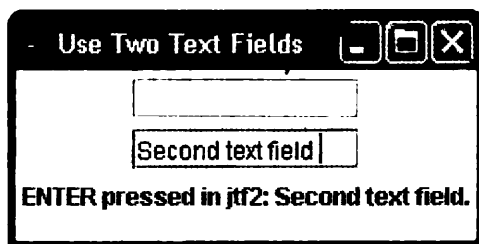


Рис. 1.4. Окно, отображаемое командой, использующей два поля редактирования

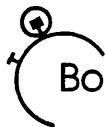
Данная программа создает два компонента — `JTextField`: `jtf1` и `jtf2`. Заметьте, что команды действия, связанные с `jtf1` и `jtf2`, задаются с помощью следующих строк кода:

```
jtf1.setActionCommand("One");  
jtf2.setActionCommand("Two");
```

В теле метода `actionPerformed()` команда действия используется для определения того, какой из компонентов сгенерировал событие.

```
public void actionPerformed(ActionEvent ae) {  
  
    if(ae.getActionCommand().equals("One"))  
        jlab.setText("ENTER pressed in tfl: "  
            + jtf1.getText());  
    else  
        jlab.setText("ENTER pressed in jtf2: "  
            + jtf2.getText());  
}
```

Поскольку в качестве команды действия для каждого поля редактирования была установлена заранее определенная строка, ее удобно использовать в качестве идентификатора источника события.



### Вопросы для текущего контроля. . .

1. Если строка текста, вводимая пользователем в компоненте `TextField`, длиннее отображаемого поля, она усекается. Да или нет?
2. Какое событие генерируется, если пользователь, работая с полем редактирования, нажимает клавишу `<Enter>`?
3. Зачем нужно явно устанавливать команду действия для компонента `TextField`?

## Проект 1.2. Создание простой шифровальной машины

`Coder.java` В данном проекте создается простая шифровальная машина, при этом используются компоненты `JLabel`, `JButton` и `TextField`. Шифр предельно прост: код каждого символа увеличивается на единицу. Так, вместо А подставляется буква В, вместо В — С и т.д. Очевидно, что расшифровать такой код не составляет труда, но он применен здесь лишь для демонстрации взаимодействия кнопок и полей редактирования. Он также демонстрирует подход, при котором события, сгенерированные различными компонентами (в данном случае полем редактирования и кнопкой), обрабатываются одним обработчиком.

### Последовательность действий

1. Создайте файл `Coder.java` и введите приведенные ниже строку комментариев и выражения `import`.

```
// Проект 1.2. Простая шифровальная машина.
```

```
import java.awt.*;
```

1. Нет. Длина строки текста может превышать ширину поля.
2. `ActionEvent`.
3. Если команда действия не установлена явно, ее роль выполняет текст, содержащийся в поле редактирования. После определения команды действия ее можно использовать для идентификации компонента, независимо от его текущего содержимого.

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

2. Начните класс `Coder` следующими строками кода:

```
class Coder implements ActionListener {

    JTextField jtfPlaintext;
    JTextField jtfCiphertext;
```

Заметьте, что данный класс реализует интерфейс `ActionListener`. В нем определены переменные, которые впоследствии будут ссылаться на поля редактирования. В поле `jtfPlaintext` будет вводиться обычный текст. Поле `jtfCiphertext` будет содержать его зашифрованную версию.

3. Начните код конструктора `Coder()` так, как показано ниже.

```
Coder() {

    // Создание нового контейнера JFrame.
    JFrame jfrm = new JFrame("A Simple Code Machine");

    // Установка диспетчера компоновки FlowLayout.
    jfrm.getContentPane().setLayout(new FlowLayout());

    // Установка начальных размеров фрейма.
    jfrm.setSize(340, 120);

    // Завершение программы при закрытии приложения
    // пользователем.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Эти действия уже знакомы вам по предыдущим примерам.

4. Создайте два компонента `JLabel`.

```
// Создание двух меток.
JLabel jlabPlaintext = new JLabel(" Plain Text: ");
JLabel jlabCiphertext = new JLabel("Cipher Text: ");
```

5. Создайте два экземпляра `JTextField` и установите ссылки на них в переменных `jtfPlaintext` и `jtfCiphertext`.

```
// Создание двух полей редактирования.
jtfPlaintext = new JTextField(20);
jtfCiphertext = new JTextField(20);
```

6. Установите команды действий для полей редактирования и свяжите с обоими компонентами обработчик событий.

```
// Установка команд действия для полей редактирования.  
jtfPlaintext.setActionCommand("Encode");  
jtfCiphertext.setActionCommand("Decode");
```

```
// Связывание обработчиков событий с полями редактирования.  
jtfPlaintext.addActionListener(this);  
jtfCiphertext.addActionListener(this);
```

В пользу установки команд действия для обоих полей редактирования можно выдвинуть три аргумента. Во-первых, эти команды позволяют идентифицировать компоненты. Во-вторых, если не установить команду действия для одного из полей редактирования, то по умолчанию в качестве такой команды будет принято текущее содержимое компонента. Если по стечению обстоятельств этот текст будет совпадать с командой действия другого компонента, возможен конфликт. Явная установка обеих команд действия предотвратит его. И в-третьих, как вы вскоре увидите, мы установили для полей редактирования такие же команды действия, которые будут установлены для кнопок. Это означает, что щелчок на кнопке и нажатие клавиши <Enter> во время работы с компонентом будут обрабатываться одинаково.

7. Включите поля редактирования и метки в состав панели содержимого.

```
// Добавление полей редактирования и меток  
// к панели содержимого.  
jfrm.getContentPane().add(jlabPlaintext);  
jfrm.getContentPane().add(jtfPlaintext);  
jfrm.getContentPane().add(jlabCiphertext);  
jfrm.getContentPane().add(jtfCiphertext);
```

Порядок включения компонентов важен, так как метки должны описывать назначение полей редактирования.

8. Создайте три кнопки: **Encode**, **Decode** и **Reset**.

```
// Создание экземпляров кнопок.  
JButton jbbtnEncode = new JButton("Encode");  
JButton jbbtnDecode = new JButton("Decode");  
JButton jbbtnReset = new JButton("Reset");
```

9. Задайте текущий экземпляр класса (переменная `this`) в качестве обработчика событий, связанных с кнопками, и включите кнопки в состав панели содержимого.

```
// Связывание обработчиков с кнопками.
jbtnEncode.addActionListener(this);
jbtnDecode.addActionListener(this);
jbtnReset.addActionListener(this);

// Включение кнопок в состав панели содержимого.
jfrm.getContentPane().add(jbtnEncode);
jfrm.getContentPane().add(jbtnDecode);
jfrm.getContentPane().add(jbtnReset);
```

10. Завершите конструктор `Coder` вызовом метода `setVisible()`.

```
// Отображение фрейма.
jfrm.setVisible(true);
}
```

11. Введите код метода `actionPerformed()`, начав его следующим образом:

```
// Обработка событий действий.
public void actionPerformed(ActionEvent ae) {

    // Если команда действия равна "Encode",
    // строка шифруется.
    if(ae.getActionCommand().equals("Encode")) {

        // Получение текста и передача его объекту
        StringBuilder.
        StringBuilder str =
            new StringBuilder(jtfPlaintext.getText());

        // Добавление единицы к коду каждого символа.
        for(int i=0; i<str.length(); i++)
            str.setCharAt(i, (char)(str.charAt(i) + 1));

        // Помещение зашифрованного текста в поле Cipher Text.
        jtfCiphertext.setText(str.toString());

    }
```



Выражение `if` проверяет, имеет ли команда действия значение "Encode". Такое значение команда будет иметь, если пользователь щелкнул мышью на кнопке `jbtnEncode` или нажал клавишу `<Enter>` при вводе текста в поле `jtfPlaintext`. Поскольку и для `jbtnEncode`, и для `jtfPlaintext` команда действия равна "Encode", оба события обрабатываются одинаково. Другими словами, события, генерируемые этими двумя управляющими элементами, будут отображаться в один обработчик, поскольку команды действия для них совпадают. Данный обработчик шифрует строку, представленную в поле Plain Text, и отображает ее в поле Cipher Text.

12. Добавьте выражение `else if`, которое проверяет, является ли команда действия строкой "Decode".

```
// Если команда действия равна "Decode",
// строка декодируется.
else if(ae.getActionCommand().equals("Decode")) {

    // Получение кодированного текста
    // и передача его объекту StringBuilder.
    StringBuilder str =
        new StringBuilder(jtfCiphertext.getText());

    // Вычитание единицы из кода каждого символа.
    for(int i=0; i<str.length(); i++)
        str.setCharAt(i, (char)(str.charAt(i) - 1));

    // Помещение декодированного текста в поле Plain Text.
    jtfPlaintext.setText(str.toString());
}
```

Обработка события осуществляется подобно тому, как это происходит для команды действия "Encode", за исключением того, что строка, содержащаяся в поле Cipher Text, дешифруется и помещается в поле Plain Text.

13. Завершите метод `actionPerformed()` обработкой команды действия "Reset". Эта команда установлена для кнопки `jbtnReset`. Поскольку команд действия только три, проверять на совпадение со строкой "Reset" нет необходимости. Если выполнение программы дошло до этой точки, значит, был щелчок на кнопке Reset.

```
// Последний вариант - команда Reset.
else {
    jtfPlaintext.setText("");
}
```

```

        jtfCiphertext.setText("");
    }
}

```

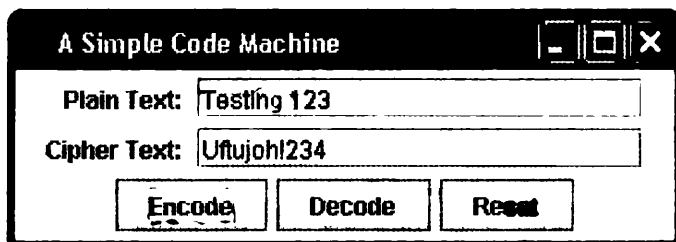
14. Завершите код программы методом `main()`.

```

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new Coder();
        }
    });
}

```

15. Полностью код программы приведен ниже. Окно, отображаемое в процессе ее работы, выглядит следующим образом:



// Проект 1.2. Простая шифровальная машина.

```

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

class Coder implements ActionListener {

    JTextField jtfPlaintext;
    JTextField jtfCiphertext;

    Coder() {

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("A Simple Code Machine");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());
    }
}

```

```
// Установка начальных размеров фрейма.
jfrm.setSize(340, 120);

// Завершение программы при закрытии окна пользователем.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Создание двух меток.
JLabel jlabPlaintext = new JLabel("   Plain Text: ");
JLabel jlabCiphertext = new JLabel("Cipher Text: ");

// Создание двух полей редактирования.
jtfPlaintext = new JTextField(20);
jtfCiphertext = new JTextField(20);

// Установка команд действия для полей редактирования.
jtfPlaintext.setActionCommand("Encode");
jtfCiphertext.setActionCommand("Decode");

// Связывание обработчиков событий с полями
// редактирования.
jtfPlaintext.addActionListener(this);
jtfCiphertext.addActionListener(this);

// Добавление полей редактирования и меток
// к панели содержимого.
jfrm.getContentPane().add(jlabPlaintext);
jfrm.getContentPane().add(jtfPlaintext);
jfrm.getContentPane().add(jlabCiphertext);
jfrm.getContentPane().add(jtfCiphertext);

// Создание экземпляров кнопок.
JButton jbtnEncode = new JButton("Encode");
JButton jbtnDecode = new JButton("Decode");
JButton jbtnReset = new JButton("Reset");

// Связывание обработчиков с кнопками.
jbtnEncode.addActionListener(this);
jbtnDecode.addActionListener(this);
jbtnReset.addActionListener(this);

// Включение кнопок в состав панели содержимого.
jfrm.getContentPane().add(jbtnEncode);
```

```
jfrm.getContentPane().add(jbtnDecode);
jfrm.getContentPane().add(jbtnReset);

// Отображение фрейма.
jfrm.setVisible(true);
}

// Обработка событий действий.
public void actionPerformed(ActionEvent ae) {

    // Если команда действия равна "Encode",
    // строка шифруется.
    if(ae.getActionCommand().equals("Encode")) {

        // Получение текста и передача его объекту
        StringBuilder.
        StringBuilder str =
            new StringBuilder(jtfPlaintext.getText());

        // Добавление единицы к коду каждого символа.
        for(int i=0; i<str.length(); i++)
            str.setCharAt(i, (char)(str.charAt(i) + 1));

        // Помещение зашифрованного текста в поле Cipher Text.
        jtfCiphertext.setText(str.toString());

    }

    // Если команда действия равна "Decode",
    // строка декодируется.
    else if(ae.getActionCommand().equals("Decode")) {

        // Получение кодированного текста
        // и передача его объекту StringBuilder.
        StringBuilder str =
            new StringBuilder(jtfCiphertext.getText());

        // Вычитание единицы из кода каждого символа.
        for(int i=0; i<str.length(); i++)
            str.setCharAt(i, (char)(str.charAt(i) - 1));

        // Помещение декодированного текста в поле Plain Text.
        jtfPlaintext.setText(str.toString());
    }
}
```

```

    }

    // Последний вариант - команда Reset.
    else {
        jtfPlaintext.setText("");
        jtfCiphertext.setText("");
    }
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new Coder();
        }
    });
}
}

```

## Альтернативные подходы к обработке событий

Программы, рассмотренные в данном модуле, очень просты. Так же просты и используемые в них обработчики событий, в роли которых выступает основной класс приложения, реализующий соответствующий интерфейс. События передаются для обработки экземпляру данного класса. Хотя такой подход вполне применим на практике, его нельзя рассматривать как единственно возможный. В программах могут применяться и другие способы обработки событий. Во-первых, для каждого события можно определить отдельный обработчик. Таким образом различные классы будут поддерживать различные события. Во-вторых, вы можете реализовать обработчики посредством неименованных внутренних классов. Например, обработчик для кнопки может иметь следующий вид:

```

jbnt.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        // Действия по обработке события.
    }
});

```

Здесь применяется неименованный внутренний класс, реализующий интерфейс `ActionListener`. Примеры использования обоих описанных выше подходов встретятся вам в последующих модулях.

ВАЖНО!

1.9

## Краткие сведения о диспетчерах компоновки

Расположение компонентов в составе контейнера определяется *диспетчером компоновки*, связанным с этим контейнером. В Java определено несколько таких диспетчеров. Большинство из них входят в состав AWT (пакет `java.awt`), но Swing также предоставляет несколько дополнительных диспетчеров компоновки. Все диспетчеры являются экземплярами классов, реализующих интерфейс `LayoutManager`. (Некоторые из диспетчеров реализуют интерфейс `LayoutManager2`.) Ниже описано несколько диспетчеров компоновки, доступных программистам, использующим средства Swing.

|                            |                                                                                                                                                                                         |
|----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>FlowLayout</code>    | Располагает компоненты в строке слева направо; следующая строка размещается под предыдущей. (Установки для некоторых стран предполагают следование компонентов в строке справа налево.) |
| <code>BorderLayout</code>  | Помещает компоненты в пяти областях, расположенных по центру и по краям контейнера. По умолчанию такой диспетчер компоновки связан с панелью содержимого                                |
| <code>GridLayout</code>    | Располагает компоненты в виде таблицы                                                                                                                                                   |
| <code>GridBagLayout</code> | Располагает компоненты в виде таблицы с ячейками различных размеров                                                                                                                     |
| <code>BoxLayout</code>     | Располагает компоненты по вертикали или по горизонтали                                                                                                                                  |
| <code>SpringLayout</code>  | Использует при размещении компонентов специальные ограничения                                                                                                                           |

До сих пор вам встречались два диспетчера: `BorderLayout` (который по умолчанию связан с панелью содержимого) и `FlowLayout`. Далее в этой книге будут описаны и использованы также другие диспетчеры компоновки. Сейчас же обратим более пристальное внимание на `BorderLayout`, поскольку именно он используется по умолчанию с панелью содержимого контейнера верхнего уровня.

Диспетчер `BorderLayout` определяет в составе контейнера пять областей, в которые могут помещаться компоненты. Первая область расположена по центру окна. Остальные четыре — по краям. Соответственно области называются центральной (*center*), северной (*north*), южной (*south*), восточной (*east*) и западной (*west*). По умолчанию компонент, помещаемый в панель содержимого, располагается в центральной области. Явным образом управлять размещением компонентов можно, используя специальную форму метода `add()`.

```
void add(Component comp, Object loc)
```

где параметр `comp` задает компонент, добавляемый к панели, а параметр `loc` определяет область, в которую этот компонент будет помещен. Допустимо одно из следующих значений параметра `loc`:

|                                  |                                |                                 |
|----------------------------------|--------------------------------|---------------------------------|
| <code>BorderLayout.CENTER</code> | <code>BorderLayout.EAST</code> | <code>BorderLayout.NORTH</code> |
| <code>BorderLayout.SOUTH</code>  | <code>BorderLayout.WEST</code> |                                 |

Диспетчер компоновки `BorderLayout` наиболее удобен, если вы создаете объект `JFrame`, который должен содержать лишь один компонент (он размещается в центре окна), или если вам надо включить группу компонентов, расположенных на панели. Подробнее этот вопрос будет обсуждаться в модуле 4.

**ВАЖНО!****1.10. Пакеты в составе Swing**

Swing — очень большая подсистема, включающая значительное количество пакетов. В JDK 5 в состав Swing входили следующие пакеты:

|                                     |                                           |                                       |
|-------------------------------------|-------------------------------------------|---------------------------------------|
| <code>javax.swing</code>            | <code>javax.swing.border</code>           | <code>javax.swing.colorchooser</code> |
| <code>javax.swing.event</code>      | <code>javax.swing.filechooser</code>      | <code>javax.swing.plaf</code>         |
| <code>javax.swing.plaf.basic</code> | <code>javax.swing.plaf.metal</code>       | <code>javax.swing.plaf.multi</code>   |
| <code>javax.swing.plaf.synth</code> | <code>javax.swing.table</code>            | <code>javax.swing.text</code>         |
| <code>javax.swing.text.html</code>  | <code>javax.swing.text.html.parser</code> | <code>javax.swing.text.rtf</code>     |
| <code>javax.swing.tree</code>       | <code>javax.swing.undo</code>             |                                       |

Основным является пакет `javax.swing`. Он должен импортироваться любой программой, создаваемой с использованием Swing. В этом пакете определены классы, реализующие основные компоненты Swing, например кнопки, метки и флажки опций. Далее в этой книге вы встретите некоторые другие пакеты.

**Тест для самоконтроля по модулю 1**

1. Большинство компонентов AWT преобразуются в платформенно-ориентированные элементы. Почему это считается проблемой и как она решается средствами Swing?
2. Большинство компонентов Swing полностью реализовано на языке Java. Да или нет?
3. Назовите четыре тяжелых контейнера верхнего уровня.
4. Какой контейнер верхнего уровня чаще всего используется в приложениях?

5. Контейнер `JFrame` содержит несколько панелей. В какую панель помещаются компоненты?
6. Чтобы обработчик получал оповещение о возникающих событиях, он должен быть \_\_\_\_\_ в источнике.
7. Какой интерфейс должен реализовывать класс, чтобы получать оповещение о событиях действий?
8. Какой метод следует вызвать при работе с компонентом `JButton` или `TextField`, чтобы установить команду действия?
9. Назовите три диспетчера компоновки.
10. В секундомере, созданном в рамках проекта 1.1, используются две кнопки: одна для запуска секундомера, а другая для его остановки. Однако можно использовать только одну кнопку, которая будет запускать остановленный и останавливать идущий секундомер. При использовании такого подхода имеет смысл заменять текст на кнопке (надпись `Start` заменять на `Stop` и наоборот). Поскольку по умолчанию текст, отображаемый на кнопке, является в то же время командой действия, вы можете использовать одну кнопку для двух различных целей. Ваша задача — переписать проект 1.1 в соответствии с данным подходом.

Для решения этой задачи надо использовать метод `setText()`, предоставляемый классом `JButton`. Этот метод устанавливает текст на кнопке. Его заголовок выглядит следующим образом:

```
void setText(String msg)
```

где параметр `msg` задает строку текста, которая должна выводиться на кнопке. Пользуясь этим методом, вы можете менять текст на кнопке в процессе выполнения программы.

11. Если вы пользуетесь JDK 5 либо более поздней версией, перепишите код проекта 1.2 так, чтобы в нем не вызывался метод `getContentPane()`.





# Модуль 2

## Метки, кнопки и обрамление

- 2.1.** Работа с рамками
- 2.2.** Выравнивание содержимого метки
- 2.3.** Включение изображений в состав меток
- 2.4.** Деактивизация метки
- 2.5.** Объявление мнемонических обозначений для меток
- 2.6.** Основные принципы работы кнопок и класс `AbstractButton`
- 2.7.** Включение изображения в состав объекта `JButton`
- 2.8.** Определение кнопки по умолчанию
- 2.9.** Использование неименованных внутренних классов в качестве обработчиков событий
- 2.10.** Создание кнопок с двумя состояниями с помощью класса `JToggleButton`
- 2.11.** Создание флажков опций с помощью класса `JCheckBox`
- 2.12.** Создание переключателей опций с помощью класса `JRadioButton`

Данный модуль начинается с подробного рассмотрения двух наиболее часто используемых компонентов Swing: меток и кнопок. Здесь также будет обсуждаться важная характеристика меток — оформление. (Снабдив метку оформлением, вы получаете возможность визуально выделить ее содержимое.) Кроме того, в данном модуле вы познакомитесь с альтернативными средствами обработки событий и узнаете еще об одном диспетчере компоновки — `GridLayout`.

**ВАЖНО!**

## **2.1. Общие сведения об оформлении**

К любому компоненту Swing можно добавить оформление. Однако для большинства компонентов, например кнопок, полей редактирования и окон списков, предусмотрены собственные рамки (их отображение задается при определении стилей). Поэтому для подобных элементов не следует специально задавать оформление, так как оно может конфликтовать с оформлением, заданным посредством стилей. Исключением из данного правила являются два компонента: метки и панели. Панели будут описаны в модуле 4. В этом модуле мы будем использовать оформления для демонстрации некоторых возможностей меток.

Каждое оформление Swing является экземпляром класса, реализующего интерфейс `javax.swing.border.Border`. Несмотря на то что разработчик имеет возможность определить новую рамку, в этом, как правило, не возникает необходимости, поскольку Swing предоставляет целый ряд предопределенных стилей, доступных посредством класса `javax.swing.BorderFactory`. В этом классе определено несколько фабричных методов, которые создают различные типы оформлений: от обычных рамок, состоящих из линий толщиной в один пиксель, до рельефных оформлений. Можно также создать рамку, которая будет включать в себя короткий заголовок, а также пустое, или невидимое, оформление. Пустые рамки удобно применять для создания зазора между компонентами.

В данной главе будут рассматриваться три типа рамок: состоящие из линий, рельефные и пустые. Для того чтобы создать оформление, сформированное из обычных линий, надо использовать следующий фабричный метод:

```
static Border createLineBorder(Color lineColor)
```

Здесь параметр `lineColor` определяет цвет линии, используемой при формировании оформления. Например, для того, чтобы создать рамку черного цвета, надо установить значение `Color.BLACK`. В этом случае будет создана рамка из линий, толщина которых принимается по умолчанию. Для того чтобы явно задать толщину линий, надо использовать другой вариант метода:

```
static Border createLineBorder(Color lineColor, int width)
```

Здесь значение параметра `width` определяет толщину линии в пикселях.

Рельефная рамка создается подобно рассмотренной выше, но выглядит так, как будто она “выгравирована” вокруг компонента. Создать рельефные рамки можно несколькими способами. Один из них — использование метода, показанного ниже.

```
static Border createEtchedBorder()
```

При этом создается обрамление, использующее конфигурацию по умолчанию.

Для того чтобы создать пустую рамку, которая формирует зазор между компонентами, надо использовать следующий вариант метода `createEmptyBorder()`:

```
static Border createEmptyBorder(int topWidth, int leftWidth,  
                                int bottomWidth, int rightWidth)
```

Параметры, передаваемые методу, задают размеры зазоров с каждой стороны компонента, выраженные в пикселях.

Создав рамку, надо связать ее с компонентом, вызвав для этой цели метод `setBorder()`, который определен в классе `JComponent`. Объявление этого метода выглядит следующим образом:

```
void setBorder(Border border)
```

Здесь параметр `border` определяет используемое обрамление. Следует заметить, что одна и та же рамка может быть связана с несколькими компонентами. Другими словами, вам нет необходимости создавать новый объект `Border` для каждого компонента, для которого вы собираетесь отобразить рамку.

Рассмотрим программу, которая демонстрирует применение обрамления, состоящего из линий, и рельефной рамки. Пример применения пустой рамки будет обсуждаться в следующем разделе. В процессе работы программа отображает окно, показанное на рис. 2.1.

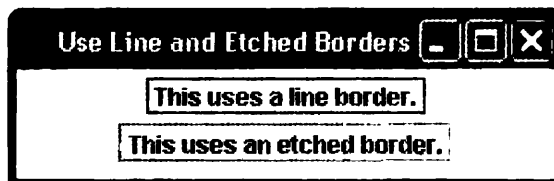


Рис. 2.1. Выходные данные, генерируемые демонстрационной программой

## 76 Модуль 2. Метки, кнопки и обрамление

---

```
// Демонстрация использования рамки, состоящей из линий,  
// и рельефного обрамления.  
import java.awt.*;  
import javax.swing.*;  
  
class BorderDemo {  
  
    BorderDemo() {  
  
        // Создание нового контейнера JFrame.  
        JFrame jfrm = new JFrame("Use Line and Etched Borders");  
  
        // Связывание с контейнером диспетчера компоновки FlowLayout.  
        jfrm.getContentPane().setLayout(new FlowLayout());  
  
        // Установка начального размера фрейма.  
        jfrm.setSize(280, 90);  
  
        // Завершение программы по закрытию пользователем приложения.  
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        // Создание метки и установка для нее рамки, состоящей  
        // из линий.  
        JLabel jlab = new JLabel(" This uses a line border. ");  
        // Связывание обрамления с jlab.  
        jlab.setBorder(BorderFactory.createLineBorder(Color.BLACK));  
  
        // Создание второй метки и установка для нее рельефной рамки.  
        JLabel jlab2 = new JLabel(" This uses an etched border. ");  
        // Связывание обрамления с jlab2.  
        jlab2.setBorder(BorderFactory.createEtchedBorder());  
  
        // Включение меток в состав панели содержимого.  
        jfrm.getContentPane().add(jlab);  
        jfrm.getContentPane().add(jlab2);  
  
        // Отображение фрейма.  
        jfrm.setVisible(true);  
    }  
  
    public static void main(String args[]) {  
        // Создание фрейма в потоке обработки событий.  
        SwingUtilities.invokeLater(new Runnable() {
```

```
public void run() {  
    new BorderDemo();  
}  
});  
}
```



## Вопросы для текущего контроля

1. Каждое обрамление Swing является экземпляром класса, реализующего интерфейс `javax.swing.border.Border`. Да или нет?
2. Желательно задавать обрамление для каждого компонента Swing. Да или нет?
3. Какой фабричный метод создает обрамление, состоящее из линий?

## Подробно о метках

В модуле 1 приводились общие сведения о метках. Теперь мы рассмотрим эти компоненты более детально. Несмотря на то что метки не реагируют на действия пользователя, они обладают рядом свойств, позволяющих оптимально настроить метку для каждой конкретной задачи. Более того, эти компоненты обладают такой высокой степенью гибкости, что пользуясь только ими, можно придать интерфейсу приложения требуемый внешний вид. Правильно применяя метки Swing, можно значительно улучшить почти любой пользовательский интерфейс.

Для поддержки меток предназначен класс `JLabel`. Конструкторы этого класса описаны в табл. 2.1. В модуле 1 вы познакомились с простейшим вариантом метки, содержащей только строку текста. Однако возможны гораздо более сложные метки. Например, вы можете задавать выравнивание содержимого этого компонента. Можно также создать метку, включающую пиктограмму (графическое изображение) либо объединяющую в себе и пиктограмму, и текст. Метку можно перевести в неактивное состояние. Существует также возможность связать метку с другим компонентом, например с полем редактирования. С помощью метки можно отображать сведения о командных клавишах, предназначенных для выполнения различных операций. При нажатии такой клавиши компонент, связанный с меткой, автоматически получает фокус ввода.

1. Да.
2. Нет.
3. `createLineBorder()`.

Таблица 2.1. Конструкторы класса JLabel

| Конструктор                                  | Описание                                                                                                                                                             |
|----------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| JLabel()                                     | Создает метку с пустым содержимым                                                                                                                                    |
| JLabel(String str)                           | Создает метку, которая отображает строку, переданную посредством параметра str                                                                                       |
| JLabel(Icon icon)                            | Создает метку, которая отображает изображение, переданное посредством параметра icon                                                                                 |
| JLabel(String str, int horzAlign)            | Создает метку, которая отображает строку, переданную посредством параметра str. Строка выравнивается по горизонтали в соответствии со значением horzAlign            |
| JLabel(Icon icon, int horzAlign)             | Создает метку, которая отображает изображение, переданное посредством параметра icon. Изображение выравнивается по горизонтали в соответствии со значением horzAlign |
| JLabel(String str, Icon icon, int horzAlign) | Создает метку, которая отображает строку, str и изображение icon. Содержимое метки выравнивается по горизонтали в соответствии со значением horzAlign                |

Данный раздел очень важен, поскольку в нем приводятся сведения о различных действиях, которые можно выполнить с меткой (например, загрузка пиктограммы, деактивизация компонента, выравнивание текста и изображений). Эти действия применимы не только к объектам JLabel, но и ко многим другим компонентам Swing. Советую внимательно ознакомиться с этим материалом.

**ВАЖНО!****2.2. Выравнивание содержимого метки**

Если вам надо создать метку, проще всего сделать это, используя конструктор JLabel.

```
JLabel(String str)
```

Этот вариант конструктора мы использовали в модуле 1. Здесь параметр str задает строку, которая должна отображаться в составе метки. Данный конструктор выводит строку, выравнивая ее по “ведущему краю” компонента (для английского текста это левая граница). В вертикальном направлении текст располагается по центру метки. Однако такое поведение, принимаемое по умолчанию, не единственно возможное. Вы можете явным образом задавать выравнивание текста в составе метки.

Указать выравнивание по горизонтали можно двумя способами. Во-первых, существует следующий конструктор:

```
JLabel(String str, int horzAlign)
```

При его использовании строка выравнивается по горизонтали так, как указано с помощью параметра `horzAlign`. Допустимо одно из следующих значений:

|                                      |                                     |
|--------------------------------------|-------------------------------------|
| <code>SwingConstants.LEFT</code>     | <code>SwingConstants.RIGHT</code>   |
| <code>SwingConstants.CENTER</code>   | <code>SwingConstants.LEADING</code> |
| <code>SwingConstants.TRAILING</code> |                                     |

В интерфейсе `SwingConstants` определены константы, управляющие поведением компонентов Swing. Этот интерфейс реализуется `JLabel` и некоторыми другими классами, соответствующими различным компонентам. Таким образом, для обращения к этим константам можно также использовать имя `JLabel`, например `JLabel.RIGHT`. Несмотря на такую возможность, в рамках данной книги мы будем ссылаться на интерфейс `SwingConstants`, поскольку подобное обращение дает более точное представление о том, где определена конкретная константа.

Несмотря на то что проще всего указывать тип выравнивания при вызове конструктора, Swing допускает альтернативные подходы. Так, например, после создания метки можно вызвать метод `setHorizontalAlignment()`. Объявление этого метода выглядит следующим образом:

```
void setHorizontalAlignment(int horzAlign)
```

Здесь значением параметра `horzAlign` должна быть одна из констант выравнивания, описанных выше.

Для выравнивания по вертикали предназначен метод `setVerticalAlignment()`. Он определяется так, как показано ниже.

```
void setVerticalAlignment(int vertAlign)
```

Значение, передаваемое методу с помощью параметра `vertAlign`, должно быть равно одной из следующих констант:

|                                 |                                    |                                    |
|---------------------------------|------------------------------------|------------------------------------|
| <code>SwingConstants.TOP</code> | <code>SwingConstants.CENTER</code> | <code>SwingConstants.BOTTOM</code> |
|---------------------------------|------------------------------------|------------------------------------|

Как уже говорилось ранее, по умолчанию используется вертикальное выравнивание по центру компонента, поэтому значение `CENTER` имеет смысл указывать только в том случае, если ранее вы задавали другой тип выравнивания и хотите восстановить поведение компонента, принятое по умолчанию.



Изменяя тип выравнивания, необходимо учитывать следующее: результаты ваших действий могут не проявиться визуально. Например, если вы работаете с диспетчером компоновки `FlowLayout`, для метки устанавливаются предпочтительные размеры, обеспечивающие лишь вывод данных в составе метки. При этом текст будет располагаться в одной и той же позиции, независимо от того, задано ли для метки выравнивание по верхнему, нижнему краю или по центру. Выравнивание проявляется только в тех метках, размеры которых большие, чем необходимо для размещения содержимого. Так, например, происходит при использовании диспетчера компоновки `GridLayout`, который автоматически устанавливает размеры метки в соответствии с пространством, доступным для нее.

Рассмотрим программу, демонстрирующую различные типы вертикального и горизонтального выравнивания. Заметьте, что выравнивание по горизонтали задается с помощью конструктора `JLabel`, а выравнивание по вертикали — путем вызова метода `setVerticalPosition()`. Окно, отображаемое в процессе работы программы, показано на рис. 2.2.

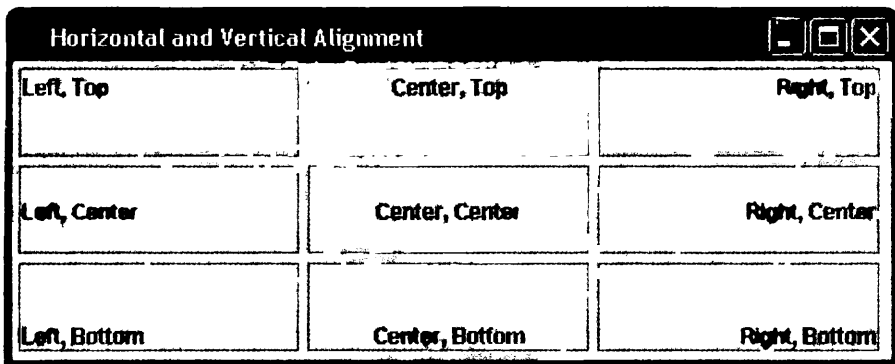


Рис. 2.2. Окно программы, демонстрирующей выравнивание текста

```
// Программа, демонстрирующая выравнивание текста
// по горизонтали и по вертикали
import javax.swing.*;
import java.awt.*;
import javax.swing.border.*;

class AlignLabelDemo {

    AlignLabelDemo() {
        JLabel[] jlabs = new JLabel[9];

        // Создание нового компонента JFrame.
```

```
JFrame jfrm = new JFrame("Horizontal and Vertical  
Alignment");
```

```
// Установка диспетчера компоновки GridLayout.
```

```
// Создается таблица из 3 строк и 3 столбцов
```

```
// с зазором 4 пикселя между компонентами.
```

```
jfrm.getContentPane().setLayout(new GridLayout(3, 3, 4, 4));
```

```
// Определение исходного размера фрейма.
```

```
jfrm.setSize(500, 200);
```

```
// Завершение программы при закрытии пользователем окна.
```

```
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
// Горизонтальное выравнивание по левой границе и
```

```
// вертикальное выравнивание по верхней границе.
```

```
jlabs[0] = new JLabel("Left, Top", SwingConstants.LEFT);
```

```
jlabs[0].setVerticalAlignment(SwingConstants.TOP);
```

```
// Горизонтальное выравнивание по центру и
```

```
// вертикальное выравнивание по верхней границе.
```

```
jlabs[1] = new JLabel("Center, Top", SwingConstants.CENTER);
```

```
jlabs[1].setVerticalAlignment(SwingConstants.TOP);
```

```
// Горизонтальное выравнивание по правой границе и
```

```
// вертикальное выравнивание по верхней границе.
```

```
jlabs[2] = new JLabel("Right, Top", SwingConstants.RIGHT);
```

```
jlabs[2].setVerticalAlignment(SwingConstants.TOP);
```

```
// Горизонтальное выравнивание по левой границе и
```

```
// вертикальное выравнивание по центру.
```

```
// Такое выравнивание применяется по умолчанию
```

```
// для большинства языков.
```

```
jlabs[3] = new JLabel("Left, Center", SwingConstants.LEFT);
```

```
// Горизонтальное и вертикальное выравнивание по центру.
```

```
jlabs[4] = new JLabel("Center, Center",
```

```
SwingConstants.CENTER);
```

```
// Горизонтальное выравнивание по правой границе и
```

```
// вертикальное выравнивание по центру.
```

```
jlabs[5] = new JLabel("Right, Center", SwingConstants.RIGHT);
```

## 82 Модуль 2. Метки, кнопки и обрамление

---

```
// Горизонтальное выравнивание по правой границе и
// вертикальное выравнивание по нижней границе.
jlabs[6] = new JLabel("Left, Bottom", SwingConstants.LEFT);
jlabs[6].setVerticalAlignment(SwingConstants.BOTTOM);

// Горизонтальное выравнивание по центру и
// вертикальное выравнивание по нижней границе.
jlabs[7] = new JLabel("Center, Bottom",
    SwingConstants.CENTER);
jlabs[7].setVerticalAlignment(SwingConstants.BOTTOM);

// Горизонтальное выравнивание по правой границе и
// вертикальное выравнивание по нижней границе.
jlabs[8] = new JLabel("Right, Bottom", SwingConstants.RIGHT);
jlabs[8].setVerticalAlignment(SwingConstants.BOTTOM);

// Добавление рамок для отображения вокруг меток.
// Создание рельефной рамки.
Border border = BorderFactory.createEtchedBorder();

// Добавление рамки к каждой метке.
for(int i=0; i<9; i++)
    jlabs[i].setBorder(border);

// Добавление меток к панели содержимого.
for(int i=0; i<9; i++)
    jfrm.getContentPane().add(jlabs[i]);

// Добавление пустой рамки к панели содержимого.
JPanel cp = ((JPanel) jfrm.getContentPane());
// Пустая рамка создает зазор между краем фрейма
// и содержащимися в нем элементами.
cp.setBorder(BorderFactory.createEmptyBorder(4, 4, 4, 4));

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new AlignLabelDemo();
        }
    });
}
```

## Спросим у опытного программиста

**Вопрос.** Вы говорили о том, что есть возможность установить размер компонента. Как это сделать?

**Ответ.** По умолчанию размер компонента определяется его содержимым, кроме того, при создании компонента действуют правила, зависящие от используемого диспетчера компоновки. Однако при необходимости вы можете явно указать предпочтительный размер компонента. Для этого надо вызвать метод `setPreferredSize()`, определенный в классе `JComponent`. Объявление этого метода выглядит следующим образом:

```
void setPreferredSize(Dimension newPS)
```

С помощью параметра `newPS` задаются новые предпочтительные размеры компонента. Класс `Dimension` входит в состав пакета `java.awt`. Один из его конструкторов выглядит так, как показано ниже.

```
Dimension(int w, int h)
```

Здесь параметр `w` задает ширину, а параметр `h` — высоту.

После того как вы зададите предпочтительный размер, диспетчер компоновки будет использовать эти сведения при размещении компонента. Следует помнить, что некоторые диспетчеры компоновки, например `GridLayout`, не учитывают предпочтительные размеры.

Существует также возможность установки минимального и максимального размеров. Этой цели служат методы `setMinimumSize()` и `setMaximumSize()`, которые также определены в классе `JComponent`. Метод `setMaximumSize()` определяется следующим образом:

```
void setMaximumSize(Dimension newSize)
```

Подобно методу `setPreferredSize()`, размеры, задаваемые посредством `setMinimumSize()` и `setMaximumSize()`, носят “рекомендательный” характер, и диспетчер компоновки может не учитывать их.

Помимо выравнивания содержимого меток, в данной программе имеется еще одно новшество. В ней используется диспетчер компоновки `GridLayout`. Это диспетчер создает таблицу, состоящую из прямоугольных ячеек, в каждой из которых располагается отдельный компонент. Размеры всех ячеек одинаковы; в соответствии с ними устанавливаются размеры компонентов. Другими словами, компонент полностью заполняет отведенную для него ячейку. В результате диспетчер `GridLayout` оказывается очень удобным для демонстрации выравнивания содержимого текстовых меток, так как он позволяет установить размер метки больше размера текстовой строки, содержащейся в составе компонента. В этом одно из отличий `GridLayout` от других диспетчеров, например `FlowLayout`, в котором по умолчанию размер метки устанавливается в точности таким, как необходимо для того, чтобы разместить в ней требуемый текст.

В классе `GirdLayout` определены три конструктора. Один из них имеет следующий вид:

```
GridLayout(int rows, int columns, int horzGap, int vertGap)
```

Здесь число строк и столбцов таблицы определяется посредством параметров `rows` и `columns`. Размер свободного пространства (в пикселях) между соседними строками и между соседними столбцами задается с помощью параметров `vertGap` и `horzGap`. Если значение `rows` равно нулю, число строк определяется количеством компонентов, включенных в таблицу. Если нулевое значение имеет параметр `column`, то число компонентов, помещенных в таблицу, определяет количество столбцов. Очевидно, что параметры `rows` и `columns` не могут одновременно быть равными нулю.

Приведенное ниже выражение создает таблицу, содержащую три строки и три столбца. Зазор между ячейками равен четырем пикселям.

```
jfrm.getContentPane().setLayout(new GridLayout(3, 3, 4, 4));
```

По умолчанию компоненты включаются в таблицу, начиная с левого верхнего угла. Заполнение таблицы осуществляется построчно слева направо, после заполнения очередной строки происходит переход на следующую строку. (На самом деле таблица может заполняться и справа налево, это зависит от ориентации, заданной для контейнера. Для европейских языков принято направление заполнения слева направо.) При желании вы можете поэкспериментировать с таблицами других размеров, например  $2 \times 5$ , и посмотреть, как изменится внешний вид окна.

ВАЖНО!

2.3

## Включение графического изображения в состав метки

В Swing содержимое меток не ограничивается текстом. В составе меток также может находиться графическое изображение либо одновременно и изображение и текст. С помощью меток удобно отображать, например, логотипы или фотоснимки. Правильно используя метки, можно обеспечить качественный внешний вид графического пользовательского интерфейса.

Для того, чтобы включить изображение в состав метки, надо использовать один из приведенных ниже конструкторов `JLabel`.

```
JLabel (Icon icon)
```

```
JLabel (Icon icon, int horzAlign)
```

Изображение передается конструктору посредством параметра `icon`. Объект должен представлять собой экземпляр класса, реализующего интерфейс `javax.swing.Icon`. (Как вы вскоре убедитесь, проще всего использовать для этой цели класс `javax.swing.ImageIcon`.) По умолчанию пиктограмма выравнивается по центру метки, однако вы можете также задавать горизонтальное выравнивание, используя второй из приведенных выше конструкторов, или применять методы, рассмотренные в предыдущем разделе.

Если вам надо включить в состав метки и текст и изображение, вам потребуется конструктор

```
JLabel (String str, Icon icon, int horzAlign)
```

Здесь параметр `str` определяет строку текста, а параметр `icon` задает изображение. Горизонтальное выравнивание конструкции, представляющей собой сочетание текста и изображения, задается посредством параметра `horzAlign`.

При создании метки, содержащей и текст и изображение, картинка располагается на "ведущем крае" текста. Взаимное размещение текста и графики можно изменить, используя приведенные ниже методы.

```
void setVerticalTextPosition(int loc)
```

```
void setHorizontalTextPosition(int loc)
```

В методе `setVerticalTextPosition()` значение параметра `loc` должно быть равно одной из рассмотренных ранее констант, применяющихся для вертикального выравнивания. Для метода `setHorizontalTextPosition()` параметр с этим же именем должен представлять собой константу, предназначенную для горизонтального выравнивания. Например, для того, чтобы в со-

ставе метки разместить текст над изображением, надо использовать следующее выражение:

```
jlab.setVerticalTextPosition(SwingConstants.TOP);
```

Получить изображение можно различными способами. Проще всего сделать это, применяя класс `ImageIcon`, содержащийся в пакете `javax.swing`. Класс `ImageIcon` реализует интерфейс `Icon`, поэтому экземпляр данного класса может быть передан конструктору `JLabel` в качестве изображения. В классе `ImageIcon` предусмотрено несколько конструкторов для создания изображений. Здесь мы будем использовать тот из них, который приведен ниже.

```
ImageIcon(string filename)
```

При вызове данного конструктора параметр `filename` определяет файл, содержащий графическое изображение. Файл должен содержать изображение в одном из форматов, поддерживаемых Java, например GIF или JPEG.

Рассмотрим программу, демонстрирующую использование изображений в составе меток. Программа иллюстрирует возможность изменения относительных позиций текста и изображения. Выходные данные, отображаемые при работе программы, показаны на рис. 2.3.

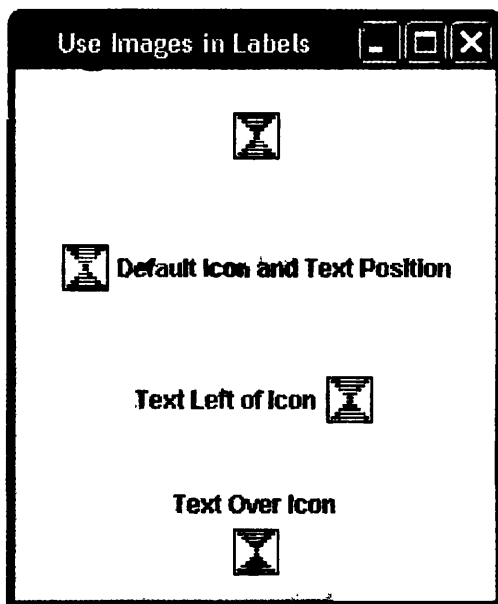


Рис. 2.3. Окно, отображаемое программой, демонстрирующей использование изображений в составе меток

```
// Использование изображений в составе меток
import javax.swing.*;
import java.awt.*;
import javax.swing.border.*;

class IconLabelDemo {

    IconLabelDemo() {

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Use Images in Labels");

        // Установка диспетчера компоновки GridLayout,
        // формирующего таблицу из четырех строк и одного столбца.
        jfrm.getContentPane().setLayout(new GridLayout(4, 1));

        // Установка начального размера фрейма.
        jfrm.setSize(250, 300);

        // Завершение программы при закрытии приложения пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание объекта ImageIcon на базе изображения,
        // загруженного из файла.
        ImageIcon myIcon = new ImageIcon("myIcon.gif");

        // Создание метки, содержащей пиктограмму.
        JLabel jlabIcon = new JLabel(myIcon);

        // Создание метки, содержащей текст и изображение.
        JLabel jlabIconTxt =
            new JLabel("Default Icon and Text Position",
                       myIcon,
                       SwingConstants.CENTER);

        // Создание метки с текстом и изображением и размещение
        // текста слева от изображения.
        JLabel jlabIconTxt2 = new JLabel("Text Left of Icon", myIcon,
                                           SwingConstants.CENTER);
        jlabIconTxt2.setHorizontalTextPosition(SwingConstants.LEFT);

        // Создание метки с текстом и изображением и размещение
        // текста над изображением.
```



```
JLabel jlabIconTxt3 = new JLabel("Text Over Icon", myIcon,
                                SwingConstants.CENTER);
jlabIconTxt3.setVerticalTextPosition(SwingConstants.TOP);
jlabIconTxt3.setHorizontalTextPosition(SwingConstants.CENTER);

// Включение меток в состав панели содержимого.
jfrm.getContentPane().add(jlabIcon);
jfrm.getContentPane().add(jlabIconTxt);
jfrm.getContentPane().add(jlabIconTxt2);
jfrm.getContentPane().add(jlabIconTxt3);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Создание фрейма в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new IconLabelDemo();
        }
    });
}
```

Обратите внимание на получение изображения, используемого программой, и создание метки. При выполнении приведенного ниже выражения изображение загружается из файла `myIcon.gif` и оформляется в виде объекта `ImageIcon` с именем `myIcon`.

```
ImageIcon myIcon = new ImageIcon("myIcon.gif");
```

Затем в программе следует выражение, которое использует объект `myIcon` при создании метки, включающей изображение.

```
JLabel jlabIcon = new JLabel(myIcon);
```

В результате формируется метка, содержащая только пиктограмму, которая по умолчанию отображается в центре компонента. В программе создаются еще три метки, каждая из которых содержит текст и изображение с разным взаимным расположением. Например, следующий фрагмент кода формирует метку, в которой текст располагается над изображением и выравнивается в горизонтальном направлении по центру:

```
JLabel jlabIconTxt3 = new JLabel("Text Over Icon", myIcon,  
                                SwingConstants.CENTER);  
jlabIconTxt3.setVerticalTextPosition(SwingConstants.TOP);  
jlabIconTxt3.setHorizontalTextPosition(SwingConstants.CENTER);
```

## Вопросы для текущего контроля

1. Какие действия выполняет метод `setHorizontalAlignment()`?
2. Какой диспетчер компоновки располагает компоненты в виде таблицы?
3. С помощью какого метода можно задать вертикальную позицию текста относительно изображения в составе метки?

**ВАЖНО!**

## 2.4. Деактивизация метки

Метки часто используются для описания других интерфейсных элементов, например полей редактирования или кнопок. Когда доступ к элементу, описываемому меткой, запрещается, желательно изменить внешний вид и самой метки. Средствами Swing это сделать достаточно легко.

Активное или неактивное состояние метки устанавливается путем вызова метода `setEnabled()`, определенного в классе `JComponent`. Объявление этого метода выглядит так:

```
void setEnabled(boolean state)
```

Если значение параметра `state` равно `true`, метка активна. Значение `false` деактивизирует метку.

Если метка неактивна, текст отображается серым цветом. Если в метке содержится изображение, то его цвет также изменяется. При необходимости вы можете задать отдельное изображение, которое будет выводиться в неактивной метке. Для этого предназначен метод `setDisabledIcon()`.

```
void setDisabledIcon(Icon icon)
```

С помощью параметра `icon` задается изображение, которое будет выводиться в составе метки при ее деактивизации.

1. Этот метод устанавливает горизонтальное выравнивание содержимого метки в ее пределах.
2. `GridLayout`
3. `setVerticalTextPosition()`.

Рассмотрим программу, демонстрирующую неактивные метки. В программе создаются три метки, содержащие как текст, так и изображение. Для сравнения первая метка отображается активной. Вторая метка неактивна, но для нее не определяется отдельная пиктограмма, так что основное изображение становится более тусклым. Третья метка также неактивна, но для нее задана отдельная пиктограмма. Внешний вид окна, отображаемого программой, показан на рис. 2.4.



*Рис. 2.4. Результаты действия программы, демонстрирующей неактивные метки*

```
// Использование неактивных меток
import javax.swing.*;
import java.awt.*;
import javax.swing.border.*;

class DisabledLabelDemo {

    DisabledLabelDemo() {

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Use Images in Labels");

        // Установка диспетчера компоновки GridLayout,
        // формирующего таблицу из трех строк и одного столбца.
        jfrm.getContentPane().setLayout(new GridLayout(3, 1));
```

```
// Установка начального размера фрейма.
jfrm.setSize(240, 250);

// Завершение программы при закрытии приложения пользователем.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Загрузка пиктограмм из файла.
ImageIcon myIcon = new ImageIcon("myIcon.gif");
// Загрузка изображения, которое будет выводиться
// для деактивизированной метки.
ImageIcon myDisIcon = new ImageIcon("myDisIcon.gif");

// Создание метки, содержащей текст и изображение.
JLabel jlabIconTxt = new JLabel("This label is enabled.",
                                myIcon,
                                SwingConstants.CENTER);

// Создание и деактивизация метки,
// содержащей текст и изображение.
JLabel jlabIconTxt2 = new JLabel("This label is disabled.",
                                myIcon,
                                SwingConstants.CENTER);
jlabIconTxt2.setEnabled(false); // Деактивизация метки.
    // Метка будет отображаться более тусклым цветом.

// Создание и деактивизация метки, содержащей текст
// и изображение. На этот раз в составе метки выводится
// специальная пиктограмма.
JLabel jlabIconTxt3 = new JLabel("Use the disabled icon.",
                                myIcon,
                                SwingConstants.CENTER);
jlabIconTxt3.setDisabledIcon(myDisIcon); //Установка
    // пиктограммы, которая должна отображаться
    // при деактивизации метки.
jlabIconTxt3.setEnabled(false); // Деактивизация метки.
    // Метка будет выводиться более тусклым цветом,
    // и в ее составе будет отображаться пиктограмма,
    // специально предусмотренная для данной ситуации.

// Включение меток в состав панели содержимого.
jfrm.getContentPane().add(jlabIconTxt);
jfrm.getContentPane().add(jlabIconTxt2);
```

```

jfrm.getContentPane().add(jlabIconTxt3);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Создание фрейма в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new DisabledLabelDemo();
        }
    });
}
}

```

**ВАЖНО!****2.5**

## **Отображение мнемонических обозначений клавиш в составе меток**

Чаще всего метки используются для описания назначения других компонентов, например полей редактирования. Если поле предназначено для ввода имени, перед ним часто указывают метку, отображающую строку Name. В подобной ситуации уместно включить в состав метки мнемоническое обозначение командной клавиши, которая позволит быстро передать фокус ввода. Для поля ввода имени это может быть клавиша <N> (первая буква в слове name). Нажатие клавиши <N> вместе с клавишей <Alt> передает фокус ввода полю редактирования.

Процесс определения мнемонического обозначения для метки включает в себя два этапа. В первую очередь задается мнемонический символ. Затем следует связать с меткой компонент, который должен получить фокус ввода. Выполнение обоих действий не вызывает затруднений.

Для того чтобы отобразить мнемоническое обозначение в составе метки, надо вызвать метод `setDisplayMnemonic()`, определенный в классе `JLabel`.

```
void setDisplayedMnemonic(int ch)
```

Здесь параметр `ch` определяет символ, выполняющий функции командной клавиши. Обычно такой символ отображается с подчеркиванием. Если в тексте метки содержится несколько одинаковых символов, подчеркнут будет первый из них.

После установки мнемонического обозначения надо связать метку с компонентом, который будет получать фокус ввода при нажатии командной клавиши. Для этой цели служит метод `setLabelFor()`, который также определен в классе `JLabel`.

```
void setLabelFor(Component comp)
```

где `comp` — это ссылка на компонент, который получит фокус в том случае, если командная клавиша будет нажата в комбинации с клавишей `<Alt>`.

Ниже приведен текст программы, демонстрирующей использование мнемонических обозначений. В программе создаются два поля редактирования: одно из них помечено текстом **E-mail Address**, а другое — строкой **Name**. В метках, описывающих эти поля, мнемоническим обозначением для **E-mail Address** является **e**, а мнемоническим обозначением для **Name** — **n**. Когда пользователь нажмет комбинацию клавиш `<Alt+N>`, поле **Name** получит фокус ввода. При нажатии `<Alt+E>` фокус ввода будет передан полю **E-mail Address**. Выходные данные, отображаемые при работе программы, показаны на рис. 2.5.

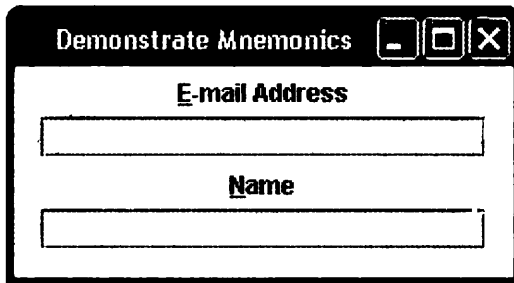


Рис. 2.5. Окно, которое отображается в процессе работы программы, демонстрирующей использование мнемонических обозначений

```
// Использование мнемонических обозначений
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class MnemDemo {

    MnemDemo () {

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Demonstrate Mnemonics");
```

```
// Установка диспетчера компоновки FlowLayout.
jfrm.getContentPane().setLayout(new FlowLayout());

// Установка начального размера фрейма.
jfrm.setSize(260, 140);

// Завершение программы при закрытии приложения пользователем.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Создание двух меток.
JLabel jlab1 = new JLabel("E-mail Address");
JLabel jlab2 = new JLabel("Name");

// Назначение мнемонических обозначений меткам.
jlab1.setDisplayedMnemonic('e');
jlab2.setDisplayedMnemonic('n');

// Создание двух полей редактирования.
JTextField jtf1 = new JTextField(20);
JTextField jtf2 = new JTextField(20);

// Связывание меток с компонентами.
jlab1.setLabelFor(jtf1);
jlab2.setLabelFor(jtf2);

// Назначение команд действий полям редактирования.
jtf1.setActionCommand("jtf1");
jtf2.setActionCommand("jtf2");

// Включение полей редактирования и меток в состав
// панели содержимого.
jfrm.getContentPane().add(jlab1);
jfrm.getContentPane().add(jtf1);
jfrm.getContentPane().add(jlab2);
jfrm.getContentPane().add(jtf2);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Создание фрейма в потоке обработки событий.
```

```
SwingUtilities.invokeLater(new Runnable() {  
    public void run() {  
        new MnemDemo();  
    }  
});  
}
```

## Получение и установка текста метки

Как было сказано в модуле 1, после создания метки есть возможность изменять текст, отображаемый в ее составе. Для этой цели предусмотрен метод `setText()`:

```
void setText(String str)
```

Здесь параметр `str` задает строку, которая должна выводиться в составе метки. Получить текст, отображаемый в метке, можно, вызвав метод `getText()`:

```
String getText()
```

В результате выполнения этого метода возвращается строка, отображаемая в составе метки в данный момент.

## Использование HTML-кода в составе метки

Перед тем как закончить разговор о метках, необходимо упомянуть еще об одной возможности, предоставляемой Swing. Для определения текста в составе метки можно использовать HTML-код. В этом случае строка текста должна начинаться последовательностью символов `<html>`. В результате текст автоматически будет сформатирован с учетом HTML-дескрипторов, содержащихся в нем. Одним из преимуществ такого подхода является возможность создания меток, отображающих несколько строк текста. Например, ниже приведено выражение, с помощью которого создается метка, представляющая две строки: первой выводится строка `Top`, а под ней — строка `Bottom`.

```
JLabel jlabhtml = new JLabel("<html>Top<br>Bottom");
```



**Спросим у опытного программиста**

**Вопрос.** При использовании мнемонических обозначений первое вхождение выбранного символа отображается с подчеркиванием. А что делать, если надо подчеркнуть второе или третье вхождение символа? Например, как сделать, чтобы в метке Enter E-mail Address с подчеркиванием отображалась буква E в слове E-mail?

**Ответ.** При необходимости вы можете явным образом указать индекс мнемонического обозначения, вызвав метод `setDisplayMnemonicIndex()`:

```
void setDisplayedMnemonicIndex(int idx)
    throws IllegalArgumentException
```

Здесь параметр `idx` задает индекс, т.е. порядковый номер символа, который должен отображаться с подчеркиванием. Так, чтобы в метке Enter E-mail Address подчеркнуть букву E в слове E-mail, вам надо использовать выражение, подобное приведенному ниже.

```
jlab.setDisplayedMnemonicIndex(6);
```

HTML-код может быть использован не только в метках, но и в других компонентах. Если интерфейсный элемент отображает текст, он может быть задан с помощью HTML-кода. В проекте 2.1 вы увидите пример использования HTML-кода для кнопки.

**Вопросы для текущего контроля**

1. Поскольку метки лишь отображают информацию, их нельзя деактивировать. Да или нет?
2. Предположим, что вы хотите использовать HTML-текст в составе метки. С какой последовательности символов должна начинаться строка?
3. Какие действия выполняет метод `setLabelFor()`?

- 
1. Нет.
  2. `<html>`.
  3. Он связывает метку с другим элементом, который получает фокус ввода при нажатии командной клавиши, определенной посредством метки.

## Спросим у опытного программиста

**Вопрос.** Я слышал, что объекты Swing являются компонентами JavaBeans. Так ли это?

**Ответ.** Да, компоненты Swing являются и компонентами JavaBeans. Если вы незнакомы со спецификацией JavaBeans, не стоит беспокоиться, поскольку, для того, чтобы изучить и использовать материал, приведенный в данной книге, совсем не обязательно знать что-то о JavaBeans. Тем не менее полезно знать, что все компоненты Swing удовлетворяют требованиям, изложенным в спецификации JavaBeans, и следуют соглашению об именах свойств. Так, для свойств, допускающих чтение и запись, определены методы, имена которых начинаются с `get` и `set`. Для свойств, предназначенных для чтения, определены лишь `get`-методы, а если свойство допускает только запись, для него доступен лишь `set`-метод. Например, `JLabel` (как и многие другие компоненты) предоставляет методы `setText()` и `getText()`. Это означает, что в `JLabel` поддерживается свойство `text`, допускающее чтение и запись. В JavaBeans поддерживаются также `is`-методы. Они определены для логических свойств и возвращают значение `true` или `false`. Например, метод `isEnabled()` возвращает значение `true`, если компонент активизирован. JavaBeans — важное средство, используемое при создании Java-программ. Если вы знакомы с ним, у вас есть возможность расширить свои знания.

**ВАЖНО!**

## 2.6 Общие сведения о кнопках

В модуле 1 были приведены сведения о классе `JButton`. Однако `JButton` — лишь один из классов Swing, представляющих различные типы кнопок. Все доступные классы, предназначенные для поддержки кнопок, перечислены ниже.

|                            |                                                 |
|----------------------------|-------------------------------------------------|
| <code>JButton</code>       | Обычная кнопка                                  |
| <code>JToggleButton</code> | Кнопка с двумя состояниями (включено/выключено) |
| <code>JCheckBox</code>     | Флажок опции                                    |
| <code>JRadioButton</code>  | Группа переключателей опций                     |

Все классы кнопок являются подклассами класса `AbstractButton`, который, в свою очередь, расширяет класс `JComponent`. В классе `AbstractButton` содержится много методов, поддерживающих основные функциональные возможности, общие для всех кнопок. С помощью этих методов можно размещать надписи на кнопках, делать кнопки неактивизированными, задавать мнемонические обозначения и производить выравнивание. В классе `AbstractButton` также определены методы, позволяющие задавать пиктограммы для отображения на кнопке в различных состояниях: деактивизированной, нажатой, выбранной, кнопке, на которой размещен курсор мыши. Очевидно, что в классе `AbstractButton` должны быть также предусмотрены методы для включения и удаления обработчиков событий, связанных с кнопками. Методы для работы с кнопками описаны в табл. 2.2. Многие из них будут использованы в данном модуле.



На заметку

Класс `AbstractButton` также является суперклассом класса `JMenuItem`, но разговор о меню мы начнем лишь в модуле 7.

Таблица 2.2. Часто используемые методы, определенные в классе `AbstractButton`

| Метод                                                        | Описание                                                                                                                                                                                                                                                                                             |
|--------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void addActionListener<br/>ActionListener al)</code>   | Регистрирует обработчик для событий действий                                                                                                                                                                                                                                                         |
| <code>void addChangeListener<br/>ChangeListener cl)</code>   | Регистрирует обработчик для событий изменения состояния                                                                                                                                                                                                                                              |
| <code>void addItemListener<br/>(ItemListener il)</code>      | Регистрирует обработчик для событий элемента                                                                                                                                                                                                                                                         |
| <code>void doClick()</code>                                  | Имитирует щелчок на кнопке                                                                                                                                                                                                                                                                           |
| <code>String getText()</code>                                | Возвращает текст, отображаемый на кнопке                                                                                                                                                                                                                                                             |
| <code>ButtonModule getModel()</code>                         | Возвращает модель для кнопки                                                                                                                                                                                                                                                                         |
| <code>boolean isSelected()</code>                            | Возвращает значение <code>true</code> , если кнопка выбрана и значение <code>false</code> — в противном случае. (Применяется для кнопок с двумя состояниями.)                                                                                                                                        |
| <code>void setEnabled<br/>(boolean state)</code>             | Если значение <code>state</code> равно <code>true</code> , кнопка доступна. Значение <code>false</code> деактивизирует кнопку.                                                                                                                                                                       |
| <code>void setDisabledIcon<br/>(Icon icon)</code>            | Устанавливает пиктограмму, которая должна отображаться тогда, когда кнопка деактивизирована. Изображение задается с помощью параметра <code>icon</code>                                                                                                                                              |
| <code>void setHorizontalAlignment<br/>(int horzAlign)</code> | Задаёт выравнивание текста или изображения на кнопке по горизонтали. Значение <code>horzAlign</code> должно быть равно одной из констант, определенных в <code>SwingConstants</code> : <code>CENTER</code> , <code>LEFT</code> , <code>RIGHT</code> , <code>LEADING</code> или <code>TRAILING</code> |

## Окончание табл. 2.2.

| Метод                                                 | Описание                                                                                                                                                                                                                                                                                                                                               |
|-------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void setHorizontalTextPosition(int loc)</code>  | Применяется для кнопок, содержащих текст и изображение и задает расположение текста относительно изображения по горизонтали. Значение <code>loc</code> должно быть равно одной из констант, определенных в <code>SwingConstants</code> : <code>CENTER</code> , <code>LEFT</code> , <code>RIGHT</code> , <code>LEADING</code> или <code>TRAILING</code> |
| <code>void setMnemonic(int ch)</code>                 | Определяет для кнопки мнемоническое обозначение, определяемое параметром <code>ch</code>                                                                                                                                                                                                                                                               |
| <code>void setPressedIcon(Icon icon)</code>           | Устанавливает пиктограмму, которая должна отображаться тогда, когда кнопка находится в нажатом состоянии. Изображение задается с помощью параметра <code>icon</code>                                                                                                                                                                                   |
| <code>void setRolloverIcon(Icon icon)</code>          | Устанавливает пиктограмму, которая должна отображаться тогда, когда курсор находится над кнопкой. Изображение задается с помощью параметра <code>icon</code>                                                                                                                                                                                           |
| <code>void setRolloverSelectedIcon(Icon icon)</code>  | Устанавливает пиктограмму, которая должна отображаться тогда, когда курсор находится над кнопкой, находящейся в выбранном состоянии. Изображение задается с помощью параметра <code>icon</code> . (Применяется для кнопок с двумя состояниями.)                                                                                                        |
| <code>void setSelected(boolean state)</code>          | Если значение параметра <code>state</code> равно <code>true</code> , кнопка выбирается. Если значение <code>state</code> равно <code>false</code> , выбор кнопки отменяется. (Применяется для кнопок с двумя состояниями.)                                                                                                                             |
| <code>void setSelectedIcon(Icon icon)</code>          | Устанавливает пиктограмму, которая должна отображаться тогда, когда кнопка выбрана. (Применяется для кнопок с двумя состояниями.)                                                                                                                                                                                                                      |
| <code>void setText(String str)</code>                 | Устанавливает текст для отображения на кнопке. Строка текста задается с помощью параметра <code>str</code>                                                                                                                                                                                                                                             |
| <code>void setVerticalAlignment(int vertAlign)</code> | Задаёт выравнивание текста или изображения на кнопке по вертикали. Значение параметра <code>vertAlign</code> должно быть равно одной из констант, определенных в <code>SwingConstants</code> : <code>CENTER</code> , <code>TOP</code> или <code>BOTTOM</code>                                                                                          |
| <code>void setVerticalTextPosition(int loc)</code>    | Для кнопок, содержащих и текст и пиктограмму, данный метод задает вертикальную позицию текста относительно изображения. Значение параметра <code>loc</code> должно быть равно одной из констант, определенных в <code>SwingConstants</code> : <code>CENTER</code> , <code>TOP</code> или <code>BOTTOM</code>                                           |

Модель, используемая при создании кнопок и их поддержке, определена посредством интерфейса `ButtonModel`. Реализацией этого интерфейса с характеристиками, принятыми по умолчанию, является класс `DefaultButtonModel`. Интерфейс `ButtonModel` определяет некоторые свойства, доступные посредством `get`- и `set`-методов. В большинстве случаев у разработчиков не возникает необходимости непосредственного взаимодействия с моделью кнопки. Тем не менее она доступна, и в проекте 2.1 показано, как работать с ней.

События, связанные с кнопками, можно условно разделить на три категории: *события действий* (action event), *события элемента* (item event) и *события изменения состояния* (change event). События действий генерируются при выполнении пользователем некоторых действий, например, по щелчку на кнопке. События элемента генерируются при выборе кнопки или отмене выбора. События изменения состояния, как и следует из их названия, возникают тогда, когда состояние кнопки изменяется. Заметьте, что существуют кнопки или варианты кнопок, для которых некоторые типы событий не применяются. Например, объекты `JButton` не генерируют события элемента. Перед тем как начинать подробное обсуждение кнопок Swing, необходимо рассмотреть события, которые они могут генерировать.

### Обработка событий действий

Событие действия генерируется тогда, когда пользователь щелкает на кнопке. Как было сказано в модуле 1, события действий представляются классом `ActionEvent`. Для обработки событий данного типа используются классы, реализующие интерфейс `ActionListener`. Этот интерфейс определяет лишь один метод, `actionPerformed()`, который объявляется следующим образом:

```
void actionPerformed(ActionEvent ae)
```

Конкретное событие передается посредством параметра `ae`.

Объект `ActionEvent`, передаваемый методу `actionPerformed()`, предоставляет доступ к информации о событии. Наиболее важными, наверное, являются те данные, которые позволяют идентифицировать компонент, сгенерировавший событие. Идентифицировать этот компонент можно двумя способами: по строке, представляющей команду действия (см. модуль 1), или посредством ссылки на объект. Строку команды действия для компонента, сгенерировавшего событие, можно получить, вызвав метод `getActionCommand()` объекта `ActionEvent`. Ниже приведено объявление этого метода.

```
String getActionCommand()
```

Например, по щелчку на кнопке генерируется событие действия, для которого строка команды совпадает с текстом, отображаемым на этой кнопке.

Команды действий, вероятно, являются наиболее универсальным способом идентификации компонентов, генерирующих события действий. Однако существует и другой способ: вы можете определить компонент, получая ссылку на него. Для этого надо вызвать метод `getSource()` объекта `ActionEvent`. Этот метод определен в классе `EventObject`, который является суперклассом для всех классов, описывающих события. Объявление этого метода выглядит следующим образом:

```
Object getSource()
```

Преимущество использования метода `getSource()` состоит в том, что если вам потребуется выполнять действия непосредственно с компонентом, вы можете сделать это с помощью данной ссылки.

Обнаружить, была ли нажата какая-либо из клавиш-модификаторов (например, `<Alt>`, `<Ctrl>`, `<META>`, и `<Shift>`), позволяет метод `getModifiers()`. Он возвращает целочисленное значение (`int`), указывающее на то, какие из клавиш-модификаторов были нажаты в момент генерации события. Заголовок этого метода выглядит так:

```
int getModifiers()
```

Возвращаемое значение формируется посредством объединения констант `ALT_MASK`, `CTRL_MASK`, `META_MASK` и `SHIFT_MASK`, объявленных в классе `ActionEvent`.

Для некоторых приложений имеет значение время возникновения события. Получить эту информацию можно, вызвав метод `getWhen()`, который имеет следующий вид:

```
long getWhen()
```

## Обработка событий элемента

События элемента возникают при установке или сбросе флажка опций, а также при изменении состояния переключателя опций. (Обычная кнопка не генерирует события данного типа.) События элемента представляются посредством класса `ItemEvent`. Для обработки этих событий используются классы, реализующие интерфейс `ItemListener`. В данном интерфейсе объявлен лишь один метод, `itemStateChanged()`, заголовок которого приведен ниже.

```
void itemStateChanged(ItemEvent ie)
```

Конкретное событие представляется параметром `ie`.

Для того чтобы получить ссылку на элемент, претерпевший изменения, надо вызвать метод `getItem()` объекта `ItemEvent`. Ниже приведено объявление этого метода.

```
Object getItem()
```

Тип возвращаемой ссылки необходимо привести к типу компонента, например `JCheckBox` или `JRadioButton`.

В результате вызова метода `getItemSelectable()` класса `ItemEvent` возвращается ссылка `ItemSelectable` на объект, сгенерировавший событие. Заголовок этого метода имеет следующий вид:

```
ItemSelectable getItemSelectable()
```

`ItemSelectable` — где интерфейс, который реализуют компоненты, допускающие выбор.

При возникновении события элемента компонент может быть в одном из двух состояний: выбран или не выбран. В классе `ItemEvent` определены следующие статические целочисленные константы (`static int`), представляющие эти два состояния:

```
SELECTED      DESELECTED
```

Чтобы получить информацию о новом состоянии, надо вызвать метод `getStateChange()`, определенный в классе `ItemEvent`. Объявление этого метода выглядит следующим образом:

```
int getStateChange()
```

Метод возвращает либо значение `ItemEvent.SELECTED`, либо `ItemEvent.DESELECTED`. Выяснить состояние кнопки (выбрана или не выбрана) можно также путем вызова метода `isSelected()`, определенного в классе `AbstractButton`.

## Обработка событий изменения состояния

События изменения состояния возникают при изменении модели компонента. Они представляются экземплярами класса `ChangeEvent`. Для обработки этих событий используются классы, реализующие интерфейс `ChangeListener`. В данном интерфейсе объявлен лишь один метод, `stateChanged()`, заголовок которого имеет следующий вид:

```
void stateChanged(ChangeEvent ce)
```

Конкретное событие представляется параметром `ce`.

Для того чтобы идентифицировать компонент, сгенерировавший событие `ChangeEvent`, надо вызвать метод `getSource()`. Доступ к этому методу предоставляет объект события, передаваемый в качестве параметра методу `stateChanged()`. Метод `getSource()`, определенный в классе `EventObject`, обсуждался выше.

## Подробные сведения о классе `JButton`

Как было сказано в модуле 1, класс `JButton` представляет обычную кнопку. По щелчку на кнопке она передает событие `ActionEvent` всем зарегистрированным обработчикам. Кнопка данного типа также передает и событие `ChangeEvent`, но необходимость обрабатывать его, как правило, не возникает, так как для большинства приложений важен лишь факт щелчка на кнопке. Помимо выполнения этих основных функций, класс `JButton` предоставляет средства детального управления внешним видом и поведением кнопки. Например, кнопка может содержать изображение, ее можно деактивизировать, есть возможность выравнивать надпись на кнопке по горизонтали и по вертикали. Кроме того, можно задавать отдельные изображения для различных состояний кнопки. Большинство средств управления кнопкой унаследовано от класса `AbstractButton`. Конструкторы `JButton` перечислены в табл. 2.3.

Таблица 2.3. Конструкторы класса `JButton`

| Конструктор                                 | Описание                                                                                                                                                                  |
|---------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JButton()</code>                      | Создает кнопку без надписи                                                                                                                                                |
| <code>JButton(String str)</code>            | Создает кнопку, на которой отображается текст, заданный посредством параметра <code>str</code>                                                                            |
| <code>JButton(String str, Icon icon)</code> | Создает кнопку, на которой отображается текст, заданный посредством параметра <code>str</code> , и изображение, указанное с помощью параметра <code>icon</code>           |
| <code>JButton(Icon icon)</code>             | Создает кнопку, на которой отображается изображение, заданное с помощью параметра <code>icon</code>                                                                       |
| <code>JButton(Action act)</code>            | Создает кнопку, для которой текст, изображение и другие параметры задаются посредством параметра <code>act</code> . Класс <code>Action</code> будет рассмотрен в модуле 7 |



**ВАЖНО!**

## **2.7. Размещение изображения на кнопке**

В состав элемента `JButton` можно включать строку символов, пиктограмму или одновременно и текст, и изображение. Кроме того, различные состояния кнопки могут быть представлены посредством разных изображений. В модуле 1 рассматривались только кнопки с надписью в виде строки текста. Здесь мы обсудим возможность помещения на кнопку изображений.

Для того чтобы создать кнопку, содержащую пиктограмму, вам надо воспользоваться одним из приведенных ниже конструкторов.

```
JButton(Icon icon)
JButton(String str, Icon icon)
```

Здесь параметры `str` и `icon` представляют соответственно строку текста и пиктограмму, отображаемые на кнопке. По умолчанию пиктограмма размещается на “ведущим крае”, а текст располагается рядом с ней. При необходимости вы можете изменить взаимное расположение текста и изображения. Изображение, определяемое посредством приведенных выше конструкторов, является изображением по умолчанию. Если другие изображения не заданы, оно используется во всех случаях. Задать изображение или установить вместо одной пиктограммы другую позволяет метод `setIcon()`. Получить пиктограмму можно с помощью метода `getIcon()`. Оба этих метода определены в классе `AbstractButton`.

В дополнение к изображению по умолчанию, задаваемому при вызове конструктора, можно установить пиктограммы, которые будут отображаться при выполнении определенных действий с кнопкой. В частности, вы можете задать изображение, которое будет выводиться на деактивизированной кнопке, изображение для нажатой кнопки или изображение, которое будет отображаться тогда, когда курсор мыши будет располагаться на кнопке. Для этой цели используются следующие методы:

```
void setDisabledIcon(Icon disabledIcon)
void setPressedIcon(Icon pressedIcon)
void setRolloverIcon(Icon rolloverIcon)
```

После установки одного из этих изображений оно будет выводиться при возникновении соответствующего события. Заметьте, что в некоторых реализациях `Swing` не предусмотрено изменение внешнего вида кнопки при наведении на нее курсора мыши. Определить, поддерживается ли данная возможность, можно, вызвав метод `isRolloverEnabled()`. Вы также можете непосредственно установить данное свойство, вызвав метод `setRolloverEnabled(true)`.

Запретить или разрешить доступ к кнопке позволяет метод `setEnabled()`:

```
void setEnabled(boolean state)
```

Если значение параметра `state` равно `false`, кнопка становится неактивизированной. Если значение `state` равно `true`, доступ к кнопке разрешается. Неактивизированная кнопка отображается серым цветом. Если изображение для деактивизированной кнопки определено, оно выводится на экран. В противном случае выводится изображение по умолчанию, но отображается более тусклым цветом.

Определить, разрешено ли обращение к кнопке, можно, вызвав метод `isEnabled()`:

```
boolean isEnabled()
```

Этот метод унаследован от класса `Component`. Он возвращает значение `true`, если кнопка доступна, и `false` — в противном случае.

Очевидно, что для установки изображений для различных состояний кнопки потребуется дополнительная работа, но, поступая подобным образом, можно существенно улучшить внешний вид пользовательского интерфейса. (С другой стороны, нельзя забывать о том, что дополнительные изображения увеличивают время загрузки программы по сети.) Ниже приведен код программы, в которой создаются две кнопки и с каждой из них связывается пиктограмма. В процессе работы программы можно заметить, что при наведении курсора мыши на кнопку и при щелчке на кнопке изображение на ней меняется. При каждом щелчке на кнопке **First** состояние кнопки **Second** изменяется с активизированного на неактивизированное и наоборот. Когда кнопка неактивизирована, на ней выводится пиктограмма, соответствующая данному состоянию. Текущее изображение также заменяется новым и при щелчке на кнопке. Выходные данные, отображаемые при работе программы, показаны на рис. 2.6.

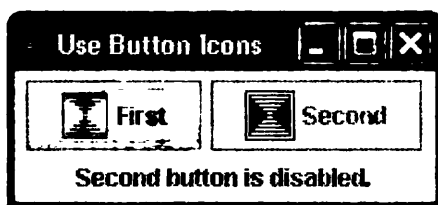


Рис. 2.6. Окно, отображаемое программой `ButtonIcons`

## 106 Модуль 2. Метки, кнопки и обрамление

---

```
// Определение пиктограмм для отображения на кнопке
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class ButtonIcons implements ActionListener {

    JLabel jlab;
    JButton jbtnFirst;
    JButton jbtnSecond;

    ButtonIcons() {
        // Загрузка пиктограмм, используемых JButton.
        ImageIcon myIcon = new ImageIcon("myIcon.gif");
        ImageIcon myDisIcon = new ImageIcon("myDisIcon.gif");
        ImageIcon myROIcon = new ImageIcon("myROIcon.gif");
        ImageIcon myPIcon = new ImageIcon("myPIcon.gif");

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Use Button Icons");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начального размера фрейма.
        jfrm.setSize(220, 100);

        // Завершение программы при закрытии приложения пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание текстовой метки.
        jlab = new JLabel("Press a button.");

        // Создание двух кнопок.
        // Для них задаются пиктограммы по умолчанию.
        jbtnFirst = new JButton("First", myIcon);
        jbtnSecond = new JButton("Second", myIcon);

        // Установка изображения, отображаемого при наведении
        // на кнопку курсора мыши.
        jbtnFirst.setRolloverIcon(myROIcon);
        jbtnSecond.setRolloverIcon(myROIcon);
    }
}
```

```
// Установка изображения, отображаемого на активизированной
// кнопке.
jbtnFirst.setPressedIcon(myPIcon);
jbtnSecond.setPressedIcon(myPIcon);

// Установка изображения, отображаемого на
// деактивизированной кнопке.
jbtnFirst.setDisabledIcon(myDisIcon);
jbtnSecond.setDisabledIcon(myDisIcon);

// Связывание с кнопками обработчиков событий.
jbtnFirst.addActionListener(this);
jbtnSecond.addActionListener(this);

// Включение кнопок в состав панели содержимого.
jfrm.getContentPane().add(jbtnFirst);
jfrm.getContentPane().add(jbtnSecond);

// Включение метки в состав панели содержимого.
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

// Обработка событий кнопок.
public void actionPerformed(ActionEvent ae) {

    if(ae.getActionCommand().equals("First")) {
        jlab.setText("First button was pressed.");
        if(jbtnSecond.isEnabled()) {
            jlab.setText("Second button is disabled.");
            jbtnSecond.setEnabled(false);
        }
        else {
            jlab.setText("Second button is enabled.");
            jbtnSecond.setEnabled(true);
        }
    } else
        jlab.setText("Second button was pressed. ");
}

public static void main(String args[]) {
```

```
// Создание фрейма в потоке обработки событий.
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        new ButtonIcons();
    }
});
}
```

В данной программе создаются четыре пиктограммы: `myIcon` (изображение по умолчанию), `myDisIcon` (кнопка деактивизирована), `myROIcon` (курсор мыши находится на кнопке) и `myPIcon` (кнопка активизирована). Эти изображения связываются с кнопками `jbtnFirst` и `jbtnSecond`. Таким образом, для кнопок задаются изображения, соответствующие каждому из возможных состояний. Затем объект `this` регистрируется как обработчик событий, связанных с кнопками, и кнопки добавляются на панель содержимого.

Обратите внимание на то, что класс `ButtonIcons` реализует интерфейс `ActionListener`. Как было сказано ранее, в интерфейсе `ActionListener` объявлен только один метод, `actionPerformed()`. Этот метод определен в `ButtonIcons`. Поскольку класс `ButtonIcons` реализует интерфейс `ActionListener`, он имеет возможность принимать оповещения о событиях действий. Именно по этой причине допустимо использовать объект `this` в качестве параметра метода `addActionListener()`.

Метод `actionPerformed()` обрабатывает события действий для обеих кнопок. При каждом щелчке на кнопке `First` состояние кнопки `Second` переключается между активизированным и деактивизированным. Данное поведение реализуется посредством представленного ниже фрагмента кода.

```
if(ae.getActionCommand().equals("First")) {
    jlab.setText("First button was pressed.");
    if(jbtnSecond.isEnabled()) {
        jlab.setText("Second button is disabled.");
        jbtnSecond.setEnabled(false);
    }
    else {
        jlab.setText("Second button is enabled.");
        jbtnSecond.setEnabled(true);
    }
}
```

Посредством объекта, описывающего событие, извлекается команда действия, которая затем сравнивается со значением "First", т.е. с командой действия для первой кнопки. (Как вы помните, по умолчанию команда действия для кнопки представляет собой строку текста.) Если в результате сравнения получается значение true, то посредством вызова метода `isEnabled()` определяется состояние объекта `jbtnSecond`. Если метод `isEnabled()` возвращает значение true, это означает, что кнопка Second доступна, поэтому она деактивируется путем вызова метода `setEnabled(false)` объекта `jbtnSecond`. В противном случае (значение false) кнопка неактивизирована и активизируется посредством выражения `jbtnSecond.setEnabled(true)`.

**ВАЖНО!**  
**2.8.**

## Определение кнопки по умолчанию

Если в приложении используются кнопки, часто возникает необходимость, чтобы одна из них была выбрана по умолчанию. Для того чтобы активизировать такую кнопку, достаточно нажать клавишу <Enter>. Чтобы создать кнопку по умолчанию, надо вызвать метод `setDefaultButton()` объекта `JRootPane`, содержащего кнопку. Ниже приведено объявление этого метода.

```
void setDefaultButton(JButton button)
```

Параметр `button` представляет здесь кнопку, определяемую в качестве кнопки по умолчанию.

Поскольку метод `setDefaultButton()` принадлежит корневой панели контейнера, то, чтобы установить кнопку `jbtnFirst` в качестве кнопки по умолчанию, надо использовать следующее выражение:

```
jfrm.getRootPane().setDefaultButton(jbtnFirst);
```

Его надо поместить в текст программы непосредственно перед вызовом метода `setVisible()`. Если кнопка по умолчанию определена, нажатие клавиши <Enter> приведет к генерации кнопкой события `ActionEvent`.

## Дополнительные средства, предоставляемые классом JButton

Класс `JButton` предоставляет разработчикам графических пользовательских интерфейсов ряд дополнительных возможностей. Например, вы можете использовать для определения текста, отображаемого на кнопке, HTML-код, подобно тому, как это делается в метках. Один из аргументов в пользу приме-

нения HTML-кода — возможность отображать несколько строк текста. Например, приведенный ниже конструктор создает кнопку, на которой слово Press отображается над словом Me.

```
JButton jbtn = new JButton("<html>Press<br>Me");
```

В тексте, отображаемом на кнопке, можно определять мнемонические обозначения. Если заданная таким образом кнопка будет нажата в комбинации с клавишей <Alt>, это приведет к активизации кнопки. Например, в предыдущем примере можно использовать объект jbtn для того, чтобы установить командную клавишу <P>. Соответствующее выражение имеет следующий вид:

```
jbtn.setMnemonic('p');
```

После его выполнения действия с кнопкой можно выполнять не только с помощью мыши, но и комбинации клавиш <Alt+P>.

Если на кнопке имеется и текст, и изображение, вы можете задать расположение текста относительно пиктограммы, вызывая методы `setVerticalTextPosition()` и `setHorizontalTextPosition()`.



### Вопросы для текущего контроля

1. Что такое `AbstractButton`?
2. Какой метод позволяет задать пиктограмму, которая будет отображаться при наведении курсора мыши на кнопку?
3. Что такое кнопка по умолчанию?

**ВАЖНО!**

## 2.9 Использование неименованного внутреннего класса для обработки событий

Перед тем как перейти к рассмотрению флажков и переключателей опций, необходимо сделать небольшое отступление, касающееся обработки событий, генерируемых компонентами Swing. В предыдущем примере, а также в одной

1. `AbstractButton` — это суперкласс всех кнопок Swing.
2. `setRolloverIcon()`.
3. Кнопка по умолчанию — это кнопка, которая активизируется по нажатию пользователем клавиши <Enter>.

из программ, рассмотренных в модуле 1, события обрабатывались основным классом (классом, находящимся в области видимости пакета), реализующим интерфейс `ActionListener`. Такой подход часто применяется на практике. Однако для некоторых приложений целесообразнее использовать другой способ обработки событий, который позволяет достичь требуемых результатов меньшими усилиями. Обработчик события можно создать в виде неименованного внутреннего класса, который передается методу `addActionListener()`. Например, неименованный внутренний класс, выступающий в роли обработчика события для кнопки `First` из предыдущего примера, будет выглядеть так, как показано ниже.

```
jbtnFirst.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        if(jbtnSecond.isEnabled()) {
            jlab.setText("Second button is disabled.");
            jbtnSecond.setEnabled(false);
        }
        else {
            jlab.setText("Second button is enabled.");
            jbtnSecond.setEnabled(true);
        }
    }
});
```

Если вы незнакомы с синтаксисом неименованных внутренних классов, то данный код вряд ли будет понятен вам. Работает он следующим образом. Параметром метода `addActionListener()` является новый объект — экземпляр неименованного класса, реализующего интерфейс `ActionListener`. В интерфейсе `ActionListener` объявлен только один метод; он должен быть определен во внутреннем классе. Экземпляр неименованного внутреннего класса становится обработчиком событий действий, генерируемых `jbtnFirst`. Он не принимает события из других источников. Ниже приведен код предыдущей программы, который модернизирован для использования обработчика событий, выполненного в виде неименованного внутреннего класса. Обратите внимание на то, что класс `ButtonIcons` уже не реализует интерфейс `ActionListener`. Вместо этого в качестве обработчика используется неименованный внутренний класс. В результате текст программы становится более коротким, а код обработчика — лучше управляемым, так как он непосредственно задается при вызове метода `addActionListener()`. Программа выполняет точно такие же действия, как и ранее, и отличается лишь кодом обработчика.



## 112 Модуль 2. Метки, кнопки и обрамление

---

```
// Использование неименованных внутренних классов
// в качестве обработчиков событий.

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

class ButtonIcons {

    JLabel jlab;
    JButton jbtnFirst;
    JButton jbtnSecond;

    ButtonIcons() {
        ImageIcon myIcon = new ImageIcon("myIcon.gif");
        ImageIcon myDisIcon = new ImageIcon("myDisIcon.gif");
        ImageIcon myROIcon = new ImageIcon("myROIcon.gif");
        ImageIcon myPIcon = new ImageIcon("myPIcon.gif");

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Use Button Icons");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начального размера фрейма.
        jfrm.setSize(220, 100);

        // Завершение программы при закрытии приложения пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание текстовой метки.
        jlab = new JLabel("Press a button.");

        // Создание двух кнопок.
        jbtnFirst = new JButton("First", myIcon);
        jbtnSecond = new JButton("Second", myIcon);

        // Установка изображения, отображаемого при помещении
        // на кнопку курсора мыши.
        jbtnFirst.setRolloverIcon(myROIcon);
        jbtnSecond.setRolloverIcon(myROIcon);
    }
}
```

```
// Установка изображения, отображаемого на активизированной
// кнопке.
jbtnFirst.setPressedIcon(myPIcon);
jbtnSecond.setPressedIcon(myPIcon);

// Установка изображения, отображаемого на
// деактивизированной кнопке.
jbtnFirst.setDisabledIcon(myDisIcon);
jbtnSecond.setDisabledIcon(myDisIcon);

// Связывание с кнопками обработчиков событий.
// Обратите внимание на то, что обработчики реализованы
// в виде неименованных внутренних классов.
jbtnFirst.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        if(jbtnSecond.isEnabled()) {
            jlab.setText("Second button is disabled.");
            jbtnSecond.setEnabled(false);
        }
        else {
            jlab.setText("Second button is enabled.");
            jbtnSecond.setEnabled(true);
        }
    }
});

jbtnSecond.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("Second button was pressed.");
    }
});

// Включение кнопок в состав панели содержимого.
jfrm.getContentPane().add(jbtnFirst);
jfrm.getContentPane().add(jbtnSecond);

// Включение метки в состав панели содержимого.
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}
```

```
public static void main(String args[]) {
    // Создание фрейма в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new ButtonIcons();
        }
    });
}
```

В коммерческих приложениях в качестве обработчиков используются как обычные, так и неименованные внутренние классы. В некоторых случаях целесообразнее применять класс, находящийся в области видимости пакета, так как при этом удастся избежать дублирования кода для нескольких обработчиков, выполняющих одни и те же действия. Этот подход также может быть более эффективным потому, что не требует создания отдельных классов. С другой стороны, неименованные внутренние классы делают код приложения более понятным. Однако такие обработчики нельзя использовать для нескольких источников события. С каждым источником должен быть связан отдельный класс, что увеличивает размер кода. Одним словом, выбор конкретного решения диктуется спецификой приложения. Правила “на все случаи жизни” не существует.

### Проект 2.1. ~~Использование событий~~ Изменение состояния JButton и ButtonModel

ChangeDemo.java

Данный проект иллюстрирует две важные детали, имеющие отношение к JButton: события изменения состояния и модель ButtonModel. Как было сказано ранее, компонент JButton генерирует два типа событий: события действий и события изменения состояния. Как правило, для выявления щелчков на кнопке используются события действий. Однако в программе можно также предусмотреть реакцию на события изменения состояния. Такие события возникают при изменении модели, лежащей в основе кнопки. Они позволяют предусматривать в программе действия, связанные с различными этапами действия с кнопкой; в частности, таким образом можно отслеживать наведение курсора на кнопку, активизацию и деактивизацию кнопки. Так, например, о наведении курсора на кнопку можно сигнализировать звуковым сигналом, а при активизации кнопки и ее деактивизации частота звука может изменяться.

Модель для JButton определена посредством интерфейса ButtonModel. Помимо прочего, в ButtonModel предусмотрено пять свойств, описывающих состояние кнопки (кнопка доступна, курсор размещен на кнопке, кнопка удер-

живается, кнопка нажата и кнопка выбрана). Из них только первые четыре свойства применимы к `JButton` (выбранной может быть лишь кнопка, допускающая два состояния.) Эти свойства доступны посредством четырех методов, определенных в `ButtonModel`.

|                                   |                                                                                                                                             |
|-----------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| <code>boolean isArmed()</code>    | Возвращает значение <code>true</code> , если кнопка была нажата, но не отпущена                                                             |
| <code>boolean isEnabled()</code>  | Возвращает значение <code>true</code> , если кнопка доступна для использования                                                              |
| <code>boolean isPressed()</code>  | Возвращает значение <code>true</code> , если кнопка была нажата                                                                             |
| <code>boolean isRollover()</code> | Возвращает значение <code>true</code> , если курсор был наведен на кнопку                                                                   |
| <code>boolean isSelected()</code> | Если возвращаемое значение равно <code>true</code> , значит, кнопка выбрана. (Данный метод применим только для кнопок с двумя состояниями.) |

Используя эти методы, можно определять текущее состояние кнопки при изменении модели.

Для того чтобы иметь возможность работать с информацией о кнопке, надо сначала получить ссылку на модель кнопки, вызывая метод `getModel()`. Объявление этого метода выглядит следующим образом:

```
ButtonModel getModel()
```

Имея ссылку на модель, можно вызвать вышеупомянутый метод и выяснить текущее состояние. В данном проекте мы создадим программу, в которой используются одна кнопка и обработчик генерируемых ею событий. При возникновении события отображается текущее состояние кнопки.

## Последовательность действий

1. Создайте файл `ChangeDemo.java` и введите приведенные ниже строку комментариев и выражения `import`.

```
// Демонстрация событий изменения состояния и модели кнопки
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
```

2. Создайте класс `ChangeDemo`, начав его следующим кодом:

```
class ChangeDemo {
```

```
JButton jbtn;  
JLabel jlab;  
ChangeDemo() {  
  
    // Создание нового контейнера JFrame.  
    JFrame jfrm = new JFrame("Button Change Events");  
  
    // Установка диспетчера компоновки FlowLayout.  
    jfrm.getContentPane().setLayout(new FlowLayout());  
  
    // Установка начального размера фрейма.  
    jfrm.setSize(250, 160);  
  
    // Завершение программы при закрытии  
    // приложения пользователем.  
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
    // Создание пустой метки.  
  
    jlab = new JLabel();  
  
    // Создание кнопки.  
    jbtn = new JButton("Press for Change Event Test");
```

Приведенный код должен быть понятен вам. С помощью данных выражений создается фрейм, а также формируются метка и кнопка.

3. Добавьте обработчик, который будет реагировать на события изменения состояния и отображать текущее состояние кнопки.

```
// Добавление обработчика события изменения состояния.  
jbtn.addChangeListener(new ChangeListener() {  
    public void stateChanged(ChangeEvent ce) {  
        ButtonModel mod = jbtn.getModel();  
        String what = "";  
  
        if(mod.isEnabled()) what += "Enabled<br>";  
        if(mod.isRollover()) what += "Rollover<br>";  
        if(mod.isArmed()) what += "Armed<br>";  
        if(mod.isPressed()) what += "Pressed<br>";  
  
        // Текст на метке задается с помощью HTML-кода.  
        jlab.setText("<html>Current state:<br>" + what);  
    }  
});
```

Этот обработчик представляет собой неименованный внутренний класс. При каждом возникновении события извлекается модель кнопки для `jbtn`. Затем, в зависимости от результатов опроса состояния кнопки, формируется строка, содержащая сведения о соответствующих свойствах кнопки. И наконец, свойства отображаются в `jlab`. Заметьте, что в данном случае используется HTML-код. Это упрощает представление свойств в списке.

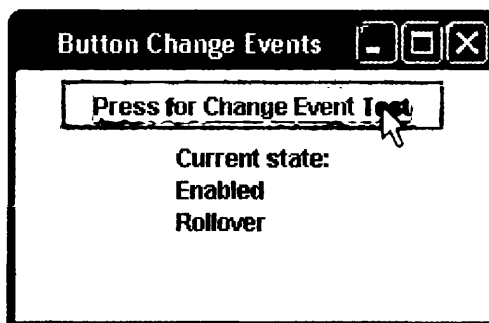
4. Завершите класс `ChangeDemo` следующими строками кода:

```
// Добавление компонентов к панели содержимого.  
jfrm.getContentPane().add(jbtn);  
jfrm.getContentPane().add(jlab);  
  
// Отображение фрейма.  
jfrm.setVisible(true);  
}
```

5. Завершите программу методом `main()`; его код приведен ниже.

```
public static void main(String args[]) {  
    // Создание фрейма в потоке обработки событий.  
  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            new ChangeDemo();  
        }  
    });  
}
```

6. Запустив программу, вы увидите, что состояние кнопки изменяется при наведении на нее курсора мыши, а также при ее активизации и деактивизации. Окно программы показано ниже.



### 7. Полностью код программы имеет следующий вид:

```
// Демонстрация событий изменения состояния
// и модели кнопки import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

class ChangeDemo {

    JButton jbtn;
    JLabel jlab;

    ChangeDemo() {

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Button Change Events");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начального размера фрейма.
        jfrm.setSize(250, 160);

        // Завершение программы при закрытии
        // приложения пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание пустой метки.
        jlab = new JLabel();

        // Создание кнопки.
        jbtn = new JButton("Press for Change Event Test");

        // Добавление обработчика события изменения
        // состояния.
        jbtn.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent ce) {
                ButtonModel mod = jbtn.getModel();
                String what = "";
```

```

        if(mod.isEnabled()) what += "Enabled<br>";
        if(mod.isRollover()) what += "Rollover<br>";
        if(mod.isArmed()) what += "Armed<br>";
        if(mod.isPressed()) what += "Pressed<br>";

        // Текст на метке задается с помощью HTML-кода.
        jlab.setText("<html>Current state:<br>" + what);
    }
});

// Добавление компонентов к панели содержимого.
jfrm.getContentPane().add(jbtn);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Создание фрейма в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new ChangeDemo();
        }
    });
}
}

```

**ВАЖНО!**  
**2.10**

## Использование класса JToggleButton

Swing предоставляет очень полезную разновидность кнопки, которая называется *кнопкой-переключателем* (toggle button). Кнопка-переключатель выглядит как обычная кнопка, но в отличие от нее имеет два состояния: нажатое и отпущенное. Таким образом, когда вы щелкнете на кнопке-переключателе, она останется нажатой (обычная кнопка в этом случае вернется в исходное положение). Щелкнув на кнопке второй раз, вы восстановите ее предыдущее состо-



яние. Таким образом, по каждому щелчку кнопка переключается из текущего состояния в противоположное.

Для поддержки кнопки-переключателя используется класс `JToggleButton`. Он является подклассом класса `AbstractButton` и предоставляет конструкторы, перечисленные в табл. 2.4. Помимо создания стандартных кнопок-переключателей, `JToggleButton` выполняет еще одну функцию: он является суперклассом для двух других компонентов Swing, которые также имеют два устойчивых состояния. Это классы `JCheckBox` и `JRadioButton`, которые будут описаны далее в этом модуле. Таким образом, `JToggleButton` определяет базовые функции для всех компонентов с двумя состояниями.

Таблица 2.4. Конструкторы класса `JToggleButton`

| Конструктор                                           | Описание                                                                                                                                                                                                                                                                                                                                   |
|-------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JToggleButton()</code>                          | Создает пустую кнопку-переключатель, т.е. кнопку, для которой не определено ни изображение, ни текст                                                                                                                                                                                                                                       |
| <code>JToggleButton(String str)</code>                | Создает кнопку-переключатель, на которой отображается текст. Строка текста передается посредством параметра <code>str</code>                                                                                                                                                                                                               |
| <code>JToggleButton(Icon icon)</code>                 | Создает кнопку-переключатель, на которой отображается пиктограмма. Изображение передается посредством параметра <code>icon</code>                                                                                                                                                                                                          |
| <code>JToggleButton(String str, boolean state)</code> | Создает кнопку-переключатель, на которой отображается текст, переданный посредством параметра <code>str</code> . Если значение параметра <code>state</code> равно <code>true</code> , исходным состоянием кнопки является нажатое (выбранное). В противном случае кнопка изначально находится в отпущенном (не выбранном) состоянии        |
| <code>JToggleButton(Icon icon, boolean state)</code>  | Создает кнопку-переключатель, на которой отображается пиктограмма, переданная посредством параметра <code>icon</code> . Если значение параметра <code>state</code> равно <code>true</code> , исходным состоянием кнопки является нажатое (выбранное). В противном случае кнопка изначально находится в отпущенном (не выбранном) состоянии |
| <code>JToggleButton(String str, Icon icon)</code>     | Создает кнопку-переключатель, на которой отображается текст, переданный посредством параметра <code>str</code> , и пиктограмма, переданная посредством параметра <code>icon</code>                                                                                                                                                         |

Окончание табл. 2.4.

| Конструктор                                                      | Описание                                                                                                                                                                                                                                                                                                                                                                                                |
|------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JToggleButton(String str, Icon icon, boolean state)</code> | Создает кнопку-переключатель, на которой отображается текст, переданный посредством параметра <code>str</code> , и пиктограмма, переданная посредством параметра <code>icon</code> . Если значение параметра <code>state</code> равно <code>true</code> , исходным состоянием кнопки является нажатое (выбранное). В противном случае кнопка изначально находится в отпущенном (не выбранном) состоянии |
| <code>JToggleButton(Action act)</code>                           | Создает кнопку-переключатель, для которой текст, изображение и другие параметры задаются посредством параметра <code>act</code> . Класс <code>Action</code> будет рассмотрен в модуле 7                                                                                                                                                                                                                 |

Модель `JToggleButton` отличается от модели `JButton`. Модель, используемая `JToggleButton`, определяется классом `JToggleButton.ToggleButtonModel`. В обычных условиях при использовании стандартной кнопки-переключателя не возникает необходимость непосредственного взаимодействия с моделью.

Подобно `JButton`, объект `JToggleButton` при каждом щелчке генерирует событие действия и событие изменения состояния. Однако в отличие от `JButton` объект `JToggleButton` также генерирует и событие элемента. Это событие используется для реализации такого понятия, как выбор. Когда кнопка, представляемая объектом `JToggleButton`, нажата, она считается выбранной. При отпускании кнопки выбор отменяется.

Для обработки событий элемента надо использовать класс, реализующий интерфейс `ItemListener`. При возникновении события элемента оно передается методу `itemStateChanged()`, определенному в `ItemListener`. В теле метода `itemStateChanged()` можно использовать метод `getItem()` для получения ссылки на объект `JToggleButton`, сгенерировавший событие. Определить, нажата кнопка или отпущена, позволяет метод `getStateChange()`. Если кнопка нажата, возвращается значение `ItemEvent.SELECTED`. В противном случае возвращается значение `ItemEvent.DESELECTED`. Кроме того, для определения состояния кнопки, сгенерировавшей событие, можно использовать ее метод `isSelected()`, унаследованный от класса `AbstractButton`.

Приведенный ниже пример иллюстрирует использование кнопки-переключателя. Внешний вид окна, отображаемого программой, показан на рис. 2.7. Обратите внимание на обработчик событий элемента. Для определения состояния кнопки он вызывает метод `isSelected()`.



*Рис. 2.7. Окно, которое отображается в процессе работы программы, демонстрирующей работу кнопки-переключателя*

// Демонстрация работы кнопки-переключателя

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

class ToggleDemo {

    JLabel jlab;
    JToggleButton jtbn;

    ToggleDemo() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Demonstrate a Toggle Button");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(290, 80);

        // Завершение программы при закрытии приложения пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание метки, отображающей строку текста.
        jlab = new JLabel("Button is off.");

        // Создание кнопки-переключателя.
        jtbn = new JToggleButton("On/Off");

        // Установка обработчика события элемента jtbn,
        // выполненного в виде неименованного внутреннего класса.
        // Для обработки событий кнопки-переключателя
```

```
// используется ItemListener
jbtn.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent ie) {
        if(jbtn.isSelected()) // Для определения состояния
            // кнопки используется метод isSelected()
            jlab.setText("Button is on.");
        else
            jlab.setText("Button is off.");
    }
});
// Включение кнопки-переключателя и метки
// в состав панели содержимого.
jfrm.getContentPane().add(jbtn);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Создание фрейма в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new ToggleDemo();
        }
    });
}
```

## Использование метода `getItem()` для определения источника события элемента

В предыдущем примере для обработки событий элемента, генерируемых кнопкой-переключателем, использовался неименованный внутренний класс. Это означает, что обработчик принимает события только от конкретной кнопки и необходимость в идентификации источника не возникает. Однако возможна ситуация, когда один обработчик `ItemEvent` объявлен для нескольких элементов. В этом случае приходится определять, какой из объектов сгенерировал событие. Самое простое решение подобной задачи — вызов метода `getItem()` объекта `ItemEvent`, ссылка на который передается методу `itemStateChanged`. Метод `getItem()` возвращает ссылку на компонент, являющийся источником события. Эту ссылку можно сравнить с компонентами, определенными в вашей программе.

Ниже приведен код программы, демонстрирующий использование метода `getItem()`. Класс `TwoTBDemo` реализует интерфейс `ItemListener`. При выполнении программы создаются две кнопки-переключателя, ссылки на которые хранятся в переменных `jbtnAlpha` и `jbtnBeta`. События элемента, генерируемые этими кнопками, обрабатываются одним и тем же методом `itemStateChanged()`. Выходные данные, отображаемые при работе программы, показаны на рис. 2.8.

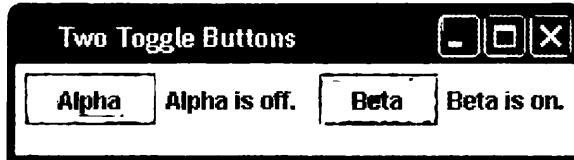


Рис. 2.8. Окно, отображаемое программой `TwoTBDemo`

```
// Использование двух кнопок-переключателей

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

class TwoTBDemo implements ItemListener {

    JLabel jlabAlpha;
    JLabel jlabBeta;
    JToggleButton jbtnAlpha;
    JToggleButton jbtnBeta;

    TwoTBDemo() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Two Toggle Buttons");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(290, 80);

        // Завершение программы при закрытии приложения пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

```

// Создание двух меток.
jlabAlpha = new JLabel("Alpha is off. ");
jlabBeta = new JLabel("Beta is off.");

// Создание двух кнопок-переключателей.
jtnbAlpha = new JToggleButton("Alpha");
jtnbBeta = new JToggleButton("Beta");

// Определение обработчиков событий для кнопок.
jtnbAlpha.addItemListener(this);
jtnbBeta.addItemListener(this);

// Включение кнопок-переключателей и меток
// в состав панели содержимого.
jfrm.getContentPane().add(jtnbAlpha);
jfrm.getContentPane().add(jlabAlpha);
jfrm.getContentPane().add(jtnbBeta);
jfrm.getContentPane().add(jlabBeta);

// Отображение фрейма.
jfrm.setVisible(true);
}

// Обработка событий элемента для обеих кнопок.
public void itemStateChanged(ItemEvent ie) {

    // Получение ссылки на кнопку, являющуюся
    // источником события.
    JToggleButton tb = (JToggleButton) ie.getItem();

    // Определение кнопки, состояние которой было изменено,
    // посредством ссылки.
    if(tb == jtnbAlpha)
        if(tb.isSelected())
            jlabAlpha.setText("Alpha is on. ");
        else
            jlabAlpha.setText("Alpha is off. ");

    else if(tb == jtnbBeta)
        if(tb.isSelected())
            jlabBeta.setText("Beta is on.");
        else
            jlabBeta.setText("Beta is off.");
}

```

```
}

public static void main(String args[]) {
    // Создание фрейма в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new TwoTBDemo();
        }
    });
}
}
```

Заметьте, что в теле метода `itemStateChanged()` для получения ссылки на источник события вызывается метод `getItem()`.

```
JToggleButton tb = (JToggleButton) ie.getItem();
```

Полученная ссылка сравнивается с переменными `jtbAlpha` и `jtbBeta`; так определяется, какая из кнопок сгенерировала событие. Подобная процедура может использоваться для определения источника любого события элемента.

### Дополнительные средства, предоставляемые классом `JToggleButton`

Подобно `JButton`, объект `JToggleButton` предоставляет различные средства настройки. Например, вы можете задавать пиктограммы, индицирующие нажатое и отпущенное состояние кнопки. Также можно задать отдельную пиктограмму для случая, когда курсор мыши наведен на кнопку. Установить изображение по умолчанию позволяет следующий вариант конструктора `JToggleButton`:

```
JToggleButton(String str, Icon icon)
```

В результате его выполнения создается кнопка, которая содержит текст, определенный посредством параметра `str`, и изображение, заданное с помощью параметра `icon`. Добавить пиктограммы, предназначенные для других целей, позволяют следующие методы: `setDisabledIcon()`, `setPressedIcon()`, `setRolloverIcon()`, `setRolloverSelectedIcon()` и `setSelectedIcon()`. Заметьте, что, для того, чтобы можно было использовать пиктограммы, задаваемые этими методами, должно быть определено изображение по умолчанию. Позицию текста относительно изображения можно задать, вызывая методы `setVerticalTextPosition()` и `setHorizontalTextPosition()`.

В тексте, отображаемом на кнопке, можно определять мнемонические обозначения. Если заданная таким образом командная клавиша будет нажата в комбинации с клавишей <Alt>, это приведет к изменению состояния кнопки. Так, чтобы в предыдущем примере задать командную клавишу <A> для `JtbnAlpha`, надо включить в текст программы следующую строку:

```
JtbnAlpha.setMnemonic('a');
```

После выполнения данного выражения переключать состояние кнопки можно будет не только с помощью мыши, но и комбинации клавиш <Alt+A>.

Запретить или разрешить доступ к элементу `JToggleButton` можно путем вызова метода `setEnabled()`, т.е. так же, как и при работе с кнопкой `JButton`.



### Вопросы для текущего контроля.

1. Что такое кнопка с двумя состояниями?
2. Какой метод позволяет определить, нажата или отпущена кнопка `JToggleButton`?
3. Какой интерфейс обработки обычно используется при обработке событий, генерируемых элементом `JToggleButton`?

**ВАЖНО!**

**2.11.**

## Создание флажков опций с помощью класса `JCheckBox`

Если обычные кнопки используются чаще других элементов, то второе место по популярности наверняка занимают флажки опций. В Swing этот элемент реализуется с помощью класса `JCheckBox`. Класс `JCheckBox` является потомком классов `AbstractButton` и `JToggleButton`, поэтому те же приемы, которые используются для управления кнопками-переключателями, применимы и для флажков опций.

В классе `JCheckBox` определены конструкторы, описанные в табл. 2.5.

1. Кнопка с двумя состояниями может находиться в нажатом (выбранном) или отпущенном (не выбранном) положении.
2. `getStateChange()` или `isSelected()`.
3. `ItemListener`.



Таблица 2.5. Конструкторы класса *JCheckBox*

| Конструктор                                                  | Описание                                                                                                                                                                                                                                                                                                    |
|--------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JCheckBox()</code>                                     | Создание флажка опции, с которым не связан ни текст, ни изображение                                                                                                                                                                                                                                         |
| <code>JCheckBox(String str)</code>                           | Создание флажка опции, с которым связана строка текста, заданная посредством параметра <code>str</code>                                                                                                                                                                                                     |
| <code>JCheckBox(Icon icon)</code>                            | Создание флажка опции, с которым связано изображение, заданное посредством параметра <code>icon</code>                                                                                                                                                                                                      |
| <code>JCheckBox(String str, boolean state)</code>            | Создание флажка опции, с которым связана строка текста, заданная посредством параметра <code>str</code> . Если значение параметра <code>state</code> равно <code>true</code> , флажок изначально установлен, в противном случае он сброшен                                                                  |
| <code>JCheckBox(Icon icon, boolean state)</code>             | Создание флажка опции, с которым связано изображение, заданное посредством параметра <code>icon</code> . Если значение параметра <code>state</code> равно <code>true</code> , флажок изначально установлен, в противном случае он сброшен                                                                   |
| <code>JCheckBox(String str, Icon icon)</code>                | Создание флажка опции, с которым связана строка текста, заданная с помощью параметра <code>str</code> , и изображение, указанное посредством параметра <code>icon</code>                                                                                                                                    |
| <code>JCheckBox(String str, Icon icon, boolean state)</code> | Создание флажка опции, с которым связана строка текста, заданная с помощью параметра <code>str</code> , и изображение, указанное посредством параметра <code>icon</code> . Если значение параметра <code>state</code> равно <code>true</code> , флажок изначально установлен, в противном случае он сброшен |
| <code>JCheckBox(Action act)</code>                           | Создание флажка опции, для которого текст, изображение и другие параметры задаются посредством параметра <code>act</code> . Класс <code>Action</code> будет рассмотрен в модуле 7                                                                                                                           |

Подобно любой кнопке с двумя состояниями, состояние флажка опции можно изменить двумя способами. Во-первых, это может сделать пользователь, а во-вторых, изменить состояние можно из программы. Для изменения состояния флажка опции (а также любой другой кнопки с двумя состояниями) программными средствами надо использовать метод `setSelected()`:

```
void setSelected(boolean state)
```

Если значение параметра `state` равно `true`, флажок опции устанавливается. Значение `state`, равное `false`, приводит к сбросу флажка.

При установке или сбросе флажка опции генерируется событие элемента. Для обработки этого события используется метод `itemStateChanged()`. В теле `itemStateChanged()` можно использовать метод `getItem()` для

получения ссылки на объект `JCheckBox`, сгенерировавший событие. Метод `getStateChange()` позволяет определить, установлен флажок опции или сброшен. Если флажок установлен, метод возвращает `ItemEvent.SELECTED`. В противном случае возвращается значение `ItemEvent.DESELECTED`. Существует и альтернативное решение: можно вызвать метод `isSelected()` флажка опции, сгенерировавшего событие (метод `isSelected()` унаследован от класса `AbstractButton`).

При каждом изменении состояния флажка опции генерируется событие элемента. Помимо него, данный компонент генерирует также события действий, но гораздо проще использовать `ItemListener`, поскольку данный обработчик предоставляет непосредственный доступ к методу `getStateChange()`.

Ниже приведен код программы, демонстрирующий работу с флажками опций. В процессе работы программы создается окно, содержащее четыре флажка опций. Первоначально доступен лишь один из них — первый, с надписью `Enable Options`. Остальные три флажка неактивизированы. При установке флажка `Enable Options` разрешается доступ к остальным трем флажкам. При каждом изменении состояния флажка из группы `Options` отображается информация о выполненном действии. Кроме того, при этом обновляется список установленных флажков. Окно, создаваемое при работе программы, показано на рис. 2.9.

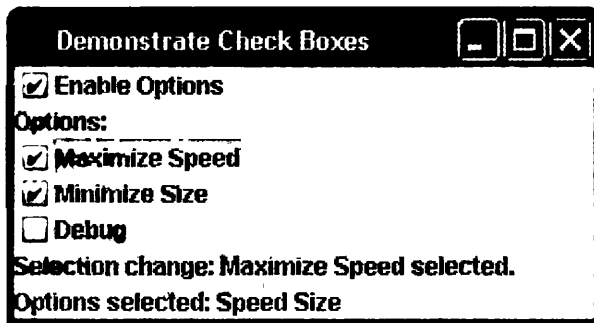


Рис. 2.9. Выходные данные программы, демонстрирующей работу с флажками опций

// Демонстрация работы с флажками опций

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

```
class CBDemo implements ItemListener {
```

```
JLabel jlabOptions;  
JLabel jlabWhat;  
JLabel jlabChange;  
JCheckBox jcbOptions;  
JCheckBox jcbSpeed;  
JCheckBox jcbSize;  
JCheckBox jcbDebug;  
  
CBDemo() {  
    // Создание нового контейнера JFrame.  
    JFrame jfrm = new JFrame("Demonstrate Check Boxes");  
  
    // Установка диспетчера компоновки GridLayout,  
    // формирующего таблицу из семи строк и одного столбца.  
    jfrm.getContentPane().setLayout(new GridLayout(7, 1));  
  
    // Установка начальных размеров фрейма.  
    jfrm.setSize(300, 150);  
  
    // Завершение программы при закрытии приложения пользователем.  
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
    // Создание двух меток.  
    jlabOptions = new JLabel("Options:");  
    jlabChange = new JLabel("");  
    jlabWhat = new JLabel("Options selected:");  
  
    // Создание четырех флажков опций.  
    jcbOptions = new JCheckBox("Enable Options");  
    jcbSpeed = new JCheckBox("Maximize Speed");  
    jcbSize = new JCheckBox("Minimize Size");  
    jcbDebug = new JCheckBox("Debug");  
  
    // Три флажка опций изначально недоступны.  
    jcbSpeed.setEnabled(false);  
    jcbSize.setEnabled(false);  
    jcbDebug.setEnabled(false);  
  
    // Установка обработчика события элемента для jcbOptions.  
    jcbOptions.addItemListener(new ItemListener() {  
        public void itemStateChanged(ItemEvent ie) {  
            if(jcbOptions.isSelected()) {
```

```
// При каждом изменении состояния флажка jcbOptions,
// т.е. при каждом событии, связанном с ним,
// состояние остальных флажков переключается с
// деактивизированного в активизированное и наоборот.
jcbSpeed.setEnabled(true);
jcbSize.setEnabled(true);
jcbDebug.setEnabled(true);
}
else {
    jcbSpeed.setEnabled(false);
    jcbSize.setEnabled(false);
    jcbDebug.setEnabled(false);
}
}
});

// События, генерируемые флажками опций из группы Options,
// обрабатываются одним методом - itemStateChanged(),
// реализованным в классе CBDemo.
jcbSpeed.addItemListener(this);
jcbSize.addItemListener(this);
jcbDebug.addItemListener(this);

// Включение флажков опций и меток
// в состав панели содержимого.
jfrm.getContentPane().add(jcbOptions);
jfrm.getContentPane().add(jlabOptions);

jfrm.getContentPane().add(jcbSpeed);
jfrm.getContentPane().add(jcbSize);
jfrm.getContentPane().add(jcbDebug);
jfrm.getContentPane().add(jlabChange);
jfrm.getContentPane().add(jlabWhat);

// Отображение фрейма.
jfrm.setVisible(true);
}

// Обработчик событий для всех флажков из группы Options.
public void itemStateChanged(ItemEvent ie) {
    String opts = "";

    // Получение ссылки на флажок опции, являющийся
```

## 132 Модуль 2. Метки, кнопки и обрамление

---

```
// источником события.
JCheckBox cb = (JCheckBox) ie.getItem();

// Предоставление пользователю информации
// о выполненных действиях.
// Для того чтобы определить, установлен или сброшен
// флажок опции, используется метод getStateChange().
if(ie.getStateChange() == ItemEvent.SELECTED)
    jlabChange.setText("Selection change: " +
                       cb.getText() + " selected.");
else
    jlabChange.setText("Selection change: " +
                       cb.getText() + " cleared.");

// Формирование строки, содержащей сведения о всех
// установленных флажках из группы Options.
if(jcbSpeed.isSelected()) opts += "Speed ";
if(jcbSize.isSelected()) opts += "Size ";
if(jcbDebug.isSelected()) opts += "Debug ";

// Отображение информации об установленных флажках.
jlabWhat.setText("Options selected: " + opts);
}

public static void main(String args[]) {
    // Создание фрейма в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new CBDemo();
        }
    });
}
```

Просматривая текст программы, следует обратить внимание на некоторые ее особенности. Обработчиком событий для `jcbOptions` является неименованный внутренний класс. Такое решение имеет смысл, так как данный обработчик полностью выделен для одного элемента. Для обработки событий,

имеющих отношение к остальным трем флажкам опций, используется метод `itemStateChanged()`, определенный в классе `CBDEmo`. Это также оправданно, так как алгоритмы обработки событий для каждого флажка опций практически совпадают. Таким образом, для них можно использовать один и тот же фрагмент кода и исключить ненужное дублирование. Как видно из данного примера, обрабатывать события можно по-разному, выбирая конкретный способ обработки, исходя из особенностей приложения.

## Дополнительные возможности при использовании флажков опций

Подобно обычным кнопками и кнопкам-переключателям, класс, реализующий флажки опций, предоставляет дополнительные возможности, позволяющие без труда обеспечить требуемый внешний вид и поведение элемента. Например, вы можете задать пиктограммы, которые будут отображаться при установленном или сброшенном флажке. Пиктограмму, соответствующую сброшенному состоянию флажка опции, проще всего задать с помощью конструктора `JCheckBox`, которому в качестве параметра передается изображение. Изображение для установленного флажка позволяет указать метод `setSelectedIcon()`. Поскольку эти две пиктограммы неразрывно связаны друг с другом, необходимо задать их обе. Одну без другой использовать нельзя.

При необходимости можно задать выравнивание для флажка опций, а также расположение текста относительно пиктограммы. Для этой цели используются те же методы, что и при работе с другими типами кнопок: `setVerticalAlignment()`, `setHorizontalAlignment()`, `setVerticalTextPosition()` и `setHorizontalTextPosition()`. Можно также определить в составе надписи мнемоническое обозначение. Для этого предназначен метод `setMnemonic()`.

### Спросим у опытного программиста

- Вопрос.** Существует ли способ имитировать программными средствами щелчок мышью на компоненте?
- Ответ.** Да. Вы можете вызвать для этой цели метод `doClick()`. Этот метод определен в классе `AbstractButton` и генерирует то же событие, которое возникает тогда, когда пользователь щелкает на элементе.

**ВАЖНО!**  
**2.12**

## Создание переключателей опций с помощью класса JRadioButton

Последний компонент, рассматриваемый в этом модуле, — переключатель опций. Данный интерфейсный элемент предназначен для выбора одного из взаимоисключающих значений. Для поддержки переключателей опций предназначен класс `JRadioButton`, который является подклассом класса `AbstractButton`. Непосредственный суперкласс `JRadioButton` — это класс `JToggleButton`, обеспечивающий поддержку кнопок с двумя состояниями. Конструкторы, определенные в классе `JRadioButton`, описаны в табл. 2.6.

Таблица 2.6. Конструкторы класса `JRadioButton`

| Конструктор                                          | Описание                                                                                                                                                                                                                                                                                 |
|------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JRadioButton()</code>                          | Создание кнопки переключателя опций, с которой не связан ни текст, ни изображение                                                                                                                                                                                                        |
| <code>JRadioButton(String str)</code>                | Создание кнопки переключателя опций, с которой связывается строка текста, передаваемая посредством параметра <code>str</code>                                                                                                                                                            |
| <code>JRadioButton(Icon icon)</code>                 | Создание кнопки переключателя опций, с которой связывается изображение, передаваемое посредством параметра <code>icon</code>                                                                                                                                                             |
| <code>JRadioButton(String str, boolean state)</code> | Создание кнопки переключателя опций, с которой связывается строка текста, передаваемая посредством параметра <code>str</code> . Если значение параметра <code>state</code> равно <code>true</code> , кнопка изначально выбрана, в противном случае она находится в невыбранном состоянии |
| <code>JRadioButton(Icon icon, boolean state)</code>  | Создание кнопки переключателя опций, с которой связывается изображение, передаваемое посредством параметра <code>icon</code> . Если значение параметра <code>state</code> равно <code>true</code> , кнопка изначально выбрана, в противном случае она находится в невыбранном состоянии  |
| <code>JRadioButton(String str, Icon icon)</code>     | Создание кнопки переключателя опций, с которой связывается строка текста, передаваемая с помощью параметра <code>str</code> , и изображение, заданное посредством параметра <code>icon</code>                                                                                            |

Окончание табл. 2.6.

| Конструктор                                                     | Описание                                                                                                                                                                                                                                                                                                                                                 |
|-----------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JRadioButton(String str, Icon icon, boolean state)</code> | Создание кнопки переключателя опций, с которой связывается строка текста, передаваемая с помощью параметра <code>str</code> , и изображение, заданное посредством параметра <code>icon</code> . Если значение параметра <code>state</code> равно <code>true</code> , кнопка изначально выбрана, в противном случае она находится в невыбранном состоянии |
| <code>JCheckBox(Action act)</code>                              | Создание кнопки переключателя опций, для которой текст, изображение и другие параметры задаются посредством параметра <code>act</code> . Класс <code>Action</code> будет рассмотрен в модуле 7                                                                                                                                                           |

Очередной выбор кнопки из группы автоматически отменяет выбор, сделанный ранее. Таким образом, в каждый момент времени выбранной может быть только одна кнопка. Группа кнопок формируется путем добавления элементов к объекту `ButtonGroup`. Этот класс принадлежит пакету `javax.swing` и предоставляет только конструктор по умолчанию. Добавление кнопок к группе производится с помощью следующего метода:

```
void add(AbstractButton button)
```

Параметр `button` представляет собой ссылку на кнопку, включаемую в состав группы. Несмотря на то что класс `ButtonGroup` можно использовать для создания группы из любых кнопок с двумя состояниями, чаще всего он используется с переключателями опций.

При выборе или отмене выбора элемент `JRadioButton` генерирует события действий, события элемента и события изменения состояния. Чаще всего обрабатываются события действий, т.е. в программе создается класс, реализующий интерфейс `ActionListener`. Как вы помните, в интерфейсе `ActionListener` объявлен только метод `actionPerformed()`. В теле этого метода можно использовать различные средства идентификации выбранной кнопки. Во-первых, можно организовать проверку команды действия, вызвав метод `getActionCommand()`. По умолчанию команда действия совпадает с надписью на кнопке, но при необходимости ее можно задать явно, вызвав метод `setActionCommand()` объекта `JRadioButton`. Во-вторых, можно вызвать метод `getSource()` объекта `ActionEvent` и проверить, на какую кнопку указывает полученная ссылка. И наконец, можно непосредственно проверить каждую кнопку и выяснить, какая из них выбрана в данный момент. Эту информацию позволяет получить метод `isSelected()`. Как вы помните, возникновение события действия означает, что выбор кнопки изменился (а в каждый момент времени может быть выбрана одна и только одна кнопка).



При использовании переключателей опций обычно необходимо изначально выбрать одну из кнопок. Сделать это можно, вызвав метод `setSelected()`.

Ниже приведен код программы, которая очень похожа на программу, рассмотренную ранее, но три флажка опций заменены переключателями. Теперь может быть выбрана только одна из трех кнопок, расположенных ниже флажка `Enable Options`. Окно, создаваемое при работе программы, показано на рис. 2.10.

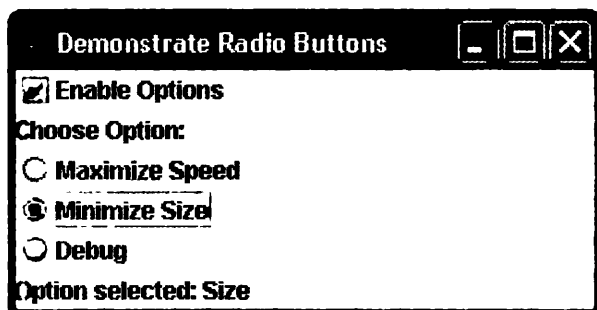


Рис. 2.10. Окно, отображаемое программой `RBDemo`

```
// Пример использования переключателей опций
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class RBDemo implements ActionListener {

    JLabel jlabOptions;
    JLabel jlabWhat;
    JCheckBox jcbOptions;
    JRadioButton jrbSpeed;
    JRadioButton jrbSize;
    JRadioButton jrbDebug;

    RBDemo() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Demonstrate Radio Buttons");

        // Установка диспетчера компоновки GridLayout,
        // формирующего таблицу из шести строк и одного столбца.
        jfrm.getContentPane().setLayout(new GridLayout(6, 1));

        // Установка начальных размеров фрейма.
```

```
jfrm.setSize(300, 150);

// Завершение программы при закрытии приложения пользователем.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Создание двух меток.
jlabOptions = new JLabel("Choose Option:");
jlabWhat = new JLabel("Option selected: Speed");

// Создание флажка опции.
jcbOptions = new JCheckBox("Enable Options");

// Создание трех кнопок переключателя опций.
// Первая кнопка, jrbSpeed, изначально выбрана.
jrbSpeed = new JRadioButton("Maximize Speed", true);
jrbSize = new JRadioButton("Minimize Size");
jrbDebug = new JRadioButton("Debug");

// Добавление кнопок переключателя опций к группе.
ButtonGroup bg = new ButtonGroup();
bg.add(jrbSpeed);
bg.add(jrbSize);
bg.add(jrbDebug);

// При запуске программы все кнопки
// переключателя опций недоступны.
jrbSpeed.setEnabled(false);
jrbSize.setEnabled(false);
jrbDebug.setEnabled(false);

// Установка обработчика событий для jcbOptions.
jcbOptions.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent ie) {
        if(jcbOptions.isSelected()) {
            jrbSpeed.setEnabled(true);
            jrbSize.setEnabled(true);
            jrbDebug.setEnabled(true);
        }
        else {
            jrbSpeed.setEnabled(false);
            jrbSize.setEnabled(false);
            jrbDebug.setEnabled(false);
        }
    }
});
```

```
    }  
    });  
  
    // События, генерируемые всеми тремя кнопками  
    // переключателя опций, обрабатываются одним методом  
    // actionPerformed(), реализованным в классе RBDemo.  
    jrbSpeed.addActionListener(this);  
    jrbSize.addActionListener(this);  
    jrbDebug.addActionListener(this);  
  
    // Включение кнопок и меток в состав панели содержимого.  
    jfrm.getContentPane().add(jcbOptions);  
    jfrm.getContentPane().add(jlabOptions);  
  
    jfrm.getContentPane().add(jrbSpeed);  
    jfrm.getContentPane().add(jrbSize);  
    jfrm.getContentPane().add(jrbDebug);  
    jfrm.getContentPane().add(jlabWhat);  
  
    // Отображение фрейма.  
    jfrm.setVisible(true);  
}  
  
// Обработчик событий для всех кнопок переключателя опций.  
public void actionPerformed(ActionEvent ie) {  
    String opts = "";  
  
    // Обработчик событий для всех флажков из группы Options.  
    public void itemStateChanged(ItemEvent ie) {  
        String opts = "";  
  
        // Получение ссылки на флажок опции, являющийся  
        // источником события.  
        JCheckBox cb = (JCheckBox) ie.getItem();  
  
        // Предоставление пользователю информации  
        // о выполненных действиях.  
        // Для того чтобы определить, установлен или сброшен  
        // флажок опции, используется метод getStateChange().  
        if(ie.getStateChange() == ItemEvent.SELECTED)  
            jlabChange.setText("Selection change: " +  
                               cb.getText() + " selected.");  
        else
```

```

jlabChange.setText("Selection change: " +
                    cb.getText() + " cleared.");

// Формирование строки, содержащей сведения
// о выбранной кнопке.
// В каждый момент времени может быть выбрана
// только одна кнопка.
if(jrbSpeed.isSelected()) opts = "Speed ";
else if(jrbSize.isSelected()) opts = "Size ";
else opts = "Debug ";

// Отображение информации о выбранной кнопке.
jlabWhat.setText("Option selected: " + opts);
}

.

public static void main(String args[]) {
    // Создание фрейма в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new RBDemo();
        }
    });
}
}

```

## Вопросы для текущего контроля

1. Какой класс реализует флажки опций?
2. Взаимно исключающие кнопки обычно являются экземплярами класса `JRadioButton`. Да или нет?
3. Для того чтобы создать группу взаимно исключающих кнопок, надо добавить кнопки к объекту \_\_\_\_\_.

- 
1. `JCheckBox`.
  2. Да.
  3. `ButtonGroup`.

## **Проект 2.2. Создание простого телефонного справочника**

`Phonebook.java` В рамках данного проекта мы создадим простой телефонный справочник, который будет хранить имена и номера телефонов. Используя эту программу, вы сможете найти номер телефона по имени или определить имя абонента, которому принадлежит тот или иной номер. Данная программа допускает настройку. Во-первых, вы можете отменить учет регистра символов при поиске. Вы также можете потребовать, чтобы слово, указанное в качестве критерия поиска, в точности совпадало с именем в списке. Кроме того, программа позволяет искать номера по началу или окончанию имени. В данном проекте используются переключатели и флажки опций, поля редактирования и метки.

### **Последовательность действий**

1. Создайте файл `ChangeDemo.java` и введите приведенные ниже строку комментариев и выражения `import`.

```
// Проект 2.2. Простой телефонный справочник
```

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

2. Создайте класс `Phonebook`, начав его следующими строками кода:

```
class Phonebook {  
  
    JTextField jtfName;  
    JTextField jtfNumber;  
  
    JRadioButton jrbExact;  
    JRadioButton jrbStartsWith;  
    JRadioButton jrbEndsWith;  
  
    JCheckBox jcbIgnoreCase;  
  
    // Краткий список имен и номеров телефонов.  
    String[][] phonelist = {  
        {"Jon", "555-8765"},
```

```
    {"Jessica", "555-5643"},  
    {"Adam", "555-1212" },  
    {"Rachel", "555-3435"},  
    {"Tom & Jerry", "555-1001"}  
};
```

Обратите внимание на двумерный массив `phonelist`. Он содержит список имен и телефонных номеров. Для реальных приложений вместо него придется использовать один из классов, поддерживающих наборы данных, но для нашей цели подойдет и такое простейшее решение.

3. Начните тело конструктора `Phonebook()` приведенным ниже фрагментом кода. Эти команды уже знакомы вам: таким образом создается фрейм.

```
Phonebook() {  
  
    // Создание нового контейнера JFrame.  
    JFrame jfrm = new JFrame("A Simple Phone List");  
  
    // Установка диспетчера компоновки GridLayout.  
    jfrm.getContentPane().setLayout(new GridLayout(0, 1));  
  
    // Установка начальных размеров фрейма.  
    jfrm.setSize(240, 220);  
  
    // Завершение программы при закрытии приложения  
    // пользователем.  
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
}
```

4. Добавьте к конструктору `Phonebook()` следующий код:

```
// Создание меток.  
JLabel jlabName = new JLabel("Name");  
JLabel jlabNumber = new JLabel("Number");  
JLabel jlabOptions = new JLabel("Search Options");  
  
// Создание полей редактирования.  
jtfName = new JTextField(10);  
jtfNumber = new JTextField(10);  
  
// Создание флажка опции Ignore Case.  
jcbIgnoreCase = new JCheckBox("Ignore Case");  
// Создание кнопок переключателя опций.  
jrbExact = new JRadioButton("Exact Match", true);
```

```
jrbStartsWith = new JRadioButton("Starts With");  
jrbEndsWith = new JRadioButton("Ends With");
```

5. Добавьте к группе кнопки переключателя опций.

```
// Добавление кнопок к группе.  
ButtonGroup bg = new ButtonGroup();  
bg.add(jrbExact);  
bg.add(jrbStartsWith);  
bg.add(jrbEndsWith);
```

Как вы помните, кнопки, принадлежащие одной группе, являются взаимно исключающими; в каждый момент времени может быть выбрана только одна из них.

6. В данной программе нужны только два обработчика событий действий, связанные с двумя полями редактирования. (Обрабатывать события, генерируемые флажками и переключателями опций, нецелесообразно, так как при необходимости можно определить состояние интересующего нас элемента.) В модуле 1 было сказано, что поле редактирования генерирует событие `ActionEvent`, когда пользователь нажимает клавишу `<Enter>`, при условии, что элемент имеет фокус ввода.

```
// Связывание обработчика события с полем редактирования Name  
jtfName.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent ae) {  
        jtfNumber.setText(lookupName(jtfName.getText()));  
    }  
});
```

```
// Связывание обработчика события с полем редактирования  
// Number.  
jtfNumber.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent ae) {  
        jtfName.setText(lookupNumber(jtfNumber.getText()));  
    }  
});
```

Поскольку в данном случае в качестве обработчиков используются неименованные внутренние классы, объект, сгенерировавший событие, известен. Поэтому в обработчике необходимо лишь вызвать соответствующий метод поиска.

7. Завершите создание конструктора `Phonebook()` добавлением компонентов к панели содержимого. Заметьте, что мы включаем в состав пане-

ли две пустые метки. Это упрощает процесс размещения компонентов.

```
// Добавление компонентов к панели содержимого
jfrm.getContentPane().add(jlabName);
jfrm.getContentPane().add(jtfName);
jfrm.getContentPane().add(jlabNumber);
jfrm.getContentPane().add(jtfNumber);
jfrm.getContentPane().add(new JLabel());
jfrm.getContentPane().add(jlabOptions);
jfrm.getContentPane().add(jcbIgnoreCase);
jfrm.getContentPane().add(new JLabel());
jfrm.getContentPane().add(jrbExact);
jfrm.getContentPane().add(jrbStartsWith);
jfrm.getContentPane().add(jrbEndsWith);

// Отображение фрейма.
jfrm.setVisible(true);
}
```

8. Создайте методы `lookupName()` и `lookupNumber()`. Код этих методов приведен ниже.

```
// Поиск по имени. Метод возвращает номер телефона.
String lookupName(String n) {
    for(int i=0; i < phonelist.length; i++) {
        if(jrbStartsWith.isSelected()) {
            if(jcbIgnoreCase.isSelected()) {
                if(phonelist[i][0].toLowerCase().
                    startsWith(n.toLowerCase()))
                    return phonelist[i][1];
            } else {
                if(phonelist[i][0].startsWith(n))
                    return phonelist[i][1];
            }
        }
        else if(jrbEndsWith.isSelected()) {
            if(jcbIgnoreCase.isSelected()) {
                if(phonelist[i][0].toLowerCase().
                    endsWith(n.toLowerCase()))
                    return phonelist[i][1];
            } else {
                if(phonelist[i][0].endsWith(n))
```



```

        return phonelist[i][1];
    }
}
else {
    if(jcbIgnoreCase.isSelected()) {
        if(phonelist[i][0].toLowerCase().
            equals(n.toLowerCase()))
            return phonelist[i][1];
    } else {
        if(phonelist[i][0].equals(n))
            return phonelist[i][1];
    }
}
}

return "Not Found";
}

// Поиск по номеру. Метод возвращает имя.
String lookupNumber(String n) {
    for(int i=0; i < phonelist.length; i++) {
        if(phonelist[i][1].equals(n))
            return phonelist[i][0];
    }
    return "Not Found";
}

```

Метод `lookupName()` ищет указанное имя и, когда находит его, возвращает соответствующий номер телефона. Метод `lookupNumber()` выполняет поиск заданного номера. Обнаружив его, метод возвращает имя абонента, с которым связан номер. Обратите внимание на метод `lookupName()`. Особенности поиска задаются флажком `Ignore Case` и переключателями опций.

9. В завершение программы, как обычно, надо создать метод `main()`.

```

public static void main(String args[]) {
    // Создание фрейма в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new Phonebook();
        }
    }
}

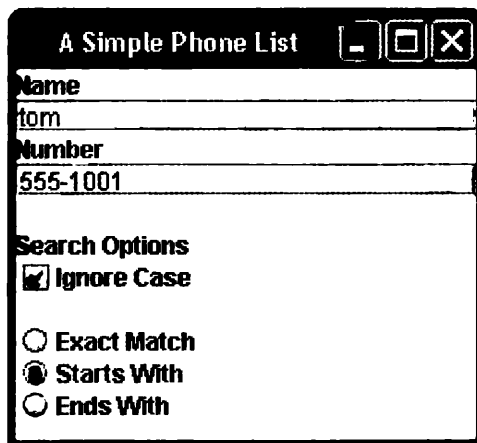
```

```

    });
}
}

```

10. В процессе работы программа отображает окно, показанное ниже.



11. Полностью код программы, поддерживающей телефонный справочник, имеет следующий вид:

```

// Проект 2.2. Простой телефонный справочник
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class Phonebook {

    JTextField jTextFieldName;
    JTextField jTextFieldNumber;

    JRadioButton jrbExact;
    JRadioButton jrbStartsWith;
    JRadioButton jrbEndsWith;

    JCheckBox jcbIgnoreCase;

    // Краткий список имен и номеров телефонов.
    String[][] phonelist = {
        {"Jon", "555-8765"},

```

```
{ "Jessica", "555-5643"},
{ "Adam", "555-1212" },
{ "Rachel", "555-3435"},
{ "Tom & Jerry", "555-1001" }
};

Phonebook() {

    // Создание нового контейнера JFrame.
    JFrame jfrm = new JFrame("A Simple Phone List");

    // Установка диспетчера компоновки GridLayout.
    jfrm.getContentPane().setLayout(new GridLayout(0, 1));

    // Установка начальных размеров фрейма.
    jfrm.setSize(240, 220);

    // Завершение программы при закрытии
    // приложения пользователем.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Создание меток.
    JLabel jlabName = new JLabel("Name");
    JLabel jlabNumber = new JLabel("Number");
    JLabel jlabOptions = new JLabel("Search Options");

    // Создание полей редактирования.
    JTextField jtfName = new JTextField(10);
    JTextField jtfNumber = new JTextField(10);

    // Создание флажка опции Ignore Case.
    JCheckBox jcbIgnoreCase = new JCheckBox("Ignore Case");

    // Создание кнопок переключателя опций.
    JRadioButton jrbExact = new JRadioButton("Exact Match", true);
    JRadioButton jrbStartsWith = new JRadioButton("Starts With");
    JRadioButton jrbEndsWith = new JRadioButton("Ends With");

    // Добавление кнопок к группе.
    ButtonGroup bg = new ButtonGroup();
```

```
bg.add(jrbExact);
bg.add(jrbStartsWith);
bg.add(jrbEndsWith);

// Связывание обработчика события
// с полем редактирования Name.
jtfName.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        jtfNumber.setText(lookupName(jtfName.getText()));
    }
});

// Связывание обработчика события
// с полем редактирования Number.
jtfNumber.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        jtfName.setText(lookupNumber(jtfNumber.getText()));
    }
});

// Добавление компонентов к панели содержимого.
jfrm.getContentPane().add(jlabName);
jfrm.getContentPane().add(jtfName);
jfrm.getContentPane().add(jlabNumber);
jfrm.getContentPane().add(jtfNumber);
jfrm.getContentPane().add(new JLabel());
jfrm.getContentPane().add(jlabOptions);
jfrm.getContentPane().add(jcbIgnoreCase);
jfrm.getContentPane().add(new JLabel());
jfrm.getContentPane().add(jrbExact);
jfrm.getContentPane().add(jrbStartsWith);
jfrm.getContentPane().add(jrbEndsWith);

// Отображение фрейма.
jfrm.setVisible(true);
}

// Поиск по имени. Метод возвращает номер телефона.
String lookupName(String n) {
    for(int i=0; i < phonelist.length; i++) {
        if(jrbStartsWith.isSelected()) {
            if(jcbIgnoreCase.isSelected()) {
                if(phonelist[i][0].toLowerCase().
```

```
        startsWith(n.toLowerCase()))
        return phonelist[i][1];
    } else {
        if(phonelist[i][0].startsWith(n))
            return phonelist[i][1];
    }
}
else if(jrbEndsWith.isSelected()) {
    if(jcbIgnoreCase.isSelected()) {
        if(phonelist[i][0].toLowerCase().
            endsWith(n.toLowerCase()))
            return phonelist[i][1];
    } else {
        if(phonelist[i][0].endsWith(n))
            return phonelist[i][1];
    }
}
else {
    if(jcbIgnoreCase.isSelected()) {
        if(phonelist[i][0].toLowerCase().
            equals(n.toLowerCase()))
            return phonelist[i][1];
    } else {
        if(phonelist[i][0].equals(n))
            return phonelist[i][1];
    }
}
}

return "Not Found";
}

// Поиск по номеру. Метод возвращает имя.
String lookupNumber(String n) {
    for(int i=0; i < phonelist.length; i++) {
        if(phonelist[i][1].equals(n))
            return phonelist[i][0];
    }
    return "Not Found";
}

public static void main(String args[]) {
    // Создание фрейма в потоке обработки событий.
```

```

SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        new Phonebook();
    }
});
}
}

```

## Тест для самоконтроля по модулю 2

1. Какой метод создает обрамление для компонента?
2. По умолчанию содержимое метки выравнивается в вертикальном направлении по центру, а в горизонтальном направлении — по \_\_\_\_\_.
3. С помощью какого метода можно задать вертикальную позицию текста относительно изображения в составе метки?
4. Зачем нужен класс ImageIcon?
5. По умолчанию деактивизированный элемент отображается тусклым цветом. Да или нет?
6. Что такое мнемоническое обозначение?
7. Назовите четыре типа кнопок, предоставляемых Swing.
8. Какой метод надо определить при реализации интерфейса ItemListener?
9. Для каких кнопок с двумя состояниями суперклассом является JToggleButton?
10. Для чего используется метод setDefaultCloseOperation()?
11. В состав какого объекта надо включить объекты JRadioButton, чтобы выбор одной кнопки автоматически отменял выбранную ранее?
12. Сложная задача. Модифицируйте программу Stopwatch, созданную в проекте 1.1, следующим образом.
  - Вместо класса Calendar и метода getTimeInMillis() используйте для получения текущего времени информацию, предоставляемую классом, который описывает событие действия. Как вы помните, эти данные доступны посредством метода getWhen().
  - Сделайте так, чтобы до щелчка на кнопке Start кнопка Stop была недоступна. Также запретите доступ к кнопке Start до щелчка на кнопке Stop.

- Добавьте флажок опции, который будет указывать, должен ли отображаться протокол учтенного времени. Если флажок установлен, информация о времени должна включаться в журнал. При сбросе флажка опции содержимое журнала следует удалить. Хранить и отображать следует лишь последние три записи. При заполнении журнала (в случае, если в нем хранятся три записи), при каждой новой записи удаляйте одну старую из конца списка.

# Модуль 3

## Полосы прокрутки, линейные регуляторы и индикаторы хода процесса

- 3.1.** Модель `BoundedRangeModel`
- 3.2.** Использование полос прокрутки
- 3.3.** Свойства полос прокрутки
- 3.4.** События регулировки
- 3.5.** Дополнительные возможности полос прокрутки
- 3.6.** Использование линейных регуляторов
- 3.7.** Установка маркеров и меток линейных регуляторов
- 3.8.** Дополнительные возможности линейных регуляторов
- 3.9.** Создание индикаторов хода процесса
- 3.10.** Дополнительные возможности индикаторов хода процесса



В данном модуле рассматриваются три элемента графического пользовательского интерфейса: полоса прокрутки, линейный регулятор и индикатор хода процесса. Эти компоненты хорошо знакомы подавляющему большинству читателей, так как они широко применяются в современных приложениях. Наиболее важным из них, наверное, следует считать полосу прокрутки, поскольку она является неотъемлемой составляющей практически каждой программы. Более того, она входила в состав того минимального набора элементов, с которого началось развитие графических интерфейсов. Несмотря на то, что с тех пор пользовательские интерфейсы прошли долгий путь развития, полосы прокрутки работают сейчас точно так же, как и тогда. А это значит, что они успешно прошли испытание временем.

Успех полос прокрутки обусловил появление других элементов, работающих по сходному принципу. Один из них — линейный регулятор, представляющий собой разновидность полосы прокрутки и использующийся для ввода целочисленных значений. Еще один вариант полосы прокрутки — индикатор хода процесса, отображающий сведения о том, какая часть задачи уже выполнена. По внешнему виду полоса прокрутки, линейный регулятор и индикатор хода процесса отличаются друг от друга и используются для различных целей, но все эти компоненты построены по одному и тому же принципу: элемент, представляющий текущее значение, перемещается в них по полю, представляющему заранее определенный диапазон величин. В Swing такое поведение определяет модель, заданная посредством интерфейса `BoundedRangeModel`. Таким образом, и полоса прокрутки, и линейный регулятор, и индикатор хода процесса используют `BoundedRangeModel` для хранения информации о своем состоянии. По этой причине все три элемента рассматриваются в одной главе.

**ВАЖНО!****3.1. Интерфейс `BoundedRangeModel`**

Модель каждого компонента, использующего в работе предопределенный диапазон целочисленных значений, определяется посредством интерфейса `BoundedRangeModel`. В этом интерфейсе определяются четыре основные величины.

- Минимальное значение.
- Максимальное значение.
- Текущее значение.
- Расширение.

Минимальное и максимальное значения определяют границы диапазона, в рамках которого должно находиться текущее значение компонента, основанного на `BoundedRangeModel`. Расширение представляет понятие “ширины” движущегося элемента, который перемещается в пределах области, соответствующей диапазону. Например, в полосе прокрутки расширение соответствует размерам ползунка, выполняющего роль маркера текущей позиции.

Интерфейс `BoundedRangeModel` задает взаимосвязь указанных выше четырех значений. Во-первых, минимальное значение должно быть меньше или равно максимальному. Текущее значение должно быть больше или равно минимальному. Текущее значение плюс расширение не могут превышать максимальное значение. Так, если вы устанавливаете максимальное значение 100 и расширение 20, то текущее значение будет меньше или равно 80 ( $100 - 20$ ). По мере необходимости вы можете задать расширение равным нулю. При этом диапазон расширится. Текущим может быть любое значение, включая минимальное и максимальное.

В интерфейсе `BoundedRangeModel` также объявлено свойство, указывающее на то, что компонент в данный момент претерпевает изменение. Это свойство, для описания которого применяется термин *is-adjusting*, доступно посредством методов `getValueIsAdjusting()` и `setValueIsAdjusting()`. Данное свойство оказывает существенную помощь в работе, поскольку полосы прокрутки и линейные регуляторы при перемещении ползунка в новую позицию генерируют большое количество событий. Обработчик событий имеет возможность дожидаться, когда метод `getValueIsAdjusting()` вернет значение `false`, и лишь тогда предпринимать требуемые действия. Можно также реагировать на все события в реальном времени. Выбор конкретного решения — за разработчиком. Методы `BoundedRangeModel`, предоставляющие доступ к различным свойствам, описаны в табл. 3.1.

Таблица 3.1. Методы, предоставляемые моделью `BoundedRangeModel`

| Метод                                      | Описание                                                                                                                                                           |
|--------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>int getExtent()</code>               | Возвращает величину расширения                                                                                                                                     |
| <code>int getMaximum()</code>              | Возвращает максимальное значение                                                                                                                                   |
| <code>int getMinimum()</code>              | Возвращает минимальное значение                                                                                                                                    |
| <code>int getValue()</code>                | Возвращает текущее значение                                                                                                                                        |
| <code>boolean getValueIsAdjusting()</code> | Возвращает значение <code>true</code> , если пользователь в данный момент изменяет состояние компонента. В противном случае возвращает значение <code>false</code> |
| <code>void setExtent(int val)</code>       | Устанавливает расширение равным <code>val</code>                                                                                                                   |

| Метод                                                                                   | Описание                                                                                                                                                              |
|-----------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void setMaximum(int max)</code>                                                   | Устанавливает максимальное значение равным <code>max</code>                                                                                                           |
| <code>void setMinimum(int min)</code>                                                   | Устанавливает максимальное значение равным <code>min</code>                                                                                                           |
| <code>void setRangeProperties(int val, int ext, int min, int max, boolean isAdj)</code> | Устанавливает все свойства                                                                                                                                            |
| <code>void setValue(int val)</code>                                                     | Устанавливает текущее значение равным <code>val</code>                                                                                                                |
| <code>void setValueIsAdjusting(boolean val)</code>                                      | Значение <code>val</code> , равное <code>true</code> , свидетельствует о выполнении последовательности изменений (например, перетаскивании ползунка полосы прокрутки) |

В интерфейсе `BoundedRangeModel` определены методы `addChangeListener()` и `removeChangeListener()`. Как следует из имен этих методов, в процессе работы рассматриваемых здесь компонентов возникают события изменения состояния. Вскоре вы увидите, что полоса прокрутки преобразует события изменения состояния в *события регулировки*.

В общем случае необходимость в непосредственном обращении к объекту `BoundedRangeModel` не возникает, так как классы, реализующие полосу прокрутки, линейный регулятор и индикатор хода процесса, предоставляют методы для взаимодействия с моделью. Тем не менее модель доступна, и вы можете работать с ней.

Реализацией интерфейса `BoundedRangeModel` с характеристиками, принятыми по умолчанию, является класс `DefaultBoundedRangeModel`. В этом классе текущее, минимальное, максимальное значения и расширение определены так, как показано ниже.

|                       |     |
|-----------------------|-----|
| Текущее значение      | 0   |
| Минимальное значение  | 0   |
| Максимальное значение | 100 |
| Расширение            | 0   |

Очевидно, что при таких установках текущее значение может находиться в пределах от 0 до 100.

## Вопросы для текущего контроля

1. Какие четыре свойства описывает `BoundedRangeModel`?
2. Какое значение вернет метод `getValueIsAdjusting()`, когда пользователь прекратит перемещать ползунок полосы прокрутки или линейного регулятора?
3. Может ли текущее значение превышать максимальное?

**ВАЖНО!**

### 3.2 Полосы прокрутки

В Swing работу полос прокрутки обеспечивает класс `JScrollbar`. Поведение экземпляров этого класса определяется моделью, т.е. интерфейсом `BoundedRangeModel` и его дочерним интерфейсом `java.awt.Adjustable`. (Следует заметить, что интерфейс `Adjustable` реализовывал класс `AWTScrollbar`, с самого начала использовавшийся для представления полос прокрутки.) Учитывая важность и широкое распространение полос прокрутки, не удивительно, что для них разработаны средства, позволяющие без труда применять данный компонент в программах. Несмотря на то что полосы прокрутки обеспечивают высокую степень гибкости, дополнительные средства, предоставляемые данными компонентами, приходится использовать сравнительно редко; их реализация по умолчанию в большинстве случаев устраивает разработчиков.

Полоса прокрутки состоит из нескольких основных частей. На каждом конце полосы расположены кнопки со стрелками; щелчком на такой кнопке можно переместить ползунок на величину *элементарного приращения* — фиксированное расстояние в направлении стрелки. Позиция ползунка отображает текущее значение в диапазоне, определяемом минимальным и максимальным значениями. Ползунок можно переместить в любую позицию. В соответствии с новой позицией устанавливается текущее значение компонента. Это значение будет отражено в модели полосы прокрутки. Щелкнув на полосе за пределами ползунка (эти области называются областями постраничного просмотра), вы переместите ползунок на расстояние, соответствующее *приращению блока*; обычно оно превышает элементарное приращение. Как правило, по щелчку на полосе

1. Текущее, минимальное и максимальное значение, а также расширение.
2. `False`.
3. Нет.

за пределами ползунка осуществляется переход на следующую или предыдущую страницу документа.

### Спросим у опытного программиста

**Вопрос.** Правда ли, что компонент `JScrollbar` не так часто используется в программах, как можно было бы ожидать. Так ли это?

**Ответ.** Несомненно, полоса прокрутки — важнейший элемент программы, созданной с использованием `Swing`. Однако независимая полоса прокрутки встречается сравнительно редко. Так происходит потому, что `Swing` предоставляет специальный компонент под названием `JScrollPane`, реализующий панель, снабженную полосой прокрутки. Компонент `JScrollPane` — это контейнер, который при необходимости автоматически отображает вертикальную и горизонтальную полосы прокрутки. Они позволяют управлять содержимым панели. Таким образом, во многих приложениях удобнее использовать панель с прокруткой, в которой полоса прокрутки автоматически отображается тогда, когда содержимое не помещается в рамках панели. `JScrollPane` — лишь один из нескольких специальных контейнеров, поддерживаемых `Swing`. Их описанию посвящен модуль 4.

В классе `JScrollbar` определены конструкторы, описанные в табл. 3.2. Для полосы прокрутки можно задать вертикальную или горизонтальную ориентацию. Конструктор по умолчанию создает вертикальную полосу прокрутки. Остальные два конструктора позволяют явно указывать ориентацию. Заметьте, что значение параметра, отвечающего за ориентацию компонента, должно быть либо `JScrollbar.VERTICAL`, либо `JScrollbar.HORIZONTAL`. (Эти значения определены в интерфейсе `java.awt.Adjustable`.) Первые два конструктора используют установки по умолчанию. Третий конструктор позволяет явно задать минимальное, максимальное и текущее значение, а также расширение. Для полос прокрутки расширение определяет размер ползунка. При этом физические размеры ползунка, отображаемого на экране, никогда не становятся меньше определенной величины; ползунок должен быть виден на экране, чтобы пользователь мог его перемещать с помощью мыши. Важно помнить, что наибольшее значение, которое может представлять полоса прокрутки, равно максимальному значению минус расширение. По умолчанию границы диапа-

зона для полосы прокрутки устанавливаются равными 0 и 100, расширение равно 10, а начальное значение — 0. Таким образом, по умолчанию значение полосы прокрутки может изменяться в пределах от 0 до 90 (100 – 10).

Таблица 3.2. Конструкторы класса `JScrollBar`

| Конструктор                                                                       | Описание                                                                                                                                                                                                                                                                                                                                                                                                              |
|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JScrollBar()</code>                                                         | Создает вертикальную полосу прокрутки с установками по умолчанию                                                                                                                                                                                                                                                                                                                                                      |
| <code>JScrollBar(int VorH)</code>                                                 | Создает полосу прокрутки, ориентация которой определяется параметром <code>VorH</code> . Значение <code>VorH</code> должно быть равно либо <code>JScrollBar.VERTICAL</code> , либо <code>JScrollBar.HORIZONTAL</code> . Значения основных параметров принимаются по умолчанию                                                                                                                                         |
| <code>JScrollBar(int VorH, int initialValue, int extent, int min, int max)</code> | Создает полосу прокрутки, ориентация которой определяется параметром <code>VorH</code> . Значение <code>VorH</code> должно быть равно либо <code>JScrollBar.VERTICAL</code> , либо <code>JScrollBar.HORIZONTAL</code> . Начальное значение, расширение, минимальное и максимальное значения задаются соответственно параметрами <code>initialValue</code> , <code>extent</code> , <code>min</code> и <code>max</code> |

ВАЖНО!

### 3.3 Свойства объекта `JScrollBar`

В объекте `JScrollBar` предусмотрено несколько свойств, определяющих его поведение. Прежде всего, данный объект поддерживает свойства, которые соответствуют минимальному, максимальному, текущему значению и расширению, описанным в `BoundedRangeModel`. Для доступа к минимальному и максимальному значениям используются следующие методы:

```
int getMinimum()
void setMinimum(int val)
```

```
int getMaximum()
void setMaximum(int val)
```

По умолчанию минимальное значение равно 0, а максимальное — 100. Доступ к текущему значению обеспечивают методы, приведенные ниже.

```
int getValue()
void setValue(int val)
```

По умолчанию это значение устанавливается равным 0. Следующие методы обеспечивают работу с расширением:

```
int getVisibleAmount()  
void setVisibleAmount(int val)
```

Заметьте, что в именах методов присутствует название `VisibleAmount`. Оно применяется потому, что эти методы определены в интерфейсе `Adjustable`, который был создан еще до появления `Swing`. Данные методы обращаются к свойству расширения, определенному в интерфейсе `BoundedRangeModel`. По умолчанию для расширения принимается значение, равное 10.

В объекте `JScrollBar` также определен метод `setValues()`, позволяющий одновременно устанавливать текущее значение, расширение, а также минимальное и максимальное значения. Применение этого метода позволяет несколько сократить исходный код программы. Объявление его выглядит следующим образом:

```
void setValues(int value, int visibleAmount, int min, int max)
```

Объект `JScrollBar` поддерживает свойство *is-adjusting*, описанное в `BoundedRangeModel`, и предоставляет для него следующие методы доступа:

```
boolean getValueIsAdjusting()  
void setValueIsAdjusting()
```

Если ползунок полосы прокрутки перемещается или полоса прокрутки выполняет последовательность команд, соответствующих переходу на страницу вверх или вниз, то метод `getValueIsAdjusting()` возвращает значение `true`. Таким образом компонент сообщает о том, что перед тем, как реагировать на производимые изменения, необходимо дождаться окончания действий пользователя. Например, перетаскивание ползунка порождает набор событий. Если вас интересует только конечное состояние полосы прокрутки, вам надо игнорировать генерируемые события до тех пор, пока пользователь не поместит ползунок в нужную позицию и не отпустит кнопку мыши.

Помимо перетаскивания ползунка, состояние полосы прокрутки можно изменить щелчком на одной из стрелок, расположенных по краям полосы, или щелчком на полосе за пределами ползунка. По щелчку на стрелке полоса прокрутки изменяет свое состояние на фиксированную величину, называемую приращением элемента. Значение приращения определяется свойством, для которого предусмотрен термин *“unit increment”*. Методы, обеспечивающие доступ к нему, приведены ниже.

```
int getUnitIncrement()
void setUnitIncrement(int val)
```

По умолчанию значение свойства равно 1.

По щелчку в области постраничного просмотра позиция ползунка изменяется на приращение блока (свойство `block increment`).

```
int getBlockIncrement()
void setBlockIncrement(int val)
```

По умолчанию значение свойства равно 10.



Существует разновидность методов `getUnitIncrement()` и `getBlockIncrement()`, которая позволяет указать направление приращения. Эти варианты методов в основном предназначены для переопределения в подклассах, в которых, в зависимости от направления прокрутки, используются различные значения приращения.

**ВАЖНО!**

### 3.4. Обработка событий регулировки объекта JScrollBar

Объект `JScrollBar` генерирует события регулировки, которые описываются экземплярами класса `java.awt.event.AdjustmentEvent`. Для обработки событий регулировки необходимо реализовать интерфейс `AdjustmentListener`. В нем объявлен только один метод, `adjustmentValueChanged()`, заголовок которого показан ниже.

```
void adjustmentValueChanged(AdjustmentEvent ae)
```

Этот метод вызывается при любом изменении значения полосы прокрутки.

Ссылку на полосу прокрутки, сгенерировавшую событие, можно получить, вызвав метод `getAdjustable()` объекта `AdjustmentEvent`. Этот метод объявлен следующим образом:

```
Adjustable getAdjustable()
```

Большинство свойств, поддерживаемых `JScrollBar`, объявлено в интерфейсе `Adjustable`, поэтому, получив ссылку, возвращаемую указанным выше методом, вы можете непосредственно работать с этими свойствами.

В классе `AdjustmentEvent` определены два метода, использовать которые очень удобно при работе с полосами прокрутки. Во-первых, вы можете получить текущее значение компонента, вызвав `getValue()`.

```
int getValue()
```



Во-вторых, приведенный ниже метод позволяет выяснить, продолжается ли регулировка полосы прокрутки.

```
boolean getValueIsAdjusting()
```

Эти два метода выполняют те же функции, что и одноименные методы класса `JScrollBar`. Благодаря тому что они присутствуют в объекте события, программист избавлен от необходимости получать ссылку на полосу прокрутки.



### **Вопросы для текущего контроля**

---

1. В дополнение к свойствам, объявленным в `BoundedRangeModel`, существуют два свойства, определяющих поведение полосы прокрутки. Назовите их.
  2. Когда ползунок полосы прокрутки перемещается, генерируется событие \_\_\_\_\_.
  3. Полосы прокрутки могут быть вертикальными, но не горизонтальными. Да или нет?
- 

## **Использование полос прокрутки**

Несмотря на обилие возможностей, полосы прокрутки на удивление просты в применении. Для того чтобы понять особенности работы с данным компонентом, рассмотрим несложный пример, в котором создаются две полосы прокрутки, каждая из которых использует значения свойств по умолчанию. Одна полоса прокрутки — вертикальная, другая — горизонтальная. Таким образом, эти два компонента различаются только ориентацией. Программа отображает текущее значение каждой полосы прокрутки. При перемещении ползунка данные обновляются. Кроме того, в процессе выполнения программы выводятся значения свойств данных компонентов. Окно, создаваемое при работе программы, показано на рис. 3.1.

- 
1. `Unit increment` и `block increment`.
  2. Регулировки.
  3. Нет. Поддерживаются как вертикальные, так и горизонтальные полосы прокрутки.

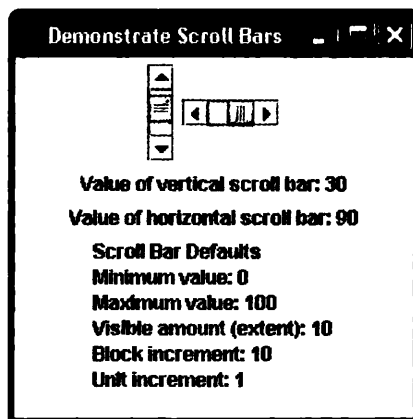


*Рис. 3.1. Данные, отображаемые в ходе работы программы SBDemo*

```
// Использование объектов JScrollBar
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

class SBDemo {

    JLabel jlabVert;
    JLabel jlabHoriz;
    JLabel jlabSBInfo;

    JScrollBar jsbVert;
    JScrollBar jsbHoriz;

    SBDemo() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Demonstrate Scroll Bars");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Определение исходного размера фрейма.
        jfrm.setSize(260, 260);

        // Завершение программы при закрытии пользователем окна.
```

```
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Метки, отображающие текущие значения полос прокрутки.
jlabVert = new JLabel("Value of vertical scroll bar: 0");
jlabHoriz = new JLabel("Value of horizontal scroll bar: 0");

// Создание вертикальной и горизонтальной полос прокрутки
// с установками по умолчанию.
jsbVert = new JScrollBar(); // По умолчанию принимается
                           // вертикальная ориентация.
jsbHoriz = new JScrollBar(Adjustable.HORIZONTAL);

// Обработчики событий регулировки для полос прокрутки.

// Перед тем как выполнить действие, соответствующее
// событию, вертикальная полоса прокрутки ожидает окончания
// действий пользователя.
jsbVert.addAdjustmentListener(new AdjustmentListener() {
    public void adjustmentValueChanged(AdjustmentEvent ae) {
        // Если полоса прокрутки претерпевает изменения,
        // никакие действия не выполняются
        if(jsbVert.getValueIsAdjusting()) return;

        // Отображение нового значения.
        jlabVert.setText("Value of vertical scroll bar: "
            + ae.getValue());
    }
});

// Горизонтальная полоса прокрутки реагирует на все
// события регулировки, независимо от того, выполняет
// ли пользователь действия, изменяющие состояние
// компонента.
jsbHoriz.addAdjustmentListener(new AdjustmentListener() {
    public void adjustmentValueChanged(AdjustmentEvent ae) {
        // Отображение нового значения.
        jlabHoriz.setText("Value of horizontal scroll bar: "
            + ae.getValue());
    }
});

// Отображение свойств полос прокрутки, установленных
// по умолчанию.
```

```
jlabSBInfo = new JLabel("<html>Scroll Bar Defaults<br>" +
    "Minimum value: " +
    jsbVert.getMinimum() + "<br>" +
    "Maximum value: " +
    jsbVert.getMaximum() + "<br>" +
    "Visible amount (extent): " +
    jsbVert.getVisibleAmount() + "<br>" +
    "Block increment: " +
    jsbVert.getBlockIncrement() + "<br>" +
    "Unit increment: " +
    jsbVert.getUnitIncrement());

// Добавление компонентов к панели содержимого.
jfrm.getContentPane().add(jsbVert);
jfrm.getContentPane().add(jsbHoriz);
jfrm.getContentPane().add(jlabVert);
jfrm.getContentPane().add(jlabHoriz);
jfrm.getContentPane().add(jlabSBInfo);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new SBDemo();
        }
    });
}
```

В этой программе есть ряд интересных особенностей. В первую очередь заметьте, что для каждой полосы прокрутки задано максимальное значение 100 и расширение, равное 10. Это означает, что наибольшее текущее значение равно 90. Как вы помните, текущее значение всегда меньше или равно максимальному значению минус расширение. В данном случае это  $90 (100 - 10)$ . Для того чтобы убедиться в этом на практике, запустите программу и переместите ползунок в крайнее положение, соответствующее наибольшему значению. Вы увидите, что текущее значение равно 90. Если вы хотите, чтобы текущее значение достигало максимального, вам надо задать нулевое расширение. Расширить

диапазон можно и другим способом: увеличить максимальное значение на величину приращения. В этом случае текущее значение сможет достичь большей величины.

Теперь рассмотрим, как происходит обработка событий регулировки. Если значение, возвращаемое методом `getValueIsAdjusting()`, равно `true`, обработчик вертикальной полосы прокрутки (`jlabVert`) прекращает работу, не выполняя никаких действий. Это означает, что при перетаскивании ползунка вертикальной полосы прокрутки текущее значение не будет обновлено до тех пор, пока ползунок не будет установлен в конечную позицию. Таким образом, наблюдать на экране значения в процессе их изменения не удастся. Такой подход применим для тех приложений, в которых нет необходимости следить за промежуточными значениями компонента.

Обработчик горизонтальной полосы прокрутки (`jlabHoriz`) обновляет выходные данные при возникновении каждого события. Следовательно, при перемещении ползунка будут отображаться и промежуточные значения. Такое решение приходится принимать при разработке некоторых приложений. Так, например, при перемещении посредством полосы прокрутки текста в окне удобно видеть, какая часть документа отображается при том или ином положении ползунка.

Программа также показывает, что приращение блока равно 10, а приращение элемента — 1. Эти значения принимаются по умолчанию и обычно вполне удовлетворяют требованиям разработчика. Однако, если диапазон изменения значений велик, например от 0 до 10000, вам потребуется большее приращение блока, возможно, придется также увеличить и приращение элемента.

Работая с полосами прокрутки, можно заметить еще одну особенность: в некоторых случаях их размеры оказываются слишком маленькими. Как вы узнаете позже, их нетрудно увеличить.

**ВАЖНО!**

### **3.5. Дополнительные возможности, предоставляемые полосами прокрутки**

Несмотря на то что установки полос прокрутки, предлагаемые по умолчанию, подходят в большинстве случаев, в некоторых случаях все же приходится изменять их. Например, иногда нужен диапазон, отличный от диапазона 0-100, или вам может потребоваться, чтобы текущее значение достигало максимального. Кроме того, в ряде случаев для поддержки большого диапазона значений приходится изменять величину элементарного приращения или приращения блока. И наконец, вы, возможно, захотите задать размеры полосы прокрутки,

которые лучше соответствовали бы вашим представлениям о требуемом внешнем виде приложения. Все эти действия выполняются сравнительно легко.

Как вы уже знаете, текущее, минимальное и максимальное значение, а также расширение можно задать посредством методов доступа, предоставляемых объектом `JScrollBar`. В ряде случаев эти значения легче задать при создании полосы прокрутки, используя приведенный ниже конструктор.

```
JScrollBar(int VorH, int initialValue, int extent, int min, int max)
```

Как вы помните, значение параметра `VorH` может быть равно либо `JScrollBar.VERTICAL`, либо `JScrollBar.HORIZONTAL`. При использовании данного конструктора элементарное приращение устанавливается равным 1, а приращение блока задается посредством параметра `extent`. Однако если значение `extent` равно 0, то приращение блока будет установлено равным 1. Например, при выполнении следующей строки кода создается вертикальная полоса прокрутки с минимальным значением 0 и максимальным 500, расширением 20 и первоначальным значением 250:

```
JScrollBar sb = new JScrollBar(ScrollBar.VERTICAL,  
    250, 25, 0, 500);
```

В данном случае значение полосы прокрутки может изменяться в диапазоне от 0 до 475 (максимальное значение минус расширение). Приращение блока равно 25, а элементарное приращение — 1. Для того чтобы задать другие значения приращения блока и элементарного приращения, вам надо использовать методы `setBlockIncrement()` и `setUnitIncrement()`.

Один из способов (и, наверное, наилучший способ) обеспечить возможность устанавливать текущее значение компонента, равное максимальному, — задать нулевую величину расширения. Поскольку текущее значение не должно превышать разность между максимальным значением и расширением, значение расширения, равное 0, решает задачу. Как было сказано ранее, физический размер ползунка выбирается пропорциональным расширению, но до тех пределов, когда расширение не становится ниже определенного значения. Поэтому, установив нулевое расширение, вы получите ползунок нормальной величины. Еще один способ обеспечить изменения текущего значения в требуемом интервале — установить максимальное значение равным верхней границе интервала плюс расширение. Если же вы не хотите, чтобы размеры ползунка отличались от размеров по умолчанию, лучше задать расширение, равное нулю. В этом случае желательно явно установить приращение блока, так как при нулевом расширении оно будет принято равным единице.

Для установки размера полосы прокрутки можно использовать метод `setPreferredSize()`, определенный в классе `JComponent`. Как вы помните из модуля 2, заголовок этого метода имеет следующий вид:

```
void setPreferredSize(Dimension size)
```

Благодаря данному методу вы можете установить размеры полосы прокрутки в соответствии с требованиями к внешнему виду приложения. Если вы хотите, чтобы полоса прокрутки была больше, чем это предусмотрено по умолчанию, но сохранить стандартную ширину, вы можете получить размеры по умолчанию, вызывая метод `getPreferredSize()`, а затем использовать значение ширины при установке размеров.

Рассмотрим программу, которая создает две полосы прокрутки с размерами, устанавливаемыми явным образом. Обе полосы имеют длину больше, чем предусмотрено по умолчанию. Кроме того, вертикальная полоса прокрутки шире, а горизонтальная — уже стандартной. Окно, создаваемое при работе программы, показано на рис. 3.2.

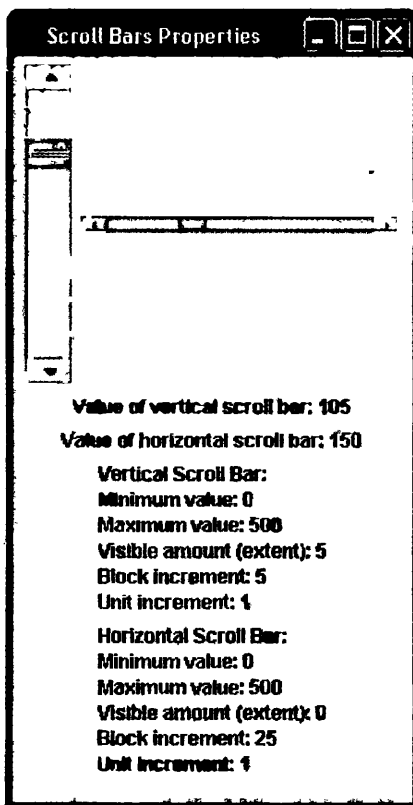


Рис. 3.2. Окно, отображаемое программой *CustomSBDemo*

```
// Установка свойств полосы прокрутки
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

class CustomSBDemo {

    JLabel jlabVert;
    JLabel jlabHoriz;
    JLabel jlabVSBInfo;
    JLabel jlabHSBInfo;

    JScrollBar jsbVert;
    JScrollBar jsbHoriz;

    CustomSBDemo() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Scroll Bars Properties");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(260, 500);

        // Завершение программы при закрытии приложения пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Метки, отображающие текущие значения полос прокрутки.
        jlabVert = new JLabel("Value of vertical scroll bar: 0");
        jlabHoriz = new JLabel("Value of horizontal scroll bar: 250");

        // Установка диапазонов, расширений и начальных
        // значений полос прокрутки.
        jsbVert = new JScrollBar(JScrollBar.VERTICAL,
                                0, // Начальное значение.
                                5, // Расширение.
                                0, // Минимальное значение.
                                500); // Максимальное значение.
        jsbHoriz = new JScrollBar(Adjustable.HORIZONTAL,
                                250, // Начальное значение.
                                0, // Расширение.
```



```
0, // Минимальное значение.
500); // Максимальное значение.

// Установка размеров полос прокрутки.

// Вертикальная полоса прокрутки шире стандартной.
jsbVert.setPreferredSize(new Dimension(30, 200));
// Горизонтальная полоса прокрутки уже стандартной.
jsbHoriz.setPreferredSize(new Dimension(200, 10));

// Установка приращения блока для горизонтальной
// полосы прокрутки.
jsbHoriz.setBlockIncrement(25);

// Обработчики событий регулировки для полос прокрутки.

// Перед тем как выполнить действие, соответствующее
// событию, вертикальная полоса прокрутки ожидает окончания
// действий пользователя.
jsbVert.addAdjustmentListener(new AdjustmentListener() {
    public void adjustmentValueChanged(AdjustmentEvent ae) {
        // Если полоса прокрутки претерпевает изменения,
        // никакие действия не выполняются.
        if(jsbVert.getValueIsAdjusting()) return;

        // Отображение нового значения.
        jlabVert.setText("Value of vertical scroll bar: "
            + ae.getValue());
    }
});

// Горизонтальная полоса прокрутки реагирует на все
// события регулировки, независимо от того, выполняет ли
// пользователь действия, изменяющие состояние компонента.
jsbHoriz.addAdjustmentListener(new AdjustmentListener() {
    public void adjustmentValueChanged(AdjustmentEvent ae) {
        // Отображение нового значения.
        jlabHoriz.setText("Value of horizontal scroll bar: "
            + ae.getValue());
    }
});

// Отображение свойств полос прокрутки.
```

```

jlabVSBInfo = new JLabel("<html>Vertical Scroll Bar:<br>" +
    "Minimum value: " +
    jsbVert.getMinimum() + "<br>" +
    "Maximum value: " +
    jsbVert.getMaximum() + "<br>" +
    "Visible amount (extent): " +
    jsbVert.getVisibleAmount() + "<br>" +
    "Block increment: " +
    jsbVert.getBlockIncrement() + "<br>" +
    "Unit increment: " +
    jsbVert.getUnitIncrement());

jlabHSBInfo = new JLabel("<html>Horizontal Scroll Bar:<br>" +
    "Minimum value: " +
    jsbHoriz.getMinimum() + "<br>" +
    "Maximum value: " +
    jsbHoriz.getMaximum() + "<br>" +
    "Visible amount (extent): " +
    jsbHoriz.getVisibleAmount() + "<br>" +
    "Block increment: " +
    jsbHoriz.getBlockIncrement() + "<br>" +
    "Unit increment: " +
    jsbHoriz.getUnitIncrement());

```

```

// Добавление компонентов к панели содержимого.

```

```

jfrm.getContentPane().add(jsbVert);
jfrm.getContentPane().add(jsbHoriz);
jfrm.getContentPane().add(jlabVert);
jfrm.getContentPane().add(jlabHoriz);
jfrm.getContentPane().add(jlabVSBInfo);
jfrm.getContentPane().add(jlabHSBInfo);

```

```

// Отображение фрейма.

```

```

jfrm.setVisible(true);

```

```

}

```

```

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() { new CustomSBDemo(); }
    });
}

```

```

}

```

Вначале обсудим процесс создания компонентов. Вертикальная полоса прокрутки имеет минимальное значение, равное 0, а максимальное — 500. Ее расширение равно 5, а начальное значение — 0. Учитывая величину расширения, несложно вычислить границы диапазона: 0–495. Приращение блока автоматически устанавливается равным значению расширения, т.е. 5. Вертикальная полоса прокрутки занимает 30 единиц в ширину и 200 единиц в высоту, т.е. она шире стандартной.

Горизонтальная полоса прокрутки имеет минимальное значение, равное 0, а максимальное — 500. Ее расширение равно 0, а начальное значение — 250. Поскольку расширение равно нулю, текущее значение может изменяться в диапазоне от 0 до 500. Горизонтальная полоса прокрутки занимает 10 единиц в высоту и 200 единиц в ширину. Таким образом, она несколько уже, чем принято по умолчанию.

Поскольку расширение равно 0, приращение блока устанавливается равным 1. Однако оно увеличивается до 25 посредством метода `setBlockIncrement()`.



### Вопросы для текущего контроля

1. Каким должно быть расширение полосы прокрутки, чтобы ее текущее значение могло изменяться в диапазоне от минимального до максимального?
2. Какой метод позволяет установить приращение блока?

**ВАЖНО!**

## 3.6. Линейные регуляторы

Линейный регулятор во многом похож на полосу прокрутки. В обоих компонентах ползунок перемещается в рамках области, представляющей диапазон значений; оба они используют в качестве модели `BoundedRangeModel`. Однако линейный регулятор применяется для других целей, и его поведение, события и конфигурация отличаются от полосы прокрутки. Если полоса прокрутки предназначена для перемещения информации (например, текста или изображения) в окне, то линейный регулятор позволяет задать значение в пределах диапазона. Так, например, линейный регулятор может быть использован в качестве регулятора уровня воспроизведения аудиоданных. Для поддержки данного компонента используется класс `Swing.JSlider`.

1. 0.
2. `setBlockIncrement()`.

Линейный регулятор состоит из полосы, по которой перемещается ползунок. Обычно рядом с полосой отображаются маркеры, соответствующие конкретным значениям, и надписи, информирующие о том, какому значению соответствует тот или иной маркер. Однако маркеры значений и надписи рядом с ними могут отсутствовать.

В классе `JSlider` определены конструкторы, показанные в табл. 3.3. Линейный регулятор может быть ориентирован по вертикали либо по горизонтали. По умолчанию принимается горизонтальная ориентация, но при необходимости ее можно изменить. Подобно полосам прокрутки, для линейных регуляторов по умолчанию устанавливается диапазон от 0 до 100, нулевое расширение и исходное значение, равное 50. И диапазон, и расширение, и начальное значение можно установить явно, однако в большинстве случаев этого не требуется, в частности, разработчиков вполне устраивает нулевое расширение.

Таблица 3.3. Конструкторы класса `JSlider`

| Конструктор                                               | Описание                                                                                                                                                                                                                                                                                                                                             |
|-----------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JSlider()</code>                                    | Создает горизонтальный линейный регулятор с диапазоном значений от 0 до 100. Расширение принимается равным 0, а начальное значение — равным 50                                                                                                                                                                                                       |
| <code>JSlider(int min, int max)</code>                    | Создает горизонтальный линейный регулятор. Диапазон определяется параметрами <code>min</code> и <code>max</code> . Расширение принимается равным 0, а начальное значение — середине диапазона                                                                                                                                                        |
| <code>JSlider(int min, int max, int val)</code>           | Создает горизонтальный линейный регулятор. Диапазон определяется параметрами <code>min</code> и <code>max</code> , а начальное значение — параметром <code>val</code> . Расширение принимается равным 0                                                                                                                                              |
| <code>JSlider(int VorH)</code>                            | Создает линейный регулятор, ориентация которого задается параметром <code>VorH</code> (он может принимать значение <code>JSlider.VERTICAL</code> или <code>JSlider.HORIZONTAL</code> ). Диапазон значений — от 0 до 100, расширение равно 0, а начальное значение — 50                                                                               |
| <code>JSlider(int VorH, int min, int max, int val)</code> | Создает линейный регулятор, ориентация которого задается параметром <code>VorH</code> (он может принимать значение <code>JSlider.VERTICAL</code> или <code>JSlider.HORIZONTAL</code> ). Диапазон определяется параметрами <code>min</code> и <code>max</code> , а начальное значение — параметром <code>val</code> . Расширение принимается равным 0 |
| <code>JSlider(BoundedRangeModel model)</code>             | Создает линейный регулятор, который использует модель, заданную посредством параметра <code>model</code>                                                                                                                                                                                                                                             |

При перемещении ползунка объект `JSlider` генерирует событие изменения состояния. (Обратите внимание: этим он отличается от объекта `JSrollBar`, который генерирует события регулировки.) Как вы уже знаете, для поддержки события изменения состояния используется класс, реализующий интерфейс `ChangeListener`. В этом интерфейсе объявлен только один метод, `stateChanged()`, заголовок которого показан ниже.

```
void stateChanged(ChangeEvent ce)
```

Для получения ссылки на линейный регулятор, сгенерировавший событие, используется метод `getSource()`. При перемещении ползунка генерируется последовательность событий. Если вас интересует только конечное положение регулятора, вам следует игнорировать все изменения до тех пор, пока метод `getValueIsAdjusting()` не вернет значение `false`.

Поскольку линейный регулятор использует в качестве модели объект `BoundedRangeModel`, он поддерживает свойства, представляющие минимальное, максимальное и текущее значения, а также расширение. Величина расширения определяет размер ползунка. По умолчанию это значение равно 0, и, как правило, необходимость изменять его не возникает. Данное значение позволяет устанавливать текущее значение в пределах всего диапазона. Если величина расширения задана больше нуля, конечная позиция ползунка будет соответствовать значению меньше максимального, а именно величине, равной максимальному значению минус расширение. (Как вы помните, эта зависимость определена в объекте `BoundedRangeModel`.)

Минимальное, максимальное, текущее значение и расширение, определяемые объектом `BoundedRangeModel`, доступны посредством методов доступа, предоставляемых `JSlider`. Эти методы приведены ниже.

```
int getMinimum()
void setMinimum(int val)

int getMaximum()
void setMaximum(int val)

int getValue()
void setValue(int val)

int getExtent()
void setExtent(int val)
```

Как было сказано ранее, в обычных условиях не приходится явно задавать расширение, поскольку его значение по умолчанию чаще всего устраивает разработчиков приложений.

ВАЖНО!

### 3.7. Установка маркеров и надписей

Несмотря на то что линейный регулятор может не содержать маркеров значений и надписей рядом с ними, желательно все же включать эти элементы, так как они позволяют пользователям более точно представлять себе, какое текущее значение установлено. На вновь созданном линейном регуляторе маркеры и надписи отсутствуют, поэтому их надо задать отдельно. Начнем с маркеров значений.

Линейные регуляторы поддерживают два набора маркеров: основные и вспомогательные. Вспомогательные маркеры располагаются между основными. В ряде случаев разработчики принимают решение использовать только основные маркеры. Для того чтобы задать маркеры, надо сначала определить их расположение. Для основных маркеров сделать это можно посредством показанного ниже метода `setMajorTickSpacing()`.

```
void setMajorTickSpacing(int incr)
```

Параметр `incr` задает здесь число единиц отсчета между маркерами. Например, если указать значение 10, то основные маркеры будут следовать через каждые 10 единиц. Если при этом используется диапазон значений по умолчанию, от 0 до 100, то маркерами будут помечены величины 0, 10, 20 и т.д. до 100. Для того чтобы задать вспомогательные маркеры, надо использовать метод `setMinorTickSpacing()`:

```
void setMinorTickSpacing(int incr)
```

Здесь, как и ранее, параметр `incr` задает число единиц отсчета между маркерами.

Для того чтобы маркеры стали видимыми, их надо “включить”, вызвав метод `setPaintTicks()`. Объявление этого метода выглядит следующим образом:

```
void setPaintTicks(boolean on)
```

Если параметр `on` равен `true`, то маркеры отображаются. Значение `false` приводит к удалению маркеров с экрана.

Описанные выше действия позволяют установить маркеры и сделать их видимыми, но на них отсутствуют надписи, сообщающие о том, каким значениям они соответствуют. Таким образом, сами по себе маркеры показывают приращение, но не величины. Обычно рядом с каждым основным маркером отображается метка, представляющая конкретное значение. Сформировать метки можно посредством метода `setLabelTable()`, показанного ниже.

```
void setLabelTable(Dictionary labs)
```

где `labs` — это набор пар “ключ–значение”, которые связывают значение каждого основного маркера с компонентом `JLabel`. Несмотря на то что подобный набор создать нетрудно, `Swing` предлагает для этого специальный инструмент, подходящий для большинства ситуаций. Чаще всего надпись рядом с маркером отображает значение, которое этот маркер предоставляет. Набор надписей для такого случая можно получить, вызвав метод `createStandardLabels()`:

```
Hashtable createStandardLabels(int incr)
```

где параметр `incr` представляет приращение для меток. Обычно оно совпадает с приращением для основных маркеров.

Для того чтобы метки стали видимыми, надо вызвать метод `setPaintLabels()`, показанный ниже.

```
void setPaintLabels(boolean on)
```

Если значение параметра `on` равно `true`, метки отображаются на экране. Значение `false` запрещает вывод меток.

При желании можно получить текущие установки для маркеров и меток и определить, отображаются ли они на экране. Для этой цели предусмотрены приведенные ниже методы, предоставляемые объектом `JSlider`.

```
Dictionary getLabelTable()
```

```
int getMajorTickSpacing()
```

```
int getMinorTickSpacing()
```

```
boolean getPaintLabels()
```

```
boolean getPaintTicks()
```



### Вопросы для текущего контроля . ————— —

1. Какой диапазон устанавливается для линейного регулятора по умолчанию?
2. Как называются два типа маркеров, используемых в линейных регуляторах?
3. Какой метод надо вызвать, чтобы маркеры отобразились на экране?

- 
1. От 0 до 100.
  2. Основные и вспомогательные.
  3. `setPaintTicks()`.

## Пример, демонстрирующий работу с линейными регуляторами

Ниже приведен код программы, демонстрирующий использование компонентов `JSlider`. Выходные данные, отображаемые при работе программы, показаны на рис. 3.3.

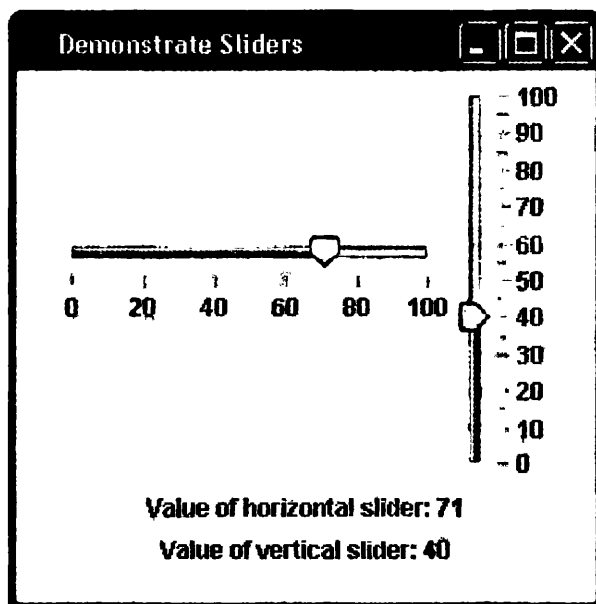


Рис. 3.3. Окно, отображаемое программой, демонстрирующей использование компонентов `JSlider`

```
// Пример, демонстрирующий работу с объектами JSlider
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;

class SliderDemo {

    JLabel jlabVert;
    JLabel jlabHoriz;

    JSlider jsldrHoriz;
    JSlider jsldrVert;

    SliderDemo() {
```



```
// Создание нового контейнера JFrame.
JFrame jfrm = new JFrame("Demonstrate Sliders");

// Установка диспетчера компоновки FlowLayout.
jfrm.getContentPane().setLayout(new FlowLayout());

// Установка начальных размеров фрейма.
jfrm.setSize(300, 300);

// Завершение программы при закрытии приложения пользователем.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Создание вертикального и горизонтального
// линейного регулятора.
jsldrVert = new JSlider(JSlider.VERTICAL);
jsldrHoriz = new JSlider(); // По умолчанию принимается
// горизонтальная ориентация компонента

// Определение расположения основных маркеров
// для обоих компонентов.
jsldrVert.setMajorTickSpacing(10);
jsldrHoriz.setMajorTickSpacing(20);

// Определение расположения вспомогательных маркеров
// для вертикального регулятора.
jsldrVert.setMinorTickSpacing(5);

// Создание стандартных числовых меток.
jsldrVert.setLabelTable(jsldrVert.createStandardLabels(10));
jsldrHoriz.setLabelTable(jsldrHoriz.createStandardLabels(20));

// Разрешение отображения маркеров.
jsldrVert.setPaintTicks(true);
jsldrHoriz.setPaintTicks(true);

// Разрешение отображения меток.
jsldrVert.setPaintLabels(true);
jsldrHoriz.setPaintLabels(true);

// Метки для представления текущего значения регуляторов.
jlabHoriz = new JLabel("Value of horizontal slider: " +
                        jsldrHoriz.getValue());
jlabVert = new JLabel("Value of vertical slider: " +
```

```
jsldrVert.getValue());

// Обработчики событий изменения состояния
// для линейных регуляторов.

// Перед тем как выполнить действие, соответствующее
// событию, горизонтальный регулятор ожидает окончания
// действий пользователя.
jsldrHoriz.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent ce) {
        // Если линейный регулятор претерпевает изменения,
        // никакие действия не выполняются.
        if(jsldrHoriz.getValueIsAdjusting()) return;

        // Отображение нового значения.
        jlabHoriz.setText("Value of horizontal slider: "
            + jsldrHoriz.getValue());
    }
});

// Вертикальный линейный регулятор реагирует на все
// события изменения состояния, независимо от того,
// перемещает ли пользователь ползунок компонента.
jsldrVert.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent ce) {
        // Отображение нового значения.
        jlabVert.setText("Value of vertical slider: "
            + jsldrVert.getValue());
    }
});

// Добавление компонентов к панели содержимого.
jfrm.getContentPane().add(jsldrHoriz);
jfrm.getContentPane().add(jsldrVert);
jfrm.getContentPane().add(jlabHoriz);
jfrm.getContentPane().add(jlabVert);

// Отображение фрейма.
jfrm.setVisible(true);
}
```

```

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new SliderDemo();
        }
    });
}
}

```

В процессе работы программы создаются два линейных регулятора. Один из них — вертикальный, другой — горизонтальный. Для обоих компонентов принят диапазон значений по умолчанию (от 0 до 100). На горизонтальном линейном регуляторе отображаются основные маркеры, следующие через 20 единиц. На вертикальном регуляторе основные маркеры расположены через 10 единиц, а вспомогательные — через 5 единиц.

Как было сказано ранее, при перемещении ползунка линейный регулятор генерирует события изменения состояния. Горизонтальный линейный регулятор вызывает метод `getValueIsAdjusting()`, пропускает все промежуточные события, возникающие в то время, пока ползунок не был установлен в конечную позицию. Таким образом, содержимое метки, на которую ссылается переменная `jlabHoriz`, не будет обновляться до тех пор, пока пользователь перемещает ползунок. Вертикальный линейный регулятор, в отличие от горизонтального, обрабатывает все события. Это означает, что на экране по мере передвижения ползунка будет отображаться текущее значение компонента.

### Спросим у опытного программиста

**Вопрос.** Можно ли выполнять действия с линейным регулятором с помощью клавиатуры.

**Ответ.** Да, это возможно. Если линейный регулятор владеет фокусом ввода, его ползунок можно перемещать посредством клавиш со стрелками. Нажатие клавиши `<PageUp>` или `<PageDown>` также вызывает передвижение ползунка; при этом приращение составляет одну десятую от величины диапазона.

ВАЖНО!

### 3.8. Дополнительные возможности, предоставляемые линейными регуляторами

Несмотря на то что маркеры значений и надписи рядом с ними могут отсутствовать, в ряде случаев линейный регулятор без них оказывается практически бесполезным. Объект `JSlider` предоставляет другие средства, которые упрощают работу пользователя с компонентом. Например, для компонента можно установить режим фиксации, т.е. настроить таким образом, чтобы при перемещении ползунка он автоматически занимал позицию, совпадающую с ближайшим маркером. Этот режим называется привязкой к маркерам. Чтобы установить его, надо вызвать метод `setSnapToTicks()`.

```
void setSnapToTicks(boolean on)
```

Если значение параметра `on` равно `true`, то при отпускании кнопки мыши после перетаскивания ползунка последний автоматически перемещается к ближайшему маркеру. Если для этого параметра задано значение `false`, ползунок остается в той позиции, в которую его поместил пользователь. Для того чтобы выяснить, установлен ли режим фиксации, надо вызвать метод `getSnapToTicks()`.

```
boolean getSnapToTicks()
```

Если режим фиксации установлен, то этот метод возвращает значение `true`.

Для того чтобы увидеть эффект от применения режима привязки к маркерам, добавьте непосредственно после кода, отвечающего за создание линейных регуляторов, следующую строку:

```
jsldrVert.setSnapToTicks(true);
```

Затем попытайтесь передвинуть ползунок вертикального регулятора. Он всегда будет останавливаться в позиции, помеченной маркером.

Еще одна возможность, предоставляемая данным компонентом, также часто оказывается полезной. Речь идет об инвертировании диапазона линейного регулятора. Для того чтобы инвертировать диапазон, надо вызвать метод `setInverted()`.

```
void setInverted(boolean on)
```

Если значение параметра `on` равно `true`, границы диапазона меняются местами. Установив данный параметр равным `false`, вы зададите нормальный порядок следования значений. Заметьте, что понятия “нормальный” и “инверти-

рованный” зависят от используемого языка и региона, для которого настроена программа. Для Северной Америки значения диапазона в обычных условиях возрастают слева направо. При инвертировании порядок следования меняется, и они увеличиваются справа налево. Для того чтобы определить, инвертирован ли диапазон, надо вызвать метод `getInverted()`.

```
boolean getInverted()
```

Если диапазон инвертирован, то этот метод возвращает значение `true`.

Для того чтобы увидеть эффект инвертирования диапазона, добавьте к рассмотренной выше программе после создания линейных регуляторов следующую строку кода:

```
jsldrHoriz.setInverted(true);
```

Как и для других компонентов Swing, вы можете установить предпочтительный размер линейного регулятора, вызвав метод `setPreferredSize()`. Таким образом, если вам не подходит размер по умолчанию, можно задать тот, который наилучшим образом будет соответствовать вашему представлению о том, каким должен быть внешний вид приложения.

## Вопросы для текущего контроля

1. Какой метод надо вызвать для того, чтобы при отпускании ползунка он перемещался в позицию, соответствующую ближайшему маркеру?
2. Можно ли инвертировать диапазон значений линейного регулятора. Если можно, то как это сделать?

## Проект 3.1. Применение пользовательских меток регулятора в интерфейсе программы воспроизведения аудиоинформации

`AudioPlayer.java` Помимо применения стандартных меток, предоставляемых методом `createStandardLabels()` класса `JSlider`, вы можете создать собственную таблицу меток. Пользовательские метки не только придают интерфейсу черты, характерные для приложений определенного типа,

1. `setSnapToTicks()`.
2. Да. Инвертировать диапазон значений линейного регулятора можно, вызвав метод `setInverted()`.

но и упрощают работу с программой, так как информируют пользователя о том, что означает та или иная позиция ползунка. В данном проекте будет показано, как добавить к линейному регулятору пользовательские метки. Вы увидите, что данная процедура достаточно проста, но все же требует выполнения некоторых дополнительных действий.

В данном случае пользовательские метки линейного регулятора применяются для формирования внешнего вида управляющей панели простой программы воспроизведения звука. На панели присутствуют пять линейных регуляторов. Они задают уровень нижних, средних и высоких частот. Четвертый линейный регулятор устанавливает баланс правой и левой колонок, а пятый — уровень воспроизведения. Как вы увидите, метки составлены так, чтобы они поясняли назначение каждого регулятора.

Кроме демонстрации работы с пользовательскими метками, данный проект иллюстрирует две другие возможности Swing. Во-первых, он поясняет принцип взаимодействия различных типов управляющих элементов, присутствующих в составе графического интерфейса. В данном случае переключатели опций позволяют быстро инициализировать линейные регуляторы в соответствии с заранее выбранными установками. Во-вторых, данный проект позволяет составить общее представление о том, что ожидает разработчика, принимающегося за программирование графического интерфейса реального приложения. Представленный здесь интерфейс чрезвычайно прост (в коммерческом приложении обычно предусмотрено гораздо большее количество установок), но он показывает трудозатраты при работе над реальными интерфейсами. Как вы увидите, несмотря на простоту интерфейса данной программы, для его реализации требуется значительный объем кода.

Внешний вид графического пользовательского интерфейса, создаваемого в рамках данного проекта, показан на рис. 3.4. Набор линейных регуляторов позволяет управлять нижними, средними и верхними частотами, а также задавать уровень воспроизведения сигнала. Кроме того, вы можете установить одну из трех сохраненных конфигураций. Первая кнопка переключателя опций задает установки по умолчанию. Остальные две вы можете запрограммировать в соответствии со своими нуждами. Выберите **Preset 1** или **Preset 2**, установите линейные регуляторы в требуемую позицию и щелкните на кнопке **Store Settings**. В результате текущие установки будут записаны в качестве значения соответствующей опции.

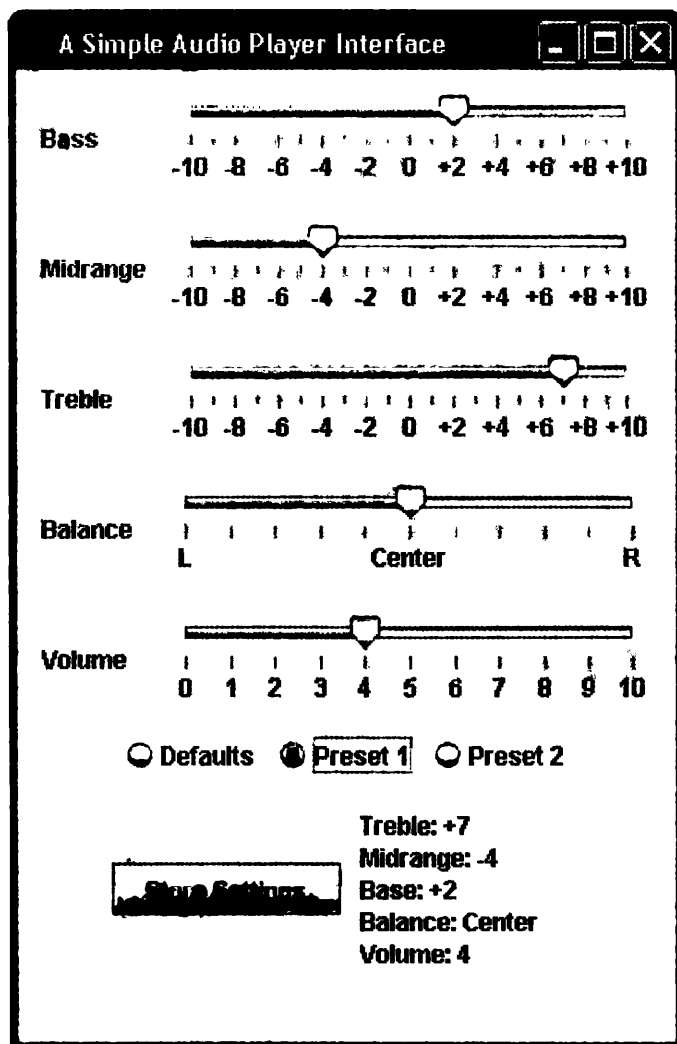


Рис. 3.4. Пользовательский интерфейс, созданный в проекте 3.1

## ПОСЛЕДОВАТЕЛЬНОСТЬ ДЕЙСТВИЙ

1. Создайте файл `AudioPlayer.java` и включите в него следующие комментарии и выражения `import`:

```
// Проект 3.1. Интерфейс Swing аудиоплеера.
```

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

```
import javax.swing.event.*;
import java.text.*;
import java.util.*;
```

2. В этом проекте создаются два класса. Первый — небольшой класс `Presets`, в котором хранятся предопределенные значения. Код этого класса приведен ниже.

```
// Класс для хранения вариантов установок.
class Presets {
    int bass;
    int midrange;
    int treble;
    int balance;
    int volume;

    Presets(int b, int m, int t, int bl, int v) {
        bass = b;
        midrange = m;
        treble = t;
        balance = bl;
        volume = v;
    }
}
```

3. Основной класс называется `AudioPlayer` и начинается следующими строками кода:

```
// Основной класс, используемый для создания
// интерфейса аудиоплеера
class AudioPlayer implements ChangeListener {

    JLabel jlabBass;
    JLabel jlabMidrange;
    JLabel jlabTreble;
    JLabel jlabBalance;
    JLabel jlabVolume;
    JLabel jlabInfo;

    JSlider jsldrBass;
    JSlider jsldrMidrange;
    JSlider jsldrTreble;
    JSlider jsldrBalance;
    JSlider jsldrVolume;
```



```
JRadioButton jrbPreset1;
JRadioButton jrbPreset2;
JRadioButton jrbDefaults;
```

```
JButton jbtnStore;
```

```
DecimalFormat df;
```

```
Presets[] presets;
```

В данном случае необходимо обратить внимание на три особенности. Во-первых, класс `AudioPlayer` реализует интерфейс `ChangeListener`. Как вы помните, линейные регуляторы генерируют события изменения состояния. Класс `AudioPlayer` предоставляет метод `stateChanged()` для поддержки этих событий. Во-вторых, в классе объявлена переменная `df`, предназначенная для хранения ссылки на объект `DecimalFormat`. Этот объект используется для форматирования значений, отображаемых в линейном регуляторе. В-третьих, в программе объявлен массив объектов `Presets`, в котором содержатся сохраненные конфигурации.

#### 4. Начните код конструктора `AudioPlayer` следующим образом:

```
AudioPlayer() {
    // Создание нового контейнера JFrame.
    JFrame jfrm = new JFrame("A Simple Audio Player
Interface");

    // Установка диспетчера компоновки FlowLayout.
    jfrm.getContentPane().setLayout(new FlowLayout());

    // Установка начальных размеров фрейма.
    jfrm.setSize(340, 520);

    // Завершение программы при закрытии приложения
    // пользователем.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Создание объекта форматирования, который будет
    // отображать знаки + и -.
    df = new DecimalFormat("+#;-#");
```

Большая часть данного кода знакома вам. Новой для вас является только команда создания объекта `DecimalFormat`. В данном случае при вызове конструктора задается формат, согласно которому положительные

значения отображаются со знаком +, а отрицательные — со знаком -. Результаты применения объекта форматирования вы увидите несколько позже.

5. Продолжите код конструктора `AudioPlayer()` следующими выражениями:

```
// Установка предопределенных значений.
setupPresets();

// Создание линейных регуляторов.
setupSliders();

// Создание меток, описывающих линейные регуляторы.
setupLabels();

// Создание кнопок переключателя опций
// для хранимых установок.
setupRButtons();

// Создание кнопки Store Settings.
jbtnStore = new JButton("Store Settings");
```

Данная последовательность команд устанавливает состояние компонентов. Поскольку код, необходимый для создания и инициализации предопределенных состояний, имеет достаточно большой объем, он организован в виде двух методов, которые вызываются из конструктора `AudioPlayer`.

6. Добавьте обработчики событий.

```
// Добавление обработчиков событий,
// генерируемых линейными регуляторами.
jsldrBass.addChangeListener(this);
jsldrMidrange.addChangeListener(this);
jsldrTreble.addChangeListener(this);
jsldrBalance.addChangeListener(this);
jsldrVolume.addChangeListener(this);

// Обработка событий, связанных с кнопкой Store Settings.
jbtnStore.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        if(jrbPreset1.isSelected())
            storePreset(presets[1]);
        else if(jrbPreset2.isSelected())
```

```

        storePreset(presets[2]);
    }
});

// Обработка событий кнопки переключателя опций Defaults.
jrbDefaults.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        loadPreset(presets[0]);
    }
});

// Обработка событий кнопки переключателя опций Preset 1.
jrbPreset1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        loadPreset(presets[1]);
    }
});

// Обработка событий кнопки переключателя опций Preset 2.
jrbPreset2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        loadPreset(presets[2]);
    }
});

```

Обработчик событий изменения состояния, генерируемых линейными регуляторами, реализован в виде независимого метода. Обработчики кнопок переключателя опций и обычной кнопки реализованы как неименованные внутренние классы.

При выборе кнопки переключателя опций соответствующие ей установки загружаются посредством метода `loadPreset()`. По щелчку на кнопке **Store Settings** происходит загрузка текущих установок; для этой цели вызывается метод `storePreset()`. Обоим этим методам передается ссылка на элемент массива предварительных установок, соответствующий выбранной кнопке переключателя опций. Однако сохранять можно только установки для **Preset 1** и **Preset 2**. Если выбрана кнопка **Defaults**, щелчок на кнопке **Store Settings** не произведет никакого эффекта.

#### 7. Завершите код `AudioPlayer()` следующим образом:

```

// Добавление компонентов к панели содержимого
Container cp = jfrm.getContentPane();

```

```
cp.add(jlabBass);
cp.add(jsldrBass);
cp.add(jlabMidrange);
cp.add(jsldrMidrange);
cp.add(jlabTreble);
cp.add(jsldrTreble);
cp.add(jlabBalance);
cp.add(jsldrBalance);
cp.add(jlabVolume);
cp.add(jsldrVolume);
cp.add(jrbDefaults);
cp.add(jrbPreset1);
cp.add(jrbPreset2);
cp.add(jbtnStore);
cp.add(new JLabel(""));
cp.add(jlabInfo);
```

```
// Отображение фрейма.
jfrm.setVisible(true);
}
```

**8. Свяжите с линейными регуляторами обработчик `stateChanged()`.**

```
// Обработка событий линейного регулятора
public void stateChanged(ChangeEvent ce) {
    showSettings(); // Обновление отображаемой информации.
}
```

Заметьте, что в данном случае в теле метода `stateChanged()` присутствует лишь вызов метода `showSettings()`, который отображает текущие установки посредством текстовых меток.

**9. Напишите код метода `showSettings()`.**

```
// Отображение текущих установок.
void showSettings() {
    String bal;

    // Получение установок баланса.
    int b = jsldrBalance.getValue();
    if(b > 0)
        bal = "Right " + df.format(jsldrBalance.getValue());
    else if(b==0)
        bal = "Center";
    else
```

```

        bal = "Left " + df.format(-jsldrBalance.getValue());

        jlabInfo.setText("<html>Treble: " +
            df.format(jsldrTreble.getValue()) +
            "<br>Midrange: " +
            df.format(jsldrMidrange.getValue()) +
            "<br>Base: " +
            df.format(jsldrBass.getValue()) +
            "<br>Balance: " + bal +
            "<br>Volume: " +
            jsldrVolume.getValue());
    }

```

Этот метод задает текст метки `jlabInfo`, в результате она отражает текущее состояние линейных регуляторов.

10. Добавьте метод `setupSliders()`, который создает, настраивает и инициализирует линейные регуляторы.

```

// Создание и инициализация линейных регуляторов.
void setupSliders() {
    // Создание линейных регуляторов.
    jsldrBass = new JSlider(-10, 10);
    jsldrMidrange = new JSlider(-10, 10);
    jsldrTreble = new JSlider(-10, 10);
    jsldrVolume = new JSlider(0, 10, 0);
    jsldrBalance = new JSlider(-5, 5);

    // Добавление основных маркеров.
    jsldrBass.setMajorTickSpacing(2);
    jsldrMidrange.setMajorTickSpacing(2);
    jsldrTreble.setMajorTickSpacing(2);
    jsldrVolume.setMajorTickSpacing(1);
    jsldrBalance.setMajorTickSpacing(1);

    // Добавление вспомогательных маркеров.
    jsldrBass.setMinorTickSpacing(1);
    jsldrMidrange.setMinorTickSpacing(1);
    jsldrTreble.setMinorTickSpacing(1);

    // Создание меток для трех линейных регуляторов.
    Hashtable table = new Hashtable();
    for(int i = -10; i <= 0; i += 2)
        table.put(new Integer(i), new JLabel("" + i));
}

```

```
for(int i = 2; i <= 10; i += 2)
    table.put(new Integer(i), new JLabel("+" + i));
jsldrTreble.setLabelTable(table);
jsldrMidrange.setLabelTable(table);
jsldrBass.setLabelTable(table);

// Создание меток для линейного регулятора Balance.
table = new Hashtable();
table.put(new Integer(0), new JLabel("Center"));
table.put(new Integer(-5), new JLabel("L"));
table.put(new Integer(5), new JLabel("R"));
jsldrBalance.setLabelTable(table);

// Создание стандартных числовых меток
// для линейного регулятора Volume.
jsldrVolume.setLabelTable(
    jsldrVolume.createStandardLabels(1));

// Разрешение отображения маркеров.
jsldrBass.setPaintTicks(true);
jsldrMidrange.setPaintTicks(true);
jsldrTreble.setPaintTicks(true);
jsldrVolume.setPaintTicks(true);
jsldrBalance.setPaintTicks(true);

// Разрешение отображения меток.
jsldrBass.setPaintLabels(true);
jsldrMidrange.setPaintLabels(true);
jsldrTreble.setPaintLabels(true);
jsldrVolume.setPaintLabels(true);
jsldrBalance.setPaintLabels(true);

// Перемещение к ближайшему маркеру.
jsldrBass.setSnapToTicks(true);
jsldrMidrange.setSnapToTicks(true);
jsldrTreble.setSnapToTicks(true);
jsldrVolume.setSnapToTicks(true);
jsldrBalance.setSnapToTicks(true);

// Установка предпочтительных размеров линейных регуляторов.
Dimension sldrSize = new Dimension(240, 60);
jsldrBass.setPreferredSize(sldrSize);
jsldrMidrange.setPreferredSize(sldrSize);
```

```

jsldrTreble.setPreferredSize(sldrSize);
jsldrVolume.setPreferredSize(sldrSize);
jsldrBalance.setPreferredSize(sldrSize);
}

```

Как видите, объем кода достаточно велик, но большая его часть уже знакома вам. Необходимо лишь обратить внимание на две особенности. Во-первых, диапазоны всех компонентов, за исключением регулятора уровня воспроизведения, содержат как положительные, так и отрицательные значения. Подобный подход нередко используется при работе с линейными регуляторами. Во-вторых, в составе данного метода присутствует код, предназначенный для создания меток регулятора. При этом применяются три различных подхода. Первые три регулятора, ориентированные на различные частоты, используют общие метки, создаваемые посредством следующего фрагмента кода:

```

// Создание меток для трех линейных регуляторов.
Hashtable table = new Hashtable();
for(int i = -10; i <= 0; i += 2)
    table.put(new Integer(i), new JLabel("" + i));
for(int i = 2; i <= 10; i += 2)
    table.put(new Integer(i), new JLabel("+" + i));
jsldrTreble.setLabelTable(table);
jsldrMidrange.setLabelTable(table);
jsldrBass.setLabelTable(table);

```

Как вы помните, установка меток регуляторов осуществляется посредством метода `setLabelTable()`. При вызове этому методу передается один параметр — ссылка на объект `Dictionary`, который должен содержать таблицу меток и соответствующие им значения. Таблица организована в виде пар значений, которые имеют следующий вид:

*значение, метка*

В обычных условиях значение должно быть экземпляром класса `Integer`, а метка — объектом `JLabel`. Однако можно использовать и другие компоненты, соответствующие вашим потребностям. Как вы, вероятно, знаете, `Dictionary` — это абстрактный класс, и непосредственно создать такой объект невозможно. Однако существует реальный класс, `Hashtable`, наследующий возможности абстрактного класса `Dictionary`. Это означает, что экземпляр данного класса может быть указан при вызове метода `setLabelTable()`. Таблица заполняется значениями от -10 до +10. Заметьте, что перед положительными значениями указывается знак +.

Пользовательские метки для линейного регулятора Balance создаются с помощью следующего кода:

```
// Создание меток для линейного регулятора Balance.
table = new Hashtable();
table.put(new Integer(0), new JLabel("Center"));
table.put(new Integer(-5), new JLabel("L"));
table.put(new Integer(5), new JLabel("R"));
jsldrBalance.setLabelTable(table);
```

Обратите внимание, что в данном случае метки отображают не числовую информацию, а содержат строковые значения — "L", "Center" и "R". Для данного регулятора метки выбраны исходя из конкретных потребностей приложения.

Заметьте, что предыдущий код работает в любой версии Java. Однако, если вы используете JDK 5 или более позднюю версию, целесообразно указать универсальное объявление для Hashtable, приведенное ниже.

```
Hashtable<Integer, JLabel> table = new Hashtable<Integer,
JLabel>();
```

Для связывания нового объекта Hashtable с линейным регулятором Balance можно использовать следующую строку кода:

```
table = new Hashtable<Integer, JLabel>();
```

Использование универсального кода позволит избежать предупреждающих сообщений при компиляции в JDK 5 или более поздних версиях. Такой подход также уменьшает вероятность ошибки, связанной с использованием типов.

11. Создайте метод `setupLabels()`. Этот метод формирует метки, описывающие назначение линейных регуляторов. Они отображают слова Bass, Midrange, Treble, Balance и Volume. В теле метода также создается метка, отображающая текущие установки.

```
// Создание меток для описания линейных регуляторов
void setupLabels() {
    // Создание меток.
    jlabTreble = new JLabel("Treble");
    jlabMidrange = new JLabel("Midrange");
    jlabBass = new JLabel("Bass");
    jlabVolume = new JLabel("Volume");
    jlabBalance = new JLabel("Balance");

    // Установка размеров меток.
```



```

Dimension labSize = new Dimension(60, 25);
jlabTreble.setPreferredSize(labSize);
jlabMidrange.setPreferredSize(labSize);
jlabBass.setPreferredSize(labSize);
jlabVolume.setPreferredSize(labSize);
jlabBalance.setPreferredSize(labSize);

// Создание информационной метки и установка ее размеров.
jlabInfo = new JLabel("");
jlabInfo.setPreferredSize(new Dimension(110, 100));

// Заполнение jlabInfo информацией, соответствующей
// установкам по умолчанию.
showSettings();
}

```

12. Добавьте код для инициализации кнопок переключателя опций, позволяющих выбрать установку.

```

// Создание и инициализация кнопок переключателя опций.
void setupRButtons() {
    // Создание кнопок.
    jrbDefaults = new JRadioButton("Defaults");
    jrbPreset1 = new JRadioButton("Preset 1");
    jrbPreset2 = new JRadioButton("Preset 2");

    // Добавление кнопок к группе.
    ButtonGroup bg = new ButtonGroup();
    bg.add(jrbDefaults);
    bg.add(jrbPreset1);
    bg.add(jrbPreset2);

    // Выбор кнопки Defaults.
    jrbDefaults.setSelected(true);
}

```

13. Добавьте метод `loadPreset()`, который устанавливает состояние линейных регуляторов, соответствующее информации, переданной в качестве параметра. Этот метод вызывается из обработчика событий переключателя опций.

```

// Загрузка ранее определенных установок.
void loadPreset(Presets info) {
    jsldrBass.setValue(info.bass);
    jsldrMidrange.setValue(info.midrange);
}

```

```
jsldrTreble.setValue(info.treble);  
jsldrBalance.setValue(info.balance);  
jsldrVolume.setValue(info.volume);  
}
```

Заметьте, что методу `loadPreset()` передается параметр типа `Presets`.

14. Добавьте метод `storePreset()`, который позволяет помещать информацию о текущем состоянии в массив `presets`. Этот метод вызывается из обработчика событий, соответствующего кнопке `Store Settings`.

```
// Сохранение установок.  
void storePreset(Presets info) {  
    info.bass = jsldrBass.getValue();  
    info.midrange = jsldrMidrange.getValue();  
    info.treble = jsldrTreble.getValue();  
    info.balance = jsldrBalance.getValue();  
    info.volume = jsldrVolume.getValue();  
}
```

Данному методу передается набор текущих установок линейного регулятора. В процессе работы он копирует их в массив, на который ссылается переменная `info`.

15. Введите приведенный ниже код метода `setupPresets()`. Этот метод инициализирует массив предварительных установок значениями по умолчанию и двумя наборами данных, заданных пользователем.

```
// Инициализация массива предварительных установок.  
void setupPresets() {  
    presets = new Presets[3];  
    presets[0] = new Presets(0, 0, 0, 0, 0);  
    presets[1] = new Presets(2, -4, 7, 0, 4);  
    presets[2] = new Presets(3, 3, -2, 1, 7);  
}
```

Значения, записываемые в элементы `presets[1]` и `presets[2]`, могут быть произвольными. Вы можете использовать любые установки, необходимо лишь, чтобы значения не выходили за пределы диапазона.

16. Завершите код класса `AudioPlayer` методом `main()`.

```
public static void main(String args[]) {  
    // Создание фрейма в потоке обработки событий.  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            new AudioPlayer();  
        }  
    });  
}
```

```

    }
  });
}
}

```

17. Полностью код программы приведен ниже. Вы можете поэкспериментировать с ней, например изменить метки. Как видите, объем кода достаточно велик. Анализируя его, можно сделать вывод, что для создания реальных интерфейсов Swing необходимо выполнить большой объем работы.

// Проект 3.1. Интерфейс Swing аудиоплеера.

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.text.*;
import java.util.*;

// Класс для хранения вариантов установок.
class Presets {
    int bass;
    int midrange;
    int treble;
    int balance;
    int volume;

    Presets(int b, int m, int t, int bl, int v) {
        bass = b;
        midrange = m;
        treble = t;
        balance = bl;
        volume = v;
    }
}

// Основной класс, используемый для создания
// интерфейса аудиоплеера.
class AudioPlayer implements ChangeListener {

    JLabel jlabBass;
    JLabel jlabMidrange;
    JLabel jlabTreble;
    JLabel jlabBalance;

```

```
JLabel jlabVolume;  
JLabel jlabInfo;  
  
JSlider jsldrBass;  
JSlider jsldrMidrange;  
JSlider jsldrTreble;  
JSlider jsldrBalance;  
JSlider jsldrVolume;  
  
JRadioButton jrbPreset1;  
JRadioButton jrbPreset2;  
JRadioButton jrbDefaults;  
  
JButton jbtnStore;  
  
DecimalFormat df;  
  
Presets[] presets;  
  
AudioPlayer() {  
    // Создание нового контейнера JFrame.  
    JFrame jfrm = new JFrame("A Simple Audio Player Interface");  
  
    // Установка диспетчера компоновки FlowLayout.  
    jfrm.getContentPane().setLayout(new FlowLayout());  
  
    // Установка начальных размеров фрейма.  
    jfrm.setSize(340, 520);  
  
    // Завершение программы  
    // при закрытии приложения пользователем.  
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
    // Создание объекта форматирования, который будет  
    // отображать знаки + и -.  
    df = new DecimalFormat("+#;-#");  
  
    // Установка предопределенных значений.  
    setupPresets();  
  
    // Создание линейных регуляторов.  
    setupSliders();  
}
```

```
// Создание меток, описывающих линейные регуляторы.
setupLabels();

// Создание кнопок переключателя опций
// для хранимых установок.
setupRButtons();

// Создание кнопки Store Settings.
jbtnStore = new JButton("Store Settings");

// Добавление обработчиков событий,
// генерируемых линейными регуляторами.
jsldrBass.addChangeListener(this);
jsldrMidrange.addChangeListener(this);
jsldrTreble.addChangeListener(this);
jsldrBalance.addChangeListener(this);
jsldrVolume.addChangeListener(this);

// Обработка событий, связанных с кнопкой Store Settings.
jbtnStore.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        if(jrbPreset1.isSelected())
            storePreset(presets[1]);
        else if(jrbPreset2.isSelected())
            storePreset(presets[2]);
    }
});

// Обработка событий кнопки переключателя опций Defaults.
jrbDefaults.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        loadPreset(presets[0]);
    }
});

// Обработка событий кнопки переключателя опций Preset 1.
jrbPreset1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        loadPreset(presets[1]);
    }
});

// Обработка событий кнопки переключателя опций Preset 2.
```

```
jrbPreset2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        loadPreset(presets[2]);
    }
});

// Добавление компонентов к панели содержимого.
Container cp = jfrm.getContentPane();
cp.add(jlabBass);
cp.add(jsldrBass);
cp.add(jlabMidrange);
cp.add(jsldrMidrange);
cp.add(jlabTreble);
cp.add(jsldrTreble);
cp.add(jlabBalance);
cp.add(jsldrBalance);
cp.add(jlabVolume);
cp.add(jsldrVolume);
cp.add(jrbDefaults);
cp.add(jrbPreset1);
cp.add(jrbPreset2);
cp.add(jbtnStore);
cp.add(new JLabel(""));
cp.add(jlabInfo);

// Отображение фрейма.
jfrm.setVisible(true);
}

// Обработка событий линейного регулятора.
public void stateChanged(ChangeEvent ce) {
    showSettings(); // update the info display
}

// Создание и инициализация линейных регуляторов.
void setupSliders() {
    // Создание линейных регуляторов.
    jsldrBass = new JSlider(-10, 10);
    jsldrMidrange = new JSlider(-10, 10);
    jsldrTreble = new JSlider(-10, 10);
    jsldrVolume = new JSlider(0, 10, 0);
    jsldrBalance = new JSlider(-5, 5);
```

```
// Добавление основных маркеров.
jsldrBass.setMajorTickSpacing(2);
jsldrMidrange.setMajorTickSpacing(2);
jsldrTreble.setMajorTickSpacing(2);
jsldrVolume.setMajorTickSpacing(1);
jsldrBalance.setMajorTickSpacing(1);

// Добавление вспомогательных маркеров.
jsldrBass.setMinorTickSpacing(1);
jsldrMidrange.setMinorTickSpacing(1);
jsldrTreble.setMinorTickSpacing(1);

// Создание меток для трех линейных регуляторов.
Hashtable table = new Hashtable();
for(int i = -10; i <= 0; i += 2)
    table.put(new Integer(i), new JLabel("" + i));
for(int i = 2; i <= 10; i += 2)
    table.put(new Integer(i), new JLabel("+" + i));
jsldrTreble.setLabelTable(table);
jsldrMidrange.setLabelTable(table);
jsldrBass.setLabelTable(table);

// Создание меток для линейного регулятора Balance.
table = new Hashtable();
table.put(new Integer(0), new JLabel("Center"));
table.put(new Integer(-5), new JLabel("L"));
table.put(new Integer(5), new JLabel("R"));
jsldrBalance.setLabelTable(table);

// Создание стандартных числовых меток
// для линейного регулятора Volume.
jsldrVolume.setLabelTable
    (jsldrVolume.createStandardLabels(1));

// Разрешение отображения маркеров.
jsldrBass.setPaintTicks(true);
jsldrMidrange.setPaintTicks(true);
jsldrTreble.setPaintTicks(true);
jsldrVolume.setPaintTicks(true);
jsldrBalance.setPaintTicks(true);

// Разрешение отображения меток.
jsldrBass.setPaintLabels(true);
```

```
jsldrMidrange.setPaintLabels(true);
jsldrTreble.setPaintLabels(true);
jsldrVolume.setPaintLabels(true);
jsldrBalance.setPaintLabels(true);

// Перемещение к ближайшему маркеру.
jsldrBass.setSnapToTicks(true);
jsldrMidrange.setSnapToTicks(true);
jsldrTreble.setSnapToTicks(true);
jsldrVolume.setSnapToTicks(true);
jsldrBalance.setSnapToTicks(true);

// Установка предпочтительных размеров линейных
// регуляторов.
Dimension sldrSize = new Dimension(240, 60);
jsldrBass.setPreferredSize(sldrSize);
jsldrMidrange.setPreferredSize(sldrSize);
jsldrTreble.setPreferredSize(sldrSize);
jsldrVolume.setPreferredSize(sldrSize);
jsldrBalance.setPreferredSize(sldrSize);
}

// Создание меток для описания линейных регуляторов.
void setupLabels() {
    // Создание меток.
    jlabTreble = new JLabel("Treble");
    jlabMidrange = new JLabel("Midrange");
    jlabBass = new JLabel("Bass");
    jlabVolume = new JLabel("Volume");
    jlabBalance = new JLabel("Balance");

    // Установка размеров меток.
    Dimension labSize = new Dimension(60, 25);
    jlabTreble.setPreferredSize(labSize);
    jlabMidrange.setPreferredSize(labSize);
    jlabBass.setPreferredSize(labSize);
    jlabVolume.setPreferredSize(labSize);
    jlabBalance.setPreferredSize(labSize);

    // Создание информационной метки и установка ее размеров.
    jlabInfo = new JLabel("");
    jlabInfo.setPreferredSize(new Dimension(110, 100));
}
```



```
// Заполнение jlabInfo информацией, соответствующей
// установкам по умолчанию.
showSettings();
}

// Создание и инициализация кнопок переключателя опций.
void setupRButtons() {
    // Создание кнопок.
    jrbDefaults = new JRadioButton("Defaults");
    jrbPreset1 = new JRadioButton("Preset 1");
    jrbPreset2 = new JRadioButton("Preset 2");

    // Добавление кнопок к группе.
    ButtonGroup bg = new ButtonGroup();
    bg.add(jrbDefaults);
    bg.add(jrbPreset1);
    bg.add(jrbPreset2);

    // Выбор кнопки Defaults.
    jrbDefaults.setSelected(true);
}

// Отображение текущих установок.
void showSettings() {
    String bal;

    // Получение установок баланса.
    int b = jsldrBalance.getValue();
    if(b > 0)
        bal = "Right " + df.format(jsldrBalance.getValue());
    else if(b==0)
        bal = "Center";
    else
        bal = "Left " + df.format(-jsldrBalance.getValue());

    jlabInfo.setText("<html>Treble: " +
        df.format(jsldrTreble.getValue()) +
        "<br>Midrange: " +
        df.format(jsldrMidrange.getValue()) +
        "<br>Base: " +
        df.format(jsldrBass.getValue()) +
        "<br>Balance: " + bal +
```

```
        "<br>Volume: " +
        jsldrVolume.getValue());

    }

    // Инициализация массива предварительных установок.
    void setupPresets() {

        presets = new Presets[3];
        presets[0] = new Presets(0, 0, 0, 0, 0);
        presets[1] = new Presets(2, -4, 7, 0, 4);
        presets[2] = new Presets(3, 3, -2, 1, 7);
    }

    // Сохранение установок.
    void storePreset(Presets info) {
        info.bass = jsldrBass.getValue();
        info.midrange = jsldrMidrange.getValue();
        info.treble = jsldrTreble.getValue();
        info.balance = jsldrBalance.getValue();
        info.volume = jsldrVolume.getValue();
    }

    // Загрузка ранее определенных установок.
    void loadPreset(Presets info) {
        jsldrBass.setValue(info.bass);
        jsldrMidrange.setValue(info.midrange);
        jsldrTreble.setValue(info.treble);
        jsldrBalance.setValue(info.balance);
        jsldrVolume.setValue(info.volume);
    }

    public static void main(String args[]) {
        // Создание фрейма в потоке обработки событий.
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new AudioPlayer();
            }
        });
    }
}
```

**ВАЖНО!****3.9. Индикаторы хода процесса**

Индикаторы хода процесса часто используются в составе графических интерфейсов. Они, несомненно, знакомы каждому читателю. Компоненты такого типа отображают на экране полосу, которая индицирует, какая часть задачи выполнена на текущий момент. По мере того как задача движется к завершению, полоса становится длиннее. Несмотря на то что индикатор хода процесса — достаточно простой элемент, он существенно повышает качество интерфейса приложения, так как визуально представляет важную для пользователя информацию. Для поддержки индикаторов хода процесса используется класс `JProgressBar`. В роли модели выступает `BoundedRangeModel`.

В классе `JProgressBar` определены конструкторы, показанные в табл. 3.4. Индикатор хода процесса может быть ориентирован по горизонтали и по вертикали. По умолчанию принимается горизонтальная ориентация, но при необходимости ее можно изменить. Если параметры не установлены явно, то принимается диапазон значений от 0 до 100, а в качестве начального выбирается значение 0.

Таблица 3.4. Конструкторы класса `JProgressBar`

| Конструктор                                           | Описание                                                                                                                                                                                                                                                                                                                                                                                            |
|-------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JProgressBar()</code>                           | Создает индикатор хода процесса с горизонтальной ориентацией. Диапазон значений от 0 до 100. Начальное значение принимается равным 0                                                                                                                                                                                                                                                                |
| <code>JProgressBar(int min, int max)</code>           | Создает индикатор хода процесса с горизонтальной ориентацией. Диапазон значений определяется посредством параметров <code>min</code> и <code>max</code> . Начальное значение принимается равным значению параметра <code>min</code>                                                                                                                                                                 |
| <code>JProgressBar(int VorH)</code>                   | Создает индикатор хода процесса, ориентация которого определяется посредством параметра <code>VorH</code> . Этот параметр может принимать значение <code>JProgressBar.VERTICAL</code> , либо <code>JProgressBar.HORIZONTAL</code> . Принимается диапазон значений от 0 до 100. Начальное значение устанавливается равным 0                                                                          |
| <code>JProgressBar(int VorH, int min, int max)</code> | Создает индикатор хода процесса, ориентация которого определяется посредством параметра <code>VorH</code> . Этот параметр может принимать значение <code>JProgressBar.VERTICAL</code> , либо <code>JProgressBar.HORIZONTAL</code> . Диапазон значений задается с помощью параметров <code>min</code> и <code>max</code> . Начальное значение принимается равным значению параметра <code>min</code> |
| <code>JProgressBar(BoundedRangeModel model)</code>    | Создает индикатор хода процесса, использующий модель, определяемую посредством параметра <code>model</code>                                                                                                                                                                                                                                                                                         |

Компоненты данного типа не генерируют событий, связанных с действиями пользователей. Они лишь отображают данные. Конечно, как и в других компонентах, при изменении состояния индикатора хода процесса возникает соответствующее событие, однако они редко обрабатываются в программах.

Поскольку индикатор хода процесса использует в качестве модели объект `BoundedRangeModel`, он поддерживает свойства, представляющие минимальное, максимальное и текущее значения. Однако в отличие от других компонентов для индикатора хода процесса понятие расширения отсутствует. Минимальное, максимальное и текущее значения, определяемые объектом `BoundedRangeModel`, доступны посредством методов, предоставляемых `JProgressBar`. Эти методы перечислены ниже.

```
int getMinimum()
void setMinimum(int val)
```

```
int getMaximum()
void setMaximum(int val)
```

```
int getValue()
void setValue(int val)
```

Как нетрудно заметить, методы доступа для расширения не предусмотрены, так как этот параметр в компонентах данного типа не применяется. По умолчанию минимальное значение принимается равным 0, максимальное — 100 и текущее — 0. Несмотря на то что эти значения можно изменить, установки по умолчанию в большинстве случаев устраивают разработчиков, так как с этим диапазоном очень удобно работать.

Свойство `value` определяет длину полосы, отображаемой данным компонентом. Таким образом, при использовании индикатора хода процесса приложение должно увеличивать значение `value` по мере выполнения текущей задачи. Желательно, чтобы это значение возрастало плавно и достигало максимума при завершении задачи. Обеспечить такое поведение в ряде случаев бывает сложно, но к этому следует стремиться, чтобы по возможности улучшить внешний вид интерфейса программы.

Объект `JProgressBar` предоставляет возможность, которую часто используют разработчики приложений: он позволяет отображать строку в составе компонента. По умолчанию в строке выводится значение в процентах, отражающее уже выполненную часть задачи. Для того чтобы текстовая строка выводилась на экране, ее отображение надо специально разрешить. Для этой цели используется метод `setStringPainted()`.

```
void setStringPainted(boolean on)
```

Если значение параметра `on` равно `true`, текстовая строка отображается на экране, в противном случае она не выводится. Выяснить, разрешено ли отображение строки, можно путем вызова метода `isStringPainted()`.

```
boolean isStringPainted()
```

Значение `true` сообщает о том, что вывод строки символов разрешен. Если метод возвращает значение `false`, это означает, что вывод строки запрещен.

## Использование индикатора хода процесса

Ниже приведен код программы, демонстрирующей использование компонента `JProgressBar`. В процессе ее выполнения создаются два индикатора: один из них ориентирован по горизонтали, другой — по вертикали. Для обоих компонентов приняты диапазоны значений по умолчанию, т.е. от 0 до 100. В программе включается отображение строки символов, отображающей выполнение задачи в процентах. Кроме того, на экран также выводится кнопка **Push Me**. По щелчку на этой кнопке текущее значение каждого индикатора хода процесса увеличивается на 10 единиц. Окно, создаваемое при работе программы, показано на рис. 3.5.

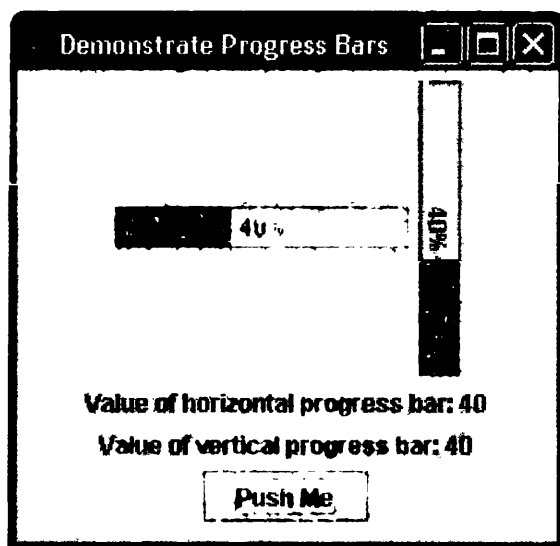


Рис. 3.5. Окно, которое отображается в процессе работы программы, демонстрирующей использование индикаторов хода процесса

```
// Демонстрация работы индикатора хода процесса
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class ProgressDemo {

    JLabel jlabVert;
    JLabel jlabHoriz;

    JProgressBar jprogHoriz;
    JProgressBar jprogVert;

    JButton jbtn;

    ProgressDemo() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Demonstrate Progress Bars");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(280, 270);

        // Завершение программы при закрытии приложения пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание индикаторов хода процесса с вертикальной
        // и горизонтальной ориентацией.
        jprogVert = new JProgressBar(JProgressBar.VERTICAL);
        jprogHoriz = new JProgressBar(); // По умолчанию
        // принимается горизонтальная ориентация

        // Отображение строки, представляющей
        // часть выполненной задачи в процентах.
        jprogVert.setStringPainted(true);
        jprogHoriz.setStringPainted(true);

        jbtn = new JButton("Push Me");

        // Метки, отображающие текущие значения
```

```

// индикаторов хода процесса.
jlabHoriz = new JLabel("Value of horizontal progress bar: " +
                        jprogHoriz.getValue());
jlabVert = new JLabel("Value of vertical progress bar: " +
                       jprogVert.getValue());

// Увеличение значения индикатора хода процесса
// по щелчку на кнопке Push Me.
// Если значение компонента достигло максимума,
// никакие действия не предпринимаются.
jbtn.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        int hVal = jprogHoriz.getValue();
        int vVal = jprogVert.getValue();

        if(hVal >= jprogHoriz.getMaximum())
            return;
        else
            jprogHoriz.setValue(hVal + 10);

        if(vVal >= jprogHoriz.getMaximum())
            return;
        else
            jprogVert.setValue(vVal + 10);

        jlabHoriz.setText("Value of horizontal progress bar: " +
                           jprogHoriz.getValue());
        jlabVert.setText("Value of vertical progress bar: " +
                           jprogVert.getValue());
    }
});

// Добавление компонентов к панели содержимого.
jfrm.getContentPane().add(jprogHoriz);
jfrm.getContentPane().add(jprogVert);
jfrm.getContentPane().add(jlabHoriz);
jfrm.getContentPane().add(jlabVert);
jfrm.getContentPane().add(jbtn);

// Отображение фрейма.
jfrm.setVisible(true);
}

```

```
public static void main(String args[]) {  
    // Фрейм создается в потоке обработки событий.  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            new ProgressDemo();  
        }  
    });  
}
```

Обратите внимание на то, как изменяется значение индикаторов хода процесса в обработчике события кнопки Push Me. Сначала программа читает текущее значение каждого компонента. Если это значение больше или равно максимальному, никакие действия не предпринимаются. В противном случае текущее значение увеличивается на 10 единиц.

**ВАЖНО!****3.10**

## **Дополнительные возможности компонента JProgressBar**

В некоторых случаях невозможно выяснить, сколько времени потребует та или иная операция, но необходимо дать пользователю знать о том, что работа программы продолжается. В такой ситуации можно использовать неопределенное состояние индикатора хода процесса. Перевести компонент в это состояние можно, вызвав метод `setIndeterminate()`.

```
void setIndeterminate(boolean on)
```

Если значение параметра `on` равно `true`, индикатор хода процесса переходит в неопределенное состояние. В данном режиме полоса небольшого размера перемещается в пределах компонента. Этим индикатор сообщает пользователю, что программа не “зависла”, а продолжает работу, но не позволяет сделать вывод о том, сколько времени осталось до завершения операции. Для того чтобы вернуть индикатор хода процесса в нормальный режим, надо вызвать тот же метод, но задать значение `false` параметра `on`. Перевести компонент в обычное состояние можно в любой момент, например, тогда, когда вы сможете получить сведения о времени завершения операции. В неопределенном состоянии в составе индикатора хода процесса не отображается строка, представляющая процент выполнения задачи. Причина очевидна: эта информация попросту недоступна. Выяснить, какой режим установлен, можно, вызвав метод `isIndeterminate()`.

```
boolean isIndeterminate()
```



Если индикатор хода процесса находится в неопределенном состоянии, метод возвращает значение `true`, в противном случае — значение `false`.

Отобразить индикатор хода процесса, находящийся в неопределенном состоянии, несложно. Надо лишь к рассмотренному выше коду программы добавить после создания индикаторов следующие строки:

```
jprogVert.setIndeterminate(true);  
jprogHoriz.setIndeterminate(true);
```

Необходимо также удалить строки, отображающие процент выполнения программ. После того как вы внесете требуемые изменения, индикаторы хода процесса будут отображать анимационную картинку, сообщающую лишь о том, что работа программы продолжается.

По умолчанию вокруг индикаторов хода процесса выводятся рамки. Такое решение обычно устраивает разработчиков. При необходимости можно запретить их отображение, вызвав метод `setBorderPainted()`.

```
void setBorderPainted(boolean on)
```

Если значение параметра `on` равно `true`, отображение рамки разрешается. Значение `false` запрещает вывод обрамления. Выяснить, разрешен ли вывод рамки, можно путем вызова метода `isBorderPainted()`.

```
boolean isBorderPainted()
```

Он возвращает значение `true`, если вывод рамки разрешен, и `false` — в противном случае.

При необходимости можно вывести в составе компонента строку, отображающую не процент выполнения задачи, а другие сведения. Для этого служит метод `setString()`.

```
void setString(String str)
```

Посредством параметра `str` передается требуемая строка. Для большинства приложений автоматически генерируемая строка вполне подходит, но следует помнить и об альтернативной возможности, предоставляемой `JProgressBar`. Получить отображаемую строку можно путем вызова метода `getString()`.

```
String getString()
```

Как и для других компонентов, вы можете задать предпочтительный размер индикатора хода процесса, вызвав метод `setPreferredSize()`, определенный в классе `JComponent`.

 Вопросы для текущего контроля

1. Если вы зададите вывод строки в составе индикатора хода процесса, какая информация будет отображаться по умолчанию посредством этой строки?
2. Как выглядит индикатор хода процесса, находящийся в неопределенном состоянии?
3. Какой диапазон значений принимается по умолчанию для индикатора хода процесса?

 Тест для самоконтроля по модулю 3

1. Полосы прокрутки, линейные регуляторы и индикаторы хода процесса используют модель \_\_\_\_\_.
2. Опишите зависимость между текущим, минимальным и максимальным значениями и расширением.
3. Когда пользователь щелкает на полосе прокрутки, ползунок изменяет свое положение в соответствии с величиной приращения блока. Да или нет?
4. Какой интерфейс следует реализовать, чтобы обеспечить обработку событий, связанных с полосой прокрутки?
5. В чём различие между основными и вспомогательными маркерами линейного регулятора?
6. События какого типа генерирует линейный регулятор?
7. Процесс установки маркеров и меток для линейного регулятора состоит из трех этапов. Опишите эти этапы.
8. Может ли индикатор хода процесса генерировать события, связанные с действиями пользователя?
9. Можно ли создать индикатор хода процесса, вокруг которого не будет отображаться рамка? Если можно, то как?

1. Часть выполненной задачи, выраженная в процентах.
2. В пределах компонента перемещается полоса небольшого размера.
3. От 0 до 100.

10. Какой интерфейс должен реализовывать обработчик событий, способный отслеживать состояние индикатора хода процесса?
11. Напишите программу, содержащую линейный регулятор и полосу прокрутки, взаимодействующие друг с другом. При каждом перемещении ползунка линейного регулятора должно соответствующим образом изменяться положение ползунка полосы прокрутки, а изменение состояния полосы прокрутки должно отражаться на линейном регуляторе.
12. В проекте 2.2 был создан простой телефонный справочник. Добавьте к нему полосу прокрутки, которая обеспечит перемещение по списку имен и номеров телефонов. Для этого добавьте метку, отображающую имя и номер телефона. При перемещении ползунка полосы прокрутки содержимое метки должно изменяться, т.е. в ней должны отображаться следующее имя и номер.

# Модуль 4

## Управление компонентами. Панели и строка подсказки

- 4.1. Использование JPanel
- 4.2. Организация компонентов с помощью JPanel
- 4.3. Использование JPanel в качестве панели содержимого
- 4.4. Использование JScrollPane
- 4.5. Добавление заголовков к панели с прокруткой
- 4.6. Дополнительные возможности JScrollPane
- 4.7. Использование JTabbedPane
- 4.8. Дополнительные возможности JTabbedPane
- 4.9. Использование JSplitPane
- 4.10. Дополнительные возможности JSplitPane
- 4.11. Установка строк подсказки

Перед тем как перейти к обсуждению остальных управляющих элементов Swing, например списков, инкрементных регуляторов, деревьев и т.д., желательно рассмотреть контейнеры, а также инструменты для организации компонентов и управления ими. Как говорилось в модуле 1, помимо контейнеров верхнего уровня, представителем которых является `JFrame`, Swing предоставляет также большое количество контейнеров других типов. Чаще остальных используются объекты `JPanel`. Кроме того, в приложениях нередко встречаются `JScrollPane`, `JTabbedPane` и `JSplitPane`. Каждый из них обладает рядом свойств, позволяющих достаточно просто решать те задачи, которые в других ситуациях были бы достаточно сложными. В данном модуле вы также узнаете, как связываются строки подсказки с компонентами пользовательского интерфейса. Вы увидите, что благодаря встроенным средствам Swing эта задача решается достаточно просто.

**ВАЖНО!****4.1. Использование объектов `JPanel`**

В коммерческих приложениях Swing редко встречаются компоненты, непосредственно включаемые в панель содержимого контейнера верхнего уровня. Вместо этого они помещаются на одну или несколько панелей, которые включаются в панель содержимого. Этот подход позволяет лучше организовать графический пользовательский интерфейс, предоставляя возможность управления группами элементов. Существуют различные типы панелей, но чаще всего используется объект `JPanel`.

`JPanel` — это контейнер общего назначения, предназначенный для размещения других компонентов. Поскольку `JPanel` является потомком класса `JComponent`, он также представляет собой компонент. Таким образом, `JPanel` — это легковесный контейнер, способный включать в себя другие компоненты, в том числе такие же объекты `JPanel`. Благодаря подобному свойству становится возможным создавать многоуровневые системы.

В классе `JPanel` определены конструкторы, описанные в табл. 4.1. Заметьте, что по умолчанию с `JPanel` связывается диспетчер компоновки `FlowLayout` и используется двойная буферизация. Двойная буферизация — это механизм, улучшающий восприятие панели пользователем при обновлении экрана. Дело в том, что непосредственное прорисовывание каждого компонента на экране вызывает эффект мерцания. Чтобы избежать этого, воспроизведение компонентов осуществляется в отдельном буфере; после окончания прорисовки содержимое буфера копируется на экран посредством одной операции, выполнение которой не прерывается. Таким образом, содержимое панели заменяется практически мгновенно. При желании вы можете отключить двойную буфе-

ризацию, но трудно представить себе ситуацию, в которой возникла бы подобная необходимость. Поддержка двойной буферизация унаследована от класса `JComponent`.

Таблица 4.1. Конструкторы класса `JPanel`

| Конструктор                                              | Описание                                                                                                                                                                                                                                                                                                |
|----------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JPanel()</code>                                    | Создает панель по умолчанию, с которой связывается диспетчер компоновки <code>FlowLayout</code> и используется двойная буферизация                                                                                                                                                                      |
| <code>JPanel(LayoutManager lm)</code>                    | Создает панель и связывает с ней диспетчер компоновки, заданный посредством параметра <code>lm</code> . Панель поддерживает двойную буферизацию                                                                                                                                                         |
| <code>JPanel(boolean doubleBuf)</code>                   | Создает панель и связывает с ней диспетчер компоновки <code>FlowLayout</code> . Если значение <code>doubleBuf</code> равно <code>true</code> , панель поддерживает двойную буферизацию. Значение <code>false</code> данного параметра приводит к отключению двойной буферизации                         |
| <code>JPanel(LayoutManager lm, boolean doubleBuf)</code> | Создает панель и связывает с ней диспетчер компоновки, заданный посредством параметра <code>lm</code> . Если значение <code>doubleBuf</code> равно <code>true</code> , панель поддерживает двойную буферизацию. Значение <code>false</code> данного параметра приводит к отключению двойной буферизации |

Заметьте, что контейнер `JPanel` позволяет задавать для него требуемый диспетчер компоновки. Это удобно при создании пользовательских интерфейсов, так как появляется возможность размещать компоненты на разных панелях и использовать для каждой из них наиболее подходящий диспетчер компоновки.

Для компонентов `JPanel` можно задавать обрамление. Оно устанавливается так же, как и рамка для кнопки (см. модуль 2). При наличии обрамления становятся видимыми границы панели. Очевидно, что рамка не является обязательным элементом каждой панели в составе пользовательского интерфейса.

В большинстве случаев для пользовательских интерфейсов требуются непрозрачные панели. В Swing определены два типа компонентов: *прозрачные* и *непрозрачные*. Непрозрачные компоненты заслоняют все элементы, находящиеся под ними. В прозрачных компонентах цвет фона не отображается, поэтому элементы, находящиеся под таким компонентом, видны на экране. В большинстве стилей компоненты `JPanel` непрозрачны по умолчанию. Исключением является `GTK+`.

## Спросим у опытного программиста

**Вопрос.** Вы сказали, что двойная буферизация унаследована `JPanel` от `JComponent`. Как определить, используется ли этот механизм в компоненте? Как включить или отключить его?

**Ответ.** Определить, поддерживает ли компонент двойную буферизацию, можно путем вызова метода `isDoubleBuffered()`.

```
boolean isDoubleBuffered()
```

Он возвращает значение `true`, если двойная буферизация включена, и `false` — в противном случае. Для включения или отключения двойной буферизации используется метод `setDoubleBuffered()`, который определяется следующим образом:

```
void setDoubleBuffered(boolean DBon)
```

Если значение параметра `DBon` равно `true`, включается двойная буферизация. Значение `false` этого параметра отключает двойную буферизацию.

Таким образом, если вам нужна непрозрачная панель (а это требуется в большинстве случаев) и если вы хотите, чтобы это свойство сохранялось во всех стилях, вам следует явно установить его, вызвав метод `setOpaque()`, который определен для всех компонентов. Его заголовок выглядит следующим образом:

```
void setOpaque(boolean opaqueOn )
```

Если значение параметра `opaqueOn` равно `true`, панель будет непрозрачной. Значение `false` делает панель прозрачной. Выяснить, прозрачна ли панель (или любой другой компонент), можно, вызвав метод `isOpaque()`.

```
boolean isOpaque()
```

Этот метод возвращает значение `true`, если компонент непрозрачен, и `false` — в противном случае.

Ниже в данном модуле будут рассмотрены два варианта применения `JPanel`. Первый — организация компонентов в группы для упрощения их управления. Второй вариант — применение `JPanel` в качестве панели содержимого.

ВАЖНО!

4.2.

## Использование панелей для организации компонентов

Как уже было сказано ранее, `JPanel` является потомком класса `JComponent`. Это означает, что `JPanel` — не только контейнер, но и компонент и может быть включен в другой контейнер. Более того, в роли включающего контейнера также может выступать объект `JPanel`; другими словами, Swing допускает вложенность панелей. На практике экземпляр `JPanel` чаще всего помещается на панель содержимого `JFrame`. Как вы помните из модуля 1, панель содержимого `JFrame` — это экземпляр `JPanel`, следовательно, один или несколько экземпляров `JPanel` можно включить в панель содержимого `JFrame`. Это обеспечивает ряд преимуществ.

Во-первых, вы можете использовать отдельный контейнер `JPanel` для размещения группы взаимосвязанных компонентов, при этом диспетчер компоновки, связанный с панелью содержимого, будет обрабатывать эти компоненты как один элемент, и у вас не возникнет необходимости принимать специальные меры для размещения каждого компонента по отдельности. При изменении размеров фрейма взаимное расположение компонентов из группы не будет нарушено. (В особенности это важно тогда, когда с панелью содержимого связан диспетчер компоновки `FlowLayout`.) Во-вторых, такой подход позволяет указать отдельный диспетчер компоновки для каждой панели. И в-третьих, вы можете добавлять или удалять группы компонентов с помощью одной операции.

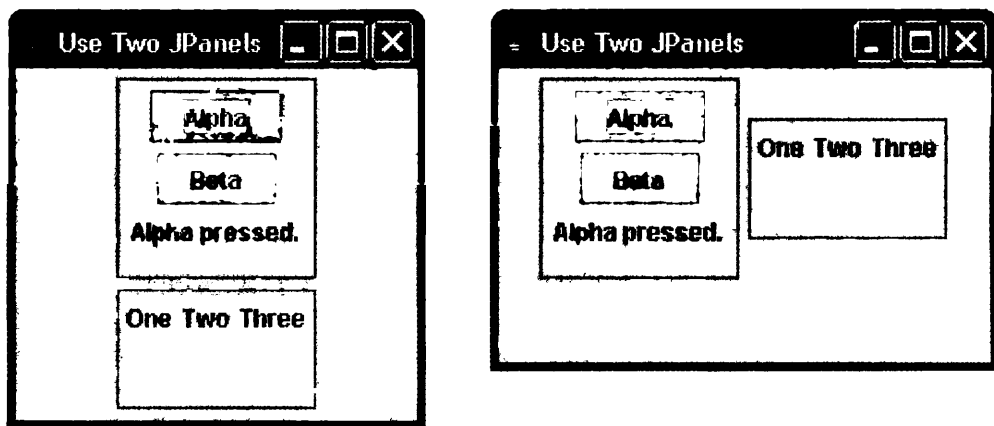


Рис. 4.1. Выходные данные программы `PanelDemo`. Заметьте, что даже при изменении размеров фрейма взаимное расположение элементов в каждой из двух панелей остается постоянным



Программа, код которой приведен ниже, демонстрирует организацию компонентов посредством панелей. В ней создаются два экземпляра `JPanel`, затем на первой панели располагаются две кнопки и метка, а на второй — три метки. Для обеих панелей устанавливаются рамки, поэтому их границы видны на экране. Панели включаются в состав панели содержимого. Данные, которые выводятся при работе программы, показаны на рис. 4.1.

```
// Создание двух объектов JPanel и включение их
// в панель содержимого.

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

class PanelDemo {

    JLabel jlab;
    JButton jbtnAlpha;
    JButton jbtnBeta;

    PanelDemo() {

        // Создание нового контейнера JFrame.
        // По умолчанию с панелью содержимого
        // связан диспетчер компоновки BorderLayout.
        JFrame jfrm = new JFrame("Use Two JPanels");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(210, 210);

        // Завершение программы при закрытии приложения пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание и инициализация первого объекта JPanel.
        JPanel jpnl = new JPanel();

        // Установка предпочтительных размеров первой панели.
        jpnl.setPreferredSize(new Dimension(100, 100));

        // Панель должна быть непрозрачной.
```

```
jpn1.setOpaque(true);

// Отображение рамки синего цвета вокруг панели.
jpn1.setBorder(
    BorderFactory.createLineBorder(Color.BLUE));

// Создание и инициализация второго объекта JPanel.
JPanel jpn12 = new JPanel();

// Установка предпочтительных размеров второй панели.
jpn12.setPreferredSize(new Dimension(100, 60));

// Панель должна быть непрозрачной.
jpn12.setOpaque(true);

// Отображение рамки красного цвета вокруг панели.
jpn12.setBorder(
    BorderFactory.createLineBorder(Color.RED));

// Создание метки.
jlab = new JLabel("Press a button.");

// Создание двух кнопок.
jbtnAlpha = new JButton("Alpha");
jbtnBeta = new JButton("Beta");

// Связывание обработчиков событий с кнопками.
jbtnAlpha.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("Alpha pressed.");
    }
});

jbtnBeta.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("Beta pressed.");
    }
});

// Включение кнопок и метки в состав первого
// объекта JPanel.
jpn1.add(jbtnAlpha);
jpn1.add(jbtnBeta);
```

```

jpn1.add(jlab);

// Включение меток в состав второго объекта JPanel1.
jpn12.add(new JLabel("One"));
jpn12.add(new JLabel("Two"));
jpn12.add(new JLabel("Three"));

// Добавление панелей к панели содержимого фрейма.
jfrm.getContentPane().add(jpn1);
jfrm.getContentPane().add(jpn12);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new PanelDemo();
        }
    });
}
}

```

Рассмотрим код программы более подробно. Прежде всего, заметьте, что в качестве контейнера верхнего уровня применяется объект `JFrame`; на него ссылается переменная `jfrm`. Для панели содержимого фрейма задается диспетчер компоновки `FlowLayout`. Как вы знаете, если используется данный диспетчер, то при изменении размеров окна взаимное расположение содержащихся в нем компонентов может измениться.

После этого создаются два экземпляра `Jpanel`, и ссылки на них присваиваются переменным `jpn1` и `jpn12`. Для каждой из панелей задается предпочтительный размер и явным образом устанавливается свойство непрозрачности. Кроме того, для панелей задается обрамление. Рамка панели, на которую ссылается переменная `jpn1`, имеет синий цвет, а рамка панели `jpn2` — красный. В результате пользователь видит область, занимаемую каждой панелью. Как вы уже знаете, в реальных приложениях рамки устанавливать не обязательно. С каждой из панелей по умолчанию связан диспетчер компоновки `FlowLayout`.

В состав каждой панели включаются компоненты. Для `jpn1` это две кнопки и метка, для `jpn12` — три метки. И наконец, обе панели включаются в состав панели содержимого фрейма и фрейм отображается на экране.

Запустив программу, вы увидите, что взаимное расположение компонентов в составе каждой панели остается постоянным, несмотря на то, что размеры фрейма изменяются. Например, если вы увеличите ширину фрейма, изменится позиция одной панели относительно другой, но не расположение компонентов в каждой панели. Этот простой пример демонстрирует некоторые принципы организации компонентов в графическом пользовательском интерфейсе и управления ими.

**ВАЖНО!**

### **4.3. Использование компонента JPanel в качестве панели содержимого**

Во всех предыдущих примерах компоненты добавлялись в существующую панель содержимого, находящуюся в составе контейнера верхнего уровня JFrame. Такой подход вполне приемлем, но существуют и другие решения, которые в некоторых случаях могут быть очень эффективными. Вместо использования панели содержимого, предоставляемой JFrame, можно создать собственную панель и связать ее с JFrame. Как вы помните, панель содержимого JFrame — это экземпляр JPanel. Таким образом, вы можете использовать в качестве панели содержимого любой объект JPanel и не ограничены тем, который автоматически формируется при создании экземпляра JFrame. Преимущество такого подхода — возможность использования произвольного объекта JPanel с требуемым содержимым.

Для того чтобы установить панель содержимого в контейнере JFrame, надо использовать метод `setContentPane()`, показанный ниже.

```
void setContentPane(Container panel)
```

Параметр `panel` здесь должен указывать на непрозрачный контейнер, являющийся подклассом `JComponent`. (Заметьте, что класс `JComponent` является подклассом класса `Container`.)

Чаще всего для формирования панели содержимого, определяемой разработчиком, создают класс, расширяющий класс `JPanel`. Экземпляр этого класса можно так заполнить компонентами, чтобы перед связыванием с контейнером верхнего уровня он был полностью инициализирован. Ниже приведен код программы, демонстрирующий данный подход. В ней создается пользовательская панель содержимого, в составе которой находятся две кнопки, `Red` и `Blue`, и метка. При активизации кнопки рамка вокруг панели содержимого окрашивается в соответствующий цвет. Окно, создаваемое при работе программы, показано на рис. 4.2.

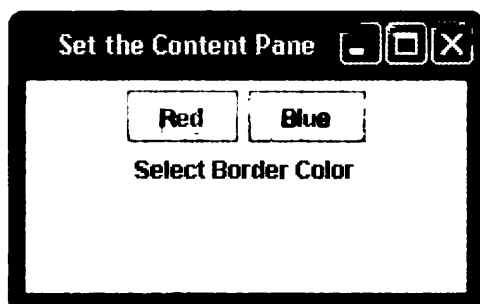


*Рис. 4.2. Данные, отображаемые программой, демонстрирующей работу компонента CustomCPDemo*

```
// Создание объекта JPanel и использование его
// в качестве панели содержимого.

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

// Данный класс создает панель, расширяющую JPanel.
// Она используется в качестве панели содержимого.
// В этом классе не предусмотрены новые функциональные
// возможности, но его экземпляр может применяться
// во всех случаях, в которых допустимо использование JPanel.
class MyContentPanel extends JPanel {
    JLabel jlab;
    JButton jbtnRed;
    JButton jbtnBlue;

    MyContentPanel() {

        // Панель должна быть непрозрачной.
        setOpaque(true);

        // Установка рамки зеленого цвета толщиной в 5 пикселей.
        setBorder(
            BorderFactory.createLineBorder(Color.GREEN, 5));

        // Создание метки.
        jlab = new JLabel("Select Border Color");
```



```
// Создание двух кнопок.
jbtnRed = new JButton("Red");
jbtnBlue = new JButton("Blue");

// Связывание обработчиков событий с кнопками.
jbtnRed.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        setBorder(
            BorderFactory.createLineBorder(Color.RED, 5));
    }
});

jbtnBlue.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        setBorder(
            BorderFactory.createLineBorder(Color.BLUE, 5));
    }
});

// Включение кнопок и метки в состав панели.
add(jbtnRed);
add(jbtnBlue);
add(jlab);

}

// Создание контейнера верхнего уровня и использование
// панели, созданной с помощью MyContentPanel,
// в качестве панели содержимого
class CustomCPDemo {
    CustomCPDemo() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Set the Content Pane");

        // Установка начальных размеров фрейма.
        jfrm.setSize(240, 150);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание экземпляра пользовательской панели содержимого.
        MyContentPanel mcp = new MyContentPanel();
```

```
// Установка mcp в качестве панели содержимого.
jfrm.setContentPane(mcp);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new CustomCPDemo();
        }
    });
}
```

В программе создаются два класса. Первый, `MyContentPanel`, является подклассом `JPanel`. Он не реализует никаких дополнительных функций по сравнению с `JPanel`. В нем объект `JPanel` лишь заполняется компонентами. В данном случае это две кнопки и метка. Заметьте, что панель явным образом определяется как непрозрачная. Это необходимо, поскольку такое требование предъявляется ко всем панелям содержимого. Также обратите внимание на то, как вызываются методы `setOpaque()` и `setBorder()`. Поскольку класс `MyContentPanel` расширяет `JPanel`, при вызове этих методов не указывается ссылка на объект `JPanel`.



### Вопросы для текущего контроля

1. В чем преимущество двойной буферизации?
  2. Используется ли по умолчанию двойная буферизация в объектах `JPanel`?
  3. Если панель прозрачна, расположенные под ней компоненты видны на экране. Да или нет?
- 
1. Двойная буферизация улучшает внешний вид пользовательского интерфейса, устраняя мерцание. Она также повышает производительность за счет вывода всего изображения в рамках одной непрерываемой операции.
  2. Да.
  3. Да.

Теперь рассмотрим второй класс — `CustomCPDemo`. В его конструкторе создается объект `JFrame`, на который указывает переменная `jfrm`. Затем создается экземпляр `MyContentPanel` (переменная `mcp`). Этот объект используется как панель содержимого контейнера `jfrm`. После вызова метода `setVisible()` отображается содержимое `mcp`.

## Спросим у опытного программиста

**Вопрос.** Если при компиляции программы, демонстрирующей применение пользовательской панели содержимого, задать опцию `-Xlint` (она разрешает отображение предупреждающих сообщений, носящих характер рекомендаций), то на экране отображается следующее сообщение:

```
warning: [serial] serializable class
MyContentPanel
has no definition of serialVersionUID
```

Что оно означает, почему выводится и как от него избавиться?

**Ответ.** Это сообщение появляется потому, что класс `MyContentPanel` расширяет класс `JPanel`, а `JPanel` реализует интерфейс `Serializable`. Интерфейс `Serializable` лишь указывает на то, что класс может быть сериализован. С каждым сериализуемым классом связывается идентификатор версии, который хранится в переменной `static final` с именем `serialVersionUID`. Если этот идентификатор явно не указан в классе, то он генерируется автоматически. Однако в документации по Java API рекомендуется задавать это значение явно. По этой причине появляется предупреждающее сообщение. Если вопросы сериализации не важны, как, например, в программах, рассматриваемых в данной книге, это сообщение можно игнорировать. Однако в реальных приложениях желательно явно указывать идентификатор. Этим вы предотвратите появление предупреждающих сообщений.

Для того чтобы объявить переменную `serialVersionUID`, можно использовать выражение наподобие следующего:

```
static final long serialVersionUID = 10101;
```

Вместо указанного здесь номера следует подставить нужное вам значение.



**ВАЖНО!**

#### 4.4. Использование объекта JScrollPane

В наборе Swing имеется несколько специальных контейнеров, помогающих в организации графического пользовательского интерфейса. К их числу относятся панель с прокруткой, панель с вкладками и разделяемая панель. Обсуждению этих контейнеров будут посвящены данный и следующие разделы. В этом разделе мы рассмотрим панель с прокруткой, реализуемую посредством класса `JScrollPane`. Такая панель автоматически осуществляет прокрутку для отображения содержащегося в ней компонента. Поскольку легковесные контейнеры также являются компонентами, `JScrollPane` может прокручивать, например, содержимое `JPanel`. Как было сказано в модуле 3, поскольку `JScrollPane` автоматизирует прокрутку, необходимость специально организовывать действия с полосой прокрутки отпадает. Размещением компонентов в составе `JScrollPane` управляет диспетчер компоновки `ScrollPaneLayout`, который разделяет панель на девять частей (рис. 4.3).

Обычно наибольшие размеры имеет часть `JScrollPane`, называемая *областью просмотра* (viewport). Это окно, в котором выводится компонент, предна-

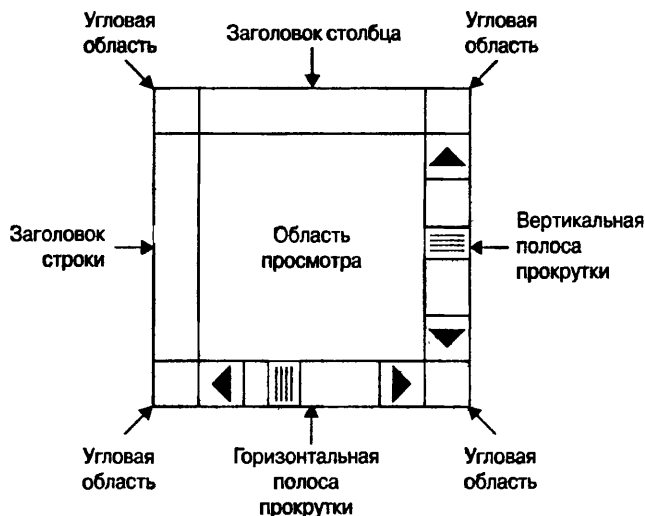


Рис. 4.3. Структура контейнера `JScrollPane`

значенный для прокрутки, вернее, его видимые части. Прокрутка компонента в области просмотра осуществляется посредством полосы прокрутки. По умолчанию `JScrollPane` по мере необходимости динамически отображает или удаляет полосу прокрутки. Например, если высота компонента больше, чем высота области просмотра, отображается вертикальная полоса прокрутки. Если компонент полностью помещается в области просмотра, то полосы прокрутки удаляются. При желании вы можете добавить заголовок строки или столбца, описывающий отображаемые данные или обеспечивающий дополнительные функциональные возможности. И наконец, в четырех углах можно разместить компоненты, которые также будут реализовывать дополнительные функции.

На первый взгляд может показаться, что, поскольку компонент `JScrollPane` включает такое большое число элементов, пользоваться им трудно. К счастью, это не так. Несмотря на то что `JScrollPane` делится на девять частей, использовать их все не обязательно. Конфигурация `JScrollPane` по умолчанию предполагает наличие лишь области просмотра и динамических полос прокрутки. Для многих приложений эта конфигурация вполне подходит. Она предельно проста в использовании: достаточно передать `JScrollPane` компонент, а остальное произойдет автоматически. Таким образом, несмотря на то, что `JScrollPane` предоставляет богатые возможности для настройки, использовать их не обязательно.

В классе `JScrollPane` определены конструкторы, приведенные в табл. 4.2. Как видно из таблицы, вы можете создать объект `JScrollPane`, не содержащий компонента для прокрутки. В этом случае вам придется указать компонент позже, уже после того, как будет создана панель с прокруткой. Однако чаще всего компонент задается в процессе создания панели. Заметьте также, что вы можете установить политику использования полос прокрутки. Эта политика определяет, когда должны отображаться полосы прокрутки. Этот вопрос будет подробно обсуждаться далее в данном разделе. Если вы не укажете политику, то полосы прокрутки будут выводиться по мере необходимости. Обычно такое поведение устраивает пользователей.

Область прокрутки в составе `JScrollPane` управляется объектом типа `JViewport`. Когда вы передадите компонент конструктору `JScrollPane`, для этого компонента будет автоматически сформирован объект `JViewport`. В большинстве случаев при работе с `JScrollPane` не возникает необходимости непосредственного взаимодействия с данным объектом.

Таблица 4.2. Конструкторы класса JScrollPane

| Конструктор                                                         | Описание                                                                                                                                                                                                                                                     |
|---------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JScrollPane()</code>                                          | Создает объект <code>JScrollPane</code> , для которого область просмотра не определяется                                                                                                                                                                     |
| <code>JScrollPane(Component comp)</code>                            | Создает объект <code>JScrollPane</code> , который автоматически прокручивает компонент, заданный посредством параметра <code>comp</code>                                                                                                                     |
| <code>JScrollPane(int vertSBP, int horizSBP)</code>                 | Создает объект <code>JScrollPane</code> , для которого область просмотра не определяется. Политика использования полос прокрутки задается с помощью параметров <code>vertSBP</code> и <code>horizSBP</code>                                                  |
| <code>JScrollPane(Component comp, int vertSBP, int horizSBP)</code> | Создает объект <code>JScrollPane</code> , который автоматически прокручивает компонент, заданный посредством параметра <code>comp</code> . Политика использования полос прокрутки задается с помощью параметров <code>vertSBP</code> и <code>horizSBP</code> |

## Простой пример использования класса JScrollPane

Несмотря на обилие предоставляемых возможностей, класс `JScrollPane` очень прост в использовании, особенно если разработчик не изменяет конфигурацию, принятую по умолчанию. Класс `JScrollPane` можно использовать для прокрутки любого компонента; для этого достаточно передать ссылку на компонент конструктору. Остальное `JScrollPane` делает автоматически. Ниже приведен простой пример, иллюстрирующий работу с данным контейнером. В этом примере осуществляется прокрутка содержимого `JLabel`, представляющего собой несколько строк HTML-кода. При работе программы выводится окно, показанное на рис. 4.4.

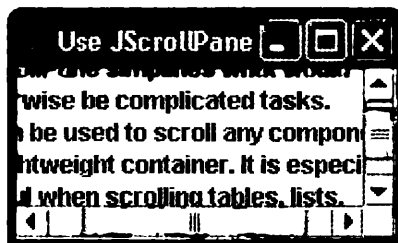


Рис. 4.4. Данные, отображаемые программой, демонстрирующей работу компонента `ScrollPaneDemo`

```
// Простой пример использования JScrollPane

import javax.swing.*;

class ScrollPaneDemo {

    ScrollPaneDemo() {

        // Создание нового контейнера JFrame.
        // Для него принимается диспетчер компоновки по умолчанию.
        JFrame jfrm = new JFrame("Use JScrollPane");

        // Установка начальных размеров фрейма.
        jfrm.setSize(200, 120);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание метки с HTML-содержимым большого объема.
        JLabel jlab =
            new JLabel("<html>JScrollPane simplifies what would<br>" +
                "otherwise be complicated tasks.<br>" +
                "It can be used to scroll any component<br>" +
                "or lightweight container. It is especially<br>" +
                "useful when scrolling tables, lists,<br>" +
                "or images.");

        // Создание панели с прокруткой и передача ей метки.
        JScrollPane jscrlp = new JScrollPane(jlab);

        // Включение панели с прокруткой в состав
        // панели содержимого фрейма.
        jfrm.getContentPane().add(jscrlp);

        // Отображение фрейма.
        jfrm.setVisible(true);
    }

    public static void main(String args[]) {
        // Фрейм создается в потоке обработки событий.
        SwingUtilities.invokeLater(new Runnable() {
```

```

    public void run() {
        new ScrollPaneDemo();
    }
});
}
}

```

Обратите особое внимание на следующую строку кода:

```
JScrollPane jscrlp = new JScrollPane(jlab);
```

Здесь конструктору JScrollPane передается ссылка jlab, указывающая на метку. После выполнения этой строки jscrlp будет отображать в области просмотра содержимое jlab, при необходимости предоставляя пользователю полосы прокрутки.

**ВАЖНО!**

**4.5.**

## **Добавление заголовков**

Как было сказано ранее, для многих приложений функциональных возможностей, предоставляемых JScrollPane по умолчанию, вполне достаточно. Однако данный компонент очень легко настроить для выполнения специальных задач. Одна из дополнительных возможностей — поддержка заголовков. JScrollPane позволяет задавать заголовок строки и заголовок столбца. Вы можете использовать любой из них или оба одновременно. Заголовок может содержать компонент любого типа. Другими словами, вы не ограничены метками и можете использовать в качестве заголовка и активные управляющие элементы, например кнопки.

Если вам надо установить заголовок строки, то проще всего сделать это путем вызова метода `setRowHeaderView()`.

```
void setRowHeaderView(Component comp)
```

Посредством параметра `comp` методу передается компонент, используемый в качестве заголовка.

Установить заголовок столбца можно, вызвав метод `setColumnHeaderView()`.

```
void setColumnHeaderView(Component comp)
```

Здесь с помощью параметра `comp` также передается компонент, который должен выполнять роль заголовка.

Установив заголовки, вы, возможно, захотите отобразить рамку вокруг области просмотра. Сделать это несложно, достаточно вызвать метод

`setViewportBorder()`, определенный в классе `JScrollPane`. Его заголовок выглядит следующим образом:

```
void setViewportBorder(Border border)
```

Как вы помните, для формирования стандартных обрамлений предусмотрен класс `BorderFactory` (см. модуль 2).

Ниже приведен пример, показывающий, как включить в панель прокрутки заголовки строки и столбца. Кроме того, в нем формируется рамка вокруг области просмотра. Обратите внимание на то, что добавление заголовков происходит очень просто. (При желании вы можете использовать гораздо более сложные заголовки, чем те, которые показаны в данном примере, но процедура их включения остается неизменной.) Окно, создаваемое при работе программы, показано на рис. 4.5.

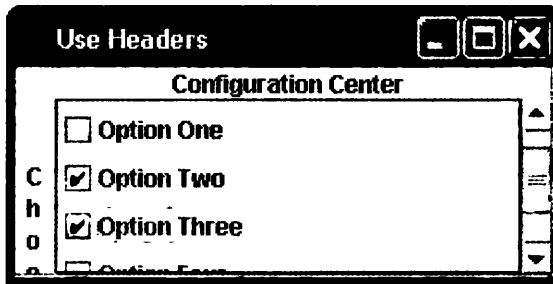


Рис. 4.5. Окно, отображаемое программой `AddHeadersDemo`

// Добавление к панели прокрутки заголовков и рамки.

```
import java.awt.*;
```

```
import javax.swing.*;
```

```
class AddHeadersDemo {
```

```
    JCheckBox jcbOpt1;
```

```
    JCheckBox jcbOpt2;
```

```
    JCheckBox jcbOpt3;
```

```
    JCheckBox jcbOpt4;
```

```
    JCheckBox jcbOpt5;
```

```
    AddHeadersDemo() {
```

```
        // Создание нового контейнера JFrame.
```

```
        // Для него принимается диспетчер компоновки по умолчанию.
```

```
        JFrame jfrm = new JFrame("Use Headers");
```

```
// Установка начальных размеров фрейма.
jfrm.setSize(280, 140);

// Завершение программы при закрытии окна пользователем.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Создание метки.
JLabel jlabOptions = new
    JLabel("Select one or more options: ");

// Создание флажков опций.
jcbOpt1 = new JCheckBox("Option One");
jcbOpt2 = new JCheckBox("Option Two");
jcbOpt3 = new JCheckBox("Option Three");
jcbOpt4 = new JCheckBox("Option Four");
jcbOpt5 = new JCheckBox("Option Five");

// В данном примере обработчики событий не нужны.

// Создание контейнера JPanel для размещения флажков опций.
JPanel jpnl = new JPanel();
jpnl.setLayout(new GridLayout(6, 1));
jpnl.setOpaque(true);

// Включение флажков опций и метки в состав JPanel.
jpnl.add(jlabOptions);
jpnl.add(jcbOpt1);
jpnl.add(jcbOpt2);
jpnl.add(jcbOpt3);
jpnl.add(jcbOpt4);
jpnl.add(jcbOpt5);

// Создание панели с прокруткой.
JScrollPane jscrlp = new JScrollPane(jpnl);

// Создание рамки вокруг области просмотра.
jscrlp.setViewportBorder(
    BorderFactory.createLineBorder(Color.BLACK));

// Создание меток, используемых для заголовков.
JLabel jlabCH = new JLabel("Configuration Center",
    SwingConstants.CENTER);
```

```
JLabel jlabRH = new JLabel("<html>C<br>h<br>o<br>s<br>e"
,
                                SwingConstants.CENTER);
jlabRH.setPreferredSize(new Dimension(20, 200));

// Формирование заголовков строки и столбца.
jscrollp.setColumnHeaderView(jlabCH);
jscrollp.setRowHeaderView(jlabRH);

// Добавление панели с прокруткой к фрейму.
jfrm.getContentPane().add(jscrollp);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new AddHeadersDemo();
        }
    });
}
```

**ВАЖНО!**

## **4.6. Дополнительные возможности панели прокрутки**

Помимо поддержки заголовков и рамки вокруг области просмотра, `JScrollPane` предоставляет ряд дополнительных возможностей. Среди них наиболее часто используются политики полос прокрутки. Эти политики определяют, когда должны отображаться полосы прокрутки. По умолчанию `JScrollPane` выводит их лишь по мере необходимости. Например, если объем информации, предназначенной для прокрутки, велик и она не помещается в области просмотра, в контейнер автоматически включается вертикальная полоса прокрутки, позволяющая перемещать информацию в области просмотра вверх и вниз. Если все данные помещаются в области просмотра, то полосы прокрутки не отображаются. Политики полос прокрутки, принятые по умолчанию, подходят в большинстве случаев, однако, если используются угловые компоненты (этот вопрос будет рассмотрен несколько позже), они могут приводить к возникнове-



нию проблем. Заметьте, что для каждой полосы прокрутки определена отдельная политика. Поэтому при желании вы можете добиться того, что вертикальная и горизонтальная полосы прокрутки будут вести себя по-разному.

Политики полос прокрутки можно задать при создании объекта `JScrollPane`. Для этого надо использовать один из следующих конструкторов:

```
JScrollPane(int vertSBP, int horizSBP)
```

```
JScrollPane(Component comp, int vertSBP, int horizSBP)
```

где параметр `vertSBP` задает политику для вертикальной, а параметр `horizSBP` — для горизонтальной полосы прокрутки. Политики определяются с помощью констант, объявленных в интерфейсе `ScrollPaneConstants`. Они описаны ниже.

| Константа                                   | Политика                                                                                                        |
|---------------------------------------------|-----------------------------------------------------------------------------------------------------------------|
| <code>HORIZONTAL_SCROLLBAR_AS_NEEDED</code> | Горизонтальная полоса прокрутки отображается только при необходимости. Данная политика принимается по умолчанию |
| <code>HORIZONTAL_SCROLLBAR_NEVER</code>     | Горизонтальная полоса прокрутки никогда не отображается                                                         |
| <code>HORIZONTAL_SCROLLBAR_ALWAYS</code>    | Горизонтальная полоса прокрутки отображается всегда                                                             |
| <code>VERTICAL_SCROLLBAR_AS_NEEDED</code>   | Вертикальная полоса прокрутки отображается только при необходимости. Данная политика принимается по умолчанию   |
| <code>VERTICAL_SCROLLBAR_NEVER</code>       | Вертикальная полоса прокрутки никогда не отображается                                                           |
| <code>VERTICAL_SCROLLBAR_ALWAYS</code>      | Вертикальная полоса прокрутки отображается всегда                                                               |

Установить политики полос прокрутки можно и для готовой панели. Для этой цели предусмотрены такие методы:

```
void setVerticalScrollBarPolicy(int vertSBP)
```

```
void setHorizontalScrollBarPolicy(int horizSBP)
```

Здесь, как и ранее, параметры `vertSBP` и `horizSBP` задают политики для вертикальной и горизонтальной полосы прокрутки.

Доступ к полосам прокрутки осуществляется посредством приведенных ниже методов.

```
JScrollBar getHorizontalScrollBar()
```

```
JScrollBar getVerticalScrollBar()
```

Каждый из этих методов возвращает ссылку на соответствующую полосу прокрутки. Используя ссылки, вы можете устанавливать атрибуты полос прокрутки, например, задавать приращение элемента или блока. Об особенностях работы с полосами прокрутки см. в модуле 3.

Работая с `JScrollPane`, можно также использовать угловые области. Они образуются на стыке полос прокрутки, заголовков строки и столбца. В этих областях можно разместить любые компоненты, например кнопки. Однако надо принимать во внимание следующие особенности. Во-первых, размеры угловых областей зависят от ширины полос прокрутки и заголовков, поэтому в большинстве случаев они очень малы. Во-вторых, угловые области видимы только тогда, когда полосы прокрутки и заголовки отображаются на экране. Например, правая нижняя угловая область видна только в том случае, если на экране присутствует и вертикальная, и горизонтальная полоса прокрутки. Если вы хотите поместить в эту область компонент, надо позаботиться о том, чтобы полосы прокрутки не исчезали с экрана. (Это один из случаев, когда приходится явным образом устанавливать политики полос прокрутки.)

Для того чтобы поместить компонент в угловую область, надо вызвать метод `setCorner()`.

```
void setCorner(String which, Component comp)
```

Здесь угол определяется значением параметра `which`. Оно должно быть равно одной из перечисленных ниже констант, которые определены в интерфейсе `ScrollPaneConstants`. (Интерфейс `ScrollPaneConstants` реализуется компонентом `JScrollPane`.)

```
LOWER_LEADING_CORNER  
LOWER_LEFT_CORNER  
LOWER_RIGHT_CORNER  
LOWER_TRAILING_CORNER  
UPPER_LEADING_CORNER  
UPPER_LEFT_CORNER  
UPPER_RIGHT_CORNER  
UPPER_TRAILING_CORNER
```

Для того чтобы эффективно использовать угловые области, надо приложить определенные усилия, однако потребность в этом возникает сравнительно редко. Однако, если вы хотите, чтобы панель прокрутки выглядела профессионально, вам придется позаботиться и об угловых областях.

**Спросим у опытного программиста**

**Вопрос.** Что такое интерфейс `Scrollable`? Насколько он важен для `JScrollPane`?

**Ответ.** Интерфейс `Scrollable` определяет атрибуты прокрутки, используемые `JScrollPane`, например предпочтительный размер области просмотра, приращение блока и элемента. Если этот интерфейс не реализуется компонентом `JScrollPane`, в нем будут использованы установки по умолчанию. Таким образом, если вы хотите реализовать достаточно сложное поведение панели с прокруткой, вам придется реализовать интерфейс `Scrollable`. Для простых приложений в этом нет необходимости.

**Вопросы для текущего контроля**

1. Контейнер `JScrollPane` обеспечивает прокрутку любого легковесного компонента `Swing`. Да или нет?
2. Какая политика задает постоянное отображение на экране вертикальной полосы прокрутки?
3. Может ли в заголовке строки или столбца содержаться активный компонент, например кнопка?

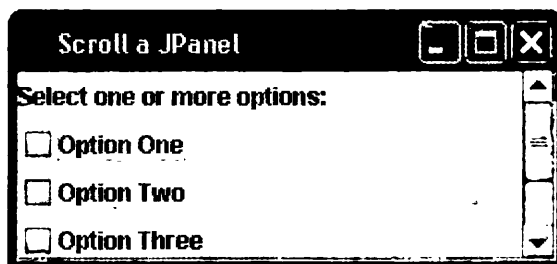
**Проект 4.1. Прокрутка содержимого `JPanel`**

`ScrollJPanelDemo.java`

Поскольку `JPanel` представляет собой легковесный контейнер, его содержимое можно прокручивать с помощью `JScrollPane`. Для того чтобы реализовать эту возможность, разработчику практически не приходится прилагать усилий. Таким образом, когда на экране не хватает места для представления нужной информации, решением проблемы может быть использование панели с прокруткой.

1. Да.
2. `VERTICAL_SCROLLBAR_ALWAYS`
3. Да.

Данный проект демонстрирует прокрутку содержимого JPanel. На панели находится метка и набор флажков опций. Панель передается конструктору JScrollPane. В результате выполнения конструктора создается объект JScrollPane, обеспечивающий автоматическую прокрутку содержимого панели. Окно, отображаемое в процессе работы программы, показано ниже.



## Последовательность действий

1. Создайте файл ScrollJPanelDemo.java и включите в него следующие комментарии и выражения import:

```
// Проект 4.1. Использование компонента JScrollPane
// для прокрутки содержимого JPanel.
```

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

2. Начните код класса ScrollJPanelDemo следующим объявлением:

```
class ScrollJPanelDemo {
```

3. Создайте конструктор ScrollJPanelDemo(), начав его строками кода, приведенными ниже.

```
ScrollJPanelDemo() {
    // Создание нового контейнера JFrame.
    // Для него принимается диспетчер компоновки по умолчанию.
    JFrame jfrm = new JFrame("Scroll a JPanel");

    // Установка начальных размеров фрейма.
    jfrm.setSize(280, 130);

    // Завершение программы при закрытии окна
    // пользователем.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

4. Добавьте фрагмент кода, предназначенный для создания метки и флажков опций.

```
// Создание метки.  
JLabel jlabOptions = new  
    JLabel("Select one or more options: ");  
  
// Создание флажков опций.  
JCheckBox jcbOpt1 = new JCheckBox("Option One");  
JCheckBox jcbOpt2 = new JCheckBox("Option Two");  
JCheckBox jcbOpt3 = new JCheckBox("Option Three");  
JCheckBox jcbOpt4 = new JCheckBox("Option Four");  
JCheckBox jcbOpt5 = new JCheckBox("Option Five");  
  
// В данном примере обработчики событий не используются.
```

Поскольку основная цель данной программы — продемонстрировать прокрутку содержимого JPanel, мы не будем создавать обработчики событий. Флажки опций нужны лишь для того, чтобы заполнить панель компонентами.

5. Создайте объект JPanel, содержимое которого будет прокручиваться, и включите в его состав метку и флажки опций.

```
// Создание объекта JPanel, в котором будут  
// содержаться флажки опций.  
JPanel jpn1 = new JPanel();  
jpn1.setLayout(new GridLayout(6, 1));  
jpn1.setOpaque(true);  
  
// Включение флажков опций и метки в состав JPanel.  
jpn1.add(jlabOptions);  
jpn1.add(jcbOpt1);  
jpn1.add(jcbOpt2);  
jpn1.add(jcbOpt3);  
jpn1.add(jcbOpt4);  
jpn1.add(jcbOpt5);
```

6. Создайте объект JScrollPane, передав конструктору ссылку на панель. Затем включите панель с прокруткой в состав панели содержимого. И в завершение разрешите отображение фрейма.

```
// Создание панели с прокруткой  
JScrollPane jscrlp = new JScrollPane(jpn1);
```

```
// Добавление панели с прокруткой к фрейму.  
jfrm.getContentPane().add(jscrlp);  
  
// Отображение фрейма.  
jfrm.setVisible(true);  
}
```

**7. Код класса, как обычно, завершите методом main().**

```
public static void main(String args[]) {  
    // Фрейм создается в потоке обработки событий.  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            new ScrollJPanelDemo();  
        }  
    });  
}
```

**8. Полностью код программы приведен ниже.**

```
// Проект 4.1. Использование компонента JScrollPane  
// для прокрутки содержимого JPanel.  
import java.awt.*;  
import javax.swing.*;  
  
class ScrollJPanelDemo {  
  
    ScrollJPanelDemo() {  
        // Создание нового контейнера JFrame.  
        // Для него принимается диспетчер компоновки  
        // по умолчанию.  
        JFrame jfrm = new JFrame("Scroll a JPanel");  
  
        // Установка начальных размеров фрейма.  
        jfrm.setSize(280, 130);  
  
        // Завершение программы при закрытии окна пользователем.  
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        // Создание метки.  
        JLabel jlabOptions = new  
        JLabel("Select one or more options: ");  
  
        // Создание флажков опций.
```

```
JCheckBox jcbOpt1 = new JCheckBox("Option One");
JCheckBox jcbOpt2 = new JCheckBox("Option Two");
JCheckBox jcbOpt3 = new JCheckBox("Option Three");
JCheckBox jcbOpt4 = new JCheckBox("Option Four");
JCheckBox jcbOpt5 = new JCheckBox("Option Five");

// В данном примере обработчики событий не используются.

// Создание объекта JPanel, в котором будут
// содержаться флажки опций.
JPanel jpn1 = new JPanel();
jpn1.setLayout(new GridLayout(6, 1));
jpn1.setOpaque(true);

// Включение флажков опций и метки в состав JPanel.
jpn1.add(jlabOptions);
jpn1.add(jcbOpt1);
jpn1.add(jcbOpt2);
jpn1.add(jcbOpt3);
jpn1.add(jcbOpt4);
jpn1.add(jcbOpt5);

// Создание панели с прокруткой.
JScrollPane jscrlp = new JScrollPane(jpn1);

// Добавление панели с прокруткой к фрейму.
jfrm.getContentPane().add(jscrlp);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new ScrollJPanelDemo();
        }
    });
}
}
```

ВАЖНО!

## 4.7. Использование класса JTabbedPane

Еще один специализированный контейнер реализуется классом `JTabbedPane`. Он управляет компонентами, размещая их на вкладках. Выбирая вкладку, вы переводите связанные с ней компоненты “на передний план”. Панели с вкладками часто встречаются в современных приложениях. Несмотря на сложную структуру подобных панелей, они на удивление просты в использовании. На сегодняшний день `JTabbedPane` — один из самых мощных компонентов Swing.

В классе `JTabbedPane` определены конструкторы, приведенные в табл. 4.3. Конструктор, вызываемый без параметров, создает панель, в которой ярлыки вкладок располагаются вдоль верхней границы окна. Если число вкладок возрастает или их размеры увеличиваются и ярлыки не помещаются в одну строку, формируется следующая строка. Конструктор, указанный в таблице вторым, позволяет задавать расположение вкладок. Возможны четыре варианта размещения ярлыков: по верхней, по нижней, по левой и по правой части окна. Они задаются посредством констант `TOP`, `BOTTOM`, `LEFT` и `RIGHT`. Эти константы объявлены в интерфейсе `SwingConstants`, который реализуется классом `JTabbedPane`. Следовательно, для использования константы можно указывать как имя класса, так и имя интерфейса. Например, допустимы выражения `JTabbedPane.TOP` и `SwingConstants.TOP`.

Таблица 4.3. Конструкторы класса `JTabbedPane`

| Конструктор                                          | Описание                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JTabbedPane ()</code>                          | Создает панель с вкладками, в которой ярлыки вкладок располагаются вдоль верхней границы окна                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <code>JTabbedPane (int where)</code>                 | Создает панель с вкладками, в которой размещение ярлыков определяется параметром <code>where</code> . Данный параметр может принимать значение <code>JTabbedPane.TOP</code> , <code>JTabbedPane.BOTTOM</code> , <code>JTabbedPane.LEFT</code> или <code>JTabbedPane.RIGHT</code>                                                                                                                                                                                                                                                             |
| <code>JTabbedPane (int where, int sclrOrWrap)</code> | Создает панель с вкладками, в которой размещение ярлыков определяется параметром <code>where</code> . Данный параметр может принимать значение <code>JTabbedPane.TOP</code> , <code>JTabbedPane.BOTTOM</code> , <code>JTabbedPane.LEFT</code> или <code>JTabbedPane.RIGHT</code> . Если число ярлыков становится большим и они не помещаются в одну строку, их организация задается параметром <code>sclrOrWrap</code> . Он может принимать значение <code>JTabbedPane.WRAP_TAB_LAYOUT</code> или <code>JTabbedPane.SCROLL_TAB_LAYOUT</code> |



Третий конструктор позволяет определять расположение вкладок и политику, управляющую размещением ярлыков в том случае, когда они не помещаются в одну строку. Возможны два варианта. Ярлыки могут располагаться в несколько строк; эта политика, принимаемая по умолчанию, задается посредством константы `JTabbedPane.WRAP_TAB_LAYOUT`. Согласно другой политике, задаваемой посредством константы `JTabbedPane.SCROLL_TAB_LAYOUT`, все ярлыки размещаются в одну строку и пользователю предоставляются средства для их прокрутки.

Компонент `JTabbedPane` использует модель `SingleSelectionModel`. Однако необходимость в непосредственном взаимодействии с моделью возникает редко.

Компонент `JTabbedPane` генерирует событие изменения состояния. Происходит это тогда, когда пользователь переходит на другую вкладку. Однако в большинстве случаев необходимость в обработке этого события не возникает, поскольку переключение вкладок осуществляется автоматически. Очевидно, что при выборе вкладки над содержащимися на ней компонентами могут выполняться определенные действия, например, читаться информация из файла, необходимая для обновления элементов. Поэтому не исключено, что вам необходимо будет предусмотреть в своей программе поддержку соответствующих событий. Однако это опять же можно рассматривать скорее как исключение из общего правила.

Использование панели с вкладками не составляет труда. В первую очередь надо создать экземпляр `JTabbedPane`, а затем заполнить панель компонентами. При добавлении каждого компонента указывается имя вкладки. Если надо разместить на вкладке группу компонентов, желательно сначала организовать ее посредством `JPanel`. (Как вы, наверное, помните, `JPanel` — это легковесный контейнер, являющийся подклассом `JComponent`, поэтому его можно рассматривать и как контейнер, и как компонент.)

Для добавления вкладок можно использовать метод `addTab()`. Существуют три варианта данного метода. Самый простой из них имеет вид

```
void addTab(String name, Component comp)
```

В результате выполнения данного метода создается вкладка с именем, определенным посредством параметра `name`, и содержащая компонент, указанный с помощью параметра `comp`.

Ниже приведен пример программы, в которой создается простая панель, содержащая четыре вкладки. Для простоты в состав вкладок включаются только метки, но в них также могут содержаться любые другие элементы, включая группу компонентов, помещенную в легковесный контейнер. Окно, создаваемое при работе программы, показано на рис. 4.6.

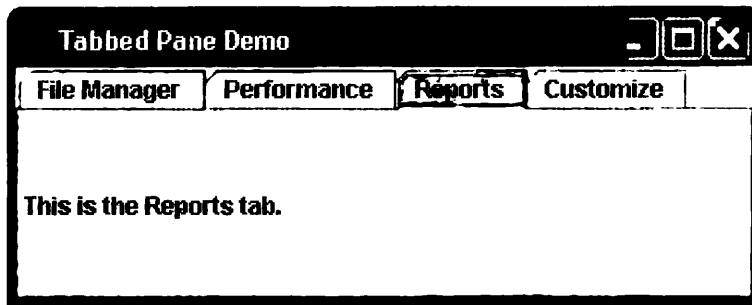


Рис. 4.6. Окно, отображаемое программой `TabbedPaneDemo`

// Пример использования компонента `JTabbedPane`.

```
import javax.swing.*;
```

```
class TabbedPaneDemo {
```

```
    TabbedPaneDemo() {
```

```
        // Создание нового контейнера JFrame.
```

```
        // Для него принимается диспетчер компоновки
```

```
        // по умолчанию, т.е. BorderLayout.
```

```
        JFrame jfrm = new JFrame("Tabbed Pane Demo");
```

```
        // Установка начальных размеров фрейма.
```

```
        jfrm.setSize(380, 150);
```

```
        // Завершение программы при закрытии окна пользователем.
```

```
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        // Создание панели с вкладками.
```

```
        JTabbedPane jtp = new JTabbedPane();
```

```
        // Добавление вкладок к панели.
```

```
        jtp.addTab("File Manager",  
            new JLabel(" This is the File Manager tab."));
```

```
        jtp.addTab("Performance",  
            new JLabel(" This is the Performance tab."));
```

```
        jtp.addTab("Reports",  
            new JLabel(" This is the Reports tab."));
```

```
        jtp.addTab("Customize",  
            new JLabel(" This is the Customize tab."));
```

```
// Включение панели с вкладками в состав панели содержимого.
jfrm.getContentPane().add(jtp);

// Отображение фрейма.
jfrm.setVisible(true);

}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new TabbedPaneDemo();
        }
    });
}
}
```

Обратите внимание на то, насколько мал объем кода, который необходимо написать для создания полнофункциональной панели с вкладками. Также заметьте, что для организации работы с вкладками не приходится вовсе прилагать усилий, — данная задача решается автоматически. В этом состоит одно из главных преимуществ специализированных контейнеров Swing: они обеспечивают богатые функциональные возможности, для использования которых требуется минимум кода.

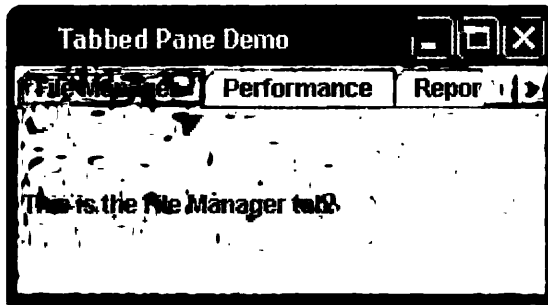
**ВАЖНО!**

**4.8.**

## **Дополнительные возможности JTabbedPane**

Компоненты `JTabbedPane` предоставляют программисту ряд возможностей по их настройке. Две из них уже упоминались при описании конструкторов `JTabbedPane`. Во-первых, вы можете задавать расположение ярлычков вкладок, а во-вторых, определять политику для случая, когда они не помещаются в одной строке. Предположим, что в предыдущем примере вызов конструктора панели был заменен следующей строкой:

```
JTabbedPane jtp = new JTabbedPane(SwingConstants.TOP,
    JTabbedPane.SCROLL_TAB_LAYOUT);
```



Теперь, уменьшив размеры окна, вы увидите, что в нем появилась полоса прокрутки, позволяющая сдвигать ярлыки вкладок. Окно программы показано ниже.

Для многих приложений возможность прокрутки ярлыков вкладок предпочтительнее размещения их в одну строку.

Часто бывает полезно отобразить на ярлыке вкладки не только текст, но и изображение. Сделать это позволяет следующий вариант метода `addTab()`:

```
void addTab(String name, Icon image, Component comp)
```

Здесь посредством параметра `image` задается графическое изображение, которое будет выводиться на ярлыке вкладки. В модуле 2 было показано, как можно создать изображение из файла, используя класс `ImageIcon`. Если вы хотите, чтобы на ярлыке было представлено изображение, а строка текста отсутствовала, задайте в качестве соответствующего параметра значение `null`.

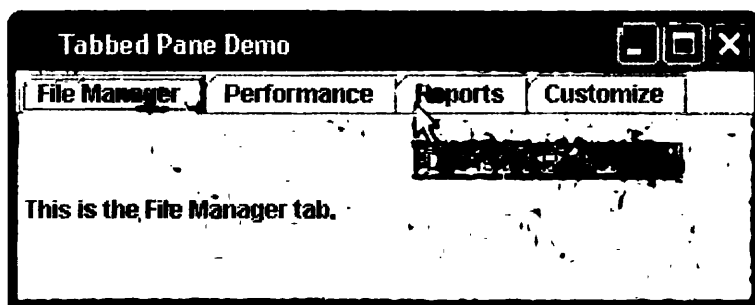
Часто для упрощения работы пользователя желательно реализовать динамическую подсказку, которая отображалась бы на экране в том случае, когда пользователь наводит курсор мыши на ярлык вкладки. Сделать это позволяет следующий вариант метода `addTab()`:

```
void addTab(String name, Icon image, Component comp,  
            String tooltip)
```

В результате выполнения формируется строка подсказки, заданная посредством параметра `tooltip`. Предположим, что в предыдущем примере вкладка `Reports` определена следующим образом:

```
jtp.addTab("Reports", null,  
          new JLabel(" This is the Reports tab."),  
          "This is the Reports tab.");
```

Тогда, если пользователь задержит курсор мыши на ярлыке данной вкладки, на экране появится подсказка `This is the Reports tab`. В этом случае окно примет вид, показанный ниже.



Если вы хотите задать строку подсказки, но не устанавливать изображение, установите в качестве параметра `image` значение `null`. (Как вы увидите далее в этом модуле, задать строку подсказки можно не только для вкладки, но и для любого компонента.)

Включить вкладку в конкретной позиции можно, вызвав метод `insertTab()`, заголовок которого приведен ниже.

```
void insertTab(String name, Icon image, Component comp,
               String tooltip, int idx)
```

Этот метод выполняет те же действия, что и `addTab()`, за исключением того, что он вставляет вкладку в позицию, указанную с помощью параметра `idx`. Индексы вкладок отсчитываются начиная с нуля. Так, если в предыдущей программе вы захотите вставить вкладку и используете для этого приведенную ниже строку кода, новая вкладка будет помещена после вкладки **Performance**; остальные вкладки сдвинутся вправо.

```
jtp.insertTab("New Tab", null, new JLabel("New Tab"),
              "New Tab Tooltip", 2);
```

Для удаления вкладок предусмотрен метод `remove()`.

```
void remove(int idx)
```

Параметр `idx` задает здесь индекс вкладки, предназначенной для удаления. Удаляя вкладку, вы также удаляете связанные с ней компоненты.

Возможности, предоставляемые `JTabbedPane`, не ограничиваются рассмотренными выше. Обсуждение их выходит за рамки данной книги. Поскольку вы, вероятнее всего, захотите детально изучить `JTabbedPane`, ниже перечислены средства, с которыми желательно ознакомиться в первую очередь.

- Установка цвета переднего плана вкладки: метод `setForegroundAt()`.
- Установка цвета фона вкладки: метод `setBackgroundAt()`.

- Установка мнемонического обозначения клавиши, используемой для перехода к вкладке: метод `setMnemonicAt()`.
- Выбор вкладки из программы: метод `setSelectedIndex()`.
- Разрешение или запрет доступа к вкладке: метод `setEnabledAt()`.
- Определение возможности доступа к вкладке: метод `isEnabledAt()`.
- Ярлык вкладки, к которой запрещен доступ, отображается серым цветом. При желании вы можете задать специальную пиктограмму для отображения такого режима. Для этого предназначен метод `setDisabledIconAt()`.



## Вопросы для текущего контроля

1. В панели с вкладками каждая вкладка должна содержать легковесный контейнер, например `JPanel`. Да или нет?
2. Какой метод надо вызвать, чтобы включить компонент в состав вкладки?
3. Предположим, что ярлыки вкладок не помещаются в одной строке. Какую политику надо использовать, чтобы вместо формирования следующей строки использовалась прокрутка?

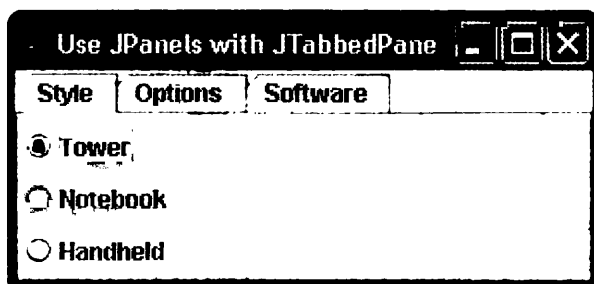
## Проект 4.2. Использование контейнеров `JPanel` в составе `JTabbedPane`

`TabbedPaneWithPanels.java` Выше был рассмотрен пример, демонстрирующий использование `JTabbedPane`, однако он не дает представления о всех возможностях данного контейнера. В большинстве случаев нецелесообразно использовать вкладку для размещения одного элемента, например метки или кнопки. Вместо этого следует поместить на вкладку легковесный контейнер, например `JPanel`, и заполнить его требуемыми компонентами. Таким образом, следует создать контейнер, содержащий группу связанных между собой компонентов, а затем сформировать вкладку для этого контейнера. Эта задача будет решена в рамках данного проекта.

1. Нет. Вы можете включить в состав панели с вкладками любой компонент, в том числе легковесный контейнер.
2. Для того чтобы включить компонент в состав вкладки, надо вызвать метод `addTab()`.
3. `JTabbedPane.SCROLL_TAB_LAYOUT`.

Проект предполагает создание панели `JTabbedPane` с тремя вкладками, которые содержат компоненты, предназначенные в помощь покупателю, выбирающему компьютер. Первая вкладка называется **Style**. На ней пользователь указывает тип компьютера: настольная система, портативная машина или карманный компьютер (**Tower**, **Notebook** или **Handheld**). Вторая вкладка, **Options**, позволяет пользователю указать требуемые возможности системы, например наличие записывающего DVD-устройства. Третья вкладка, **Software**, позволяет выбрать программное обеспечение, например, предназначенное для работы с текстом или для разработки программ. Выбор пользователя никак не влияет на действия, выполняемые программой, так как она предназначена лишь для того, чтобы продемонстрировать возможность отображения контейнеров `JPanel` посредством `JTabbedPane`.

Окно программы показано ниже.



## Последовательность действий

1. Создайте файл `TabbedPaneWithPanels.java` и включите в него следующие комментарии и выражения `import`:

```
// Проект 4.2. Использование JTabbedPane
// для отображения JPanel.
```

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

2. Начните код класса `TabbedPaneWithPanels` следующим образом:

```
class TabbedPaneWithPanels {

    JCheckBox jcbDVD;
    JCheckBox jcbScanner;
    JCheckBox jcbNtwrkRdy;
```

```
JCheckBox jcbWordProc;  
JCheckBox jcbCompiler;  
JCheckBox jcbDatabase;  
  
JRadioButton jrbTower;  
JRadioButton jrbNotebook;  
JRadioButton jrbHandheld;
```

Эти переменные будут хранить ссылки на компоненты, используемые в трех панелях.

3. Создайте конструктор `TabbedPaneWithPanels()`, начав его так, как показано ниже.

```
// Создание нового контейнера JFrame. В качестве  
// диспетчера компоновки принимается BorderLayout,  
// связанный по умолчанию с панелью содержимого  
// данного контейнера.  
JFrame jfrm = new JFrame("Use JPanels with JTabbedPane");  
  
// Установка начальных размеров фрейма.  
jfrm.setSize(280, 140);  
  
// Завершение программы при закрытии окна пользователем.  
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

4. Создайте кнопки переключателя опций, используемого для выбора типа компьютера, и включите их в контейнер `JPanel`.

```
// Создание кнопок переключателя опций для вкладки Style.  
jrbTower = new JRadioButton("Tower");  
jrbNotebook = new JRadioButton("Notebook");  
jrbHandheld = new JRadioButton("Handheld");  
ButtonGroup bg = new ButtonGroup();  
bg.add(jrbTower);  
bg.add(jrbNotebook);  
bg.add(jrbHandheld);  
  
// Создание объекта JPanel для размещения кнопок  
// переключателя опций.  
JPanel jpnl = new JPanel();  
jpnl.setLayout(new GridLayout(3, 1));  
jpnl.setOpaque(true);  
  
// Включение кнопок переключателя опций в состав панели.
```



```
jpn1.add(jrbTower);  
jpn1.add(jrbNotebook);  
jpn1.add(jrbHandheld);
```

Заметьте, что в данном случае используется диспетчер компоновки `GridLayout`. В результате выравнивание компонентов происходит автоматически и их взаимное расположение не зависит от размеров окна. Если такое размещение вас не устраивает, вы можете выбрать любой другой диспетчер компоновки.

**5. Создайте подобным образом компоненты для вкладок `Options` и `Software`.**

```
// Создание флажков опций для вкладки Options.  
jcbDVD = new JCheckBox("DVD Burner");  
jcbScanner = new JCheckBox("Scanner");  
jcbNtwrkRdy = new JCheckBox("Network Ready");  
  
// Создание объекта JPanel для хранения флажков опций.  
JPanel jpn12 = new JPanel();  
jpn12.setLayout(new GridLayout(3, 1));  
jpn12.setOpaque(true);  
  
// Включение флажков опций в контейнер JPanel.  
jpn12.add(jcbDVD);  
jpn12.add(jcbScanner);  
jpn12.add(jcbNtwrkRdy);  
  
// Создание флажков опций для вкладки Software.  
jcbWordProc = new JCheckBox("Word Processing");  
jcbCompiler = new JCheckBox("Program Development");  
jcbDatabase = new JCheckBox("Database");  
  
// Создание объекта JPanel для хранения флажков опций.  
JPanel jpn13 = new JPanel();  
jpn13.setLayout(new GridLayout(3, 1));  
jpn13.setOpaque(true);  
  
// Включение флажков опций в контейнер JPanel.  
jpn13.add(jcbWordProc);  
jpn13.add(jcbCompiler);  
jpn13.add(jcbDatabase);
```

**6. Приведенная ниже строка комментариев напоминает о том, что действия пользователя в программе не обрабатываются.**

// Обработчики событий в программе не используются.

В реальном приложении вам придется создать обработчик события для каждого компонента, расположенного на каждой вкладке.

7. Создайте контейнер `JTabbedPane` с тремя вкладками и разместите на этих вкладках три панели.

// Создание панели с вкладками. Если ярлыки вкладок

// не помещаются в одной строке, организуется их прокрутка.

```
JTabbedPane jtp = new JTabbedPane(JTabbedPane.TOP,  
JTabbedPane.SCROLL_TAB_LAYOUT);
```

// Включение контейнеров `JPanel` в состав панели с вкладками.

```
jtp.addTab("Style", jpn1);  
jtp.addTab("Options", jpn12);  
jtp.addTab("Software", jpn13);
```

8. Завершите конструктор кодом, добавляющим панель с вкладками к панели содержимого и разрешающим отображение фрейма.

```
// Добавление панели с вкладками к панели содержимого.  
jfrm.getContentPane().add(jtp);
```

```
// Отображение фрейма.  
jfrm.setVisible(true);
```

```
}
```

9. Как обычно, в состав класса `TabbedPaneWithPanels` включается метод `main()`.

```
public static void main(String args[]) {  
    // Фрейм создается в потоке обработки событий.  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            new TabbedPaneWithPanels();  
        }  
    });  
}
```

10. Полностью код программы `TabbedPaneWithPanels` приведен ниже.

```
// Проект 4.2. Использование JTabbedPane  
// для отображения JPanel.
```

```
import java.awt.*;
```

```
import javax.swing.*;

class TabbedPaneWithPanels {

    JCheckBox jcbDVD;
    JCheckBox jcbScanner;
    JCheckBox jcbNtwrkRdy;

    JCheckBox jcbWordProc;
    JCheckBox jcbCompiler;
    JCheckBox jcbDatabase;

    JRadioButton jrbTower;
    JRadioButton jrbNotebook;
    JRadioButton jrbHandheld;

    TabbedPaneWithPanels() {

        // Создание нового контейнера JFrame. В качестве
        // диспетчера компоновки принимается BorderLayout,
        // связанный по умолчанию с панелью содержимого
        // данного контейнера.
        JFrame jfrm = new JFrame("Use JPanels with JTabbedPane");

        // Установка начальных размеров фрейма.
        jfrm.setSize(280, 140);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание кнопок переключателя опций для вкладки Style.
        jrbTower = new JRadioButton("Tower");
        jrbNotebook = new JRadioButton("Notebook");
        jrbHandheld = new JRadioButton("Handheld");
        ButtonGroup bg = new ButtonGroup();
        bg.add(jrbTower);
        bg.add(jrbNotebook);
        bg.add(jrbHandheld);

        // Создание объекта JPanel для размещения кнопок
        // переключателя опций.
        JPanel jpnl = new JPanel();
        jpnl.setLayout(new GridLayout(3, 1));
```

```
jpn1.setOpaque(true);

// Включение кнопок переключателя опций в состав панели.
jpn1.add(jrbTower);
jpn1.add(jrbNotebook);
jpn1.add(jrbHandheld);

// Создание флажков опций для вкладки Options.
jcbDVD = new JCheckBox("DVD Burner");
jcbScanner = new JCheckBox("Scanner");
jcbNtwrkRdy = new JCheckBox("Network Ready");

// Создание объекта JPanel для хранения флажков опций.
JPanel jpn12 = new JPanel();
jpn12.setLayout(new GridLayout(3, 1));
jpn12.setOpaque(true);

// Включение флажков опций в контейнер JPanel.
jpn12.add(jcbDVD);
jpn12.add(jcbScanner);
jpn12.add(jcbNtwrkRdy);

// Создание флажков опций для вкладки Software.
jcbWordProc = new JCheckBox("Word Processing");
jcbCompiler = new JCheckBox("Program Development");
jcbDatabase = new JCheckBox("Database");

// Создание объекта JPanel для хранения флажков опций.
JPanel jpn13 = new JPanel();
jpn13.setLayout(new GridLayout(3, 1));
jpn13.setOpaque(true);

// Включение флажков опций в контейнер JPanel.
jpn13.add(jcbWordProc);
jpn13.add(jcbCompiler);
jpn13.add(jcbDatabase);

// Обработчики событий в программе не используются.

// Создание панели с вкладками. Если ярлыки вкладок
// не помещаются в одной строке, организуется их прокрутка.
JTabbedPane jtp = new JTabbedPane(JTabbedPane.TOP,
    JTabbedPane.SCROLL_TAB_LAYOUT);
```

```

// Включение контейнеров JPanel в состав панели с вкладками.
jtp.addTab("Style", jpnl1);
jtp.addTab("Options", jpnl2);
jtp.addTab("Software", jpnl3);

// Добавление панели с вкладками к панели содержимого.
jfrm.getContentPane().add(jtp);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater (new Runnable() {
        public void run() {
            new TabbedPaneWithPanels();
        }
    });
}
}

```

**ВАЖНО!**

## 4.9. Использование JSplitPane

JSplitPane — еще один специализированный контейнер, упрощающий решение некоторых достаточно сложных проблемы. Он формирует окно, состоящее из двух частей, в которых отображаются два компонента. Между компонентами помещается разделитель, который позволяет динамически изменять относительные размеры компонентов.

В классе JSplitPane определены конструкторы, показанные в табл. 4.4. Обратите внимание, что панель может разделяться по вертикали или по горизонтали. Ориентация разделителя определяет, будут ли компоненты отображаться рядом или один над другим. Разделяемая панель также поддерживает два способа перерисовки компонентов при изменении позиции разделителя. Компоненты могут обновляться постоянно (постоянная перерисовка) или один раз при остановке разделителя. Компоненты помещаются на разделяемую панель при ее создании. При желании вы также можете включить компоненты в уже готовую панель.

Таблица 4.4. Конструкторы класса JSplitPane

| Конструктор                                                                                                                 | Описание                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-----------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JSplitPane()</code>                                                                                                   | Создает пустую горизонтальную разделяемую панель. Если разделитель перемещается, обновление компонентов производится лишь после его остановки                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <code>JSplitPane(<br/>int vertOrHoriz)</code>                                                                               | Создает пустую разделяемую панель, ориентация которой задается посредством параметра <code>vertOrHoriz</code> , который может принимать значение <code>JSplitPane.HORIZONTAL_SPLIT</code> или <code>JSplitPane.HORIZONTAL_SPLIT</code> . Если разделитель перемещается, обновление компонентов производится лишь после его остановки                                                                                                                                                                                                                                                                                                                                                                       |
| <code>JSplitPane(<br/>int vertOrHoriz,<br/>boolean contRedraw)</code>                                                       | Создает пустую разделяемую панель, ориентация которой задается посредством параметра <code>vertOrHoriz</code> , который может принимать значение <code>JSplitPane.HORIZONTAL_SPLIT</code> или <code>JSplitPane.HORIZONTAL_SPLIT</code> . Если разделитель перемещается, обновление компонентов производится лишь после его остановки. Если значение параметра <code>contRedraw</code> равно <code>true</code> , осуществляется постоянная перерисовка компонентов при движении разделителя. В противном случае они обновляются лишь после остановки разделителя                                                                                                                                            |
| <code>JSplitPane(<br/>int vertOrHoriz,<br/>Component leftOrTop,<br/>Component rightOrBottom)</code>                         | Создает разделяемую панель, ориентация которой задается посредством параметра <code>vertOrHoriz</code> . Если разделитель перемещается, обновление компонентов производится лишь после его остановки. Панель включает в себя компоненты, заданные с помощью параметров <code>leftOrTop</code> и <code>rightOrBottom</code> . Позиции компонентов зависит от ориентации разделителя. Параметр <code>vertOrHoriz</code> может принимать значение <code>JSplitPane.HORIZONTAL_SPLIT</code> или <code>JSplitPane.VERTICAL_SPLIT</code>                                                                                                                                                                         |
| <code>JSplitPane(<br/>int vertOrHoriz,<br/>boolean contRedraw,<br/>Component leftOrTop,<br/>Component rightOrBottom)</code> | Создает разделяемую панель, ориентация которой задается посредством параметра <code>vertOrHoriz</code> . Панель включает в себя компоненты, заданные с помощью параметров <code>leftOrTop</code> и <code>rightOrBottom</code> . Позиции компонентов зависит от ориентации разделителя. Обновлением компонентов при перемещении разделителя управляет параметр <code>contRedraw</code> . Если его значение равно <code>true</code> , осуществляется постоянная перерисовка компонентов. В противном случае они обновляются лишь после остановки разделителя. Параметр <code>vertOrHoriz</code> может принимать значение <code>JSplitPane.HORIZONTAL_SPLIT</code> или <code>JSplitPane.VERTICAL_SPLIT</code> |

## Спросим у опытного программиста

**Вопрос.** Мне кажется, что постоянная перерисовка содержимого разделяемой панели дает лучшие результаты, чем однократное обновление компонентов после остановки разделителя. Почему постоянная перерисовка не используется всегда?

**Ответ.** Если для обновления компонентов требуется длительное время, то постоянная перерисовка неоправданно замедлит работу приложения. В этих случаях предпочтительнее использовать однократную перерисовку, которая производится после остановки разделителя. Однако, учитывая всевозрастающее быстродействие компьютеров, такое решение приходится принимать очень редко.

Используя разделяемую панель, обычно приходится указывать минимальные размеры содержащихся в ней компонентов. Тогда при перемещении разделителя размер компонента не станет меньше минимального. Чтобы в полной мере воспользоваться возможностями разделяемой панели, минимальный размер надо указывать настолько малым, насколько это допустимо. В некоторых случаях желательно также установить предпочтительные размеры компонентов. Они определяют исходное положение разделителя.

Ниже приведен простой пример, иллюстрирующий работу с `JSplitPane`. В процессе работы программа отображает две метки. Окно программы показано на рис. 4.7.

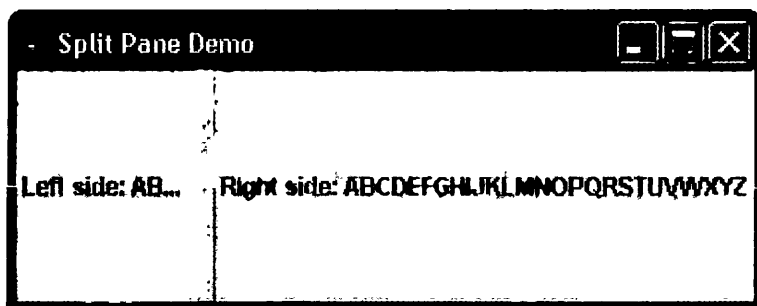


Рис. 4.7. Данные, отображаемые программой, демонстрирующей работу контейнера `SplitPaneDemo`

// Пример использования JSplitPane.

```
import javax.swing.*;
```

```
import java.awt.*;
```

```
class SplitPaneDemo {
```

```
    SplitPaneDemo() {
```

```
        // Создание нового контейнера JFrame.
```

```
        // Для него принимается диспетчер компоновки
```

```
        // по умолчанию, т.е. BorderLayout.
```

```
        JFrame jfrm = new JFrame("Split Pane Demo");
```

```
        // Установка начальных размеров фрейма.
```

```
        jfrm.setSize(380, 150);
```

```
        // Завершение программы при закрытии окна пользователем.
```

```
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        // Создание двух меток, предназначенных для отображения
```

```
        // в составе разделяемой панели.
```

```
        JLabel jlab =
```

```
            new JLabel(" Left side: ABCDEFGHIJKLMNOPQRSTUVWXYZ");
```

```
        JLabel jlab2 =
```

```
            new JLabel(" Right side: ABCDEFGHIJKLMNOPQRSTUVWXYZ");
```

```
        // Установка минимальных размеров каждой метки.
```

```
        // Этот шаг не является абсолютно необходимым, но
```

```
        // позволяет ограничить диапазон возможных положений
```

```
        // разделителя.
```

```
        jlab.setMinimumSize(new Dimension(90, 30));
```

```
        jlab2.setMinimumSize(new Dimension(90, 30));
```

```
        // Создание разделяемой панели, содержащей метки.
```

```
        JSplitPane jsp = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,  
            true, jlab, jlab2);
```

```
        // Включение разделяемой панели в состав панели содержимого.
```

```
        jfrm.getContentPane().add(jsp);
```

```
        // Отображение фрейма.
```

```
        jfrm.setVisible(true);
```

```
    }
```



```

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new SplitPaneDemo();
        }
    });
}
}

```

В данной программе создаются две метки. Для обеих устанавливается минимальная ширина, равная 90, и минимальная высота, равная 30. Это означает, что при перемещении разделителя удастся уменьшать размер компонента лишь до тех пор, пока не будет достигнуто минимальное значение. Если минимальный размер не установлен, то по умолчанию он будет принят равным размеру всей метки и перемещать разделитель не удастся. (Вы можете убедиться в этом, удалив вызов метода `setMinimumSize()`.) После установки минимальных размеров создается горизонтальная разделяемая панель и метки включаются в нее. Запустив программу, вы увидите, что при перемещении разделителя размеры меток изменяются.

**ВАЖНО!**

## **4.10. Дополнительные возможности JSplitPane**

Контейнер `JSplitPane` предоставляет дополнительные возможности, позволяющие настроить его в соответствии с требованиями пользователя. Некоторые из них описаны в данном разделе. Одна из наиболее важных — это возможность управлять ориентацией разделителя. Как было сказано ранее, разделитель может быть расположен по вертикали или по горизонтали; ориентация разделителя задается при создании панели. Кроме того, вы можете изменить данную характеристику уже существующей панели. Для этого надо вызвать метод `setOrientation()`.

```
void setOrientation(int vertOrHoriz)
```

Здесь значение параметра `vertOrHoriz` должно быть равно либо `JSplitPane.HORIZONTAL_SPLIT`, либо `JSplitPane.VERTICAL_SPLIT`.

Как правило, наиболее простым решением является указание компонентов при создании разделяемой панели, однако возможно также задавать компоненты после формирования панели. Это позволяют сделать следующие методы:

```
void setBottomComponent(Component comp)
void setLeftComponent(Component comp)
void setRightComponent(Component comp)
void setTopComponent(Component comp)
```

В каждом из них параметр `comp` задает компонент, который должен отображаться в соответствующей части панели. Несмотря на то что для включения компонентов предусмотрены четыре метода, можно обойтись двумя из них. Вызов `setBottomComponent()` для горизонтальной панели приведет к включению компонента в правую часть, а метод `setLeftComponent()` для вертикальной панели поместит компонент в ее нижнюю часть. Подобным образом ведут себя и два других метода.

Включить или отключить постоянную перерисовку можно путем вызова метода `setContinuousLayout()`.

```
void setContinuousLayout(boolean CLon)
```

Если значение параметра `CLon` равно `true`, постоянная перерисовка разрешена. В противном случае она запрещается.

В некоторых случаях могут быть очень полезны кнопки мгновенного расширения. Однако это средство доступно не во всех стилях. Так, например, при использовании стиля `Metal` эта возможность по умолчанию отключена, но вы можете включить ее. Средства мгновенного расширения — это две небольшие кнопки, помещаемые на разделитель. По щелчку на такой кнопке одна часть разделяемой панели расширяется и занимает собой все окно, а вторая пропадает из области видимости. Скрытую часть можно отобразить, активизировав другую кнопку. По умолчанию средства мгновенного расширения запрещены. Включить их можно, вызвав метод `setOneTouchExpandable()`.

```
void setOneTouchExpandable(boolean OTEon)
```

Если значение параметра `OTEon` равно `true`, кнопки мгновенного расширения помещаются на разделитель и становятся доступны для использования. Значение `false` данного параметра удаляет кнопки и отключает средства мгновенного расширения. Для того чтобы проверить, как работают кнопки мгновенного расширения, включите в код программы следующую строку, разместив ее сразу после вызова конструктора `JSplitPane()`:

```
jsp.setOneTouchExpandable(true);
```

После запуска программы вы увидите, что на разделителе появились две небольшие кнопки и он приобрел вид, показанный на рис. 4.8.

Существуют также и другие возможности. Вы можете указать, как должен вести себя разделитель при изменении размеров панели. Сделать это позволяет

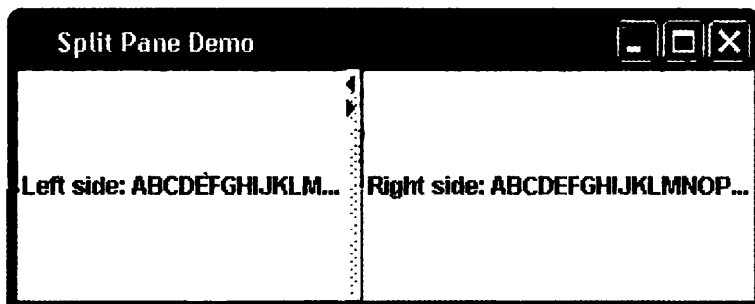


Рис. 4.8. Кнопки мгновенного расширения

метод `setResizeWeight()`. Установить размеры разделителя можно путем вызова `setDividerSize()`. Управлять расположением разделителя из программы позволяет метод `setDividerLocation()`.

## Вопросы для текущего контроля

1. Какое значение параметра, управляющего ориентацией, надо указать для создания вертикальной разделяемой панели?
2. Верно ли утверждение, что при использовании разделяемой панели часто возникает необходимость в установке минимальных размеров компонентов?
3. Какой метод надо вызывать для того, чтобы активизировать средства мгновенного расширения?

### ВАЖНО!

## 4.11 Формирование подсказки

При обсуждении панелей с вкладками было сказано о том, что для вкладки можно задать строку подсказки. Однако варианты использования подсказки не ограничиваются вкладками. Swing позволяет задать строку подсказки для любого легковесного компонента. Для этой цели в классе `JComponent` (который является суперклассом для всех легковесных компонентов Swing) предусмотр-

1. `JSplitPane.VERTICAL_SPLIT`.
2. Да.
3. `setOneTouchExpandable()`.

рен метод `setToolTipText()`. Этот метод позволяет указать строку, которая будет отображаться в том случае, если пользователь задержит курсор мыши на соответствующем компоненте. Задать строку подсказки настолько просто, что есть смысл определять ее для всех компонентов, назначение которых может быть не совсем понятно пользователю. Наличие такой подсказки повышает качество пользовательского интерфейса и упрощает работу с приложением.

Заголовок метода `setToolTipText()` имеет следующий вид:

```
void setToolTipText(String tip)
```

Здесь параметр `tip` задает текст, предназначенный для отображения на экране.

Приведенный ниже код программы показывает, насколько просто включить строку подсказки в приложение. В программе создаются две кнопки, с каждой из которых связывается строка подсказки. Строка, отображаемая при помещении курсора мыши на одну из кнопок, показана на рис. 4.9.

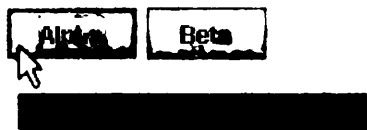


Рис. 4.9. Демонстрация строки подсказки

```
// Пример, демонстрирующий использование строк подсказки.
```

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

```
class ToolTipDemo {
```

```
    ToolTipDemo() {
```

```
        // Создание нового контейнера JFrame.
```

```
        JFrame jfrm = new JFrame("Add Tooltips");
```

```
        // Установка диспетчера компоновки FlowLayout.
```

```
        jfrm.getContentPane().setLayout(new FlowLayout());
```

```
        // Установка начальных размеров фрейма.
```

```
jfrm.setSize(300, 150);

// Завершение программы при закрытии окна пользователем.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Создание двух кнопок.
JButton jbtnAlpha = new JButton("Alpha");
JButton jbtnBeta = new JButton("Beta");

// Установка строк подсказки для кнопок.
jbtnAlpha.setToolTipText("This activates the Alpha option.");
jbtnBeta.setToolTipText("This activates the Beta option.");

jfrm.getContentPane().add(jbtnAlpha);
jfrm.getContentPane().add(jbtnBeta);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new ToolTipDemo();
        }
    });
}
```



## **Тест для самоконтроля по модулю 4**

1. Что такое двойная буферизация и зачем она нужна?
2. Какой метод надо вызвать, чтобы компонент `JPanel` стал непрозрачным?
3. Можно ли использовать в качестве панели содержимого объект `JPanel`, созданный в программе?
4. Несмотря на обширные возможности, предоставляемые элементом `JScrollPane`, пользоваться им очень сложно. Да или нет?
5. Назовите девять областей в составе `JScrollPane`.

6. Какой метод надо вызвать, чтобы установить заголовок строки JScrollPane?
7. Какой метод надо использовать для включения компонента в состав JTabbedPane?
8. Как установить строку подсказки для вкладки компонента JTabbedPane?
9. Назовите возможные варианты ориентации панели JSplitPane?
10. Что такое средства мгновенного расширения JSplitPane? Какой метод надо использовать, чтобы включить их?
11. Правда ли, что строку подсказки можно задать для любого легковесного компонента Swing? Если можно, то как сделать это?
12. Желательно не использовать строки подсказки слишком часто, так как пользовательский интерфейс оказывается перегруженным ими. Да или нет?
13. Добавьте строки подсказки для кнопок в программе CustomCPDemo, которая была рассмотрена в начале данной главы.
14. Модифицируйте пример, демонстрирующий работу с разделяемой панелью, так, чтобы в левой части этой панели отображалась панель с прокруткой, созданная в проекте 4.1.



# Модуль 5

## Списки

- 5.1.** Работа с компонентами `JList`
- 5.2.** Выбор нескольких пунктов списка
- 5.3.** Дополнительные возможности компонента `JList`
- 5.4.** Использование компонента `JComboBox`
- 5.5.** Средства редактирования в составе `JComboBox`
- 5.6.** Дополнительные возможности компонента `JComboBox`
- 5.7.** Работа с компонентами `JSpinner`
- 5.8.** Использование модели `SpinnerListModel`
- 5.9.** Использование модели `SpinnerNumberModel`
- 5.10.** Использование модели `SpinnerDateModel`



В этом модуле мы продолжаем обсуждение компонентов Swing. На этот раз речь пойдет о списках. Swing предоставляет три компонента для работы со списками: `JList`, `JComboBox` и `JSpinner`. Все они реализуют различные подходы к решению задачи, заключающейся в том, чтобы дать пользователю возможность выбрать один или несколько пунктов. Несмотря на то что все эти компоненты просты в использовании, каждый из них предоставляет большое количество дополнительных возможностей, позволяющих настраивать их в соответствии с требованиями конкретного приложения.

**ВАЖНО!**

## **5.1. Использование компонента `JList`**

Основной класс поддержки списков, предоставляемых Swing, называется `JList`. Он позволяет выбирать один или несколько пунктов. Несмотря на то что список состоит из строк, можно создать список, включающий практически любые объекты, способные отображаться на экране. Компонент `JList` используется в Java-программах настолько широко, что он вряд ли мог остаться для вас незамеченным.

Компонент `JList` основан на двух моделях. Первая из них — `ListModel`. Этот интерфейс определяет особенности доступа к данным списка. Swing предоставляет класс `DefaultListModel`, реализующий `ListModel`. В нем определено большое число методов, позволяющих работать со списком. Второй моделью является интерфейс `ListSelectionModel`, в котором объявлены методы, позволяющие определить, какой пункт или пункты списка были выбраны. В `JList` по умолчанию используется класс `DefaultListSelectionModel`, реализующий `ListSelectionModel`. Как правило, необходимость в непосредственном взаимодействии с моделями не возникает, так как класс `JList` предоставляет методы, позволяющие работать с различными функциями модели. Однако, если вы захотите организовать обработку события `ListDataEvent`, генерируемого при изменении содержимого списка, вам придется обращаться к модели.

Конструкторы, предоставляемые `JList`, перечислены в табл. 5.1. Если вы создали пустой список, то должны явным образом связать с ним модель. Конструкторы, которым в качестве параметра передается массив или вектор, автоматически создают модель. Для большинства приложений, использующих `JList`, этого вполне достаточно. Однако список, создаваемый такими конструкторами, нельзя изменить в процессе работы программы (решить эту задачу можно, связав со списком новую модель). Четвертый конструктор позволяет указывать модель при создании списка.

Таблица 5.1. Конструкторы класса *JList*

| Конструктор                               | Описание                                                                                                       |
|-------------------------------------------|----------------------------------------------------------------------------------------------------------------|
| <code>JList()</code>                      | Создает пустой элемент <code>JList</code>                                                                      |
| <code>JList(Object[] items)</code>        | Создает элемент <code>JList</code> , который содержит пункты, заданные посредством массива <code>items</code>  |
| <code>JList(Vector&lt;?&gt; items)</code> | Создает элемент <code>JList</code> , который содержит пункты, заданные посредством вектора <code>items</code>  |
| <code>JList(ListModel lm)</code>          | Создает элемент <code>JList</code> , который использует модель, заданную посредством параметра <code>lm</code> |

Несмотря на то что класс `JList` может использоваться сам по себе, чаще всего его помещают в состав контейнера `JScrollPane`. В этом случае автоматически реализуется возможность прокрутки длинных списков. Это упрощает работу над графическими пользовательскими интерфейсами и позволяет без труда изменить число пунктов списка, оставляя размер компонента `JList` постоянным. В данной книге мы будем следовать общепринятому подходу, т.е. использовать `JList` в составе `JScrollPane`.

Компонент `JList` генерирует событие `ListSelectionEvent` в тот момент, когда пользователь изменяет выбор пунктов. Это же событие генерируется при отмене выбора. Обработчик события `ListSelectionEvent` должен реализовывать интерфейс `ListSelectionListener`. В этом интерфейсе объявлен только один метод, `valueChanged()`, заголовок которого показан ниже.

```
void valueChanged(ListSelectionEvent le)
```

где параметр `le` — это ссылка на объект события. И хотя объект `ListSelectionEvent` предоставляет ряд полезных методов, детали, связанные с событием, обычно приходится выяснять путем опроса объекта `JList`.

По умолчанию `JList` позволяет пользователю выбирать несколько пунктов. Изменить данное поведение можно, вызвав метод `setSelectionMode()`, предоставляемый `JList`. Его заголовок выглядит следующим образом:

```
void setSelectionMode(int mode)
```

где параметр `mode` задает режим выбора пунктов. Он должен быть равен одной из следующих констант, объявленных в `ListSelectionModel`.

```
SINGLE_SELECTION
```

```
SINGLE_INTERVAL_SELECTION
```

```
MULTIPLE_INTERVAL_SELECTION
```

По умолчанию компонент `JList` принимает значение `MULTIPLE_INTERVAL_SELECTION`. В этом режиме пользователь может выбирать несколько интервалов содержимого списка. Значение `SINGLE_INTERVAL_SELECTION` позволяет выбирать только один интервал. В режиме `SINGLE_SELECTION` в каждый момент времени может быть выбран только один пункт. Очевидно, что в остальных двух режимах пользователь при желании также может ограничиться выбором одного пункта. Однако, если ему потребуется выбрать несколько пунктов, то компонент позволит сделать это.

Метод `getSelectedIndex()`, показанный ниже, возвращает индекс первого выбранного пункта. В режиме `SINGLE_SELECTION` этот пункт будет единственным.

```
int getSelectedIndex()
```

Отсчет индексов начинается с нуля. Таким образом, если выбран первый пункт, данный метод возвратит значение 0. Если ни один пункт не выбран, возвращается значение, равное -1.

Перед тем как приступить к более детальному обсуждению `JList`, рассмотрим простой пример, демонстрирующий использование данного компонента. Программа выводит список сортов яблок. Выбор пользователя отображается с помощью метки. Когда пользователь щелкает на кнопке **Buy Apple**, программа извлекает выбранный пункт и отображает подтверждение того, что покупка состоялась. Окно, отображаемое при работе программы, показано на рис. 5.1.



Рис. 5.1. Выходные данные программы `ListDemo`

// Пример использования компонента JList

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

```
class ListDemo {
```

```
    JList jlst;
    JLabel jlab;
    JScrollPane jscrlp;
    JButton jbtnBuy;
```

// Создание массива с именами сортов яблок.

```
String apples[] = { "Winesap", "Cortland", "Red Delicious",
    "Golden Delicious", "Gala", "Fuji",
    "Granny Smith", "Jonathan" };
```

```
ListDemo() {
```

// Создание нового контейнера JFrame.

```
JFrame jfrm = new JFrame("JList Demo");
```

// Установка диспетчера компоновки FlowLayout.

```
jfrm.getContentPane().setLayout(new FlowLayout());
```

// Установка начальных размеров фрейма.

```
jfrm.setSize(204, 200);
```

// Завершение программы при закрытии окна пользователем.

```
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

// Создание компонента JList на основе массива,

// состоящего из строк текста.

```
jlst = new JList(apples);
```

// Установка режима, позволяющего выбирать только

// один пункт списка.

```
jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

// Включение списка в состав панели с прокруткой.

```
jscrlp = new JScrollPane(jlst);
```

```
// Установка предпочтительных размеров панели с прокруткой.
jscroll.setPreferredSize(new Dimension(120, 90));

// Создание метки, отображающей выбор пользователя.
jlab = new JLabel("Please choose an apple.");

// Связывание со списком обработчика событий.
jlst.addListSelectionListener(new ListSelectionListener() {
    public void valueChanged(ListSelectionEvent le) {
        // Получение индекса текущего выбранного пункта.
        int idx = jlst.getSelectedIndex();

        // Если пользователь выбрал пункт, отображается
        // информация о выборе.
        if(idx != -1)
            jlab.setText("Current selection " + apples[idx]);
        else // В противном случае снова выводится приглашение
            // сделать выбор.
            jlab.setText("Please choose an apple.");
    }
});

// Формирование кнопки Buy Apple.
jbtnBuy = new JButton("Buy Apple");

// Связывание обработчика событий с кнопкой Buy Apple.
jbtnBuy.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        // Получение индекса пункта.
        int idx = jlst.getSelectedIndex();

        // Если пункт списка был выбран, выводится
        // информация о нем.
        if(idx != -1)
            jlab.setText("You purchased " + apples[idx]);
        else // В противном случае выводится сообщение
            // о неправильных действиях.
            jlab.setText("No apple has been selected.");
    }
});

// Включение списка и метки в состав панели содержимого.
```

```
jfrm.getContentPane().add(jscrlp);
jfrm.getContentPane().add(jbtnBuy);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new ListDemo();
        }
    });
}
```

Рассмотрим код примера более подробно. В первую очередь обратите внимание на то, что в начале программы был объявлен массив `apples` и инициализирован набором строк, представляющих различные сорта яблок. В конструкторе `ListDemo()` массив `apples` используется при создании компонента `JList`. Конструктор, которому в качестве параметра передается массив, как это происходит в данном случае, автоматически создает модель, содержащую элементы массива. Таким образом, создаваемый список будет содержать названия сортов яблок.

Далее устанавливается режим, предполагающий выбор только одного пункта списка. Другими словами, пользователь, работающий с программой, может в каждый момент времени выбрать только один сорт яблок. (Выбор нескольких пунктов списка будет рассмотрен в следующем разделе.) Затем `JList` помещается в `JScrollPane`, и для панели с прокруткой устанавливаются предпочтительные размеры, равные 120 и 90. Это делается для того, чтобы компонент выглядел компактно и был удобен в использовании.

После создания метки, предназначенной для отображения результатов выбора пользователем пункта списка, указан обработчик события, связанного с выбором. Данное событие возникает, когда пользователь выбирает пункт или изменяет свой выбор. В обработчике для получения индекса выбранного пункта вызывается метод `getSelectedIndex()`. Поскольку для списка был задан режим, ограничивающий выбор одним пунктом, индекс однозначно определяет этот пункт. Затем индекс используется для обращения к массиву `apples` и получения имени выбранного пункта. Заметьте, что в программе проверяет-

ся, равен ли индекс значению -1. Как вы помните, это значение возвращается, если ни один пункт списка не выбран. Такая ситуация может возникнуть в том случае, если событие было сгенерировано в результате отмены пользователем своего выбора. (Как вы помните, событие выбора генерируется, когда пользователь выбирает пункт списка или отменяет выбор.)

Далее создается кнопка **Buy Apple** и для нее устанавливается обработчик событий, соответствующий щелчку мышью. Данный обработчик выполняет те же действия, что и обработчик, связанный со списком, в частности, в нем также выполняется проверка на равенство индекса значению -1. Если ни один пункт списка не был выбран (так может произойти, если программа была только что запущена или если пользователь отменил выбор), отображается сообщение, информирующее пользователя о данной ситуации.

**ВАЖНО!**

## **5.2. Выбор нескольких пунктов**

В рассмотренной ранее программе пользователю предоставлялась возможность выбора только одного пункта из списка. Однако такое ограничение накладывается не всегда, более того, подобное поведение не присуще `JList` по умолчанию. Главным отличием между компонентом `JList`, позволяющим выбирать несколько пунктов, и таким же компонентом, ограничивающим выбор только одним пунктом, является обработчик события. Если можно выбрать несколько пунктов, необходимо иметь также возможность определить, какие именно пункты были выбраны. Проще всего сделать это с помощью метода `getSelectedIndices()`.

```
int[] getSelectedIndices()
```

Данный метод возвращает массив выбранных пунктов. Элементы массива следуют в порядке возрастания индексов. Если пользователь имеет возможность выбирать пункты списка в различных интервалах, то очевидно, что индексы в массиве не обязательно будут располагаться по порядку. Если ни один пункт не был выбран, длина возвращаемого массива будет равна нулю.

Ниже представлена модифицированная версия предыдущей программы, которая демонстрирует один из способов поддержки выбора пунктов, принадлежащих различным диапазонам. Окно, отображаемое при работе программы, показано на рис. 5.2.

```
// Выбор нескольких пунктов списка.
```

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

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

class MultiRangeList {

    JList jlst;
    JLabel jlab;
    JScrollPane jscrlp;
    JButton jbtnBuy;

    // Создание массива с названиями сортов яблок.
    String apples[] = { "Winesap", "Cortland", "Red Delicious",
        /
        "Golden Delicious", "Gala", "Fuji",
        "Granny Smith", "Jonathan" };

    MultiRangeList() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Multiple Range");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(180, 240);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание компонента JList. По умолчанию список
        // допускает выбор пунктов в нескольких интервалах.
        jlst = new JList(apples);

        // Включение списка в состав панели с прокруткой.
        jscrlp = new JScrollPane(jlst);

        // Установка предпочтительных размеров панели с прокруткой.
        jscrlp.setPreferredSize(new Dimension(120, 90));

        // Создание метки, отображающей выбор пользователя.
        jlab = new JLabel("Please choose an apple.");

        // Связывание обработчика событий со списком.
        jlst.addListSelectionListener(new ListSelectionListener() {
```



```
public void valueChanged(ListSelectionEvent le) {
    String what = "";

    // Получение массива с индексами выбранных пунктов.
    int indices[] = jlst.getSelectedIndices();

    // Проверка, был ли выбран хотя бы один пункт.
    // Если длина массива равна нулю, ни один пункт
    // не был выбран.
    if(indices.length == 0) {
        jlab.setText("Please choose an apple.");
        return;
    }

    // Отображение выбранных пунктов.
    for(int i = 0; i < indices.length; i++)
        what += apples[indices[i]] + "<br>";

    jlab.setText("<html>Current selection:<br>" + what);
}

});

// Создание кнопки Buy Apple.
jbtnBuy = new JButton("Buy Apple");

// Связывание с кнопкой Buy Apple обработчика событий.
jbtnBuy.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        String what = "";

        // Получение индексов выбранных пунктов.
        int indices[] = jlst.getSelectedIndices();

        // Проверка, был ли выбран хотя бы один пункт списка.
        if(indices.length == 0) {
            jlab.setText("No apple has been selected.");
            return;
        }

        // Отображение выбранных пунктов.
        for(int i = 0; i < indices.length; i++)
            what += apples[indices[i]] + "<br>";
```

```
jlab.setText("<html>Apples purchased:<br>" + what);
}
});

// Добавление списка и метки к панели содержимого.
jfrm.getContentPane().add(jscrlp);
jfrm.getContentPane().add(jbtnBuy);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new MultiRangeList();
        }
    });
}
}
```

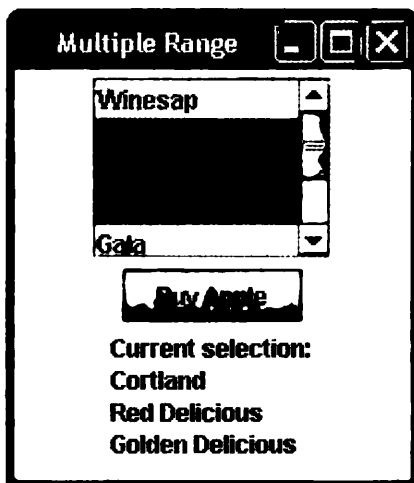


Рис. 5.2. Выходные данные, отображаемые программой *MultiRangeList*. Данная программа позволяет выбирать несколько пунктов компонента *JList*

Между данной программой и предыдущей ее версией есть два главных отличия. Во-первых, для списка принимается режим по умолчанию, позволяющий выбирать несколько пунктов, принадлежащих различным интервалам. По этой причине метод `setSelectionMode()` не вызывается. Во-вторых, следует отметить обработку событий выбора и событий действия. Для получения массива индексов выбранных пунктов вызывается метод `getSelectedIndices()`. Затем проверяется длина массива, чтобы убедиться, что в нем есть элементы. Если массив пуст, это значит, что ни один пункт не был выбран. В таком случае метка `jab` обновляется, чтобы отразить данный факт, и процедура обработки события завершается. Как вы помните, событие выбора генерируется при выборе элемента списка или при отмене выбора, поэтому метод `valueChanged()` может получить управление в тот момент, когда ни один пункт не был выбран. (Метод `actionPerformed()` может быть вызван, если пользователь щелкнет на кнопке `Buy Apple`, не выбрав ни один пункт списка.) Таким образом, необходимо подтвердить, что выбор действительно был сделан, а для этого в обработке события следует проверить длину массива. Для отображения информации о выборе пользователя формируется и выводится на экран HTML-строка, содержащая имена выбранных сортов.

**ВАЖНО!**

### **5.3. Дополнительные возможности компонента JList**

`JList` предоставляет разработчику много дополнительных возможностей. Обсуждение всех их не входит в круг задач данной книги. Однако некоторые средства мы все же рассмотрим. Начнем с альтернативного способа, позволяющего определить, какой пункт списка был выбран. Вместо того чтобы получать индекс пункта, вы можете определить значение, связанное с этим пунктом, вызвав один из следующих методов:

```
Object getSelectedValue()
```

```
Object[] getSelectedValues()
```

Метод `getSelectedValue()` возвращает ссылку на первое выбранное значение. Если ни одно из значений не было выбрано, возвращается значение `null`. Метод `getSelectedValues()` возвращает массив, содержащий ссылки на все выбранные пункты. Если ни один пункт не был выбран, длина возвращаемого массива равна нулю. Значения массива располагаются в том же порядке, что и в списке.

Если мы хотим использовать метод `getSelectedValues()`, то нам надо переписать обработчик событий выбора для предыдущей программы следующим образом:

```
// Обработчик событий выбора для списка.
jlst.addListSelectionListener(new ListSelectionListener() {
    public void valueChanged(ListSelectionEvent le) {
        String what = "";

        // Получение выбранных пунктов.
        Object[] strs = jlst.getSelectedValues();

        if(strs.length == 0) {
            jlab.setText("Please choose an apple.");
            return;
        }

        // Отображение выбранных пунктов.
        for(int i= 0; i < strs.length; i++)
            what += strs[i] + "<br>";

        jlab.setText("<html>Current selection:<br>" + what);
    }
});
```

Как вы помните, метод `getSelectedValues()` возвращает массив `Object`, поэтому чаще всего приходится выполнять приведение элементов массива к типу пунктов, реально содержащихся в списке. В данном случае необходимости в этом нет, поскольку в классе `Object` есть метод `toString()`, который автоматически вызывается при попытке получить строковое представление объекта. Однако в некоторых других приложениях следует явным образом преобразовывать типы.

Пункт списка можно вызвать из программы, используя для этой цели метод `setSelectedIndex()`.

```
void setSelectedIndex(int idx)
```

где параметр `idx` задает индекс пункта, который должен быть выбран. Данный метод часто используется для того, чтобы организовать первоначальное отображение списка с уже выбранным пунктом. Например, если вы добавите приведенную ниже строку к предыдущей программе, то при ее запуске будет выбран первый пункт списка, т.е. `Winesap`.

```
jlst.setSelectedIndex(0);
```

Если в списке поддерживается выбор нескольких пунктов, вы можете сделать выбор с помощью метода `setSelectedIndices()`.

```
void setSelectedIndices(int[] idxs)
```

Здесь целочисленный массив, передаваемый с помощью параметра `idxs`, содержит индексы пунктов, которые должны быть выбраны. Например, добавив к предыдущей программе приведенные ниже строки кода, вы обеспечите предварительный выбор первого, третьего и пятого элементов.

```
int indices[] = {0, 2, 4};  
jlst.setSelectedIndices(indices);
```

Если выбор по индексу вам по каким-то причинам не подходит, вы можете выбрать пункт списка по значению, вызвав метод `setSelectedValue()`.

```
void setSelectedValue(Object item, boolean scrollToItem)
```

С помощью параметра `item` задается значение, которое должно быть выбрано. Если значение параметра `scrollToItem` равно `true` и выбранный пункт списка не виден, прокрутка осуществляется так, чтобы он отображался на экране. Предположим, например, что в предыдущую программу включено выражение

```
jlst.setSelectedValue("Gala", true);
```

В результате его выполнения будет выбран пункт `Gala` и выполнена прокрутка списка, чтобы этот пункт был виден пользователю.

Отменить выбор любого пункта и даже всех их можно, вызвав метод `clearSelection()`.

```
void clearSelection()
```

После выполнения метода выбор всех пунктов будет отменен.

Определить, сделан ли выбор, можно, вызвав метод `isSelectionEmpty()`.

```
boolean isSelectionEmpty()
```

Он возвращает значение `true`, если ни один пункт не выбран, и `false` в противном случае.

Используя список с поддержкой выбора нескольких пунктов, бывает необходимо знать, какой из пунктов был выбран первым, а какой — последним. В терминологии `JList` первый выбранный пункт называется якорем (`anchor`), в последний — ведущим (`lead`). Если пользователь выбирает пункты списка в обратном порядке, индекс якоря будет больше или равен индексу ведущего

пункта. Получить индекс якоря и ведущего пункта можно с помощью методов `getAnchorSelectionIndex()` и `getLeadSelectionIndex()`.

```
void getAnchorSelectionIndex()
```

```
void getLeadSelectionIndex()
```

Оба метода возвращают значение, равное -1, если ни один из пунктов списка не был выбран.

Для установки пунктов списка `JList` используется метод `setListData()`.

Существуют две следующие разновидности данного метода:

```
void setListData(Object[] items)
```

```
void setListData(Vector<?> items)
```

где параметр `items` — это массив или вектор, содержащий пункты, которые надо отобразить в составе списка.

Вместо того чтобы передавать массив или вектор конструктору `JList()`, создающему список, можно перед формированием списка создать модель. Заполненную модель следует передать в качестве параметра соответствующему конструктору. Такой подход имеет два основных преимущества. Во-первых, можно сформировать модель, наилучшим образом соответствующую требованиям разрабатываемого приложения. Во-вторых, он позволяет изменять содержимое списка в процессе работы программы, добавляя элементы в модель или удаляя их. Этот подход отражен в проекте 5.1.

При выборе пользователем элемента списка или отмене выбора компонент `JList` может генерировать несколько событий. Выяснить, был ли произведен выбор, позволяет метод `getValueIsAdjusting()`. Он возвращает значение `true`, если выбор продолжается, и значение `false`, если этот процесс уже завершен. (Данный метод действует подобно одноименному методу, присутствующему в полосе прокрутки и линейном регуляторе.)

Еще одна возможность, которая может пригодиться вам — это управление воспроизведением ячейки. При необходимости вы можете определять, как будет отображаться каждый пункт списка. Класс, управляющий воспроизведением пунктов списка, должен реализовывать интерфейс `ListCellRenderer`. В этом интерфейсе объявлен только один метод `getListCellRendererComponent()`, который должен возвращать компонент, отображающий информацию для пункта списка. Для того чтобы задать специализированные средства воспроизведения ячеек, надо вызвать метод `setCellRenderer()` объекта списка.

## Спросим у опытного программиста

**Вопрос.** Могу ли я организовать выбор из программы пунктов списка, принадлежащих некоторому диапазону?

**Ответ.** Да. Для того чтобы выбрать пункты из диапазона, надо вызвать метод `setSelectionInterval()`.

```
void setSelectionInterval(int idxStart,
                        int idxStop)
```

Указанные границы входят в состав диапазона. Таким образом, пункты, соответствующие `idxStart` и `idxStop`, также будут выбраны. Заметьте: значение `idxStart` не обязательно должно быть меньше, чем `idxStop`. Параметр `idxStart` определяет якорь, а `idxStop` — ведущий пункт.

## Вопросы для текущего контроля

1. Какой вариант конструктора `JList` следует использовать для создания списка на основе массива?
2. Какое событие генерируется при выборе пользователем пункта меню?
3. Какой метод вызывается для получения индексов, соответствующих всем выбранным пунктам? Какой метод вызывается для получения значений всех выбранных пунктов?

## Проект 5.1. Работа с моделью списка

Для многих приложений, использующих `JList`, нет необходимости организовывать непосредственное взаимодействие с моделью списка. Как было показано в предыдущих примерах, модель, предоставляемая компонентом `JList`, который создается на основе массива или вектора, проста в использовании и обеспечивает достаточный набор функций. Однако если вам надо непосредственно взаимодействовать с данными, например добавлять пункты, то необходимо иметь доступ к модели. Получить его достаточно легко. Данный проект

1. `JList(Object[] items)`.
2. `ListSelectionEvent`.
3. `getSelectedIndices()` и `getSelectedValues()`.

демонстрирует работу с моделью списка. В нем модель задается при создании объекта `JList`; с ее помощью к списку добавляются и исключаются элементы.

Для того чтобы организовать взаимодействие с моделью списка, надо выполнить следующие действия.

1. Создать модель.
2. Заполнить модель пунктами списка.
3. Создать объект `JList`, передав модель конструктору.

Самый простой способ создания модели списка — использование класса `DefaultListModel`. Данная модель реализует интерфейс `ListModel`. Класс модели предоставляет большое количество методов, позволяющих управлять данными в списке, в том числе добавлять или удалять пункты списка. Модель `DefaultListModel` можно использовать непосредственно (как это будет сделано в данном проекте) или создать специальную версию модели, используя подкласс класса `DefaultListModel`.

После создания списка можно получить ссылку на модель, вызвав метод `getModel()`.

```
ListModel getModel()
```

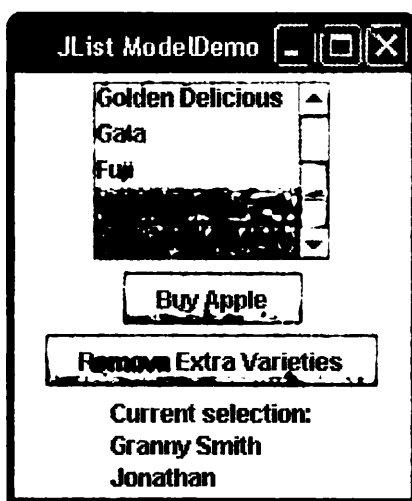
Как было сказано ранее, в данном проекте используется модель `DefaultListModel`. Этот класс содержит несколько методов (они выполняют те же функции, что и методы исходного класса `Vector`). Ниже описано назначение некоторых методов, включая те, которые используются в данном проекте.

| Метод                                             | Описание                                                                                                                                                                                                                                                                                 |
|---------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void add(int idx, Object item)</code>       | Включает пункт с индексом, заданным посредством параметра <code>idx</code>                                                                                                                                                                                                               |
| <code>void addElement(Object item)</code>         | Добавляет пункт к концу списка                                                                                                                                                                                                                                                           |
| <code>void clear()</code>                         | Удаляет пункты из списка                                                                                                                                                                                                                                                                 |
| <code>Object get(int idx)</code>                  | Возвращает пункт с индексом, соответствующим параметру <code>idx</code>                                                                                                                                                                                                                  |
| <code>int getSize()</code>                        | Возвращает число пунктов в списке                                                                                                                                                                                                                                                        |
| <code>Object remove(int idx)</code>               | Удаляет пункт с индексом, соответствующим параметру <code>idx</code> . Возвращает удаленный пункт                                                                                                                                                                                        |
| <code>void removeRange(int start, int end)</code> | Удаляет пункты списка в диапазоне, заданном посредством параметров <code>start</code> и <code>end</code> . Пункты, определяемые значениями <code>start</code> и <code>end</code> , входят в состав диапазона. Значение <code>start</code> не может быть больше значения <code>end</code> |



Используя эти методы, можно добавлять пункты к списку или удалять их. Также есть возможность определить, сколько пунктов содержится в списке. Заметьте, что в классе `DefaultListModel` определены и другие методы, которые могут помочь управлять списком.

В рамках данного проекта создаются различные варианты списка, использованного в предыдущем примере. Для добавления новых сортов яблок и удаления существующих используется модель списка. В окно программы включена дополнительная кнопка. При запуске программы на этой кнопке отображается надпись `Add More Varieties`. После щелчка мышью на кнопке новые пункты добавляются к списку. После этого надпись на кнопке заменяется на `Remove Extra Varieties`. Теперь щелчок мышью на кнопке приведет к удалению пунктов списка. Действия над списком осуществляются посредством модели. Окно, отображаемое в процессе работы программы, показано ниже.



## Последовательность действий

1. Создайте файл с именем `ListModelDemo.java` и включите в него следующие комментарии и выражения `import`.

```
// Проект 5.1. Использование интерфейса ListModel
```

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

2. Начните код класса `ListModelDemo` и его конструктор следующими строками:

```
class ListModelDemo {

    JList jlst;
    JLabel jlab;
    JScrollPane jscrip;
    JButton jbtnBuy;
    JButton jbtnAddDel;

    ListModelDemo() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("JList ModelDemo");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(180, 240);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

3. Создайте экземпляр класса `DefaultListModel` и включите в него пять названий сортов яблок.

```
// Создание и заполнение модели списка.
//
// Сначала создается экземпляр DefaultListModel.
DefaultListModel lm = new DefaultListModel();

// Затем к модели добавляются пункты.
lm.addElement("Winesap");
lm.addElement("Cortland");
lm.addElement("Red Delicious");
lm.addElement("Golder. Delicious");
lm.addElement("Gala");
```

В результате этих действий будет создана новая модель списка с именем `lm`, содержащая пять пунктов. Другими словами, после выполнения представленного выше кода `lm` будет содержать данные о пяти сортах яблок.

4. Создайте экземпляр класса `JList`, передав конструктору модель.

```
// Создание объекта JList на основании модели списка.  
jlst = new JList(lm);
```

После выполнения данной строки кода список `jlst` будет полностью инициализирован, т.е. будет содержать данные, указанные посредством `lm`.

5. Поместите объект `JList` в состав `JScrollPane` и установите предпочтительный размер.

```
// Включение списка в панель с прокруткой.  
jscrollp = new JScrollPane(jlst);
```

```
// Установка предпочтительных размеров панели с прокруткой.  
jscrollp.setPreferredSize(new Dimension(120, 90));
```

6. Создайте метку, которая будет отображать данные для пользователя.

```
// Создание метки, отображающей результаты выбора.  
jlab = new JLabel("Please choose an apple.");
```

7. Добавьте обработчик событий выбора. Заметьте, что в приведенном ниже обработчике вместо `getSelectedIndices()` для получения массива значений используется метод `getSelectedValues()`. Если бы такое изменение не было внесено, обработчик был бы аналогичен обработчику из предыдущего примера.

```
// Обработчик событий выбора для списка.  
jlst.addListSelectionListener(new ListSelectionListener() {  
    public void valueChanged(ListSelectionEvent le) {  
        String what = "";  
  
        // Получение пунктов списка.  
        Object values[] = jlst.getSelectedValues();  
  
        // Проверка, был ли выбран хотя бы один пункт.  
        if(values.length == 0) {  
            jlab.setText("Please choose an apple.");  
            return;  
        }  
  
        // Отображение выбранных пунктов.  
        for(int i = 0; i < values.length; i++)  
            what += values[i] + "<br>";  
  
        jlab.setText("<html>Current selection:<br>" + what);  
    }  
});
```

```
}  
});
```

8. Добавьте кнопку **Buy Apple** и обработчик событий, связанных с ней. Данный фрагмент кода совпадает с соответствующей частью предыдущего примера, за исключением того, что вместо `getSelectedIndices()` для получения информации о выбранных пунктах используется метод `getSelectedValues()`.

```
// Создание кнопки для "покупки" выбранного сорта.  
jbtnBuy = new JButton("Buy Apple");  
  
// Связывание обработчика событий с кнопкой Buy Apple.  
jbtnBuy.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent ae) {  
        String what = "";  
  
        // Получение выбранных пунктов.  
        Object values[] = jlst.getSelectedValues();  
  
        // Проверка, был ли выбран хотя бы один пункт списка.  
        if(values.length == 0) {  
            jlab.setText("No apple has been selected.");  
            return;  
        }  
  
        // Отображение выбранных пунктов.  
        for(int i = 0; i < values.length; i++)  
            what += values[i] + "<br>";  
  
        jlab.setText("<html>Apples purchased:<br>" + what);  
    }  
});
```

9. Создайте кнопку **Add More Varieties** и обработчик событий, связанных с ней. В обработчике добавление или удаление пунктов списка осуществляется посредством модели. Обратите внимание на этот факт.

```
// Создание кнопки, добавляющей пункты.  
jbtnAddDel = new JButton("Add More Varieties");  
  
// Связывание обработчика событий с кнопкой Add More Varieties.  
jbtnAddDel.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent ae) {
```

```

// Получение ссылки на модель.
DefaultListModel lm = (DefaultListModel) jlst.getModel();

// Проверка, были ли уже добавлены дополнительные пункты.
if(lm.getSize() > 5) {
    // При положительном результате проверки
    // пункты удаляются.
    for(int i=7; i > 4; i-) lm.remove(i);

    jbtnAddDel.setText("Add More Varieties");
} else {

    // Добавление дополнительных пунктов.
    lm.addElement("Fuji");
    lm.addElement("Granny Smith");
    lm.addElement("Jonathan");
    jbtnAddDel.setText("Remove Extra Varieties");
}
}
});

```

В начале обработчика событий извлекается ссылка на модель. Для этой цели вызывается метод `getModel()`. Далее эта модель используется для получения текущего размера списка. Если в списке содержится более пяти пунктов, шестой и последующие пункты удаляются посредством метода `remove()`. (Для удаления требуемых пунктов можно также один раз вызвать метод `removeRange()`. Попробуйте реализовать данное решение самостоятельно.) В противном случае программа добавляет к списку три пункта, вызывая для этого метод `addElement()`.

10. Завершите конструктор `ListModelDemo()` кодом, включающим панель с вкладками в панель содержимого и разрешающим отображение фрейма.

```

// Включение списка и метки в панель содержимого.
jfrm.getContentPane().add(jscrlp);
jfrm.getContentPane().add(jbtnBuy);
jfrm.getContentPane().add(jbtnAddDel);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

```

**11. Как обычно, в состав класса ListModelDemo включается метод main().**

```
public static void main(String args[]) {  
    // Фрейм создается в потоке обработки событий.  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            new ListModelDemo();  
        }  
    });  
}
```

**12. Полностью код программы ListModelDemo приведен ниже.**

```
// Проект 5.1. Использование интерфейса ListModel  
import javax.swing.*;  
import javax.swing.event.*;  
import java.awt.*;  
import java.awt.event.*;  
  
class ListModelDemo {  
  
    JList jlst;  
    JLabel jlab;  
    JScrollPane jScrollPane;  
    JButton jbtnBuy;  
    JButton jbtnAddDel;  
  
    ListModelDemo() {  
        // Создание нового контейнера JFrame.  
        JFrame jfrm = new JFrame("JList ModelDemo");  
  
        // Установка диспетчера компоновки FlowLayout.  
        jfrm.getContentPane().setLayout(new FlowLayout());  
  
        // Установка начальных размеров фрейма.  
        jfrm.setSize(180, 240);  
  
        // Завершение программы при закрытии окна пользователем.  
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        // Создание и заполнение модели списка.  
        //  
        // Сначала создается экземпляр DefaultListModel.
```

```
DefaultListModel lm = new DefaultListModel();

// No Затем к модели добавляются пункты.
lm.addElement("Winesap");
lm.addElement("Cortland");
lm.addElement("Red Delicious");
lm.addElement("Golden Delicious");
lm.addElement("Gala");

// Создание объекта JList на основании модели списка.
jlst = new JList(lm);

// Включение списка в панель с прокруткой.
jscrlp = new JScrollPane(jlst);

// Установка предпочтительных размеров панели с прокруткой.
jscrlp.setPreferredSize(new Dimension(120, 90));

// Создание метки, отображающей результаты выбора.
jlab = new JLabel("Please choose an apple.");

// Обработчик событий выбора для списка.
jlst.addListSelectionListener(new
ListSelectionListener() {
    public void valueChanged(ListSelectionEvent le) {
        String what = "";

        // Получение пунктов списка.
        Object values[] = jlst.getSelectedValues();

        // Проверка, был ли выбран хотя бы один пункт.
        if(values.length == 0) {
            jlab.setText("Please choose an apple.");
            return;
        }

        // Отображение выбранных пунктов.
        for(int i = 0; i < values.length; i++)
            what += values[i] + "<br>";

        jlab.setText("<html>Current selection:<br>" + what);
    }
});
```

```
// Создание кнопки для "покупки" выбранного сорта.
jbtnBuy = new JButton("Buy Apple");

// Связывание обработчика событий с кнопкой Buy Apple.
jbtnBuy.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        String what = "";

        // Получение выбранных пунктов.
        Object values[] = jlst.getSelectedValues();

        // Проверка, был ли выбран хотя бы один пункт списка.
        if(values.length == 0) {
            jlab.setText("No apple has been selected.");
            return;
        }

        // Отображение выбранных пунктов.
        for(int i = 0; i < values.length; i++)
            what += values[i] + "<br>";

        jlab.setText("<html>Apples purchased:<br>" + what);
    }
});

// Создание кнопки, добавляющей пункты.
jbtnAddDel = new JButton("Add More Varieties");

// Связывание обработчика событий
// с кнопкой Add More Varieties.
jbtnAddDel.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {

        // Получение ссылки на модель.
        DefaultListModel lm = (
            DefaultListModel) jlst.getModel();

        // Проверка, были ли уже добавлены дополнительные
        // пункты.
        if(lm.getSize() > 5) {
            // При положительном результате проверки
            // пункты удаляются.
```



```

        for(int i=7; i > 4; i-) lm.remove(i);

        jbtnAddDel.setText("Add More Varieties");
    } else {
        // Добавление дополнительных пунктов.
        lm.addElement("Fuji");
        lm.addElement("Granny Smith");
        lm.addElement("Jonathan");
        jbtnAddDel.setText("Remove Extra Varieties");
    }
}

});

// Включение списка и метки в панель содержимого.
jfrm.getContentPane().add(jscrlp);
jfrm.getContentPane().add(jbtnBuy);
jfrm.getContentPane().add(jbtnAddDel);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new ListModelDemo();
        }
    });
}
}

```

**ВАЖНО!**

## **5.4. Использование JComboBox**

Одной из разновидностей списков является раскрывающийся список. В составе такого списка может присутствовать либо активизирующая его кнопка, либо поле редактирования. Поскольку список раскрывается лишь тогда, когда он необходим, данный компонент очень компактен и его очень удобно использовать в случае, если размеры окна малы или интерфейс перегружен элементами. Для поддержки раскрывающихся списков в Swing предусмотрен класс `JComboBox`.

В составе компонента `JComboBox` используется модель `ComboBoxModel`. Этот интерфейс расширяет `ListModel` и объявляет дополнительные методы `getSelectedItem()` и `setSelectedItem()`. Если раскрывающийся список допускает изменение пункта, в нем используется интерфейс `MutableComboBoxModel`, расширяющий `ComboBoxModel`. Реализацией интерфейса `MutableComboBoxModel` по умолчанию является класс `DefaultComboBoxModel`. В данном классе, помимо обязательных методов, объявленных в интерфейсе, содержатся также методы, позволяющие добавлять пункты к списку и удалять их. Они работают подобно методам из класса `DefaultListModel`, который использовался в проекте 5.1. В большинстве случаев нет необходимости в непосредственном взаимодействии с моделью, так как класс `JComboBox` предоставляет методы, позволяющие выполнять практически все требуемые действия.

Конструкторы, предоставляемые `JComboBox`, перечислены в табл. 5.2.

Таблица 5.2. Конструкторы класса `JComboBox`

| Конструктор                                   | Описание                                                                                                  |
|-----------------------------------------------|-----------------------------------------------------------------------------------------------------------|
| <code>JComboBox()</code>                      | Создает пустой раскрывающийся список                                                                      |
| <code>JComboBox(Object[] items)</code>        | Создает раскрывающийся список, содержащий пункты, заданные посредством массива <code>items</code>         |
| <code>JComboBox(Vector&lt;?&gt; items)</code> | Создает раскрывающийся список, содержащий пункты, заданные посредством вектора <code>items</code>         |
| <code>JComboBox(ComboBoxModel cbm)</code>     | Создает раскрывающийся список, который использует модель, заданную посредством параметра <code>cbm</code> |

В процессе работы компонент `JComboBox` генерирует события действия. Они возникают тогда, когда пользователь выбирает пункт списка (или при вводе пользователем строки, если в состав раскрывающегося списка входит поле редактирования). Таким образом, получение команды действия означает, что пользователь выбрал пункт списка. `JComboBox` также генерирует события элемента. Это происходит при изменении состояния компонента, т.е. при выборе пункта списка или отмене выбора. Таким образом, при изменении выбора пункта списка могут возникать два события: одно, соответствующее выбору, а другое — отмене выбора. Во многих случаях достаточно организовать обработку событий действия, но при необходимости в вашем распоряжении находятся оба типа события.

Текущий выбор можно определить двумя способами. Один из них — вызов метода `getSelectedItem()`, который возвращает ссылку на выбранный пункт. Его заголовок выглядит следующим образом:

```
Object getItemSelected()
```

Если ни один пункт не выбран, возвращается значение, равное `null`.

Второй способ — вызов метода `getSelectedIndex()`.

```
int getSelectedIndex()
```

Этот метод возвращает индекс выбранного пункта списка. Индексы отсчитываются от нуля. Если ни один пункт не выбран, возвращается значение, равное `-1`.

Выбрать пункт списка можно также из программы. Для этого предусмотрены методы `setSelectedItem()` и `setSelectedIndex()`. Их заголовки имеют следующий вид:

```
void setSelectedItem(Object item)
```

```
void setSelectedIndex(int idx)
```

где параметры `item` и `idx` задают соответственно пункт и индекс пункта для выбора. Как вы помните, отсчет индексов начинается с нуля.

В большинстве случаев использование `JComboBox` не составляет труда. Достаточно передать конструктору массив, содержащий пункты, которые должны отображаться в раскрывающемся списке, и зарегистрировать обработчик событий. При каждом изменении выбора обработчик событий будет получать управление. Ниже приведен код программы, демонстрирующей ра-

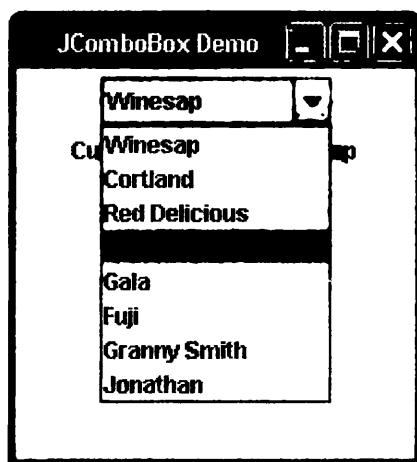


Рис. 5.3. Данные, отображаемые программой `ComboBoxDemo`

боту с раскрывающимся списком. Окно, отображаемое при работе программы, показано на рис. 5.3.

// Пример использования раскрывающегося списка.

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

class ComboBoxDemo {

    JComboBox jcbb;
    JLabel jlab;

    // Создание массива с названиями сортов яблок.
    String apples[] = { "Winesap", "Cortland", "Red Delicious",
                        "Golden Delicious", "Gala", "Fuji",
                        "Granny Smith", "Jonathan" };

    ComboBoxDemo() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("JComboBox Demo");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(220, 240);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание компонента JComboBox на основе массива строк.
        jcbb = new JComboBox(apples);

        // Создание метки, отображающей выбор пользователя.
        jlab = new JLabel();

        // Связывание обработчика событий с раскрывающимся списком.
        jcbb.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent le) {
                // Получение ссылки на выбранный пункт.
                String item = (String) jcbb.getSelectedItem();
            }
        });
    }
}
```

```
        // Отображение выбранного пункта.
        jlab.setText("Current selection: " + item);
    }
});

// Первоначально в списке выбирается первый пункт.
jcbb.setSelectedIndex(0);

// Добавление раскрывающегося списка и метки
// к панели содержимого.
jfrm.getContentPane().add(jcbb);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new ComboBoxDemo();
        }
    });
}
}
```

В данной программе отображается тот же список сортов яблок, что и в примере, демонстрирующем работу с компонентом `JList`. Однако теперь компонент занимает на экране меньше места. При создании раскрывающегося списка конструктору передается массив `apples`. В результате автоматически создается модель `ComboBoxModel`, содержащая пункты из массива.

С раскрывающимся списком `jcbb` связывается обработчик событий. Он получает управление каждый раз, когда пользователь выбирает пункт списка. Выбранный пункт извлекается в теле метода `actionPerformed()` посредством вызова `getSelectedItem()`. Поскольку событие действия генерируется только при выборе пункта, ситуация, при которой ни один пункт не является выбранным, исключается. В результате выполнения обработчика на экране отображается выбранный пункт.

Заметьте, что первый пункт списка выбирается из программы с помощью метода `setSelectedIndex()`. Так реализуется выбор по умолчанию. В результате вызова данного метода также генерируется событие действия. Применять подобное решение в каждой программе не обязательно.

**ВАЖНО!**

## **5.5. Создание раскрывающегося списка, допускающего редактирование**

Как было сказано ранее, существует возможность создать раскрывающийся список, позволяющий редактировать выбранный пункт. Если в составе раскрывающегося списка присутствует поле редактирования, пользователь может не только выбрать пункт списка, но и задать произвольное значение.

Более того, можно даже выбрать пункт из списка, а затем отредактировать его. Если нажать клавишу `<Enter>` в тот момент, когда поле редактирования владеет фокусом ввода, будет сгенерировано событие действия, причем информация, содержащаяся в поле, будет принята в качестве текущего выбора. Надо, однако, заметить, что ввод нового значения или редактирование пункта списка не оказывает влияния на сам список.

Для того чтобы сформировать раскрывающийся список с редактированием, надо сначала создать экземпляр `JComboBox`, а затем вызвать метод `setEditable(true)` полученного объекта. Заголовок метода `setEditable()` выглядит следующим образом:

```
void setEditable(boolean canEdit)
```

Если значение параметра `canEdit` равно `true`, в состав раскрывающегося списка включается поле редактирования. Значение `false` данного параметра удаляет поле редактирования из состава компонента.

Для того чтобы добавить функции редактирования к раскрывающемуся списку из предыдущего примера, надо лишь включить после вызова конструктора `JComboBox()` следующие строки:

```
// Создание раскрывающегося списка, допускающего редактирование.
jcbb.setEditable(true);
```

После того как данные изменения будут внесены, раскрывающийся список примет вид, показанный на рис. 5.4. Для того чтобы принять строку, содержащуюся в поле, в качестве текущего выбора, надо нажать клавишу `<Enter>` тогда, когда поле будет обладать фокусом ввода. Введенное вами значение не включается в состав списка.

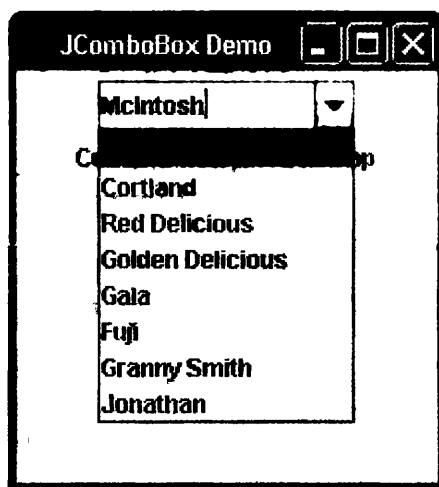


Рис. 5.4. Раскрывающийся список с редактированием

**ВАЖНО!**

## 5.6. Дополнительные возможности JComboBox

Подобно JList, компонент JComboBox предоставляет много дополнительных возможностей — гораздо больше, чем можно описать в одном модуле. В данном разделе мы рассмотрим лишь самые популярные из них. При разработке приложений очень полезным может оказаться динамическое добавление или удаление пунктов раскрывающегося списка в процессе выполнения программы. Эта возможность поддерживается методами `addItem()` и `removeItem()`.

```
void addItem(Object item)
```

```
void removeItem(Object item)
```

С помощью параметра `item` задается добавляемый или удаляемый пункт списка. Данные методы доступны только для модифицируемых раскрывающихся списков. Именно такой список поддерживает модель `DefaultComboBoxModel`, поэтому приведенные выше методы можно применять при работе с моделью по умолчанию, используемой JComboBox. Новый пункт помещается в конец списка.

При необходимости можно также удалить пункт списка, передав его индекс методу `removeItemAt()`. Для удаления всех пунктов списка предусмотрен метод `removeAllItems()`. Заголовки этих методов имеют следующий вид:

```
void removeItemAt(int idx)
```

```
void removeAllItems()
```

где параметр `idx` задает индекс пункта, предназначенного для удаления.

Определить размер списка (т.е. число содержащихся в нем пунктов) можно, вызвав метод `getItemCount()`.

```
int getItemCount()
```

Если список пуст, возвращается нулевое значение.

Ниже приведен код программы, которая демонстрирует использование некоторых из описанных методов. По сравнению с первым примером, посвященным раскрывающимся спискам, в нее внесены следующие изменения. Во-первых, создаваемый компонент допускает редактирование. Во-вторых, если пользователь вводит значение, не совпадающее ни с одним из пунктов списка, оно добавляется к списку. И в-третьих, в окно программы включается кнопка **Remove Selection**. По щелчку на этой кнопке выделенные пункты удаляются из списка. Окно, отображаемое при работе программы, показано на рис. 5.5.

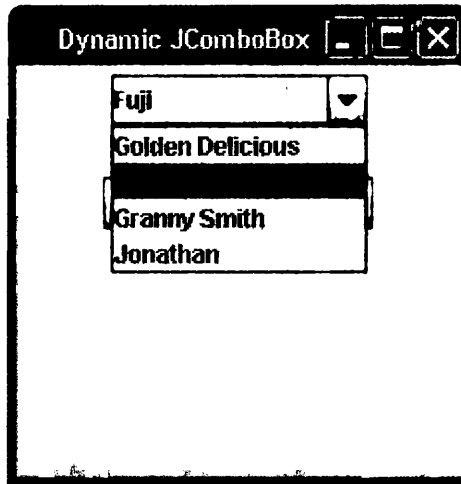


*Рис. 5.5. Окно, отображаемое программой DynamicComboBox. Данная программа позволяет динамически добавлять и удалять пункты списка. Обратите внимание на то, что некоторые из названий сортов яблок были удалены*



```
// Динамическое добавление и удаление пунктов
// раскрывающегося списка.

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

class DynamicComboBox {

    JComboBox jcbb;
    JLabel jlab;
    JButton jbtnRemove;

    // Создание массива с названиями сортов яблок.
    String apples[] = { "Winesap", "Cortland", "Red Delicious",
                        "Golden Delicious", "Gala", "Fuji",
                        "Granny Smith", "Jonathan" };

    DynamicComboBox() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Dynamic JComboBox");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(220, 240);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание компонента JComboBox.
        jcbb = new JComboBox(apples);

        // Включение средств редактирования в состав
        // раскрывающегося списка.
        jcbb.setEditable(true);

        // Создание метки, отображающей выбор пользователя.
        jlab = new JLabel();

        // Связывание обработчика событий с раскрывающимся списком.
        // Новый пункт, введенный пользователем, добавляется
```

```
// к списку.
jcbb.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        // Получение ссылки на выбранный пункт.
        String item = (String) jcbb.getSelectedItemAt();

        // Если ни один пункт не выбран, никакие действия
        // не выполняются.
        if(item==null) return;

        // Отображение выбранного пункта.
        jlab.setText("Current selection: " + item);

        // Если данный пункт отсутствует в списке,
        // он включается в его состав.
        int i;

        // Проверка наличия пункта в списке.
        for(i=0; i < jcbb.getItemCount(); i++)
            if(item.equals(jcbb.getItemAt(i)))
                break; // Пункт есть в списке.

        // Если пункта нет в списке, он добавляется.
        if(i==jcbb.getItemCount()) jcbb.addItem(item);
    }
});

// Первоначально в списке выбирается первый пункт.
jcbb.setSelectedIndex(0);

// Создание кнопки Remove Selection.
jbtnRemove = new JButton("Remove Selection");

// Связывание обработчика событий с кнопкой.
jbtnRemove.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        // Получение ссылки на выбранный пункт,
        String item = (String) jcbb.getSelectedItemAt();

        // Если ни один пункт не выбран, никакие действия
        // не выполняются.
        if(item==null) return;
```

```
        // Удаление выбранного пункта из списка.
        jcbb.removeItem(item);

        // Отображение выбранного пункта.
        jlab.setText("Removed " + item);
    }
});

// Включение раскрывающегося списка, метки и кнопки
// в состав панели содержимого.
jfrm.getContentPane().add(jcbb);
jfrm.getContentPane().add(jlab);
jfrm.getContentPane().add(jbtnRemove);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new DynamicComboBox();
        }
    });
}
}
```

В программе предусмотрен фрагмент кода для добавления к списку пунктов, введенных пользователем в поле редактирования. Добавление пункта осуществляется только в том случае, если он отсутствует в списке. (В результате дублирование пунктов не происходит.) Данный фрагмент помещен в состав обработчика событий, связанных с раскрывающимся списком. Он имеет следующий вид:

```
// Если данный пункт отсутствует в списке,
// он включается в его состав.
int i;

// Проверка наличия пункта в списке,
for(i=0; i < jcbb.getItemCount(); i++)
    if(item.equals(jcbb.getItemAt(i)))
```

```
break; // Пункт есть в списке.
```

```
// Если пункта нет в списке, он добавляется.  
if(i==jcbb.getItemCount()) jcbb.addItem(item);
```

Рассмотрим подробнее данный код. Пункты списка обрабатываются в цикле `for`, переменная цикла принимает значение от нуля до числа, на единицу меньшего, чем количество пунктов в списке. Размер списка возвращает метод `getItemCount()`. На каждой итерации выбранное значение сравнивается с пунктом списка, полученным посредством метода `getItemAt()`. Если очередной пункт совпадает с выбранным значением, то выполнение цикла прерывается. В противном случае выполняются все итерации, после чего переменная `i` становится равной величине, возвращаемой методом `getItemCount()`. Это является признаком отсутствия соответствующего пункта в списке. Тогда данное значение добавляется к списку с помощью метода `addItem()`.

В данной программе интересен также обработчик событий, связанных с кнопкой `Remove Selection`. Этот обработчик удаляет из списка выбранный пункт. Код обработчика приведен ниже.

```
jbtnRemove.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent le) {  
        // Получение ссылки на выбранный пункт.  
        String item = (String) jcbb.getSelectedItem();  
  
        // Если ни один пункт не выбран, никакие действия  
        // не выполняются.  
        if(item==null) return;  
  
        // Удаление выбранного пункта из списка.  
        jcbb.removeItem(item);  
  
        // Отображение выбранного пункта.  
        jlab.setText("Removed " + item);  
    }  
});
```

В обработчике в первую очередь вызывается метод, возвращающий ссылку на выбранный пункт списка. Если возвращается значение `null` (это может быть в том случае, если список пуст), выполнение обработчика завершается. В противном случае он удаляет выбранный пункт из списка, вызывая метод `removeItem()`.

Рассмотрим еще некоторые средства, которые, возможно, заинтересуют вас. Раскрыть список из программы позволяет метод `setVisible()`.

```
void setVisible(boolean show)
```

Если значение параметра `show` равно `true`, список отображается на экране, в противном случае он сворачивается. Кроме того, свернуть список можно, вызвав метод `hide()`.

Разрешить или запретить доступ к раскрываемому списку позволяет метод `setEnabled()`.

```
void setEnabled(boolean enable)
```

Если значение параметра `enable` равно `true`, доступ к списку разрешен. В противном случае список недоступен.

Получить ссылку на модель раскрываемого списка позволяет метод `getModel()`.

```
ComboBoxModel getModel()
```

Для установки модели служит метод `setModel()`, заголовок которого имеет следующий вид:

```
void setModel(ComboBoxModel cbm)
```

Как было сказано ранее, в большинстве случаев необходимость непосредственного взаимодействия с моделью не возникает.



## Вопросы для текущего контроля

1. Какое событие генерируется, когда пользователь выбирает пункт раскрываемого списка?
2. Какой метод позволяет получить текущий выбранный пункт раскрываемого списка?
3. Какой метод надо вызвать, чтобы создать раскрывающийся список, допускающий редактирование?

- 
1. `ActionEvent`.
  2. `getSelectedItem()`.
  3. `setEditable()`.

## Спросим у опытного программиста

- Вопрос.** Существует ли способ выяснить, раскрыт ли список в данный момент?
- Ответ** Да. Для этого надо вызвать метод `isPopupVisible()`. Он вернет значение `true`, если список отображается на экране, и `false` в противном случае.

**ВАЖНО!**

## 5.6. Использование компонента JSpinner

К числу компонентов Swing относится инкрементный регулятор, в процессе работы которого также используется список. Инкрементный регулятор представляет собой список, снабженный кнопками со стрелками (подобными тем, которые отображаются на полосе прокрутки). С помощью данного компонента можно прокручивать список и редактировать его элементы. Для прокрутки списка можно также использовать клавиши со стрелкой вверх и вниз, расположенные на клавиатуре. Текущий пункт списка отображается в поле редактирования. Инкрементные регуляторы были реализованы в версии 1.4 Java и поддерживаются классом `JSpinner`.

Компонент `JSpinner` использует модель, основанную на интерфейсе `SpinnerModel`. В этом интерфейсе определены методы, перечисленные в табл. 5.3. Заметьте, что компонент предоставляет доступ к текущему пункту (который отображается в поле редактирования, а также к предыдущему и последующему (если они существуют). Обратиться к произвольному элементу невозможно. Таким образом, данные в составе инкрементного регулятора лучше всего рассматривать как упорядоченную последовательность. Поскольку действия с компонентом `JSpinner` существенно зависят от данных (так, например, для разных типов данных требуются различные редакторы), то обычно приходится для каждого регулятора создавать свою модель. Этим инкрементный регулятор отличается от других компонентов Swing, для которых в большинстве случаев подходит модель по умолчанию.

Таблица 5.3. Методы, определенные в классе `SpinnerModel`

| Метод                                                     | Описание                                                                                                                                                                         |
|-----------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void addChangeListener(ChangeListener cl)</code>    | Устанавливает <code>cl</code> в качестве обработчика событий изменения состояния                                                                                                 |
| <code>Object getNextValue()</code>                        | Возвращает значение следующего пункта. Если текущим является последний пункт списка, возвращается значение <code>null</code>                                                     |
| <code>Object getPreviousValue()</code>                    | Возвращает значение предыдущего пункта. Если текущим является первый пункт списка, возвращается значение <code>null</code>                                                       |
| <code>Object getValue()</code>                            | Возвращает значение текущего пункта. Это значение отображается в поле редактирования                                                                                             |
| <code>void removeChangeListener(ChangeListener cl)</code> | Удаляет обработчика событий изменения состояния <code>cl</code>                                                                                                                  |
| <code>void setValue(Object val)</code>                    | Устанавливает текущее значение равным <code>val</code> . Если <code>val</code> представляет недопустимое значение, генерируется исключение <code>IllegalArgumentException</code> |

Интерфейс `SpinnerModel` частично реализуется абстрактным классом `AbstractSpinnerModel`. Этот класс использован в качестве базового для следующих стандартных моделей.

|                                 |                                                                                                  |
|---------------------------------|--------------------------------------------------------------------------------------------------|
| <code>SpinnerDateModel</code>   | Управление списком дат                                                                           |
| <code>SpinnerListModel</code>   | Управление списком, пункты которого определены посредством массива либо набора <code>List</code> |
| <code>SpinnerNumberModel</code> | Управление списком чисел                                                                         |

Для создания инкрементного регулятора, использующего последовательность числовых значений, подходит модель `SpinnerNumberModel`. Для того чтобы создать регулятор, работающий с датами, можно использовать класс `SpinnerDateModel`. Другие списки поддерживаются классом `SpinnerListModel`. С его помощью можно, например, создать регулятор, который будет перебирать строки.

Конструкторы, предоставляемые `JSpinner`, перечислены в табл. 5.4. Первый из конструкторов создает так называемый инкрементный регулятор по умолчанию, обеспечивающий перебор числовых значений. Однако такой ком-

понент имеет ограниченное применение, поскольку набор возможных чисел не ограничен. Обычно при создании инкрементного регулятора для числовых значений указывается верхняя и нижняя граница. Чтобы сделать это, либо чтобы обеспечить перебор данных других типов, надо, создавая экземпляр `JSpinner`, указать модель.

Таблица 5.4. Конструкторы класса `JSpinner`

| Конструктор                             | Описание                                                                                       |
|-----------------------------------------|------------------------------------------------------------------------------------------------|
| <code>JSpinner()</code>                 | Создает инкрементный регулятор, который перебирает целые числа из неограниченного списка       |
| <code>JSpinner(SpinnerModel spm)</code> | Создает инкрементный регулятор на базе модели, заданной посредством параметра <code>spm</code> |

Компонент `JSpinner` генерирует события изменения состояния. Такое событие возникает при изменении текущего значения. Данное изменение происходит либо при переходе к следующему значению списка, либо при вводе данных вручную в поле редактирования.

`JSpinner` предоставляет методы, обеспечивающие доступ к текущему, предыдущему и следующему значениям. Эти методы имеют следующий вид:

```
Object getValue()
Object getPreviousValue()
Object getNextValue()
```

Благодаря наличию данных методов необходимость непосредственного взаимодействия с моделью, как правило, не возникает.

**ВАЖНО!**

## 5.7. Использование модели `SpinnerListModel`

Из трех встроенных моделей инкрементных регуляторов `SpinnerListModel` является наиболее универсальной, поскольку она может использоваться для перебора списка значений произвольного типа, например строк. Для данной модели определены конструкторы, перечисленные в табл. 5.5. Класс `SpinnerListModel` реализует методы, объявленные в интерфейсе `SpinnerModel`, и, кроме того, два дополнительных метода.

```
List<?> getList()
void setList(List<?> items)
```



Таблица 5.5. Конструкторы класса `SpinnerListModel`

| Конструктор                                         | Описание                                                                                 |
|-----------------------------------------------------|------------------------------------------------------------------------------------------|
| <code>SpinnerListModel ()</code>                    | Создает модель, не содержащую данных                                                     |
| <code>SpinnerListModel (Object[] items)</code>      | Создает модель, которая содержит данные, заданные посредством массива <code>items</code> |
| <code>SpinnerListModel (List&lt;?&gt; items)</code> | Создает модель, которая содержит данные, заданные посредством списка <code>items</code>  |

Метод `getList ()` возвращает ссылку на список. Метод `setList ()` формирует новый список на базе данных, переданных в качестве параметра `items`. Обратите внимание, что список не копируется, вместо этого метод сохраняет ссылку на него.

Ниже приведен код программы, которая демонстрирует работу компонента `JSpinner`, использующего модель `SpinnerListModel`. Данный инкрементный регулятор управляет списком строк, представляющих основные цвета RGB: Red, Green и Blue. Выбранный цвет используется для отображения рамки вокруг метки, с помощью которой выводится информация о выборе пользователя. Окно, создаваемое при работе программы, показано на рис. 5.6.

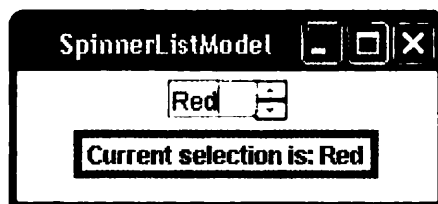


Рис. 5.6. Выходные данные программы, использующей `SpinnerListModel`

```
// Демонстрация работы инкрементного регулятора,
// основанного на модели SpinnerListModel.
```

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

```
class SpinnerDemo {
```

```
    JSpinner jspin;
    JLabel jlab;
```

```
// Создание массива цветов RGB.
String colors[] = { "Red", "Green", "Blue" };

SpinnerDemo() {
    // Создание нового контейнера JFrame.
    JFrame jfrm = new JFrame("SpinnerListModel");

    // Установка диспетчера компоновки FlowLayout.
    jfrm.getContentPane().setLayout(new FlowLayout());

    // Установка начальных размеров фрейма.
    jfrm.setSize(220, 100);

    // Завершение программы при закрытии окна пользователем.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Создание модели, поддерживающей список строк.
    SpinnerListModel spm = new SpinnerListModel(colors);

    // Создание компонента JSpinner и указание модели.
    jspin = new JSpinner(spm);

    // Установка предпочтительных размеров
    // инкрементного регулятора.
    jspin.setPreferredSize(new Dimension(60, 20));

    // Создание метки, которая отображает выбор пользователя.
    // Вокруг метки выводится цветная рамка. Поскольку Red -
    // первый пункт списка, он выбирается по умолчанию при
    // создании инкрементного регулятора.
    jlab = new JLabel(" Current selection is: Red ");
    jlab.setBorder(BorderFactory.createLineBorder(Color.RED, 4));

    // Связывание обработчика событий изменения состояния с
    // инкрементным регулятором.
    jspin.addChangeListener(new ChangeListener() {
        public void stateChanged(ChangeEvent ce) {
            // Получение выбранного значения.
            String color = (String) jspin.getValue();

            // Отображение выбранного значения с помощью метки.
            jlab.setText(" Current selection is: " +
                color + " ");
        }
    });
}
```

```

        // Формирование рамки вокруг метки. Рамка
        // отображается цветом, выбранным пользователем.
        if (color.equals("Red"))
            jlab.setBorder(
                BorderFactory.createLineBorder(Color.RED, 4));
        else if (color.equals("Green"))
            jlab.setBorder(
                BorderFactory.createLineBorder(Color.GREEN, 4));
        else
            jlab.setBorder(
                BorderFactory.createLineBorder(Color.BLUE, 4));
    }
});

// Включение инкрементного регулятора и метки
// в состав панели содержимого.
jfrm.getContentPane().add(jspin);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new SpinnerDemo();
        }
    });
}
}

```

В данной программе создается массив `colors`, содержащий строки "Red", "Green" и "Blue". Данный массив используется для создания экземпляра класса `SpinnerListModel`. Затем модель передается конструктору `JSpinner`. При отображении инкрементного регулятора в качестве текущего выбора автоматически принимается первый пункт списка, в данном случае это Red. Когда пользователь изменяет выбор, обработчик события извлекает новое значение, вызывая метод `getValue()`. Выбранное значение отображается в составе метки, после чего используется для вывода рамки.

ВАЖНО!

## 5.8. Использование модели SpinnerNumberModel

Если необходимо организовать с помощью инкрементного регулятора перебор числовых значений, следует использовать модель `SpinnerNumberModel`. В этой модели содержатся свойства, определяющие минимальную и максимальную величину шага (т.е. приращение при переходе от предыдущего значения к следующему) и само значение (соответствующее свойство унаследовано от `SpinnerModel`). Конструкторы данного класса перечислены в табл. 5.6.

Таблица 5.6. Конструкторы класса `SpinnerNumberModel`

| Конструктор                                                                                  | Описание                                                                                                                                                                                                                |
|----------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>SpinnerNumberModel()</code>                                                            | Создает модель по умолчанию, поддерживающую неограниченный список. Начальное значение равно нулю, а величина приращения — единице                                                                                       |
| <code>SpinnerNumberModel(int val, int min, int max)</code>                                   | Создает модель, в которой начальное значение устанавливается равным <code>val</code> , минимальное — <code>min</code> и максимальное — <code>max</code>                                                                 |
| <code>SpinnerNumberModel(double val, double min, double max, double stepSize)</code>         | Создает модель, в которой начальное значение устанавливается равным <code>val</code> , минимальное — <code>min</code> и максимальное — <code>max</code> . Величину приращения определяет параметр <code>stepSize</code> |
| <code>SpinnerNumberModel(Number val, Comparable min, Comparable max, Number stepSize)</code> | Создает модель, в которой начальное значение равно <code>val</code> , а приращение — <code>stepSize</code> . Минимальное и максимальное значение задается с помощью объектов <code>min</code> и <code>max</code>        |

В классе `SpinnerNumberModel` реализованы методы, объявленные в интерфейсе `SpinnerModel`, а кроме того, содержатся дополнительные методы, обеспечивающие доступ к максимальному и минимальному значениям, а также к величине шага.

```
Comparable getMaximum()
void setMaximum(Comparable max)
Comparable getMinimum()
void setMinimum(Comparable min)
Number getStepSize()
void setStepSize(Number stepSize)
```

Несмотря на то что параметры методов `setMaximum()` и `setMinimum()` имеют тип `Comparable`, вы можете указывать при вызове данных методов объекты-числа. (Все числовые типы Java реализуют интерфейс `Comparable`, поэтому допустимо использование их в качестве параметров указанных методов.) Кроме того, класс `SpinnerNumberModel` предоставляет также метод `getNumber()`.

```
Number getNumber()
```

Он возвращает текущее значение как объект `Number`.

Ниже приведен код программы, демонстрирующей использование модели `SpinnerNumberModel`. В ней создается инкрементный регулятор, предназначенный для перебора чисел от 1 до 10. Выбранное значение используется для установки толщины рамки вокруг метки, отображающей текущее значение. При изменении значения инкрементного регулятора изменяется толщина рамки. Окно, создаваемое при работе программы, показано на рис. 5.7.

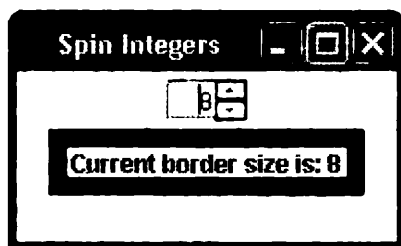


*Рис. 5.7. Данные, отображаемые программой SpinInts*

```
// Выбор целочисленных значений с помощью
// инкрементного регулятора.
```

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

```
class SpinInts {
```

```
    JSpinner jspin;
    JLabel jlab;
```

```
    SpinInts() {
```

```
// Создание нового контейнера JFrame.
JFrame jfrm = new JFrame("Spin Integers");

// Установка диспетчера компоновки FlowLayout.
jfrm.getContentPane().setLayout(new FlowLayout());

// Установка начальных размеров фрейма.
jfrm.setSize(200, 120);

// Завершение программы при закрытии окна пользователем.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Создание модели, используемой для поддержки
// целых чисел.
SpinnerNumberModel spm =
    new SpinnerNumberModel(1, 1, 10, 1);

// Создание компонента JSpinner, использующего
// подготовленную ранее модель.
jspin = new JSpinner(spm);

// Установка предпочтительных размеров
// инкрементного регулятора.
jspin.setPreferredSize(new Dimension(40, 20));

// Создание метки, отображающей выбор пользователя.
jlab = new JLabel(" Current border size is: 1 ");
jlab.setBorder(BorderFactory.createLineBorder(Color.BLACK));

// Связывание обработчика событий изменения состояния
// с инкрементным регулятором.
jspin.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent ce) {
        // Получение текущего размера.
        Integer bSize = (Integer) jspin.getValue();

        // Отображение сведений о текущем размере.
        jlab.setText(" Current border size is: " +
            bSize + " ");

        // Установка рамки вокруг метки. Толщина рамки
        // определяется значением, выбранным пользователем
        // с помощью инкрементного регулятора.
```

```

        jlab.setBorder(
            BorderFactory.createLineBorder(Color.BLACK,
                bSize.intValue()));

    }
});

// Включение инкрементного регулятора и метки
// в состав панели содержимого.
jfrm.getContentPane().add(jspin);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new SpinInts();
        }
    });
}
}
}

```

В данной программе экземпляр `SpinnerNumberModel` создается с помощью следующего выражения:

```
SpinnerNumberModel spm = new SpinnerNumberModel(1, 1, 10, 1);
```

Создаваемая модель определяет диапазон значений от 1 до 10, величину приращения, 1, и начальное значение, равное 1. Затем модель используется для формирования инкрементного регулятора. При желании вы можете изменить величину приращения и посмотреть результаты.

При изменении текущего значения инкрементного регулятора управление получает обработчик события, который использует новое значение для установки размера рамки, отображаемой вокруг кнопки. Таким образом, при каждом изменении значения инкрементного регулятора изменяется внешний вид рамки. Если вы будете удерживать одну из кнопок со стрелкой в составе регулятора, размер рамки будет непрерывно изменяться до тех пор, пока не достигнет граничного значения.

ВАЖНО!

5.10.

## Использование модели SpinnerDateModel

Инкрементные регуляторы часто используются для установки даты. Чтобы упростить разработчикам решение данной задачи, в состав Swing включен класс `SpinnerDateModel`. В нем определены конструкторы, показанные в табл. 5.7. Класс `SpinnerDateModel` содержит свойства, определяющие начальную и конечную даты, а также свойство `calendarField`, которое задает правила изменения даты. Например, вы можете указать в качестве приращения день, месяц, час или минуту. Как правило, при выборе даты принимается величина шага, равная одному дню. При установке времени удобно задать шаг, равный одной минуте. Очевидно, что конкретный выбор приращения определяется особенностями приложения. Следует иметь в виду, что при использовании некоторых стилей поле `calendarField` не принимается во внимание и изменяется поле, выбранное пользователем. Поэтому полностью полагаться на данное свойство нельзя.

Таблица 5.7. Конструкторы класса `SpinnerDateModel`

| Конструктор                                                                             | Описание                                                                                                                                                                                                                                                                                          |
|-----------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>SpinnerDateModel()</code>                                                         | Создает модель, в котором в качестве начального значения принимается текущая дата и устанавливается величина шага, равная одному дню. Диапазон значений не ограничен                                                                                                                              |
| <code>SpinnerDateModel(Date val, Comparable begin, Comparable end, int calField)</code> | Создает модель, в которой начальное значение задается посредством параметра <code>val</code> . Значения изменяются в диапазоне, определенном с помощью параметров <code>begin</code> и <code>end</code> . В поле <code>calendarField</code> записывается значение параметра <code>calField</code> |

Свойство `calendarField` может принимать одно из приведенных ниже значений (они определены в классе `Calendar`).

|                      |              |             |
|----------------------|--------------|-------------|
| AM_PM                | DAY_OF_MONTH | DAY_OF_WEEK |
| DAY_OF_WEEK_IN_MONTH | DAY_OF_YEAR  | ERA         |
| HOUR                 | HOUR_OF_DAY  | MILLISECOND |
| MINUTE               | MONTH        | SECOND      |
| WEEK_OF_MONTH        | WEEK_OF_YEAR | YEAR        |



Как вы уже знаете, в некоторых случаях величина приращения может не приниматься во внимание.

В классе `SpinnerDateModel` реализованы методы, объявленные в интерфейсе `SpinnerModel`. Кроме того, в нем содержатся дополнительные методы, предоставляющие доступ к начальной и конечной датам, а также к полю `calendarField`.

```
int getCalendarField()  
Comparable getEnd()  
Comparable getStart()  
void setCalendarField(int calField)  
void setEnd(Comparable end)  
void setStart(Comparable begin)
```

Несмотря на то что для `setEnd()` и `setStart()` предусмотрены параметры типа `Comparable`, вы можете передавать этим методам объекты `Date` (класс `Date` реализует интерфейс `Comparable`). Кроме того, класс `SpinnerDateModel` предоставляет также метод `getDate()`.

```
Date getDate()
```

Он возвращает текущее значение в виде объекта `Date`.

Ниже приведен код программы, демонстрирующей использование модели `SpinnerDateModel`. В ней создается инкрементный регулятор, позволяющий выбирать дату и время. Нижняя граница диапазона равна текущей дате минус один месяц, а верхняя соответствует текущей дате плюс один месяц. Окно, создаваемое при работе программы, показано на рис. 5.8.

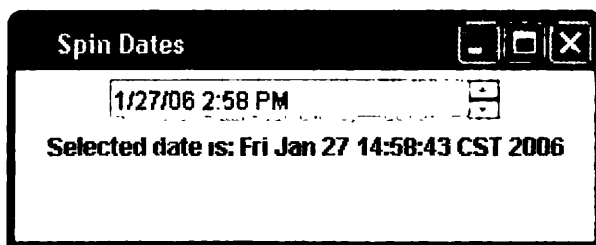


Рис. 5.8. Данные, отображаемые программой `SpinDates`

// Выбор даты с помощью инкрементного регулятора.

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

```
class SpinDates {
```

```
    JSpinner jspin;
    JLabel jlab;
```

```
SpinDates() {
```

```
    // Создание нового контейнера JFrame.
    JFrame jfrm = new JFrame("Spin Dates");
```

```
    // Установка диспетчера компоновки FlowLayout.
    jfrm.getContentPane().setLayout(new FlowLayout());
```

```
    // Установка начальных размеров фрейма.
    jfrm.setSize(300, 120);
```

```
    // Завершение программы при закрытии окна пользователем.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
    // Создание объекта Calendar, который представляет
    // текущую дату. Впоследствии этот объект будет
    // использован для формирования границ диапазона.
    GregorianCalendar g = new GregorianCalendar();
```

```
    // Получение текущей даты.
    Date curDate = new Date();
```

```
    // Установка нижней границы диапазона, равной
    // текущей дате минус один месяц.
```

```
    g.add(Calendar.MONTH, -1);
    Date begin = g.getTime();
```

```
    // Установка верхней границы диапазона, равной
    // текущей дате плюс один месяц.
```

```
    g.add(Calendar.MONTH, 2);
    Date end = g.getTime();
```

```
    // Создание модели инкрементного регулятора, в которой
```

```
// используются подготовленные ранее значения даты.
SpinnerDateModel spm =
    new SpinnerDateModel(curDate, begin, end, Calendar.HOUR);
// Создание инкрементного регулятора, использующего
// подготовленную модель.
jspin = new JSpinner(spm);

// Создание метки для отображения выбранной даты.
jlab = new JLabel(" Selected date is: " + curDate);

// Связывание обработчика событий изменения состояния
// с инкрементным регулятором.
jspin.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent ce) {
        // Получение выбранной даты.
        Date date = (Date) jspin.getValue();

        // Отображение выбранной даты.
        jlab.setText(" Selected date is: " + date + " ");
    }
});

// Включение инкрементного регулятора и метки в состав
// панели содержимого.
jfrm.getContentPane().add(jspin);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new SpinDates();
        }
    });
}
```

В программе для получения текущей даты вызывается конструктор объекта `Date`, заданный по умолчанию. Начальная и конечная даты задаются путем создания объекта `GregorianCalendar` с именем `g`, содержащего текущую дату. Значение этого объекта уменьшается на один месяц, а результат помещается в переменную `begin`. После этого значение `g` увеличивается на два месяца, а результат записывается в переменную `end`. Две полученные даты определяют границы изменения значения инкрементного регулятора. Далее в программе создается класс `SpinnerDateModel`, в котором в качестве начального значения используется текущая дата, а границы задаются переменными `begin` и `end`. Свойство `calendarField` получает значение `Calendar.HOUR`. (Следует помнить, что свойство `calendarField` игнорируется при использовании некоторых стилей.) Таким образом, компонент позволяет изменять дату в пределах месяца вперед и назад от текущей даты. При каждом изменении даты новое значение отображается с помощью метки.

### Вопросы для текущего контроля

1. Инкрементный регулятор аналогичен раскрывающемуся списку с полем редактирования, но имеет одну особенность. Какую?
2. Какую модель следует использовать для перебора числовых значений с помощью инкрементного регулятора?
3. Какая модель позволяет перебирать компоненты списка произвольного типа?

### Спросим у опытного программиста

**Вопрос.** В классе `SpinnerDateModel` методы `setStart()` и `setEnd()` получают параметры типа `Comparable`. Почему не использовать объекты `Date`? То же самое касается класса `SpinnerNumberModel`. Почему не определить для методов `setMinimum()` и `setMaximum()` параметры типа `Number` вместо `Comparable`?

1. В инкрементном регуляторе список значений не раскрывается. Пункты списка поочередно отображаются в поле редактирования.
2. `SpinnerNumberModel`.
3. `SpinnerListModel`.

**Ответ.** Задавая параметры Comparable, вы можете передавать соответствующим методам собственные классы, обеспечивающие сравнение дат или чисел. Заметьте, что интерфейс Comparable реализуется очень просто, так как в нем присутствует только один метод compareTo().

```
int compareTo(T obj)
```

Этот метод сравнивает текущий объект с объектом, передаваемым в качестве параметра. Здесь T — универсальный тип, который можно заменить конкретным типом, используемым в вашей программе.

## Тест для самоконтроля по модулю 5

1. Объект JList реализует средства, позволяющие пользователю выбирать один или несколько пунктов списка. Да или нет?
2. Что надо сделать, чтобы обеспечить прокрутку содержимого объекта JList?
3. Событие какого типа генерирует JList при выборе пользователем пункта списка? Какой метод вызывается для получения индекса, соответствующего первому из выбранных пунктов?
4. Какую модель использует компонент JList?
5. Какой компонент Swing реализует раскрывающийся список?
6. Какой метод позволяет получить выбранный пункт компонента JComboBox?
7. Какой метод надо вызвать, чтобы создать раскрывающийся список, поддерживающий редактирование?
8. Можно ли динамически добавлять пункты в раскрывающийся список в процессе выполнения программы? Если да, то какой метод надо для этого использовать?
9. Можно ли раскрывать и сворачивать раскрывающийся список из программы? Если да, то какой метод надо для этого использовать?
10. Какой класс Swing поддерживает инкрементный регулятор?

11. Какой метод инкрементного регулятора позволяет получить текущее значение? Какой метод надо вызвать, чтобы определить предыдущее значение?
12. Позволяет ли инкрементный регулятор управлять списком строк?
13. Напишите программу, в которой перебирались бы значения типа `double` в диапазоне от 0,0 до 9,9. Величину приращения установите равной 0,1.
14. Включите в программу `DynamicComboBox` компонент `JList`. Каждый раз, когда пользователь удаляет из раскрывающегося списка название сорта яблок, соответствующий пункт должен быть включен в состав `JList`. Таким образом, `JList` будет содержать удаленные записи.



# Модуль 6

## Текстовые компоненты

- 6.1.** Класс `JTextComponent`
- 6.2.** Использование компонента `JTextField`
- 6.3.** Использование компонента `JPasswordField`
- 6.4.** Использование компонента `JFormattedTextField`
- 6.5.** Создание компонента `JFormattedTextField` на основе типа данных
- 6.6.** Создание компонента `JFormattedTextField` на основе сведений о формате
- 6.7.** Поддержка событий изменения свойств компонента `JFormattedTextField`
- 6.8.** Использование компонента `JTextArea`
- 6.9.** Дополнительные возможности компонента `JTextArea`



Этот модуль посвящен компонентам, которые позволяют отображать, вводить и редактировать текст. Поскольку текстовые компоненты — важная часть пользовательского интерфейса, разработчики Swing уделили их поддержке максимум внимания. В модуле 1 вы уже познакомились с одним из текстовых компонентов — полем редактирования, поддерживаемым классом `JTextField`. Это самый простой из текстовых компонентов Swing. Помимо `JTextField`, в наборе Swing предусмотрены также и другие элементы подобного назначения. Ниже приведен список текстовых компонентов, присутствующих в пакете `javax.swing`,

```
JTextField  
JPasswordField  
JFormattedTextField  
JTextArea  
JEditorPane  
JTextPane
```

В данной книге будут рассмотрены первые четыре из них, применяемые достаточно часто. Они позволяют редактировать текст, в котором не используются стили, поэтому работа с ними не составляет труда. Последние два компонента, `JEditorPane` и `JTextPane`, несколько сложнее. Они поддерживают стили, подобные тем, которые используются в HTML- и RTF-документах. В них также могут содержаться изображения и другие компоненты.

Следует заметить, что даже самые простые текстовые компоненты предоставляют обширные возможности, в частности, допускают различные настройки. К счастью, в большинстве случаев их стандартные конфигурации удовлетворяют требованиям большинства приложений. Если же в будущем вам потребуется реализовывать настраиваемые текстовые компоненты, вы должны будете уделить время для изучения их возможностей.

## ВАЖНО!

### **6.1. Класс `JTextComponent`**

В основе всех текстовых компонентов Swing лежит абстрактный класс `JTextComponent`. Он является суперклассом и предоставляет общие функциональные возможности для всех классов, реализующих текстовые компоненты. Класс `JTextComponent` содержится в пакете `javax.swing.text`.

Модель, используемая `JTextComponent` (и, следовательно, всеми текстовыми компонентами Swing), определена посредством интерфейса `Document`, который также принадлежит пакету `javax.swing.text`.

Абстрактный класс, реализующий интерфейс `Document`, называется `AbstractDocument`. Подклассы `PlainDocument` и `DefaultStyledDocument` этого класса позволяют создавать реальные объекты. Класс `DefaultStyledDocument` является также суперклассом для класса `HTMLDocument`. Если с помощью текстовых компонентов Swing решаются относительно простые задачи, потребность в непосредственном взаимодействии с моделью, как правило, не возникает.

В классе `JTextComponent` определено специальное понятие текстового курсора (*caret*). Позиция этого курсора в документе определяет “точку приложения” следующего действия. Вы можете даже определить или установить позицию текстового курсора, изменить его цвет и вид (правда, необходимость в этом возникает крайне редко). Текстовый курсор представляется посредством экземпляра класса, реализующего интерфейс `Caret`. Данный интерфейс находится в пакете `javax.swing.text`. Реализацией текстового курсора по умолчанию является класс `DefaultCaret`.

В классе `JTextComponent` поддерживается также понятие *выделенного текста*. Существуют средства, позволяющие получить фрагмент текста, выбранный пользователем. При необходимости выделение текста можно осуществить из программы.

Класс `JTextComponent` поддерживает возможность вырезания, копирования и вставки текста посредством буфера обмена, которая очень часто требуется пользователям в процессе работы.

Класс `JTextComponent` также предоставляет дополнительные средства, например навигационные фильтры, ограничивающие перемещение текстового курсора, методы ввода данных и поддержку соответствующих им событий, а также связывание и отображение клавиш. Обсуждение этих инструментов не входит в круг задач данной книги, поэтому, если они потребуются в работе, вам придется изучить их самостоятельно.

В табл. 6.1 перечислены некоторые методы, определенные в классе `JTextComponent`. Они доступны для всех текстовых компонентов Swing. Обратите особое внимание на методы, поддерживающие действия с буфером обмена: `copy()`, `cut()` и `paste()`. Эти методы существенно упрощают обмен текстовыми данными с другими приложениями. Заметьте также, что существуют методы, позволяющие извлечь выделенный текст или выбрать его из программы. Например, метод `getSelectedText()` возвращает выделенный фрагмент. Для того чтобы выбрать текст, надо вызвать либо метод `select()`, либо `setCaretPosition()`, сопроводив последний вызов обращением к методу `moveCaretPosition()`. (Второй подход считается предпочтительнее первого; он будет использован в примерах, приведенных в данной книге.)

Таблица 6.1. Методы класса JTextComponent

| Метод                                                                | Описание                                                                                                                                                                                                                                           |
|----------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void addCaretListener(CaretListener cl)</code>                 | Регистрирует обработчик событий текстового курсора                                                                                                                                                                                                 |
| <code>void copy()</code>                                             | Копирует выделенный текст в буфер обмена                                                                                                                                                                                                           |
| <code>void cut()</code>                                              | Копирует выделенный текст в буфер обмена и удаляет его из компонента                                                                                                                                                                               |
| <code>int getCaretPosition()</code>                                  | Возвращает текущую позицию текстового курсора. Позиция определяется как число символов от начала содержимого компонента                                                                                                                            |
| <code>Document getDocument()</code>                                  | Возвращает модель                                                                                                                                                                                                                                  |
| <code>Insets getMargin()</code>                                      | Возвращает объект Insets, содержащий границы компонента                                                                                                                                                                                            |
| <code>String getSelectedText()</code>                                | Возвращает строку, которая содержит выделенный текст                                                                                                                                                                                               |
| <code>int getSelectionEnd()</code>                                   | Возвращает позицию последнего выделенного символа. Позиция определяется как число символов от начала содержимого компонента                                                                                                                        |
| <code>int getSelectionStart()</code>                                 | Возвращает позицию первого выделенного символа. Позиция определяется как число символов от начала содержимого компонента                                                                                                                           |
| <code>String getText()</code>                                        | Возвращает текст, содержащийся в составе компонента                                                                                                                                                                                                |
| <code>String getText(int start, int num)</code>                      | Возвращает строку, которая содержит фрагмент текста длиной num, начинающийся с позиции start                                                                                                                                                       |
| <code>boolean isEditable()</code>                                    | Возвращает значение true, если текст доступен для редактирования. Если компонент находится в режиме только для чтения, возвращается значение false                                                                                                 |
| <code>void moveCaretPosition(int newLoc)</code>                      | Устанавливает текущую позицию текстового курсора равной newLoc. Позиция определяется как число символов от начала содержимого компонента                                                                                                           |
| <code>void paste()</code>                                            | Копирует содержимое буфера обмена (если он не пуст) в состав компонента. Если фрагмент текста в составе компонента выделен, содержимое буфера обмена заменяет его. В противном случае данные включаются в позиции, определяемой текстовым курсором |
| <code>void read(Reader input, Object what) throws IOException</code> | Копирует данные из потока input в состав компонента. Параметр what описывает поток. Его значение может быть равно null                                                                                                                             |

| Метод                                                     | Описание                                                                                                                                                                                                                                                                                                           |
|-----------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void select(int start, int end)</code>              | Выделяет фрагмент текста, заданный значениями <code>start</code> и <code>end</code> . Каждый из этих параметров задает число символов от начала содержимого компонента. Значение <code>start</code> должно быть меньше или равно <code>end</code> , и оба они не должны выходить за пределы содержимого компонента |
| <code>void selectAll()</code>                             | Выделяет весь текст в составе компонента                                                                                                                                                                                                                                                                           |
| <code>void setCaretPosition(int newLoc)</code>            | Устанавливает позицию текстового курсора равной <code>newLoc</code> . Позиция определяется как число символов от начала содержимого компонента                                                                                                                                                                     |
| <code>void setEditable(boolean canEdit)</code>            | Если значение параметра <code>canEdit</code> равно <code>true</code> , текст в составе компонента доступен для редактирования. Значение <code>false</code> указывает на то, что компонент находится в режиме только для чтения                                                                                     |
| <code>void setMargin(Insets margins)</code>               | Устанавливает границы текста в соответствии со значением параметра <code>margins</code>                                                                                                                                                                                                                            |
| <code>void setText(String str)</code>                     | Помещает в состав компонента текст, заданный с помощью параметра <code>str</code>                                                                                                                                                                                                                                  |
| <code>void write(Writer output) throws IOException</code> | Записывает содержимое компонента в указанный поток                                                                                                                                                                                                                                                                 |

При каждом изменении расположения текстового курсора генерируется событие `CaretEvent`. Для обработки таких событий используется класс, реализующий интерфейс `CaretListener`. (Данный интерфейс находится в пакете `javax.swing.event`.) В составе `CaretListener` определен только один метод — `caretUpdate()`. Его заголовок выглядит следующим образом:

```
void caretUpdate(CaretEvent ce)
```

Класс `CaretEvent` предоставляет два приведенных ниже метода.

```
int getDot()
```

```
int getMark()
```

Метод `getDot()` возвращает текущую позицию текстового курсора, а метод `getMark()` — позицию, соответствующую началу выделенного текста. Если текст не выделен, методы `getDot()` и `getMark()` возвращают одинаковые значения.



## Вопросы для текущего контроля

1. Является ли `JTextComponent` суперклассом для всех текстовых компонентов `Swing`?
2. Что означает позиция текстового курсора?
3. Событие какого типа генерируется в случае, если позиция текстового курсора изменяется?

**ВАЖНО!**

## 6.2. Компонент `JTextField`

Наиболее простым из текстовых компонентов является поле редактирования, поддерживаемое классом `JTextField`. Этот компонент, широко используемый в составе графических интерфейсов, позволяет вводить одну строку текста. Вы познакомились с `JTextField` в модуле 1. Здесь мы рассмотрим данный компонент более детально.

В классе `JTextField` определены конструкторы, приведенные в табл. 6.2. Заметьте, что некоторые из конструкторов позволяют задавать ширину компонента в столбцах. Как вы помните, поле редактирования позволяет вводить строку, длина которой превышает его размер, отображаемый на экране, поэтому, если в конструкторе присутствует параметр `cols`, он задает лишь физический размер компонента. По умолчанию `JTextField` использует модель `PlainDocument`.

Как было сказано в модуле 2, если пользователь нажимает клавишу `<Enter>` при вводе информации в текстовом поле, генерируется событие `ActionEvent`. Для его обработки надо использовать класс, реализующий интерфейс `ActionListener`. С компонентом `JTextField` связывается команда действия. По умолчанию она принимается равной текущему содержимому поля редактирования, однако вы можете установить команду явным образом, вызвав метод `setActionCommand()`.

```
void setActionCommand(String cmd)
```

1. Да.
2. Позиция текстового курсора определяет, к какой точке содержимого документа будет применена следующая операция редактирования. Например, при вставке текста он будет помещен в эту позицию.
3. `CaretEvent`.

Таблица 6.2. Конструкторы класса `JTextField`

| Конструктор                                                   | Описание                                                                                                                                                                                                                                        |
|---------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JTextField()</code>                                     | Создает пустое поле редактирования                                                                                                                                                                                                              |
| <code>JTextField(int cols)</code>                             | Создает пустое поле редактирования, ширина которого определяется значением параметра <code>cols</code>                                                                                                                                          |
| <code>JTextField(String str)</code>                           | Создает поле редактирования, которое содержит строку, указанную посредством параметра <code>str</code> . Размеры компонента будут установлены такими, чтобы в нем могла отобразиться заданная строка                                            |
| <code>JTextField(String str, int cols)</code>                 | Создает поле редактирования, которое содержит строку, указанную посредством параметра <code>str</code> . Размеры компонента задаются посредством параметра <code>cols</code>                                                                    |
| <code>JTextField(Document model, String str, int cols)</code> | Создает поле редактирования, которое использует модель, заданную посредством параметра <code>model</code> . Компонент содержит строку, указанную с помощью параметра <code>str</code> . Ширину компонента определяет параметр <code>cols</code> |

Строка, переданная посредством параметра `cmd`, становится новой командой действия. При этом текст в поле редактирования не изменяется. Установленная строка команды действия остается постоянной, независимо от того, какой текст вводится в поле редактирования. Как правило, разработчики прибегают к явной установке команды действия для того, чтобы обеспечить распознавание компонента, сгенерировавшего событие. Так приходится поступать в том случае, когда во фрейме находится несколько управляющих элементов, для которых установлен общий обработчик событий. Если в качестве обработчика выступает неименованный внутренний класс, проблема идентификации источника события не возникает.

Поскольку `JTextField` является подклассом класса `JTextComponent`, вы можете использовать для работы с текстом любые методы, доступные в суперклассе. Рассмотрим несколько примеров. Чтобы получить строку, отображаемую в поле редактирования, надо обратиться к экземпляру `JTextField` и вызвать метод `getText()`. Он имеет следующий вид:

```
String getText()
```

Задать текст для компонента `JTextField` позволяет метод `setText()`, заголовок которого приведен ниже.

```
void setText(String text)
```

Строка передается компоненту посредством параметра `text`.

Получить выделенный фрагмент текста позволяет метод `getSelectedText()`.

```
String getSelectedText()
```

Если текст не был выделен, возвращается значение `null`.

Установить позицию курсора из программы дает возможность метод `setCaretPosition()`.

```
void setCaretPosition(int newLoc)
```

Чтобы выделить фрагмент текста из программы, надо вызвать метод `moveCaretPosition()`.

```
void moveCaretPosition(int newLoc)
```

При этом будет выделен фрагмент между начальным положением текстового курсора и указанной позицией.

Можно также поместить текст в буфер обмена или извлечь его из буфера. Для этой цели предусмотрены методы `cut()`, `copy()` и `paste()`.

```
void cut()
```

```
void copy()
```

```
void paste()
```

Метод `cut()` удаляет выделенный текст и помещает его в буфер обмена. Метод `copy()` также выполняет копирование в буфер, но не удаляет текст из компонента. Метод `paste()` включает содержимое буфера обмена в поле редактирования. Если фрагмент текста в составе компонента выделен, то в результате выполнения метода `paste()` он будет заменен содержимым буфера обмена. В противном случае текст вставляется в позицию, обозначенную текстовым курсором.

Ниже приведен код программы, демонстрирующей использование компонента `JTextField` и вызов некоторых из его методов. В результате работы программы создается поле редактирования, ширина которого позволяет отобразить в нем до пятнадцати символов. После нажатия клавиши `<Enter>` текущее содержимое поля редактирования выводится на экран. Если фрагмент текста был выделен, он также отобразится. В окне программы размещаются две кнопки: `Cut` и `Paste`. Они вызывают стандартные функции вырезания и вставки. Если текст был выбран, по щелчку на кнопке `Cut` он удаляется и помещается в буфер обмена. По щелчку на кнопке `Paste` в поле редактирования вставляется текст из буфера. Очевидно, что, помимо указанных кнопок, вы можете использовать для вырезания и вставки текста стандартные команды `Windows`, связанные с комбинациями клавиш `<Ctrl+X>` и `<Ctrl+V>`. Окно, создаваемое при работе программы, показано на рис. 6.1.

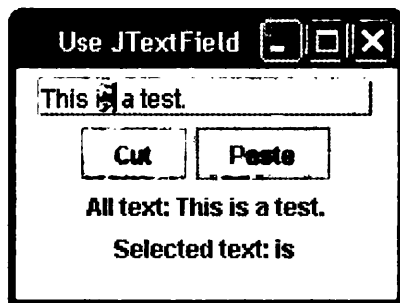


Рис. 6.1. Окно программы *JTextFieldDemo*

// Демонстрация возможностей поля редактирования.

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

class JTextFieldDemo {

    JLabel jlabAll;
    JLabel jlabSelected;

    JTextField jtf;

    JButton jbtnCut;
    JButton jbtnPaste;

    public JTextFieldDemo() {

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Use JTextField");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(200, 150);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```



```
// Создание меток.
jlabAll = new JLabel("All text: ");
jlabSelected = new JLabel("Selected text: ");

// Создание поля редактирования.
jtf = new JTextField("This is a test.", 15);

// Связывание обработчика событий действий
// с полем редактирования.
// Каждый раз, когда пользователь нажимает клавишу <Enter>,
// содержимое компонента отображается на экране.
// Также выводится выделенный текст.
jtf.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        jlabAll.setText("All text: " + jtf.getText());
        jlabSelected.setText("Selected text: " +
                               jtf.getSelectedText());
    }
});

// Создание кнопок Cut и Paste.
jbtnCut = new JButton("Cut");
jbtnPaste = new JButton("Paste");

// Связывание обработчика событий с кнопкой Cut.
jbtnCut.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        // Выделенный текст вырезается и помещается в буфер обмена.
        jtf.cut();
        jlabAll.setText("All text: " + jtf.getText());
        jlabSelected.setText("Selected text: " +
                               jtf.getSelectedText());
    }
});

// Связывание обработчика событий с кнопкой Paste.
jbtnPaste.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        // Текст извлекается из буфера обмена и вставляется
        // в поле редактирования.
        jtf.paste();
    }
});
```

```

// Создание обработчика событий текстового курсора
// и связывание его с полем редактирования.
// Это позволяет приложению реагировать на изменения
// содержимого компонента в реальном времени.
jtf.addCaretListener(new CaretListener() {
    public void caretUpdate(CaretEvent ce) {
        // При каждом изменении позиции текстового курсора
        // отображается содержимое поля редактирования
        // и выделенный текст.
        jlabAll.setText("All text: " + jtf.getText());
        jlabSelected.setText("Selected text: " +
                               jtf.getSelectedText());
    }
});

// Включение компонентов в состав панели содержимого.
jfrm.getContentPane().add(jtf);
jfrm.getContentPane().add(jbtnCut);
jfrm.getContentPane().add(jbtnPaste);
jfrm.getContentPane().add(jlabAll);
jfrm.getContentPane().add(jlabSelected);

// Установка текстового курсора после пятого символа.
jtf.setCaretPosition(5);
// Перемещение текстового курсора в позицию после
// седьмого символа. Эти действия приводят к выделению
// слова "is".
jtf.moveCaretPosition(7);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new JTextFieldsDemo();
        }
    });
}
}

```

Большая часть кода должна быть понятна вам, следует обратить внимание лишь на некоторые детали. Во-первых, при создании поля редактирования `jtf` задается начальный размер, равный 15, и строка "This is a test" в качестве исходного содержимого. Во-вторых, прежде чем окно станет видимым, выбирается несколько символов. Для их выделения используются следующие команды:

```
// Установка текстового курсора после пятого символа.  
jtf.setCaretPosition(5);  
// Перемещение текстового курсора в позицию после  
// седьмого символа. Эти действия приводят к выделению  
// слова "is".  
jtf.moveCaretPosition(7);
```

Как вы уже знаете, установка и перемещение текстового курсора приводит к выбору соответствующего фрагмента текста. Таким образом, сразу после запуска программы будет выделено слово "is".

Каждый раз, когда пользователь завершает действия с полем редактирования, нажимая клавишу <Enter>, генерируется событие `ActionEvent`. В ответ на возникновение этого события вызывается метод `actionPerformed()` обработчика. Для извлечения текста, содержащегося в поле, в теле `actionPerformed()` предусмотрен вызов метода `getText()` объекта `jtf`. Полученный текст отображается посредством метки, на которую ссылается переменная `jlabAll`. Кроме того, с помощью метки `jlabSelected` выводится выделенный текст, который извлекается путем вызова метода `getSelectedText()`.

В данной программе также используется обработчик событий текстового курсора. Каждый раз, когда его позиция изменяется, генерируется событие `CaretEvent` и вызывается метод `caretUpdate()`. Этот метод обновляет две метки, `jlabAll` и `jlabSelected`, в результате чего они отражают текущее содержимое `jtf` и выделенный текст. Все изменения в составе поля редактирования сразу же находят отражение в метках.

И наконец, рассмотрим обработчики событий, связанные с кнопками `Cut` и `Paste`. При активизации кнопки `Cut` вызывается метод `cut()`, который удаляет выделенный текст и помещает его в буфер обмена. По щелчку на кнопке `Paste` вызывается метод `paste()`, который извлекает текст из буфера и вставляет его в поле редактирования. Очевидно, что те же результаты будут получены при использовании стандартных комбинаций клавиш <Ctrl+X> и <Ctrl+V>. Обработчики событий кнопок использованы здесь лишь для того, чтобы продемонстрировать, что те же действия могут быть выполнены из программы.

## Вопросы для текущего контроля

1. Какие методы, унаследованные `TextField` от родительского класса `Component`, позволяют работать с буфером обмена?
2. Какое событие будет сгенерировано, если, работая с полем редактирования, пользователь нажмет клавишу `<Enter>`?
3. Какой метод надо вызвать, чтобы получить текст, выделенный в компоненте `TextField`?

## Спросим у опытного программиста

**Вопрос.** Можно ли изменить шрифт, используемый для отображения текста в составе компонента `TextField`?

**Ответ.** Да. Для изменения шрифта в `TextField` (или любом другом компоненте Swing) служит метод `setFont()`. Получить текущий шрифт позволяет метод `getFont()`. Эти методы определены в классе `JComponent`, и заголовки их имеют следующий вид:

```
void setFont(Font newFont)
```

```
Font getFont()
```

Класс `Font` принадлежит пакету `java.awt`. Подробное обсуждение класса `Font` выходит за рамки данной книги, однако ничто не мешает вам поэкспериментировать с различными шрифтами. Создаются они предельно просто. Ниже представлен один из конструкторов класса `Font`.

```
Font(  
    String fontName, int fontStyle, int pointSize)
```

1. Методы `copy()`, `cut()` и `paste()`.
2. `ActionEvent`.
3. `getSelectedText()`.

Здесь параметр `fontName` задает имя шрифта, например `Dialog` или `Serif`. Параметр `fontStyle` должен иметь одно из приведенных ниже значений.

`Font.PLAIN`

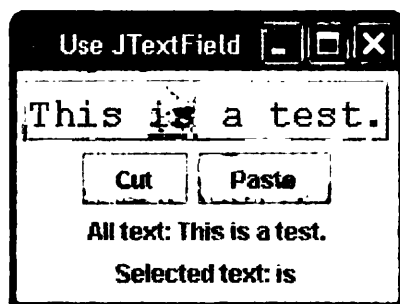
`Font.BOLD`

`Font.ITALIC`

Параметр `pointSize` определяет размер шрифта в пунктах. Чтобы изменить шрифт, включите в программу `JTextFieldDemo` после создания поля редактирования следующую строку кода:

```
jtf.setFont(new Font("monospaced",
    Font.PLAIN, 20));
```

В результате текст в поле будет отображаться моноширинным шрифтом размером 20 пунктов.



**ВАЖНО!**

### 6.3. Компонент `JPasswordField`

Для поддержки поля ввода пароля используется подкласс класса `JTextField` с именем `JPasswordField`. Когда пользователь вводит текст в данном поле, отображения символов в режиме “эхо” не происходит. Вместо этого выводятся символы-заменители; чаще всего в роли такого символа выступает звездочка (\*). В результате вводимый пароль не отображается на экране.

В классе `JPasswordField` определены конструкторы, показанные в табл. 6.3. Подобно `JTextField`, компонент `JPasswordField` генерирует событие действия в случае, если при работе с полем пользователь нажмет клавишу `<Enter>`. Соответствующее событие возникает также при каждом перемещении текстового курсора. Несмотря на то что `JPasswordField` является

подклассом `JTextField`, обращение к методам `cut()` и `copy()` запрещено для повышения уровня безопасности. По тем же соображениям не рекомендован к применению метод `getText()`.

Таблица 6.3. Конструкторы, определенные в классе `JPasswordField`

| Конструктор                                                       | Описание                                                                                                                                                                                                                                      |
|-------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JPasswordField()</code>                                     | Создает пустой компонент <code>JPasswordField</code>                                                                                                                                                                                          |
| <code>JPasswordField(int cols)</code>                             | Создает пустой компонент <code>JPasswordField</code> , ширина которого определяется значением параметра <code>cols</code>                                                                                                                     |
| <code>JPasswordField(String str)</code>                           | Создает компонент <code>JPasswordField</code> , содержащий строку, которая задается посредством параметра <code>str</code> . Размеры компонента устанавливаются такими, чтобы указанная строка могла быть отображена в нем                    |
| <code>JPasswordField(String str, int cols)</code>                 | Создает поле ввода пароля, которое содержит строку, указанную посредством параметра <code>str</code> . Размеры компонента задаются посредством параметра <code>cols</code>                                                                    |
| <code>JPasswordField(Document model, String str, int cols)</code> | Создает поле ввода пароля, которое использует модель, заданную посредством параметра <code>model</code> . Компонент содержит строку, указанную с помощью параметра <code>str</code> . Ширину компонента определяет параметр <code>cols</code> |

Для того чтобы получить пароль, введенный пользователем, надо вызвать метод `getPassword()`.

```
char[] getPassword()
```

Заметьте, что пароль возвращается не в виде строки, а в виде массива символов. Такое решение принято опять же для того, чтобы повысить уровень защиты. Строка не может быть изменена, поэтому, если она содержит пароль, удалить его явным образом невозможно. Пароль будет присутствовать в памяти до тех пор, пока процедура «сборки мусора» не удалит содержащий его объект, а до активизации этой процедуры может пройти достаточно много времени. Таким образом, нельзя полностью исключить вероятность того, что пароль будет получен другим процессом. К счастью, эта проблема разрешима. Поскольку метод `getPassword()` возвращает массив символов, есть возможность после окончания работы с массивом очистить его содержимое, повысив таким образом безопасность программы. Идеальным решением при работе с методом `JPasswordField` является очистка не только массива, хранящего содержимое поле ввода пароля,

но и массива, использованного для сравнения. Очистить массив можно, например, заполнив его нулевыми значениями.

В большинстве случаев при вводе пароля с помощью компонента `JPasswordField` символы отображаются звездочками (\*). Однако при желании вы можете установить другой символ-заменитель, вызвав метод `setEchoChar()`:

```
void setEchoChar(char echr),
```

где параметр `echr` задает требуемый символ. Учтите, что при переключении стилей символ-заменитель может быть изменен.

Ниже приведен код программы, демонстрирующей использование компонента `JPasswordField`. Обратите внимание на то, что после выполнения требуемых действий массив, содержащий пароль, очищается. Окно, создаваемое при работе программы, показано на рис. 6.2.

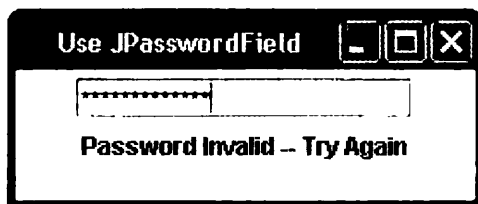


*Рис. 6.2. Данные, отображаемые программой `JPasswordFieldDemo`*

// Пример, демонстрирующий работу с объектом `JPasswordField`

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

class JPasswordFieldDemo {

    JLabel jlabPW;

    JPasswordField jpswd;

    /
    public JPasswordFieldDemo() {

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Use JPasswordField");
```

```
// Установка диспетчера компоновки FlowLayout.
jfrm.getContentPane().setLayout(new FlowLayout());

// Установка начальных размеров фрейма.
jfrm.setSize(240, 100);

// Завершение программы при закрытии окна пользователем.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Создание метки.
jlabPW = new JLabel("Enter Password");

// Создание поля ввода пароля.
jpswd = new JPasswordField(15);

// Связывание с полем ввода пароля обработчика
// событий действия.
// Каждый раз, когда пользователь нажимает
// клавишу <Enter>, содержимое компонента
// проверяется на соответствие паролю.
jpswd.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        char pw[] = { 't', 'e', 's', 't' };

        char [] userSeq = jpswd.getPassword();

        // Проверка строки, введенной пользователем.
        if(Arrays.equals(userSeq, pw))
            jlabPW.setText("Password Valid");
        else
            jlabPW.setText("Password Invalid -- Try Again");

        // После завершения работы очищаются два массива.
        Arrays.fill(pw, (char) 0);
        Arrays.fill(userSeq, (char) 0);
    }
});

// Включение компонентов в состав панели содержимого.
jfrm.getContentPane().add(jpswd);
jfrm.getContentPane().add(jlabPW);
```



```

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new JPasswordFieldDemo();
        }
    });
}
}

```

Обратите внимание на то, что в программе очищается массив `pw` (содержащий строку пароля) и массив `userSeq` (в котором хранится информация, полученная от `jpswd`). Для очистки массивов вызывается метод `Arrays.fill()`. Класс `Arrays` содержится в пакете `java.util` и предоставляет большое количество статических методов для выполнения различных операций с массивами. Метод `fill()` заполняет массив, переданный в качестве первого параметра, значениями, заданными посредством второго параметра.

**ВАЖНО!**

## 6.4. Компонент `JFormattedTextField`

Еще один подкласс `JTextField` позволяет устанавливать формат для работы с текстом. Этот компонент, который появился в составе Swing, начиная с Java 1.4, называется `JFormattedTextField`. Несмотря на то что `JFormattedTextField` предоставляет различные средства, обеспечивающие детальный контроль над процессом редактирования форматированной информации, они используются редко, так как в большинстве случаев конфигурация, установленная по умолчанию, вполне устраивает разработчиков.

Конструкторы, предоставляемые `JFormattedTextField`, приведены в табл. 6.4. Заметьте, что некоторым из конструкторов в качестве параметра передается объект `AbstractFormatter` или `AbstractFormatterFactory`. Экземпляр `AbstractFormatter` используется для форматирования текста. Этот объект предоставляется экземпляром класса `AbstractFormatterFactory`. Однако для сравнительно простых задач, которые в обычных приложениях составляют большинство, необходимость организовывать взаимодействие этих классов не возникает. Поэтому чаще всего используются следующие

два конструктора, в которых действия, необходимые для создания компонента `JFormattedTextField`, скрыты от разработчика:

```
JFormattedTextField(Object contents)
```

```
JFormattedTextField(Format fmt)
```

Данные конструкторы автоматически создают средства форматирования и соответствующий объект-фабрику, обеспечивая совместимость с типом данных, указанных посредством параметра `contents`, или со спецификатором формата, указанным с помощью параметра `fmt`. Рассмотрим эти два конструктора более подробно.

Таблица 6.4. Конструкторы класса *JFormattedTextField*

| Конструктор                                                                                                         | Описание                                                                                                                                                                                                                                                    |
|---------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JFormattedTextField()</code>                                                                                  | Создает пустой компонент <code>JFormattedTextField</code> , не связывая с ним средства форматирования                                                                                                                                                       |
| <code>JFormattedTextField(Object contents)</code>                                                                   | Создает поле редактирования с форматированием, которое отображает данные, указанные с помощью параметра <code>contents</code> . Данный компонент использует средства форматирования, совместимые с типом данных, переданных с помощью <code>contents</code> |
| <code>JFormattedTextField(Format fmt)</code>                                                                        | Создает пустой компонент <code>JFormattedTextField</code> и связывает с ним средства форматирования, совместимые с форматом <code>fmt</code>                                                                                                                |
| <code>JFormattedTextField(JFormattedTextField.<br/>AbstractFormatter absFmt)</code>                                 | Создает пустой компонент <code>JFormattedTextField</code> , который использует средства форматирования, указанные с помощью параметра <code>absFmt</code>                                                                                                   |
| <code>JFormattedTextField(JFormattedTextField.<br/>AbstractFormatterFactory<br/>absFmtFact)</code>                  | Создает пустой компонент <code>JFormattedTextField</code> , который использует фабрику форматирования, указанную с помощью параметра <code>absFmtFact</code>                                                                                                |
| <code>JFormattedTextField(JFormattedTextField.<br/>AbstractFormatterFactory<br/>absFmtFact, Object contents)</code> | Создает поле редактирования с форматированием, которое отображает данные, указанные с помощью параметра <code>contents</code> . Компонент использует объект <code>AbstractFormatterFactory</code> , заданный с помощью параметра <code>absFmtFact</code>    |

**ВАЖНО!**  
**6.5****Создание компонента****JFormattedTextField на основе  
типа данных**

Для простых форматов проще всего создать компонент `JFormattedTextField` с помощью следующего конструктора:

```
JFormattedTextField(Object contents)
```

Он автоматически формирует объект `AbstractFormatterFactory` на основе типа данных, на которые ссылается параметр `contents`. Например, с помощью приведенного ниже выражения создается компонент `JFormattedTextField`, поддерживающий целочисленные значения.

```
JFormattedTextField jtf = new JFormattedTextField(10);
```

Поскольку в данном случае параметр `contents` ссылается на целое число, выбирается целочисленный формат. Несмотря на то что данный подход очень удобен, он не обеспечивает достаточного уровня контроля над создаваемым объектом. В некоторых случаях приходится указывать формат. Соответствующий конструктор рассматривается в следующем разделе.

**ВАЖНО!**  
**6.6****Создание компонента****JFormattedTextField на основе  
сведений о формате**

Задать формат для компонента `JFormattedTextField` позволяет конструктор

```
JFormattedTextField(Format fmt)
```

В данном случае формат, указанный посредством параметра `fmt`, автоматически используется для создания объекта `AbstractFormatter`, который затем помещается в `AbstractFormatterFactory`. `Format` — это абстрактный класс, в котором предусмотрены основные средства для указания формата. Подклассами `Format` являются два важных класса: `DateFormat` и `NumberFormat` (они также абстрактные). Оба эти класса предоставляют фабричные методы для создания различных форматов, позволяющих работать с датой и временем.

В данном модуле рассматривается пример, в котором используются два фабричных метода.

```
static final NumberFormat getCurrencyInstance()  
static final DateFormat getDateInstance(int dateStyle)
```

Метод `getCurrencyInstance()`, предоставляемый классом `NumberFormat`, возвращает формат, поддерживающий представление валюты для региона по умолчанию. В нем предусмотрены два десятичных знака, знак валюты и разделитель трехзначных групп. Метод `getDateInstance()`, предоставляемый классом `DateFormat`, возвращает формат, поддерживающий дату. Допустимо одно из следующих значений параметра `dateStyle`:

|                               |                                |
|-------------------------------|--------------------------------|
| <code>DateFormat.SHORT</code> | <code>DateFormat.MEDIUM</code> |
| <code>DateFormat.LONG</code>  | <code>DateFormat.FULL</code>   |

Как следует из имен форматов, они задают различные представления даты. Сокращенная запись (`DateFormat.SHORT`) выглядит так: 9/12/09. Для остальных форматов записи имеют большую длину. Формат `DateFormat.MEDIUM` используется в примере, рассматриваемом в данном разделе. Он генерирует формат даты наподобие следующего: Sep 12, 2009.

При необходимости можно без труда создать другой тип формата, называемый *форматом маски*. Маска позволяет задавать в точности то представление, которое вам необходимо. Формат маски поддерживается классом `MaskFormatter`, который принадлежит пакету `javax.swing.text`. Этот класс является подклассом класса `JFormattedTextField.AbstractFormatter`. Таким образом, для использования `MaskFormatter` надо указать следующий конструктор `JFormattedTextField`:

```
JFormattedTextField(JFormattedTextField.AbstractFormatter absFmt)
```

В классе `MaskFormatter` предусмотрены два конструктора. Один из них — конструктор по умолчанию. Другой выглядит следующим образом (он будет использован в данном разделе):

```
MaskFormatter(String fmtMask)
```

где параметр `fmtMask` — строка, определяющая формат, которому должны соответствовать данные в поле редактирования. Эта строка включает литеральные символы (их можно пропустить) и символы, отмечающие позиции.

Предположим, например, что в программе имеется следующая строка:

```
new MaskFormatter("ID: AA-LL-UU");
```

| Символ | Допустимые данные                                                      |
|--------|------------------------------------------------------------------------|
| A      | Буквы и цифры                                                          |
| h      | Шестнадцатеричные цифры                                                |
| L      | Буквы (они будут преобразованы в нижний регистр)                       |
| U      | Буквы (они будут преобразованы в верхний регистр)                      |
| #      | Цифры                                                                  |
| *      | Все символы                                                            |
| ?      | Все буквы                                                              |
| `      | Код, применяемый для literalного представления символов форматирования |

Данное выражение создает объект `MaskFormatter`, который допускает ввод трех пар символов. В первой паре могут присутствовать любые буквы. Вторая пара также допускает ввод произвольных букв, но они будут преобразованы в нижний регистр. И наконец, третья пара, как и первые две, допускает любые буквы, но они преобразуются в верхний регистр. Первые четыре символа "ID: " — literalные, и редактировать их запрещено. Если при создании `MaskFormatter` вы укажете недопустимый формат, то будет сгенерировано исключение `ParseException`.

**ВАЖНО!**

## 6.7. Обработка событий изменения свойств компонента `JFormattedTextField`

Подобно своему суперклассу `JTextField`, `JFormattedTextField` генерирует события действия в том случае, когда пользователь завершает ввод данных нажатием клавиши `<Enter>`. Однако иногда приходится предусматривать обработку событий изменения свойств. По этой причине желательно более подробно рассмотреть работу компонента `JFormattedTextField`.

Начнем с основной характеристики `JFormattedTextField` — поддержки значения. Оно соответствует данным, отображаемым в поле, но не зависит от них. Когда пользователь вводит текст посредством данного компонента, текст преобразуется в значение. Если значение установлено из программы, средства форматирования преобразуют его в строку, которая выводится посредством компонента. Таким образом, значение поля редактирования с поддержкой формата отделено от отображаемого текста. У вас может возникнуть вполне зако-

номерный вопрос: когда и при каких условиях будет обновлено значение, т.е. когда оно будет соответствовать введенному тексту?

В конфигурации по умолчанию измененный текст в поле не будет преобразован в значение до тех пор, пока пользователь не нажмет клавишу <Enter> либо не передаст фокус ввода другому компоненту. Когда происходит одно из этих действий, внесенные изменения “принимаются во внимание” и значение компонента обновляется в соответствии с новыми данными. (Очевидно, что это произойдет только в том случае, если информация соответствует формату, определенному для данного компонента.) Таким образом, в течение некоторого времени текст, отображаемый в поле, и реальное значение компонента могут не совпадать. Если вы перейдете к другому интерфейсному элементу, изменения будут учтены в значении компонента, независимо от того, нажимали ли вы клавишу <Enter>. Чтобы постоянно отслеживать изменения форматированного текста, надо вместе события действия обрабатывать события изменения свойств. Такое событие будет сгенерировано при каждом изменении текста в поле.

События изменения свойств описываются экземплярами класса `PropertyChangeEvent`, принадлежащего пакету `java.beans`. Для того чтобы обрабатывать события изменения свойств, необходимо реализовать интерфейс `PropertyChangeListener`. В нем объявлен только метод `propertyChange()`, заголовок которого имеет следующий вид:

```
void propertyChange(PropertyChangeEvent pe)
```

В классе `PropertyChangeEvent` определены методы, которые позволяют получить имя измененного свойства, а также его старое и новое значение.

Задать обработчик событий изменения состояния можно с помощью одного из двух приведенных ниже методов.

```
void addPropertyChangeListener(PropertyChangeEvent pl)
```

```
void addPropertyChangeListener(  
    String propName, PropertyChangeEvent pl)
```

Первый из этих методов задает обработку событий для всех свойств. Второй позволяет указать с помощью параметра `propName` требуемое свойство. В компоненте `JFormattedTextField` свойство, соответствующее значению, имеет имя `value`.

Установить значение для форматированного текста можно из программы. Для этой цели используется метод `setValue()`. Получить текущее значение позволяет метод `getValue()`. Заголовки этих методов имеют следующий вид:

```
void setValue(Object val)
```

```
Object getValue()
```

где параметр `val` определяет новое значение поля редактирования с форматированием. Установка значения также приводит к изменению текста в поле редактирования. Очевидно, что значение должно соответствовать формату, заданному для поля.

## Программа, демонстрирующая работу компонента `JFormattedTextField`

Ниже приведен код программы, демонстрирующей использование компонента `JFormattedTextField`. В ней создаются три поля для работы с форматированным текстом; для всех их заданы различные форматы. В первом поле, предназначенном для редактирования идентификационного номера сотрудника, используется формат маски. Идентификационный номер включает код отдела из двух цифр и номер сотрудника, состоящий из трех цифр. Две части кода разделяются дефисом, например 12-576. Во втором поле, предназначенном для редактирования сведений о заработной плате сотрудника, используется числовой формат. Третье поле предназначено для ввода или редактирования даты приема сотрудника на работу. В нем используется формат даты. Внешний вид окна, создаваемого при работе программы, показан на рис. 6.3.

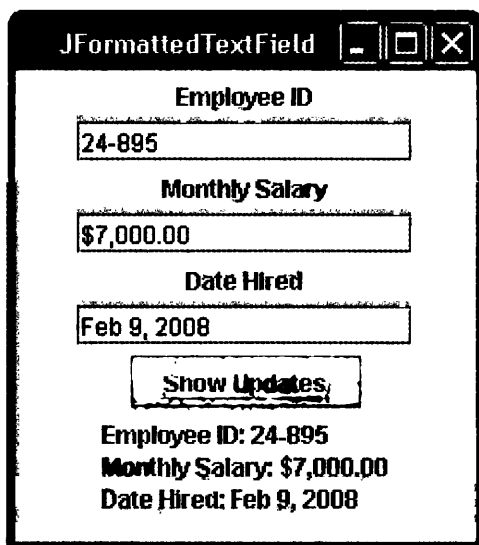


Рис. 6.3. Данные, отображаемые программой `FormattedTFDemo`

// Использование поля редактирования с поддержкой форматирования.

```
import java.beans.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;
import javax.swing.text.*;
import java.util.*;

class FormattedTFDemo {

    NumberFormat cf;
    DateFormat df;

    JLabel jlab;

    JFormattedTextField jftfSalary;
    JFormattedTextField jftfDate;
    JFormattedTextField jftfEmpID;

    JButton jbtnShow;

    public FormattedTFDemo() {

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("JFormattedTextField");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(240, 270);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание метки.
        jlab = new JLabel();

        // Создание поля редактирования форматированного текста,
        // применяемого для ввода идентификационных номеров.
        // Данный компонент использует формат маски.
```



```
try {
    MaskFormatter mf = new MaskFormatter("##-###");
    jftfEmpID = new JFormattedTextField(mf);
} catch (ParseException exc) {
    System.out.println("Invalid Format");
    return;
}
jftfEmpID.setColumns(15);
jftfEmpID.setValue("24-895");

// Создание поля редактирования форматированного текста,
// применяемого для работы с финансовой информацией.
// Данный компонент использует формат денежных единиц.
cf = NumberFormat.getCurrencyInstance();
cf.setMaximumIntegerDigits(5);
cf.setMaximumFractionDigits(2);
jftfSalary = new JFormattedTextField(cf);
jftfSalary.setColumns(15);
jftfSalary.setValue(new Integer(7000));

// Создание поля редактирования форматированного текста,
// применяемого для ввода даты.
df = DateFormat.getDateInstance(DateFormat.MEDIUM);
jftfDate = new JFormattedTextField(df);
jftfDate.setColumns(15);
jftfDate.setValue(new Date()); // Инициализация текущей датой.

// Связывание обработчика изменения свойств с полем
// ввода идентификационного номера.
jftfEmpID.addPropertyChangeListener("value",
    new PropertyChangeListener() {
        // Метод получает управление при изменении значения
        // компонента.
        public void propertyChange(PropertyChangeEvent pe) {
            jlab.setText("Employee ID changed.");
        }
    });

// Связывание обработчика изменения свойств с полем
// для работы с информацией о зарплате.
jftfSalary.addPropertyChangeListener("value",
    new PropertyChangeListener() {
        // Метод получает управление при изменении значения
```

```
// компонента.
public void propertyChange(PropertyChangeEvent pe) {
    jlab.setText("Monthly salary changed.");
}
});

// Связывание обработчика изменения свойств с полем
// для работы с датами.
jftfDate.addPropertyChangeListener("value",
    new PropertyChangeListener() {
        // Метод получает управление при изменении значения
        // компонента.
        public void propertyChange(PropertyChangeEvent pe) {
            jlab.setText("Date hired changed.");
        }
    });

// Создание кнопки Show Updates.
jbtnShow = new JButton("Show Updates");

// Связывание обработчика событий с кнопкой Show Updates.
jbtnShow.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        // Отображение форматированных данных. Заметьте, что
        // информация о заработной плате и дате форматируется
        // с помощью тех же объектов, которые используются
        // в соответствующих полях. Идентификационный номер
        // сотрудника уже сформатирован.
        jlab.setText("<html>Employee ID: " +
            jftfEmpID.getValue() +
            "<br>Monthly Salary: " +
            cf.format(jftfSalary.getValue()) +
            "<br>Date Hired: " +
            df.format(jftfDate.getValue()));
    }
});

// Включение компонентов в состав панели содержимого.
jfrm.getContentPane().add(new JLabel("Employee ID"));
jfrm.getContentPane().add(jftfEmpID);
jfrm.getContentPane().add(new JLabel("Monthly Salary"));
jfrm.getContentPane().add(jftfSalary);
jfrm.getContentPane().add(new JLabel("Date Hired"));
```

```
jfrm.getContentPane().add(jftfDate);
jfrm.getContentPane().add(jbtnShow);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new FormattedTFDemo();
        }
    });
}
```

Данная программа демонстрирует некоторые важные особенности использования компонента `JFormattedTextField`. После уже знакомых вам действий, обычно выполняемых в начале программы, создаются три текстовых поля с форматированием. В первом из них используется объект `MaskFormatter`, который позволяет вводить идентификационный код сотрудника. Строка формата имеет вид `##-###`. Следовательно, номер 12-465 соответствует этой строке, а АВ-934 — нет, поскольку использование букв запрещено. Второе поле редактирования использует числовой формат для ввода сведений о заработной плате. В третьем поле применен формат представления даты средней длины. Заметьте, что все три поля инициализированы корректными данными. Это важно. Если данные, заданные при инициализации, будут некорректны, могут возникнуть проблемы на этапе выполнения программы.

С каждым из полей связывается обработчик событий изменения свойств. Как вы уже знаете, такой обработчик получает управление при изменении свойства связанного с ним компонента. Обработчики лишь информируют пользователя об изменениях. В реальном приложении можно предусмотреть более содержательные действия, выполняемые в ответ на возникновение событий данного типа. И наконец, еще один обработчик связывается с кнопкой **Show Updates**. По щелчку на кнопке из трех полей редактирования, поддерживающих формат, извлекается содержимое и отображается на экране. Обратите внимание на то, что заработная плата и дата приема на работу отображаются в том же формате, который использован в полях редактирования. В результате данные в окне отображаются так же, как и в полях редактирования.

## Спросим у опытного программиста

**Вопрос.** Вы сказали, что поведение компонента `JFormattedTextField` может изменяться при потере фокуса ввода. Как это происходит?

**Ответ.** По умолчанию, когда компонент `JFormattedTextField` теряет фокус ввода, изменения, внесенные пользователем, принимаются и значение компонента обновляется. Однако это происходит только в том случае, если значение корректно. При вводе недопустимых данных восстанавливается предыдущее значение компонента. В большинстве случаев такое поведение компонента устраивает разработчика. Однако при желании вы можете изменить его, вызвав метод `setFocusLostBehavior()`.

```
void setFocusLostBehavior(int what)
```

Здесь параметр `what` должен иметь одно из приведенных ниже значений, определенных в классе `JFormattedTextField`.

```
COMMIT    REVERT    COMMIT_OR_REVERT    PERSIST
```

По умолчанию принимается значение `COMMIT_OR_REVERT`. Установка политики потери фокуса ввода, равной `COMMIT`, предполагает ввод корректных значений. Недопустимое значение сохраняется в поле редактирования, но значение компонента не изменяется. Значение `REVERT` задает извлечение и отображение текущего значения. Значение `PERSIST` задает использование текущих отредактированных данных, но значение изменяется.

Следует также заметить следующее: независимо от того, какая из политик потери фокуса ввода используется, нажатие клавиши `<Enter>` при работе с компонентом приводит к установке нового значения, которое соответствует данным, введенным в поле (если эти данные допустимы).

Перед тем как продолжить работу с материалом данного модуля, желательно немного поэкспериментировать с компонентом. Попробуйте, например, задать другую маску или другой числовой формат. Измените поведение компонента при потере фокуса ввода. Компонент `JFormattedTextField` предоставляет обширные возможности, поэтому имеет смысл потратить время и усилия для ознакомления с ними.

**ВАЖНО!****6.5. Компонент JTextArea**

Как вы уже знаете, `JTextField`, а также его подклассы `JPasswordField` и `JFormattedTextField` ориентированы на работу с одной строкой текста. С их помощью можно, например, ввести имя файла или URL. Если же необходимо отредактировать несколько строк, следует воспользоваться другим текстовым компонентом Swing. Во многих случаях для этой цели подходит компонент `JTextArea`, реализующий текстовую область.

Этот компонент позволяет пользователям вводить и редактировать несколько строк текста. Он прост в использовании, но позволяет работать только с неформатированными последовательностями символов. Этот компонент не поддерживает стили; по умолчанию в нем используется модель `PlainDocument`. Следовательно, данный компонент ориентирован на обработку обычного текста. Так, например, вы можете применять его тогда, когда вам надо предоставить пользователю возможность дополнить данные, введенные посредством формы, своими замечаниями. `JTextArea` поддерживает стандартные команды редактирования; конкретный набор команд зависит от используемого стиля.

Несмотря на то что компонент `JTextArea` позволяет редактировать текст, состоящий из нескольких строк, он не поддерживает прокрутку. Если прокрутка необходима, следует поместить `JTextArea` в состав `JScrollPane`. Именно в таком виде компонент `JTextArea` используется в большинстве приложений.

Конструкторы, предоставляемые классом `JTextArea`, перечислены в табл. 6.5. Если вы не собираетесь помещать текстовую область в состав панели с прокруткой, вам придется указать при вызове конструктора число строк и столбцов. Если компонент `JTextArea` включается в состав `JScrollPane`, размеры текстовой области устанавливаются в соответствии с предпочтительными размерами панели. При работе с компонентом `JTextArea` необходимо учитывать одну особенность: если он используется без панели с прокруткой, то его размеры будут увеличиваться с тем, чтобы в нем помещался весь введенный текст. В результате возникнут проблемы с компоновкой. Это еще один аргумент в пользу использования панели с прокруткой. (Можно, конечно, задать предпочтительные размеры текстовой области, но все же применение панели с прокруткой — более удачное решение.)

В отличие от `JTextField`, `JTextArea` не генерирует события действия. Набор событий ограничивается теми, которые были унаследованы от `JTextComponent`. В частности, при работе с данным компонентом возникают события, связанные с текстовым курсором. Он также генерирует события документа; это происходит при изменении модели. Для большинства приложений достаточно обработчика событий текстового курсора. Более того, работая

с текстовой областью, можно вовсе отказаться от обработки событий. Приложению достаточно извлекать текст по мере необходимости. Например, если вы включили в форму текстовую область с тем, чтобы пользователь вводил в нее свои замечания при покупке, содержимое компонента можно извлекать в обработчике событий, связанном с кнопкой Send (эту кнопку пользователь активирует, чтобы передать заполненную форму).

Подобно остальным текстовым компонентам, `JTextArea` является подклассом `JTextComponent`. Это означает, что при работе с данным компонентом можно использовать все методы, определенные в `JTextComponent`. Как и при работе с компонентом `JTextField`, вы можете извлечь содержимое текстовой области с помощью метода `getText()`. Выделенный текст доступен посредством метода `getSelectedText()`. Можно также выделять текст из программы, вызывая сначала метод `setCaretPosition()`, а затем `moveCaretPosition()`. Действия с буфером обмена можно выполнять, вызывая методы `cut()`, `copy()` и `paste()`.

Таблица 6.5. Конструкторы класса `JTextArea`

| Конструктор                                                  | Описание                                                                                                                                                                                                 |
|--------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JTextArea()</code>                                     | Создает пустую текстовую область с нулевыми размерами                                                                                                                                                    |
| <code>JTextArea(String str)</code>                           | Создает текстовую область, которая инициализируется строкой <code>str</code>                                                                                                                             |
| <code>JTextArea(int numRows, int numCols)</code>             | Создает пустую текстовую область, размеры которой (число строк и число столбцов) задаются соответственно параметрами <code>numRows</code> и <code>numCols</code>                                         |
| <code>JTextArea(String str, int numRows, int numCols)</code> | Создает текстовую область, которая инициализируется строкой <code>str</code> . Ее размеры (число строк и число столбцов) задаются соответственно параметрами <code>numRows</code> и <code>numCols</code> |
| <code>JTextArea(Document model)</code>                       | Создает текстовую область, которая использует модель, заданную посредством параметра <code>model</code>                                                                                                  |

При вводе текста пользователем длина строки может быть больше ширины текстовой области. Если вы поместите компонент `JTextArea` в панель с прокруткой, при появлении слишком длинных строк будет автоматически отображаться горизонтальная полоса прокрутки. Однако пользоваться таким компонентом не очень удобно. В некоторых случаях предпочтительнее, если при достижении текстом правой границы компонента он будет переводиться на следующую строку. Такой режим можно установить, вызвав метод `setLineWrap()`.

```
void setLineWrap(boolean wrapOn)
```

Если значение параметра `wrapOn` равно `true`, текст, превышающий ширину области, автоматически переносится на новую строку. По умолчанию принимается значение данного параметра, равное `false`, т.е. автоматический перенос не производится. Определить текущий режим переноса позволяет метод `getLineWrap()`. Если автоматический перенос установлен, то этот метод возвращает значение `true`.

Существуют два способа перехода на следующую строку: может осуществляться перенос отдельных символов или целого слова. По умолчанию происходит перенос символов. Перевести компонент в режим переноса по границе слова можно, вызвав метод `setWrapStyleWord()`.

```
void setWrapStyleWord(boolean breakOnWord)
```

Если значение параметра `breakOnWord` равно `true`, длинные строки разрываются по границе слова. Перед тем как перейти к детальному обсуждению возможностей компонента `JTextArea`, желательно рассмотреть простую программу, демонстрирующую использование компонента `JTextArea`. В программе создается небольшая текстовая область, которая помещается в состав панели с прокруткой. Каждый раз, когда возникает событие текстового курсора, подсчитывается количество слов, содержащихся в текстовой области, и полученное значение отображается на экране. Обратите внимание, что для реализации такого достаточно сложного поведения требуется очень малый объем кода. Внешний вид окна, создаваемого при работе программы, показан на рис. 6.4.

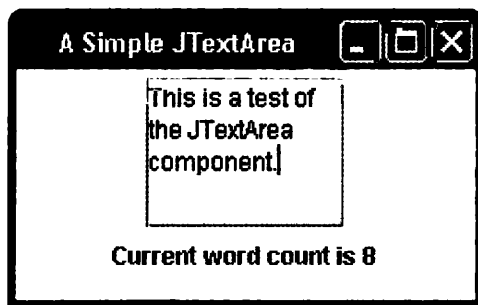


Рис. 6.4. Окно, отображаемое программой `SimpleTextAreaDemo`

```
// Пример использования компонента JTextArea.
```

```
import java.io.*;
```

```
import java.awt.*;
```

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

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

class SimpleTextAreaDemo {

    JLabel jlabWC;

    JTextArea jta;

    public SimpleTextAreaDemo() {

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("A Simple JTextArea");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(240, 150);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание метки для отображения счетчика слов.
        jlabWC = new JLabel("Current word count is 0");

        // Создание поля редактирования.
        jta = new JTextArea();

        // Установка режима переноса по границе слова.
        jta.setLineWrap(true);
        jta.setWrapStyleWord(true);

        // Включение текстовой области в панель с прокруткой.
        JScrollPane jscrlp = new JScrollPane(jta);
        jscrlp.setPreferredSize(new Dimension(100, 75));

        // Связывание обработчика событий текстового курсора
        // с компонентом. Данный обработчик отображает
        // счетчик слов.
        jta.addCaretListener(new CaretListener() {
            // При каждом перемещении текстового курсора
            // отображается счетчик слов.
        });
    }
}
```



```

public void caretUpdate(CaretEvent ce) {
    int wc; // Хранение числа слов.

    // Получение текущего текста.
    String str = jta.getText();

    if(str.length() == 0)
        wc = 0; // Если текст отсутствует, число слов равно нулю.
    else {
        // Разделение строки на отдельные слова. Разделителями
        // являются символы, которые не могут присутствовать
        // в слове, например пробелы или знаки пунктуации.
        String [] strsplit = str.split("\\W+");

        // Счетчик слов равен числу строк, которые
        // возвращает метод split().
        wc = strsplit.length;

        // Учет ведущих символов-разделителей.
        if(strsplit.length > 0 &&
            strsplit[0].length() == 0) wc--;
    }

    // Отображение счетчика слов.
    jlabWC.setText("Current word count is " + wc);
}

// Включение компонентов в состав панели содержимого.
jfrm.getContentPane().add(jscrlp);
jfrm.getContentPane().add(jlabWC);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new SimpleTextAreaDemo();
        }
    });
}
}

```

Рассмотрим код программы более подробно. Прежде всего обратите внимание на то, как создается и подготавливается к использованию текстовая область `jta`. Эту задачу выполняет следующий фрагмент кода:

```
// Создание текстовой области.  
jta = new JTextArea();  
  
// Установка режима переноса по границе слова.  
jta.setLineWrap(true);  
jta.setWrapStyleWord(true);
```

Поскольку для многих приложений предпочтительнее перенос по границе слова, команды, подобные приведенным выше, часто можно встретить в реальных программах. Конечно, есть приложения, в которых перенос строки не нужен вовсе: ни по границе слова, ни по границе символа. Однако если компонент `JTextArea` используется для ввода комментариев, то желательно, чтобы слово, не помещающееся в область, переносилось на новую строку; при этом интерфейс выглядит более профессионально.

Следующие строки кода помещают компонент `jta` в состав панели с прокруткой и задают предпочтительные размеры панели:

```
// Включение текстовой области в панель с прокруткой.  
JScrollPane jscrlp = new JScrollPane(jta);  
jscrlp.setPreferredSize(new Dimension(100, 75));
```

Установить размеры панели необходимо, так как они определяют размеры текстовой области. Следует также помнить, что, если предпочтительные размеры панели установлены, размеры (число строк и столбцов), указанные при создании текстовой области, не принимаются во внимание.

Затем в программе регистрируется обработчик событий текстового курсора. Эти события обрабатываются так же, как и для компонента `JTextField`: каждый раз, когда возникает событие текстового курсора, вызывается метод `caretUpdate()`. В данной программе обработчик извлекает строку из текстовой области путем вызова метода `getText()`. После этого производится подсчет слов, для чего используется метод `split()` класса `String`. Данный метод разделяет строку на подстроки. Роль разделителя выполняет последовательность символов, соответствующая регулярному выражению. В данном случае указано выражение `\W+`. Оно означает любую последовательность символов, которые не являются частью слова. Таким образом, в результате выполнения метода `split()` текст будет преобразован в массив слов, который и возвращается при завершении метода. Длина массива равна числу слов в составе текста.

Это значение отображается в составе метки. Поскольку рассматриваемое здесь событие возникает при каждом изменении позиции текстового курсора, счетчик слов обновляется в реальном времени, по мере ввода текста.

**ВАЖНО!**

## **6.9. Дополнительные возможности JTextArea**

В дополнение к функциям, унаследованным от `JTextComponent`, компонент `JTextArea` предоставляет разработчику ряд дополнительных возможностей. Например, по умолчанию табуляторы расположены в составе `JTextArea` через восемь позиций, однако, вызвав метод `setTabSize()`, вы можете установить другое значение.

```
void setTabSize(int newSize)
```

Здесь параметр `newSize` задает новое расстояние между табуляторами.

В составе `JTextArea` есть несколько методов, позволяющих добавлять текст из программы. Эти методы приведены ниже.

```
void append(String str)
```

```
void insert(String str, int idx)
```

```
void replaceRange(String str, int begin, int end)
```

Параметр `str` содержит текст, который надо добавить к содержимому компонента. Метод `append()` добавляет строку в конец текста. Метод `insert()` включает ее в позицию, указанную посредством параметра `idx` (этот параметр задает смещение от начала текста). Заменить подстроку указанным текстом позволяет метод `replaceRange()`. В результате его выполнения заменяются символы в диапазоне от `begin` до `end - 1`. Параметры `begin` и `end` представляют собой смещение относительно начала текста.

Для подсчета строк во фрагменте текста надо вызвать метод `getLineCount()`.

```
int getLineCount()
```

При этом необходимо помнить, что учитываются лишь строки, оканчивающиеся символом перевода строки. Таким образом, "строки", сформированные путем автоматического переноса, не будут приняты во внимание.

## Вопросы для текущего контроля

1. Чем компонент `JTextArea` отличается от компонента `JTextField`?
2. Обеспечивает ли компонент `JTextArea` автоматическую прокрутку текста?
3. Какой метод надо вызвать, чтобы перевести компонент `JTextArea` в режим переноса по границе слова?

## Проект 6.1. Создание простого текстового редактора

`SimpleTextEditor.java`

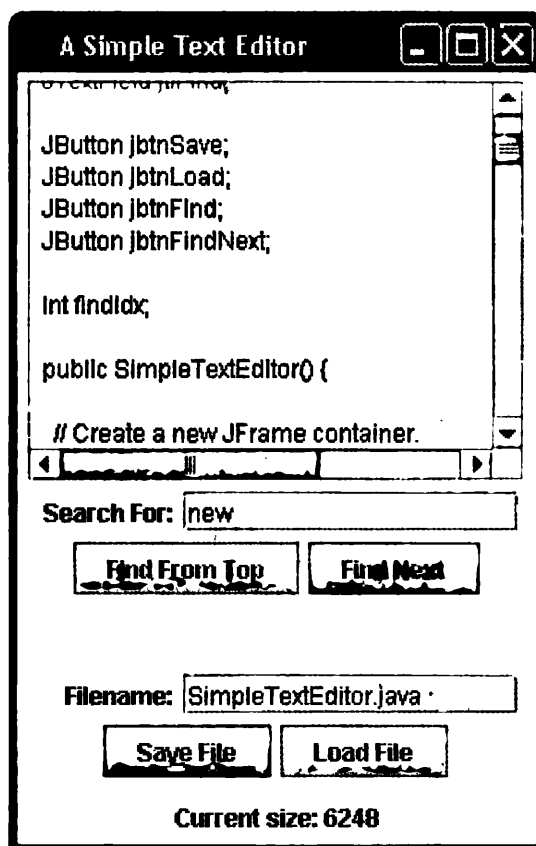
Данный проект демонстрирует возможности текстовых компонентов Swing на примере простого текстового редактора. С его помощью вы можете загружать файлы или сохранять их на диске. Редактор также поддерживает функции поиска строк. Для редактирования файла используется компонент `JTextArea`, следовательно, он автоматически поддерживает основные команды, например вырезание и вставку текста. Для ввода имени файла используется поле редактирования. Такой же компонент предусмотрен и для ввода строки поиска. Обратите внимание, что для создания полнофункционального текстового редактора потребовался относительно небольшой объем кода. Окно программы показано ниже.

Поскольку в редакторе используется компонент `JTextArea`, он может работать только с неформатированным текстом, например, с его помощью можно написать исходный код программы или изменить конфигурационный файл. Однако и такой простой редактор может быть полезен. Его можно использовать как заголовок для более сложной программы, реализующей, например, записную книжку. Рекомендации по модернизации редактора вы найдете в вопросах 13–15 в конце данной главы.

1. Компонент `JTextField` позволяет вводить только одну строку текста. В отличие от него `JTextArea` поддерживает редактирование текста, состоящего из нескольких строк.
2. Нет. Для автоматической прокрутки надо поместить данный компонент в состав панели с прокруткой.
3. `setWrapStyleWord(true)`.

Проект  
6.1

Создание простого текстового редактора



## Последовательность действий

1. Создайте файл `SimpleTextEditor.java` и включите в него следующие комментарии и выражения `import`:

// Проект 6.1. Создание простого текстового редактора.

```

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

```

2. Начните код класса `SimpleTextEditor` следующим образом:

```

class SimpleTextEditor {

```

```

    JLabel jlabMsg;

```

```
JTextArea jta;  
  
JTextField jtffName;  
JTextField jtffFind;  
  
JButton jbtnSave;  
JButton jbtnLoad;  
JButton jbtnFind;  
JButton jbtnFindNext;  
  
int findIdx;
```

Как видите, в этом фрагменте кода объявлены переменные экземпляра. Метка `jlabMsg` используется для отображения сообщений об операциях, выполненных редактором, например, она выводит значение счетчика символов или результаты поиска. На текстовую область, посредством которой осуществляется редактирование, указывает переменная `jta`. В поле редактирования `jtffName` вводится имя файла. В поле `jtffFind` задается строка для поиска. Кнопки `jbtnSave` и `jbtnLoad` управляют соответственно сохранением и загрузкой файла. Кнопка `jbtnFind` запускает процедуру поиска, а кнопка `jbtnFindNext` предназначена для продолжения поиска. Переменная `findIdx` содержит индекс позиции в файле, с которой должен начинаться поиск.

**3. Начните код конструктора `SimpleTextEditor()` следующим образом:**

```
public SimpleTextEditor() {  
  
    // Создание нового контейнера JFrame.  
    JFrame jfrm = new JFrame("A Simple Text Editor");  
  
    // Установка диспетчера компоновки FlowLayout.  
    jfrm.getContentPane().setLayout(new FlowLayout());  
  
    // Установка начальных размеров фрейма.  
    jfrm.setSize(270, 420);  
  
    // Завершение программы при закрытии окна пользователем.  
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
}
```

**4. Создайте метки для отображения данных в окне. Для них следует указать предпочтительные размеры и тип выравнивания. Это упрощает компоновку элементов.**

```
// Создание метки для отображения сообщений.
jlabMsg = new JLabel();
jlabMsg.setPreferredSize(new Dimension(200, 30));
jlabMsg.setHorizontalAlignment(SwingConstants.CENTER);

// Создание метки без текста для выделения пустого
// пространства.
JLabel jlabSeparator = new JLabel();
jlabSeparator.setPreferredSize(new Dimension(200, 30));

// Создание меток Search For и Filename.
JLabel jlabFind = new JLabel("Search For:");
jlabFind.setPreferredSize(new Dimension(70, 20));
jlabFind.setHorizontalAlignment(SwingConstants.RIGHT);

JLabel jlabFilename = new JLabel("Filename:");
jlabFilename.setPreferredSize(new Dimension(70, 20));
jlabFilename.setHorizontalAlignment(SwingConstants.RIGHT);
```

5. Создайте компонент `JTextArea`, с помощью которого будет осуществляться редактирование.

```
// Создание текстовой области.
jta = new JTextArea();

// Включение текстовой области
// в состав панели с прокруткой.
JScrollPane jscrlp = new JScrollPane(jta);
jscrlp.setPreferredSize(new Dimension(250, 200));
```

Как и в большинстве приложений, использующих текстовые области, `jta` помещается в состав панели с прокруткой. Таким образом, обеспечивается автоматическое отображение полос прокрутки, позволяющих работать с файлами большого размера. Заметьте также, что выполняется перенос на новую строку. При редактировании исходных текстов программ автоматический перенос не упростит, а скорее усложнит работу. Если вы считаете иначе, включите автоматический перенос и сравните два режима работы.

6. Создайте поле редактирования для ввода имени файла.

```
// Поле редактирования для ввода имени файла.
jtffName = new JTextField(15);
```

## 7. Создайте обработчик событий текстового курсора.

```
// Связывание обработчика событий текстового курсора
// с текстовой областью. Обработчик отображает
// число символов в файле и обновляет эти данные при
// каждом изменении положения текстового курсора.
// В переменную findIdx записывается текущая позиция
// текстового курсора.
jta.addCaretListener(new CaretListener() {
    public void caretUpdate(CaretEvent ce) {
        String str = jta.getText();
        jlabMsg.setText("Current size: " + str.length());
        findIdx = jta.getCaretPosition();
    }
});
```

При каждом перемещении текстового курсора обработчик будет отображать текущий размер файла. Он также записывает текущую позицию текстового курсора в переменную `findIdx`. Таким образом, при активизации кнопки **Find Next** поиск будет производиться, начиная с текущей позиции.

## 8. Создайте кнопки **Save File** и **Load File** и свяжите с ними обработчики событий. За сохранение и загрузку файлов отвечают методы `save()` и `load()`.

```
// Создание кнопок Save File и Load File.
jbtnSave = new JButton("Save File");
jbtnLoad = new JButton("Load File");

// Связывание обработчика событий с кнопкой Save File.
jbtnSave.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        save();
    }
});

// Связывание обработчика событий с кнопкой Load File.
jbtnLoad.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        load();
    }
});
```



9. Создайте поле редактирования, в котором будет задаваться строка для поиска. Создайте также кнопки Find From Top и Find Next и свяжите с ними обработчики событий.

```
// Создание поля редактирования Search For.
jtfFind = new JTextField(15);

// Создание кнопок Find From Top и Find Next.
jbtnFind = new JButton("Find From Top");
jbtnFindNext = new JButton("Find Next");

// Связывание обработчика событий с кнопкой Find From Top.
jbtnFind.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        findIdx = 0;
        find(findIdx);
    }
});

// Связывание обработчика событий с кнопкой Find Next.
jbtnFindNext.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        find(findIdx+1);
    }
});
```

Для осуществления реального поиска текста, заданного в поле Search For, обработчики событий, связанных с кнопками Find From Top и Find Next, вызывают метод `find()`. При активизации кнопки Find From Top переменная `findIdx` устанавливается равной 0 и передается методу `find()`. В результате поиск производится с начала файла. Повторный поиск начинается с позиции, на единицу превышающей текущую. Метод `find()` будет описан несколько позже.

10. Включите компоненты в панель содержимого и завершите код конструктора. Поскольку в панель содержимого помещается много компонентов, ссылка на нее сохраняется в переменной `cp`. В результате уменьшается, как объем кода, так и количество вызовов метода `getContentPane()`.

```
// Добавление компонентов к панели содержимого.
Container cp = jfrm.getContentPane();
cp.add(jscrlp);
cp.add(jlabFind);
cp.add(jtfFind);
```

```
cp.add(jbtnFind);
cp.add(jbtnFindNext);
cp.add(jlabSeparator);
cp.add(jlabFilename);
cp.add(jtffName);
cp.add(jbtnSave);
cp.add(jbtnLoad);
cp.add(jlabMsg);

// Отображение фрейма.
jfrm.setVisible(true);
}
```

11. Код метода `save()` приведен ниже. Этот метод предназначен для сохранения файла.

```
// Метод, предназначенный для сохранения файла.
void save() {
    FileWriter fw;

    // Извлечение имени файла из поля редактирования.
    String fname = jtffName.getText();

    // Проверка наличия имени.
    if(fname.length() == 0) {
        jlabMsg.setText("No filename present.");
        return;
    }

    // Сохранение файла.
    try {
        fw = new FileWriter(fname);
        jta.write(fw);
        fw.close();
    } catch(IOException exc) {
        jlabMsg.setText("Error opening or writing file.");
        return;
    }

    jlabMsg.setText("File written successfully.");
}
```

Код метода `save()` очень прост. Сначала из поля редактирования `jtffName` извлекается имя файла. Затем выполняется проверка, содержатся ли в имени файла символы (в некоторых случаях пользователи забывают ввести имя файла). Далее файл открывается, и с помощью метода `write()` в него записывается содержимое текстовой области `jta`. Метод `write()` определен в классе `JTextComponent`; он автоматически записывает в указанный поток данные, находящиеся в составе компонента.

**12. Добавьте метод `load()`, код которого показан ниже.**

```
// Метод, предназначенный для загрузки файла.
void load() {
    FileReader fw;

    // Извлечение имени файла из поля редактирования.
    String fname = jtffName.getText();

    // Проверка наличия имени файла.
    if(fname.length() == 0) {
        jlabMsg.setText("No filename present.");
        return;
    }

    // Загрузка файла.
    try {
        fw = new FileReader(fname);
        jta.read(fw, null);
        fw.close();
    } catch(IOException exc) {
        jlabMsg.setText("Error opening or reading file.");
        return;
    }

    // При загрузке нового файла текущая позиция обнуляется.
    findIdx = 0;

    jlabMsg.setText("File loaded successfully.");
}
```

Данный метод похож на метод `save()`, но он загружает файл, имя которого указано в поле редактирования `jtffName`. В теле метода файл открывается для чтения, а затем вызывается метод `read()`, определенный в классе `JTextComponent`. Этот метод читает данные из указанного по-

тока в текстовую область, заменяя ими текущее содержимое компонента. Заметьте, что в переменную `findIdx` записывается значение 0. Поскольку в поле редактирования были загружены новые данные из файла, старое значение текущей позиции не имеет смысла.

13. Создайте метод `find()`, который будет осуществлять поиск строки, заданной в поле редактирования `jtFind`.

```
// Поиск текста.
void find(int start) {
    // Получение текущего текста в виде строки.
    String str = jta.getText();

    // Извлечение строки поиска.
    String findStr = jtFind.getText();

    // Поиск первого вхождения указанной строки.
    int idx = str.indexOf(findStr, start);

    // Проверка, найдено ли соответствие.
    if(idx > -1) {

        // Если поиск завершен успешно, текстовый
        // курсор помещается в позицию, соответствующую
        // найденному тексту.
        jta.setCaretPosition(idx);
        findIdx = idx;
        jlabMsg.setText("String found.");
    }
    else
        jlabMsg.setText("String not found.");

    // Передача фокуса ввода окну редактора.
    jta.requestFocusInWindow();
}
```

В первую очередь из текстовой области `jta` извлекается строка, представляющая содержимое данного компонента, и сохраняется в переменной `str`. Ключевые слова для поиска извлекаются из поля редактирования `jtFind` и записываются в переменную `findStr`. Поиск осуществляет метод `indexOf()`, определенный в классе `String`. Этот метод ищет в текущей строке (в данном случае `str`) последовательность символов, которая соответствовала бы строке, заданной в качестве первого параметра (в данном примере это переменная `findStr`). В качестве второго

параметра методу передается индекс, определяющий позицию, с которой должен начаться поиск. Если метод возвращает значение, равное  $-1$ , это означает, что поиск завершился неудачно. В противном случае возвращается индекс начала подстроки, соответствующей строке поиска. В данной программе этот индекс сохраняется в локальной переменной `idx`.

В случае успешного поиска курсор текстовой области устанавливается в позицию, соответствующую началу найденной последовательности символов. В переменную `findIdx` записывается значение, содержащееся в `idx`. Сообщение о том, успешно ли завершился поиск, отображается с помощью метки. Перед завершением метод `find()` передает фокус ввода компоненту `jta` (т.е. текстовой области). Для этого вызывается метод `requestFocusInWindow()`. Данный метод впервые определен в классе `Component` и переопределен в классе `JComponent`. Он передает фокус ввода вызывающему компоненту. Передать фокус ввода текстовой области необходимо, в противном случае не будет видно расположение найденного текста.

14. Завершите код класса методом `main()`.

```
public static void main(String args[]) {  
    // Фрейм создается в потоке обработки событий.  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            new SimpleTextEditor();  
        }  
    });  
}
```

15. Полностью код программы `SimpleTextEditor` выглядит следующим образом:

```
// Проект 6.1. Создание простого текстового редактора.
```

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

```
class SimpleTextEditor {
```

```
    JLabel jlabMsg;
```

```
JTextArea jta;

JTextField jtffName;
JTextField jtffFind;

JButton jbtnSave;
JButton jbtnLoad;
JButton jbtnFind;
JButton jbtnFindNext;

int findIdx;

public SimpleTextEditor() {

    // Создание нового контейнера JFrame.
    JFrame jfrm = new JFrame("A Simple Text Editor");

    // Установка диспетчера компоновки FlowLayout.
    jfrm.getContentPane().setLayout(new FlowLayout());

    // Установка начальных размеров фрейма.
    jfrm.setSize(270, 420);

    // Завершение программы при закрытии окна пользователем.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Создание метки для отображения сообщений.
    jlabMsg = new JLabel();
    jlabMsg.setPreferredSize(new Dimension(200, 30));
    jlabMsg.setHorizontalAlignment(SwingConstants.CENTER);

    // Создание метки без текста для выделения пустого
    // пространства.
    JLabel jlabSeparator = new JLabel();
    jlabSeparator.setPreferredSize(new Dimension(200, 30));

    // Создание меток Search For и Filename.
    JLabel jlabFind = new JLabel("Search For:");
    jlabFind.setPreferredSize(new Dimension(70, 20));
    jlabFind.setHorizontalAlignment(SwingConstants.RIGHT);
```

```
JLabel jlabFilename = new JLabel("Filename:");
jlabFilename.setPreferredSize(new Dimension(70, 20));
jlabFilename.setHorizontalAlignment(SwingConstants.
RIGHT);

// Создание текстовой области.
jta = new JTextArea();

// Включение текстовой области
// в состав панели с прокруткой.
JScrollPane jscrlp = new JScrollPane(jta);
jscrlp.setPreferredSize(new Dimension(250, 200));

// Поле редактирования для ввода имени файла.
jtfFName = new JTextField(15);

// Связывание обработчика событий текстового курсора
// с текстовой областью. Обработчик отображает
// число символов в файле и обновляет эти данные при
// каждом изменении положения текстового курсора.
// В переменную findIdx записывается текущая позиция
// текстового курсора.
jta.addCaretListener(new CaretListener() {
    public void caretUpdate(CaretEvent ce) {
        String str = jta.getText();
        jlabMsg.setText("Current size: " + str.length());
        findIdx = jta.getCaretPosition();
    }
});

// Создание кнопок Save File и Load File.
jbtnSave = new JButton("Save File");
jbtnLoad = new JButton("Load File");

// Связывание обработчика событий с кнопкой Save File.
jbtnSave.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        save();
    }
});
```

```
    }  
    });  
  
    // Связывание обработчика событий с кнопкой Load File.  
    jbtnLoad.addActionListener(new ActionListener() {  
        public void actionPerformed(ActionEvent le) {  
            load();  
        }  
    });  
  
    // Создание поля редактирования Search For.  
    jtfFind = new JTextField(15);  
  
    // Создание кнопок Find From Top и Find Next.  
    jbtnFind = new JButton("Find From Top");  
    jbtnFindNext = new JButton("Find Next");  
  
    // Связывание обработчика событий с кнопкой Find From Top.  
    jbtnFind.addActionListener(new ActionListener() {  
        public void actionPerformed(ActionEvent le) {  
            findIdx = 0;  
            find(findIdx);  
        }  
    });  
  
    // Связывание обработчика событий с кнопкой Find Next.  
    jbtnFindNext.addActionListener(new ActionListener() {  
        public void actionPerformed(ActionEvent le) {  
            find(findIdx+1);  
        }  
    });  
  
    // Добавление компонентов к панели содержимого.  
    Container cp = jfrm.getContentPane();  
    cp.add(jscrlp);  
    cp.add(jlabFind);  
    cp.add(jtfFind);  
    cp.add(jbtnFind);  
    cp.add(jbtnFindNext);  
    cp.add(jlabSeparator);  
    cp.add(jlabFilename);
```



```
cp.add(jtffName);
cp.add(jbtnSave);
cp.add(jbtnLoad);
cp.add(jlabMsg);

// Отображение фрейма.
jfrm.setVisible(true);
}

// Метод, предназначенный для сохранения файла.
void save() {
    FileWriter fw;

    // Извлечение имени файла из поля редактирования.
    String fname = jtffName.getText();

    // Проверка наличия имени.
    if(fname.length() == 0) {
        jlabMsg.setText("No filename present.");
        return;
    }

    // Сохранение файла.
    try {
        fw = new FileWriter(fname);
        jta.write(fw);
        fw.close();
    } catch(IOException exc) {
        jlabMsg.setText("Error opening or writing file.");
        return;
    }

    jlabMsg.setText("File written successfully.");
}

// Метод, предназначенный для загрузки файла.
void load() {
    FileReader fw;

    // Извлечение имени файла из поля редактирования.
    String fname = jtffName.getText();

    // Проверка наличия имени файла.
```

```
if(fname.length() == 0) {
    jlabMsg.setText("No filename present.");
    return;
}

// Загрузка файла.
try {
    fw = new FileReader(fname);
    jta.read(fw, null);
    fw.close();
} catch(IOException exc) {
    jlabMsg.setText("Error opening or reading file.");
    return;
}

// При загрузке нового файла текущая позиция обнуляется.
findIdx = 0;

jlabMsg.setText("File loaded successfully.");
}

// Поиск текста.
void find(int start) {
    // Получение текущего текста в виде строки.
    String str = jta.getText();

    // Извлечение строки поиска.
    String findStr = jtfFind.getText();

    // Поиск первого вхождения указанной строки.
    int idx = str.indexOf(findStr, start);

    // Проверяется, найдено ли соответствие.
    if(idx > -1) {
        // Если поиск завершен успешно, текстовый
        // курсор помещается в позицию, соответствующую
        // найденному тексту.
        jta.setCaretPosition(idx);
        findIdx = idx;
        jlabMsg.setText("String found.");
    }
    else
        jlabMsg.setText("String not found.");
}
```

```
// Передача фокуса ввода окну редактора.
jta.requestFocusInWindow();
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new SimpleTextEditor();
        }
    });
}
}
```



## **Тест для самоконтроля по модулю 6**

1. Что такое `JTextComponent`? Можно ли создать экземпляр класса `JTextComponent`?
2. Назовите шесть текстовых компонентов, определенных в составе Swing.
3. Какое событие генерируется при перемещении текстового курсора компонента `JTextField`? Какое событие будет сгенерировано, если, работая с компонентом `JTextField`, пользователь нажмет клавишу `<Enter>`?
4. Какой компонент следует использовать для ввода пароля?
5. В чем состоит главное преимущество компонента `JFormattedTextField` по сравнению с `JTextField`?
6. Какую маску надо использовать для определения формата ввода телефонных номеров, которые имеют вид подобный следующему:  
1 (555) 555-5555?
7. Назовите четыре политики потери фокуса ввода компонента `JFormattedTextField` и объясните их назначение.
8. События какого типа генерируются при изменении значения поля редактирования с поддержкой формата?
9. Компонент `JTextArea` обычно включается в состав `JScrollPane`. Да или нет?
10. Какой метод надо вызвать, чтобы перевести текстовую область в режим автоматического перехода на новую строку?

11. Какой метод надо вызвать, чтобы изменить в текстовой области размер табуляторов?
12. Можно ли вырезать и вставлять текст в текстовых компонентах? Можно ли делать это из программы? Если да, то какие методы следует использовать?
13. Добавьте в программу `SimpleTextEditor`, созданную в рамках проекта 6.1, возможность замены текста. Замене подлежит текст, заданный в поле редактирования `Search For`. Добавьте еще одно поле редактирования, в которой будет вводиться новая строка. Создайте также кнопку `Replace`, по каждому щелчку на которой будет заменяться одно вхождение исходной строки.
14. Измените код программы `SimpleTextEditor` так, чтобы вместо счетчика символов отображался счетчик строк.
15. В программе `SimpleTextEditor` задайте комбинации клавиш для активизации кнопок `Find From Top` и `Find Next`. Для `Find From Top` это должна быть комбинация `<Alt+F>`, а для `Find Next` — `<Alt+N>`.
16. Реализуйте в программе `SimpleTextEditor` те возможности, которые вы считаете уместными. Например, вы можете добавить флажок опции, отменяющий учет регистра символов при поиске.



# Модуль 7

## Меню

- 7.1.** Общие сведения о меню
- 7.2.** Классы `JMenuBar`, `JMenu` и `JMenuItem`
- 7.3.** Создание главного меню
- 7.4.** Определение мнемонических обозначений и клавиш быстрого доступа для пунктов меню
- 7.5.** Связывание изображений и строк подсказки с пунктами меню
- 7.6.** Использование объектов `JRadioButtonMenuItem` и `JCheckBoxMenuItem`
- 7.7.** Создание контекстного меню
- 7.8.** Создание панели инструментов
- 7.9.** Использование действий
- 7.10.** Окончательный вариант программы, использующей меню и панель инструментов

Данная глава посвящена одному из самых важных элементов современных графических интерфейсов — меню. Меню является неотъемлемой частью почти каждого приложения, поскольку оно предоставляет пользователю доступ к функциональным возможностям программного продукта. Без меню обходятся разве что самые простые программы. Учитывая важность данного компонента, Swing обеспечивает расширенные средства его поддержки. Именно при работе с меню становится понятно, насколько мощной является библиотека Swing.

Средства поддержки меню, имеющиеся в составе Swing, обеспечивают работу следующих компонентов.

- Строка меню, которая обычно является главным меню приложения.
- Стандартные меню, содержащие пункты (в том числе те, посредством которых вызываются подчиненные меню).
- Контекстные меню, обычно активизируемые щелчком правой кнопкой мыши.
- Панели инструментов, предоставляющие быстрый доступ к функциональным возможностям программ. Часто элементы панели инструментов дублируют пункты меню.
- Действия, позволяющие управлять двумя или несколькими различными компонентами посредством одного объекта. Несмотря на то что действия используются при решении различных задач, чаще всего с их помощью организуется управление меню и панелями инструментов.

Меню Swing также поддерживают клавиши быстрого доступа, позволяющие выбрать требуемые пункты, не раскрывая меню и подменю.

**ВАЖНО!**

## **7.1. Общие сведения о меню**

Для поддержки системы меню в библиотеке Swing предусмотрены классы, перечисленные в табл. 7.1. На первый взгляд может показаться, что разобраться с ними сложно, но это впечатление ошибочное. Средства создания меню Swing очень просты в использовании. Swing допускает настройку каждого компонента, однако в большинстве случаев конфигурация, принятая по умолчанию, вполне удовлетворяет требованиям приложений. Так, например, стандартные средства позволяют без труда связывать с пунктами меню изображения или клавиши быстрого доступа.

Таблица 7.1. Классы Swing, используемые для создания меню

| Класс                | Описание                                                            |
|----------------------|---------------------------------------------------------------------|
| JMenuBar             | Объект, реализующий меню верхнего уровня приложения                 |
| JMenu                | Стандартное меню. Содержит один или несколько объектов JMenuItem    |
| JMenuItem            | Объект, реализующий пункт меню                                      |
| JCheckBoxMenuItem    | Пункт меню в виде флажка опции                                      |
| JRadioButtonMenuItem | Пункт меню в виде кнопки переключателя опций                        |
| JSeparator           | Разделитель между пунктами меню                                     |
| JPopupMenu           | Контекстное меню. Обычно активизируется щелчком правой кнопкой мыши |

В данном разделе мы вкратце рассмотрим классы, используемые при создании меню. Для формирования главного, или основного, меню приложения надо создать объект `JMenuBar`. Этот класс можно рассматривать как своеобразный контейнер для меню. С экземпляром `JMenuBar` можно связать объекты `JMenu`. Каждый такой объект определяет отдельное меню, содержащее один или несколько пунктов. Пункты, отображаемые в составе `JMenu`, поддерживаются с помощью объектов `JMenuItem`, допускающих активизацию пользователем.

Помимо меню, связанных с главным меню, вы также можете создавать контекстные меню. Для этого нужно сформировать объект `JPopupMenu`, а затем связать с ним объекты `JMenuItem`. Как правило, контекстное меню активизируется щелчком правой кнопкой мыши; при этом курсор мыши должен находиться на компоненте, для которого данное меню было создано.

Помимо стандартных пунктов в состав меню можно включать флажки и переключатели опций. Для создания пункта меню, представляющего собой флажок опции, применяется объект `JCheckBoxMenuItem`. Пункты меню в виде кнопок переключателя опций можно сформировать с помощью объектов `JRadioButtonMenuItem`. Оба этих класса являются подклассами `JMenuItem`. Их можно использовать как в стандартных, так и в контекстных меню.

С помощью объекта `JToolBar` можно создать панель инструментов — независимый компонент, который, подобно меню, позволяет обращаться к функциям приложения. Как правило, элементы панели инструментов дублируют пункты меню и используются для ускоренного доступа к соответствующим программным средствам. Например, в текстовом процессоре с помощью панели инструментов часто организуют быстрый доступ к средствам форматирования.

`JSeparator` — это вспомогательный класс, используемый для формирования разделителей в составе меню.



Одной из важных особенностей меню Swing является тот факт, что все классы, предназначенные для поддержки пунктов меню, являются подклассами класса `AbstractButton`. Как вы, наверное, помните, `AbstractButton` — это суперкласс для всех компонентов, реализующих кнопки, например `JButton`. Таким образом, любой пункт меню представляет собой кнопку. Внешний вид пунктов меню отличается от привычных вам кнопок, но их поведение во многом совпадает с поведением кнопок. Например, при выборе пункта меню, как и по щелчку на кнопке, генерируется событие действия.

Еще одна особенность состоит в том, что `JMenuItem` является суперклассом `JMenu`. Это позволяет создавать подменю — меню в составе другого меню. Чтобы сформировать подменю, надо сначала создать и заполнить пунктами объект `JMenu`, после чего его можно связать с другим объектом `JMenu`. Соответствующий пример будет рассмотрен в следующем разделе.

В системе меню Swing также определены два важных интерфейса: `SingleSelectionModel` и `MenuElement`. Интерфейс `SingleSelectionModel` определяет действия компонента, содержащего несколько элементов, но позволяющего в каждый момент времени выбрать только один из них. Эта модель используется объектами `JMenuBar` и `JPopupMenu`. Реализацией данного интерфейса по умолчанию является класс `DefaultSingleSelectionModel`. Интерфейс `MenuElement` определяет поведение пункта меню. Он реализуется всеми классами, используемыми для поддержки меню, за исключением `JSeparator`. Как правило, необходимость обращаться к методам, определенным в этих интерфейсах, не возникает. Исключения составляют задачи, связанные с настройкой системы меню.

Как было сказано ранее, при выборе пункта меню генерируется событие действия. По умолчанию строка команды действия совпадает с именем выбранного пункта. Таким образом, идентифицировать выбранный пункт можно путем анализа команды действия. При необходимости вы также можете определить для каждого пункта меню обработчик события действия, выполненный в виде неименованного внутреннего класса. Однако следует иметь в виду, что во многих приложениях система меню включает большое число пунктов. Создание отдельного класса-обработчика для каждого пункта приведет к увеличению объема кода.

Меню также могут генерировать и другие типы событий. При каждом выборе или отмене выбора пункта меню генерируется событие `MenuEvent`. Для обработки таких событий используются объекты `MenuListener`. Если меню реагирует на действия с клавиатурой, то возникает событие `MenuKeyEvent`, для которого предусмотрен интерфейс `MenuKeyListener`. Перетаскивание мыши при работе с меню приводит к возникновению событий `MenuDragMouseEvent`, обработчиками для которых являются объекты `MenuDragMouseListener`.

При отображении и сокрытии контекстного меню генерируется событие `PopupMenuEvent`. Соответствующий обработчик должен реализовывать интерфейс `PopupMenuListener`. В большинстве случаев необходимость обработки указанных выше событий не возникает. Чаще всего приложение должно реагировать лишь на события действия. Однако, если вам необходим более детальный контроль за работой программы, в вашем распоряжении есть и другие события.



### Вопросы для текущего контроля...

1. Какой класс используется для создания строки меню верхнего уровня?
2. С помощью какого класса создается пункт меню?
3. Может ли пункт меню вести себя как флажок или переключатель опций?

**ВАЖНО!**

## 7.2. Классы `JMenuBar`, `JMenu` и `JMenuItem`

Для того чтобы создать меню, необходимо иметь хотя бы общее представление о трех классах: `JMenuBar`, `JMenu` и `JMenuItem`. Эти три класса представляют собой минимальный набор средств, используемых для создания главного меню приложения. Классы `JMenu` и `JMenuItem` также используются для реализации контекстных меню. Таким образом, указанные выше классы составляют основу системы меню.

### Класс `JMenuBar`

Как было сказано ранее, объект `JMenuBar` представляет собой своеобразный контейнер для меню. Подобно другим компонентам, он является подклассом класса `JComponent` (который, в свою очередь, является потомком `Container` и `Component`). Для данного класса предусмотрен только конструктор по умолчанию. Таким образом, вновь созданная строка меню пуста, и перед использованием ее надо заполнить конкретными пунктами. В приложении может содержаться только одна строка меню.

1. `JMenuBar`.
2. `JMenuItem`.
3. Да.

В классе `JMenuBar` определено несколько методов, но в большинстве случаев можно ограничиться использованием только одного из них — метода `add()`. Этот метод связывает со строкой меню объект `JMenu`. Он имеет следующий вид:

```
JMenu add(JMenu menu)
```

где параметр `menu` представляет собой экземпляр класса `JMenu`, связываемый со строкой меню. В результате выполнения метод возвращает ссылку на меню. Конкретные пункты располагаются в строке меню слева направо в том порядке, в котором они были включены в состав данного компонента. Если вы хотите поместить очередной пункт в заданную позицию, вам надо использовать приведенный ниже вариант метода `add()`, унаследованный от класса `Container`.

```
Component add(Component menu, int idx)
```

Здесь параметр `menu` определяет меню, а параметр `idx` задает индекс соответствующего пункта. Отсчет индексов начинается с нуля; нулевое значение индекса соответствует первому пункту слева.

В некоторых случаях приходится удалять ненужные более меню. Для этой цели используется метод `remove()`, унаследованный от класса `Container`. Существуют две разновидности данного метода:

```
void remove(Component menu)
```

```
void remove(int idx)
```

Параметр `menu` представляет собой ссылку на меню, подлежащее удалению, а `idx` задает индекс удаляемого меню. Отсчет индексов начинается с нуля.

В некоторых случаях может оказаться полезным метод `getMenuCount()`.

```
int getMenuCount()
```

Он возвращает число элементов, содержащихся в составе строки меню.

В классе `JMenuBar` имеются и другие методы, которые можно использовать при создании приложений. Например, получить массив ссылок на меню, связанные со строкой меню, позволяет метод `getSubElements()`. Определить, какое меню выбрано в данный момент, можно, вызвав метод `isSelected()`.

После создания и заполнения строки меню ее надо включить в состав `JFrame`. Для этой цели следует вызвать метод `setJMenuBar()` данного объекта (строка меню включается в состав контейнера верхнего уровня, а не в панель содержимого). Заголовок метода `setJMenuBar()` имеет следующий вид:

```
void setJMenuBar(JMenuBar mb)
```

где параметр `mb` представляет собой ссылку на строку меню. Расположение строки меню определяется используемым стилем. Как правило, она отображается в верхней части окна.

## Класс `JMenu`

Объект `JMenu` инкапсулирует меню, которое заполняется пунктами, реализованными с помощью объектов `JMenuItem`. Как уже упоминалось выше, класс `JMenu` является дочерним по отношению к классу `JMenuItem`. Это означает, что объект `JMenu` может быть связан с другим объектом такого же типа, т.е. можно создавать подменю в составе меню. В классе `JMenu` определены конструкторы, показанные в табл. 7.2. Независимо от конструктора, вновь созданное меню пусто, и его надо заполнить пунктами.

Таблица 7.2. Конструкторы класса `JMenu`

| Конструктор                                      | Описание                                                                                                                                               |
|--------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JMenu()</code>                             | Создает меню, не присваивая ему имени                                                                                                                  |
| <code>JMenu(String name)</code>                  | Создает меню с именем, заданным посредством параметра <code>name</code>                                                                                |
| <code>JMenu(String name, boolean tearOff)</code> | Создает меню с именем, заданным посредством параметра <code>name</code> . На момент написания данной книги параметр <code>tearOff</code> игнорировался |
| <code>JMenu(Action action)</code>                | Создает меню, управляемое действием <code>action</code>                                                                                                |

В классе `JMenu` определено большое количество методов. В данном разделе будут вкратце описаны некоторые из них, используемые наиболее часто. Для включения пункта в состав меню используется метод `add()`. Существует несколько вариантов этого метода, в том числе два, приведенных ниже.

```
JMenuItem add(JMenuItem item)
```

```
JMenuItem add(Component item, int idx)
```

Здесь `item` — это пункт, включаемый в состав меню. Первый из двух приведенных выше вариантов метода `add()` помещает пункт в конце меню. Второй вариант метода размещает пункт в позиции, указанной с помощью параметра `idx`. Как и ранее, отсчет индексов начинается с нуля. Оба метода возвращают пункт, включенный в состав меню. Заполнить меню пунктами позволяет также метод `insert()`.

Для организации пунктов в составе меню можно использовать разделители (экземпляры класса `JSeparator`). Для включения разделителя надо вызвать метод `addSeparator()`.

```
void addSeparator()
```

Разделитель помещается в конец меню. Для того чтобы вставить его в требуемую позицию, надо использовать метод `insertSeparator()`.

```
void insertSeparator(int idx)
```

Здесь параметр `idx` задает индекс позиции, в которой должен находиться разделитель. Отсчет индексов начинается с нуля.

Удалить пункт из меню можно с помощью метода `remove()`.

```
void remove(JMenuItem menu)
```

```
void remove(int idx)
```

Параметр `menu` представляет собой ссылку на пункт, подлежащий удалению, а `idx` задает индекс удаляемого пункта меню.

Определить число пунктов в составе меню позволяет метод `getMenuComponentCount()`.

```
int getMenuComponentCount()
```

Метод `getMenuComponents()` возвращает массив пунктов, содержащихся в составе меню. Заголовок этого метода выглядит следующим образом:

```
Component[] getMenuComponents()
```

Он возвращает массив объектов типа `Component`.

### **Спросим у опытного программиста**

**Вопрос.** Я знаю, что в классе `JMenu` определены методы `addSeparator()` и `insertSeparator()`, включающие разделители в состав меню. Зачем нужен класс `JSeparator`?

**Ответ.** Объект `JSeparator` может быть помещен в ту позицию, в которой должен отображаться разделитель. Следует, однако, помнить, что внешний вид разделителя определяется используемым стилем.

## Класс JMenuItem

Объект JMenuItem инкапсулирует пункт меню. С пунктом меню может быть связано либо конкретное действие, как, например, с пунктами Save и Close, либо подменю. Как было сказано ранее, класс JMenuItem является дочерним по отношению к классу AbstractButton, поэтому пункт меню можно рассматривать как разновидность кнопки. При выборе пункта меню генерируется событие действия (подобное тому, которое генерирует класс JButton по щелчку на кнопке). В классе JMenuItem определены конструкторы, показанные в табл. 7.3. При создании пункта меню можно задавать мнемоническое обозначение, что позволит выбирать пункт меню нажатием соответствующих клавиш.

Таблица 7.3. Конструкторы класса JMenuItem

| Конструктор                        | Описание                                                                                                                                                       |
|------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| JMenuItem()                        | Создает пункт меню, не присваивая ему имени                                                                                                                    |
| JMenuItem(String name)             | Создает пункт меню с именем, заданным с помощью параметра name                                                                                                 |
| JMenuItem(Icon image)              | Создает пункт меню, в составе которого выводится изображение, заданное с помощью параметра image                                                               |
| JMenuItem(String name, Icon image) | Создает пункт меню с именем, заданным с помощью параметра name. В составе пункта выводится изображение, заданное с помощью параметра image                     |
| JMenuItem(String name, int mnem)   | Создает пункт меню с именем, заданным с помощью параметра name. Для данного пункта задается мнемоническое обозначение, определяемое посредством параметра mnem |
| JMenuItem(Action action)           | Создает пункт меню, используя информацию, заданную посредством параметра action                                                                                |

Поскольку пункты меню являются потомками AbstractButton, вам доступны функциональные возможности, определенные в этом классе. Например, вы можете разрешить или запретить доступ к кнопке, вызвав метод `setEnabled()`.

```
void setEnabled(boolean enable)
```

Если значение параметра `enable` равно `true`, доступ к пункту меню разрешается. Значение `false` данного параметра делает пункт меню недоступным.



## Вопросы для текущего контроля:

1. С помощью какого метода можно включить пункт в состав меню?
2. Какие действия выполняет метод `addSeparator()`?
3. Запретить доступ к пункту меню невозможно. Да или нет?

## ВАЖНО!

7.3

## Создание главного меню

В подавляющем большинстве приложений присутствует главное, или основное, меню. Оно выполнено в виде строки меню и предоставляет доступ ко всем (или почти ко всем) функциям приложения. Средства, имеющиеся в составе Swing, позволяют достаточно просто создать главное меню и управлять им. В данном разделе мы рассмотрим формирование главного меню, а в последующих разделах речь пойдет о том, как снабдить его дополнительными возможностями.

Формирование главного меню включает несколько этапов. В первую очередь надо создать объект `JMenuBar`, который будет содержать другие меню. Далее следует создать меню, доступные посредством главного. Для формирования меню надо сначала создать объект `JMenu`, а затем включить в его состав объекты `JMenuItem`. Готовое меню надо связать со строкой меню. Строка меню помещается в состав фрейма путем вызова метода `setJMenuBar()`. И, наконец, с каждым пунктом меню надо связать обработчик событий действия; события данного типа будут вызываться при выборе пунктов меню.

Для того чтобы лучше понять особенности создания меню и управления им, следует рассмотреть конкретный пример. Ниже приведен код программы, создающей простую строку меню, содержащую три пункта. Первому пункту соответствует стандартное меню, **File**, в состав которого входят пункты **Open**, **Close**, **Save** и **Exit**. Второе меню называется **Options** и содержит два подменю: **Colors** и **Priority**. Третье меню, соответствующее пункту **Help**, содержит только один пункт — **About**. При выборе пункта меню его имя отображается с помощью метки, находящейся на панели содержимого. Окно, создаваемое при работе программы, показано на рис. 7.1.

1. `add()`.
2. Этот метод включает в меню разделитель.
3. Нет.

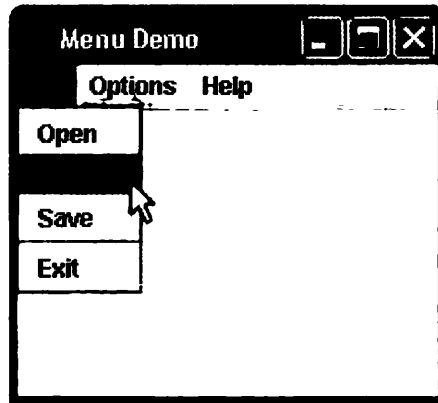


Рис. 7.1. Данные, отображаемые программой MenuDemo

// Создание простой строки меню.

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

```
class MenuDemo implements ActionListener {
```

```
    JLabel jlab;
```

```
    MenuDemo() {
```

```
        // Создание нового контейнера JFrame.
```

```
        JFrame jfrm = new JFrame("Menu Demo");
```

```
        // Установка диспетчера компоновки FlowLayout.
```

```
        jfrm.getContentPane().setLayout(new FlowLayout());
```

```
        // Установка начальных размеров фрейма.
```

```
        jfrm.setSize(220, 200);
```

```
        // Завершение программы при закрытии окна пользователем.
```

```
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        // Создание метки, с помощью которой отображается
```

```
        // информация о выборе пользователя.
```

```
        jlab = new JLabel();
```



```
// Создание строки меню.
JMenuBar jmb = new JMenuBar();

// Создание меню File.
JMenu jmFile = new JMenu("File");
// Создание пунктов, которые будут находиться
// в составе меню File.
JMenuItem jmiOpen = new JMenuItem("Open");
JMenuItem jmiClose = new JMenuItem("Close");
JMenuItem jmiSave = new JMenuItem("Save");
JMenuItem jmiExit = new JMenuItem("Exit");
// Включение пунктов в состав меню File.
jmFile.add(jmiOpen);
jmFile.add(jmiClose);
jmFile.add(jmiSave);
jmFile.addSeparator();
jmFile.add(jmiExit);
// Связывание меню File со строкой меню
jmb.add(jmFile);

// Создание меню Options.
JMenu jmOptions = new JMenu("Options");

// Создание подменю Colors.
JMenu jmColors = new JMenu("Colors");
JMenuItem jmiRed = new JMenuItem("Red");
JMenuItem jmiGreen = new JMenuItem("Green");
JMenuItem jmiBlue = new JMenuItem("Blue");
jmColors.add(jmiRed);
jmColors.add(jmiGreen);
jmColors.add(jmiBlue);
// Включение меню Colors в состав меню Options.
jmOptions.add(jmColors);

// Создание подменю Priority.
JMenu jmPriority = new JMenu("Priority");
JMenuItem jmiHigh = new JMenuItem("High");
JMenuItem jmiLow = new JMenuItem("Low");
jmPriority.add(jmiHigh);
jmPriority.add(jmiLow);
// Включение меню Priority в состав меню Options.
jmOptions.add(jmPriority);
```

```
// Создание пункта Reset и включение его в меню Options.
JMenuItem jmiReset = new JMenuItem("Reset");
jmOptions.addSeparator();
jmOptions.add(jmiReset);

// Связывание меню Options со строкой меню.
jmb.add(jmOptions);

// Создание меню Help и связывание его со строкой меню.
JMenu jmHelp = new JMenu("Help");
JMenuItem jmiAbout = new JMenuItem("About");
jmHelp.add(jmiAbout);
jmb.add(jmHelp);

// Связывание обработчика событий с пунктами меню.
jmiOpen.addActionListener(this);
jmiClose.addActionListener(this);
jmiSave.addActionListener(this);
jmiExit.addActionListener(this);
jmiRed.addActionListener(this);
jmiGreen.addActionListener(this);
jmiBlue.addActionListener(this);
jmiHigh.addActionListener(this);
jmiLow.addActionListener(this);
jmiReset.addActionListener(this);
jmiAbout.addActionListener(this);

// Включение метки на панель содержимого.
jfrm.getContentPane().add(jlab);

// Включение меню в состав фрейма.
jfrm.setJMenuBar(jmb);

// Отображение фрейма.
jfrm.setVisible(true);
}

// Обработка событий действия для пунктов меню.
public void actionPerformed(ActionEvent ae) {
    // Получение команды действия, соответствующей
    // выбранному пункту.
    String comStr = ae.getActionCommand();
```

```
// Если пользователь выбрал пункт Exit меню File,  
// выполнение программы завершается.  
if(comStr.equals("Exit")) System.exit(0);  
  
// В противном случае отображается информация  
// о выборе пользователя.  
jlab.setText(comStr + " Selected");  
}  
  
public static void main(String args[]) {  
    // Фрейм создается в потоке обработки событий.  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            new MenuDemo();  
        }  
    });  
}
```

Рассмотрим подробно процесс создания меню, начиная с конструктора MenuDemo. В начале конструктора находятся выражения, знакомые вам по предыдущим примерам. Затем создается строка меню, и ссылка на этот компонент сохраняется в переменной jmb.

```
// Создание строки меню.  
JMenuBar jmb = new JMenuBar();
```

Далее создается меню File, ссылка на которое помещается в переменную jmFile. Для создания данного меню и его пунктов используются следующие выражения:

```
// Создание меню File.  
JMenu jmFile = new JMenu("File");  
// Создание пунктов, которые будут находиться  
// в составе меню File.  
MenuItem jmiOpen = new JMenuItem("Open");  
JMenuItem jmiClose = new JMenuItem("Close");  
JMenuItem jmiSave = new JMenuItem("Save");  
JMenuItem jmiExit = new JMenuItem("Exit");
```

При активизации этого меню в нем будут отображаться пункты Open, Close, Save и Exit. Созданные пункты включаются в меню File следующим образом:

```
jmFile.add(jmiOpen);
```

```
jmFile.add(jmiClose);  
jmFile.add(jmiSave);  
jmFile.addSeparator();  
jmFile.add(jmiExit);
```

Работа над меню **File** завершается связыванием его со строкой меню.

```
jmb.add(jmFile);
```

После выполнения приведенных выше выражений в строке меню будет содержаться один пункт — **File**. Меню **File** будет включать четыре пункта: **Open**, **Close**, **Save** и **Exit**, расположенные в той же последовательности, в которой они перечислены здесь. Заметьте также, что перед пунктом **Exit** помещен разделитель. Он будет отображаться на экране, отделяя пункт **Exit** от предыдущих четырех пунктов.

Для создания меню **Options** применяется та же последовательность операций, что и для меню **File**. Однако меню **Options** содержит два подменю, **Colors** и **Priority**, а также пункт **Reset**. Подменю следует сформировать, а затем связать их с меню **Options**. Пункт **Reset** включается в состав меню в последнюю очередь. Готовое меню **Options** связывается со строкой меню. Такие же действия выполняются для создания меню **Help**.

Заметьте, что класс **MenuDemo** реализует интерфейс **ActionListener**, и события действия, сгенерированные при выборе пунктов меню, обрабатываются методом **actionPerformed()**, определенным в составе данного класса. Поэтому при вызове метода, связывающего обработчик событий с источником, указывается параметр **this**. Обратите также внимание на то, что с пунктами **Colors** и **Priority** не связываются обработчики, поскольку эти пункты служат лишь для вызова подменю.

Готовая строка меню помещается в состав фрейма. Для этого используется выражение

```
jfrm.setJMenuBar(jmb);
```

Как было сказано ранее, строка меню включается в состав контейнера верхнего уровня, а не на панель содержимого.

Метод **actionPerformed()** обрабатывает события действия, которые генерирует меню. Посредством вызова **getActionCommand()** объекта события он извлекает строку команды действия, связанной с выбранным пунктом. Строка команды действия сохраняется в переменной **comStr**. Затем выполняется проверка на совпадение со строкой **"Exit"**.

```
if(comStr.equals("Exit")) System.exit(0);
```

В случае положительного результата проверки работа программы завершается; для завершения программы вызывается метод `System.exit()`. Параметр, переданный данному методу, передается вызывающему процессу и интерпретируется как код завершения. (В качестве вызывающего процесса обычно выступает операционная система или браузер.) По соглашениям, код 0 означает успешное завершение программы. Любое другое значение свидетельствует о наличии ошибки.

Попробуйте поэкспериментировать с программой `MenuDemo`. Добавьте новые меню или включите в состав существующих меню дополнительные пункты. Перед тем как переходить к следующему разделу, необходимо четко представлять себе все особенности создания меню.

**ВАЖНО!**

## **7.4. Мнемонические обозначения и клавиши быстрого доступа**

Меню, созданное в предыдущем примере, вполне работоспособно, но его можно улучшить. В реальных приложениях поддерживается выбор пунктов меню с помощью клавиатуры. Для этого используются два механизма: мнемонические обозначения и клавиши быстрого доступа. В применении к меню мнемоническое обозначение задает клавишу, позволяющую выбрать пункт активизированного меню. Другими словами, данный механизм позволяет выбирать с помощью клавиатуры пункты того меню, которое уже отображается на экране. Клавиши быстрого доступа позволяют выбирать пункты любого меню, причем предварительная активизация этого меню не требуется.

Мнемонические обозначения можно задать как для объекта `JMenuItem`, так и для объекта `JMenu`. Существуют два способа формирования мнемонического обозначения для `JMenuItem`. Его можно задать при создании объекта (см. табл. 7.3). Это также позволяет сделать метод `setMnemonic()`. Для того чтобы создать мнемоническое обозначение для объекта `JMenu`, следует вызвать метод `setMnemonic()`. В обоих классах данный метод унаследован от класса `AbstractButton`. Его заголовок имеет следующий вид:

```
void setMnemonic(int mnem)
```

Как было сказано в модуле 2 (посвященном обсуждению кнопок), параметр `mnem` задает мнемоническое обозначение. Его значением должна быть одна из констант, определенных в классе `java.awt.event.KeyEvent`, например `KeyEvent.VK_F`. (Существует другой вариант метода `setMnemonic()`, которому в качестве параметра передается символ, но он не рекомендован к использованию.) Мнемонические обозначения не зависят от регистра символов, поэ-

тому, если вы передадите методу `setMnemonic()` параметр `VK_A`, для доступа к соответствующему объекту можно использовать как символ `a`, так и `A`.

По умолчанию первая из букв имени пункта, которая соответствует мнемоническому обозначению, отображается с подчеркиванием. Если вы хотите, чтобы подчеркнута была другая буква, вам следует вызвать метод `setDisplayMnemonicIndex()` и передать ему в качестве параметра индекс требуемого символа. Метод `setDisplayMnemonicIndex()` унаследован от `AbstractButton` как классом `JMenu`, так и классом `JMenuItem`. Заголовок этого метода имеет следующий вид:

```
void setDisplayedMnemonicIndex(int idx)
```

Индекс символа, который должен отображаться с подчеркиванием, задается посредством параметра `idx`.

Клавиши быстрого доступа можно связывать как с объектом `JMenu`, так и с объектом `JMenuItem`. Эта задача решается с помощью метода `setAccelerator()`.

```
void setAccelerator(KeyStroke ks)
```

Здесь `ks` — это комбинация клавиш, которую надо нажать для выбора пункта меню. Класс `KeyStroke` содержит несколько фабричных методов, которые создают различные типы комбинаций клавиш. Три таких метода показаны ниже.

```
static KeyStroke getKeyStroke(char ch)
```

```
static KeyStroke getKeyStroke(Character ch, int modifier)
```

```
static KeyStroke getKeyStroke(int ch, int modifier)
```

Здесь параметр `ch` определяет символ быстрого доступа. В первом варианте метода символ задается с помощью значения типа `char`. Во втором варианте методу передается объект `Character`. В третьем случае это константа из класса `KeyEvent`, который упоминался выше. Значение `modifier` формируется из одной или нескольких констант, определенных в классе `java.awt.event.InputEvent`.

---

```
InputEvent.ALT_MASK
```

```
InputEvent.CTRL_MASK
```

```
InputEvent.META_MASK
```

```
InputEvent.SHIFT_MASK
```

---

Таким образом, если вы зададите символ `VK_A` и модификатор `InputEvent.CTRL_MASK`, для быстрого доступа будет использоваться комбинация клавиш `<Ctrl+A>`.

Приведенные ниже выражения задают мнемонические обозначения и клавиши быстрого доступа для меню File, созданного в предыдущем разделе в программе MenuDemo. На рис. 7.2 показано, как выглядит активизированное меню File.

```
// Связывание мнемонических обозначений и клавиш
// быстрого доступа с меню File.
JMenu jmFile = new JMenu("File");
// Доступ к меню File можно получить с помощью клавиши <F>
jmFile.setMnemonic(KeyEvent.VK_F);

// В каждом случае в качестве мнемонического обозначения
// используется первая буква имени пункта. Для быстрого
// доступа используется та же клавиша, нажатая в комбинации
// с клавишей <Ctrl>
JMenuItem jmiOpen = new JMenuItem("Open",
                                   KeyEvent.VK_O);
jmiOpen.setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_O,
                           InputEvent.CTRL_MASK));

JMenuItem jmiClose = new JMenuItem("Close",
                                    KeyEvent.VK_C);
jmiClose.setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_C,
                           InputEvent.CTRL_MASK));
```

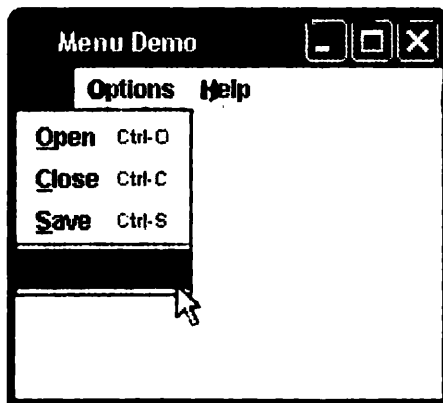


Рис. 7.2. Меню File после связывания с ним мнемонических обозначений и клавиш быстрого доступа

```
JMenuItem jmiSave = new JMenuItem("Save",
                                   KeyEvent.VK_S);
jmiSave.setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_S,
                           InputEvent.CTRL_MASK));

JMenuItem jmiExit = new JMenuItem("Exit",
                                   KeyEvent.VK_E);
jmiExit.setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_E,
                           InputEvent.CTRL_MASK));
```

После того как предложенные изменения будут внесены, меню File станет доступно с помощью комбинации клавиш <Alt+F>. Когда меню будет активизировано, вы сможете выбирать его пункты посредством клавиши <O>, <C>, <S> или <E>. Кроме того, в любой момент вы сможете выбрать пункт меню File с помощью комбинаций клавиш <Ctrl+O>, <Ctrl+C>, <Ctrl+S> и <Ctrl+E>.

## Спросим у опытного программиста

**Вопрос.** Клавиши быстрого доступа позволяют выбирать пункт меню, независимо от того, отображается ли само меню на экране. Есть ли при этом смысл задавать также мнемонические обозначения?

**Ответ.** Несмотря на то что клавиши быстрого доступа позволяют обращаться к пункту меню, не активизируя само меню, у них есть существенный недостаток: конкретную клавишу необходимо сопровождать модификатором: <Ctrl> или <Alt>. Мнемонические обозначения, напротив, позволяют пользователю выбирать пункт нажатием одной клавиши, однако, чтобы это стало возможно, меню должно отображаться на экране. Например, если вы зададите для пункта Save букву S в качестве мнемонического обозначения и клавиши быстрого доступа <Ctrl+S>, пользователь сможет выбрать данный пункт открытого меню, нажав лишь клавишу <S>, без клавиши <Ctrl>. Это различие может показаться несущественным, но из подобных мелочей складывается общее впечатление пользователя о программном продукте. Поэтому, если вы хотите, чтобы ваше приложение выглядело профессионально, вам следует определить и мнемонические обозначения, и клавиши быстрого доступа.



ВАЖНО!

## 7.5. Связывание изображений и подсказок с пунктами меню

Создавая меню, вы можете связывать изображения с его пунктами и даже использовать изображения вместо текста. Проще всего сделать это при создании пункта меню. Например, приведенные ниже конструкторы `JMenuItem` позволяют включить в состав пункта меню пиктограмму.

```
JMenuItem(Icon image)
```

```
JMenuItem(String name, Icon image)
```

Первый из данных конструкторов создает пункт меню, в составе которого имеется изображение, указанное посредством параметра `image`. Второй конструктор задает пункт меню с именем, определяемым параметром `name`, и изображением, на которое указывает параметр `image`. Например, чтобы создать пункт меню `About`, содержащий изображение, надо использовать следующие выражения:

```
ImageIcon icon = new ImageIcon("AboutIcon.gif");
```

```
JMenuItem jmiAbout = new JMenuItem("About", icon);
```

В результате выполнения данного фрагмента кода пиктограмма, заданная посредством параметра `icon`, будет отображаться в составе меню `Help` рядом со строкой `About` (рис. 7.3). Также можно включить пиктограмму в уже готовый пункт меню. Для этого следует вызвать метод `setIcon()`, который, подобно другим методам, унаследован от класса `AbstractButton`. Задать взаимное расположение текста и изображения позволяет метод `setHorizontalTextPosition()`.

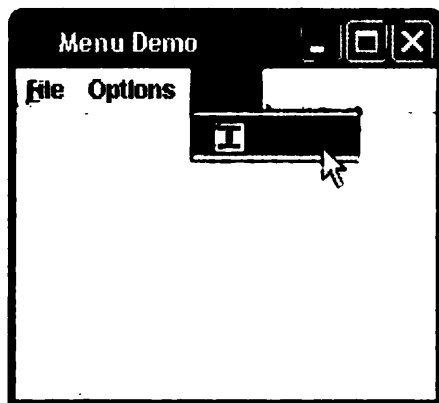


Рис. 7.3. Пункт меню `About`, в состав которого включена пиктограмма

По мере необходимости вы можете задать изображение, соответствующее запрету доступа к пункту меню. Это позволяет сделать метод `setDisabledIcon()`. В обычных условиях, если пункт меню недоступен, пиктограмма, используемая по умолчанию, отображается в серых тонах. Если пиктограмма, соответствующая запрету доступа, задана, она будет отображаться при переводе пункта меню в данный режим. Метод `setDisabledIcon()` унаследован от класса `AbstractButton` и используется так же, как и для кнопок.

Поскольку все классы меню являются потомками класса `JComponent`, они поддерживают строку подсказки. Для того чтобы связать подсказку с пунктом меню, надо вызвать метод `setToolTipText()` и ввести текст, который должен отображаться в случае, если пользователь задержит курсор мыши на данном пункте. Например, для того, чтобы связать строку подсказки с пунктом `About`, надо включить в состав программы следующее выражение:

```
jmiAbout.setToolTipText("Info about the MenuDemo program.");
```

Несмотря на то что строка подсказки связывается с пунктами меню достаточно редко, такая возможность существует, и это еще раз свидетельствует о том, насколько мощные средства предоставляет набор `Swing`.



## Вопросы для текущего контроля

1. Методам `setMnemonic()` и `setAccelerator()` в качестве параметра передается объект `KeyEvent`. Да или нет?
2. Какая маска соответствует клавише `<Ctrl>`?
3. Какой метод позволяет связать с пунктом меню строку подсказки?

## Проект 7.1. Динамическое добавление и удаление пунктов меню

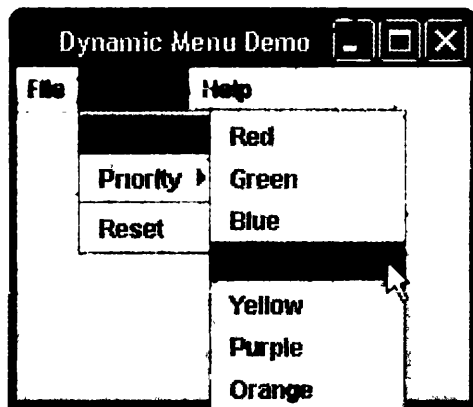
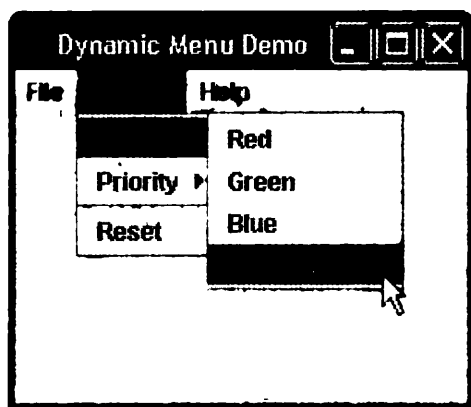
`DynMenuDemo.java`

Существует возможность изменять состав меню в процессе выполнения программы. Например, вы можете по мере необходимости добавлять или удалять пункты. Можно также изменить из программы имя пункта меню. Таким образом, в программе может использоваться динамическое меню, конфигурация которого будет изменяться в зави-

1. Нет, методу `setAccelerator()` передается объект `KeyStroke`.
2. `InputEvent.CTRL_MASK`.
3. `setToolTipText()`.

симости от текущих потребностей пользователя. Данный проект, представляющий собой модификацию программы MenuDemo, рассмотренной в начале модуля, демонстрирует добавление новых цветов к меню Colors и удаление существующих.

К меню Colors добавляется новый пункт More Colors. При выборе данного пункта в состав меню включаются цвета Yellow, Purple и Orange, а имя пункта More Colors изменяется на Less Colors. При выборе пункта Less Colors указанные выше цвета удаляются из меню, а данный пункт снова получает имя More Colors. Окна, отображаемые на разных этапах работы программы, показаны ниже.



## Последовательность действий

1. Скопируйте код программы MenuDemo, рассмотренной в начале данного модуля, в файл DynMenuDemo.java. (При желании вы можете использовать версию MenuDemo, в которой предусмотрена поддержка клавиш быстрого доступа, мнемонических сокращений и пиктограмм. Однако, чтобы размеры кода не были слишком велики, рассмотренный ниже исходный текст программы не реализует эти дополнительные возможности.)
2. Измените имя класса MenuDemo на DynMenuDemo. Добавьте три переменных экземпляра типа JMenuItem: jmiYellow, jmiPurple и jmiOrange. Преобразуйте локальную переменную jmColors в переменную экземпляра. После внесения этих изменений начало программы должно выглядеть так, как показано ниже.

```
// Проект 7.1. Динамическое добавление и удаление
// пунктов меню.
```

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

```
import javax.swing.*;

class DynMenuDemo implements ActionListener {

    JLabel jlab;

    JMenuItem jmiYellow;
    JMenuItem jmiPurple;
    JMenuItem jmiOrange;

    JMenu jmColors;
```

3. Создайте пункт меню **More Colors** и поместите ссылку на него в переменную `jmiMoreLess`. Включите этот пункт в состав меню `jmColors`.

```
// Создание пункта меню More Colors / Less Colors.
JMenuItem jmiMoreLess = new JMenuItem("More Colors");
jmColors.add(jmiMoreLess);
```

4. Создайте пункты, соответствующие цветам **Yellow**, **Purple** и **Orange**.

```
// Создание дополнительных цветов. Соответствующие им пункты
// будут добавляться и удаляться по мере необходимости.
jmiYellow = new JMenuItem("Yellow");
jmiPurple = new JMenuItem("Purple");
jmiOrange = new JMenuItem("Orange");
```

5. Свяжите обработчики событий с новыми пунктами меню.

```
// Связывание обработчиков событий с пунктами меню,
// соответствующих дополнительным цветам.
jmiMoreLess.addActionListener(this);
jmiYellow.addActionListener(this);
jmiPurple.addActionListener(this);
jmiOrange.addActionListener(this);
```

6. Измените метод `actionPerformed()`, включив в него код для поддержки новых пунктов.

```
// Обработка событий действий, соответствующих пунктам меню.
public void actionPerformed(ActionEvent ae) {
    // Получение команды действия для выбранного
    // пункта меню.
    String comStr = ae.getActionCommand();

    // Если пользователь выбрал пункт Exit,
```

```

// выполнение программы должно завершиться.
if(comStr.equals("Exit"))
    System.exit(0);
else if(comStr.equals("More Colors")) {
    jmColors.add(jmiYellow);
    jmColors.add(jmiPurple);
    jmColors.add(jmiOrange);
    JMenuItem mi = (JMenuItem) ae.getSource();
    mi.setText("Less Colors");
} else if(comStr.equals("Less Colors")) {
    jmColors.remove(jmiYellow);
    jmColors.remove(jmiPurple);
    jmColors.remove(jmiOrange);
    JMenuItem mi = (JMenuItem) ae.getSource();
    mi.setText("More Colors");
}

// В противном случае выводится информация о
// выборе пользователя.
jlab.setText(comStr + " Selected");
}

```

Обратите внимание, что при выборе пункта **More Colors** к меню **Colors** добавляются три новых цвета, а имя пункта **jmiMoreLess** изменяется на **Less Colors**. Когда пользователь выбирает пункт **Less Colors**, цвета, добавленные ранее, удаляются из меню, а имя пункта **jmiMoreLess** изменяется на **More Colors**. Для удаления пунктов из меню используется метод `remove()`.

7. После внесения всех указанных изменений код программы должен выглядеть так, как показано ниже.

// Проект 7.1. Динамическое добавление и удаление пунктов меню.

```

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

class DynMenuDemo implements ActionListener {

    JLabel jlab;

    JMenuItem jmiYellow;
    JMenuItem jmiPurple;

```

```
JMenuItem jmiOrange;

JMenu jmColors;

DynMenuDemo() {
    // Создание нового контейнера JFrame.
    JFrame jfrm = new JFrame("Dynamic Menu Demo");

    // Установка диспетчера компоновки FlowLayout.
    jfrm.getContentPane().setLayout(new FlowLayout());

    // Установка начальных размеров фрейма.
    jfrm.setSize(220, 200);

    // Завершение программы при закрытии окна пользователем.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Создание метки, с помощью которой отображается
    // информация о выборе пользователя.
    jlab = new JLabel();

    // Создание строки меню.
    JMenuBar jmb = new JMenuBar();

    // Создание меню File.
    JMenu jmFile = new JMenu("File");
    JMenuItem jmiOpen = new JMenuItem("Open");
    JMenuItem jmiClose = new JMenuItem("Close");
    JMenuItem jmiSave = new JMenuItem("Save");
    JMenuItem jmiExit = new JMenuItem("Exit");
    jmFile.add(jmiOpen);
    jmFile.add(jmiClose);
    jmFile.add(jmiSave);
    jmFile.addSeparator();
    jmFile.add(jmiExit);
    jmb.add(jmFile);

    // Создание меню Options.
    JMenu jmOptions = new JMenu("Options");

    // Создание подменю Colors.
    jmColors = new JMenu("Colors");
    JMenuItem jmiRed = new JMenuItem("Red");
```

```
JMenuItem jmiGreen = new JMenuItem("Green");
JMenuItem jmiBlue = new JMenuItem("Blue");
jmColors.add(jmiRed);
jmColors.add(jmiGreen);
jmColors.add(jmiBlue);

// Создание пункта меню More Colors / Less Colors.
JMenuItem jmiMoreLess = new JMenuItem("More Colors");
jmColors.add(jmiMoreLess);

// Включение меню Colors в состав меню Options.
jmOptions.add(jmColors);

// Создание дополнительных цветов. Соответствующие
// им пункты
// будут добавляться и удаляться по мере необходимости.
jmiYellow = new JMenuItem("Yellow");
jmiPurple = new JMenuItem("Purple");
jmiOrange = new JMenuItem("Orange");

// Создание подменю Priority.
JMenu jmPriority = new JMenu("Priority");
JMenuItem jmiHigh = new JMenuItem("High");
JMenuItem jmiLow = new JMenuItem("Low");
jmPriority.add(jmiHigh);
jmPriority.add(jmiLow);

// Включение меню Priority в состав меню Options.
jmOptions.add(jmPriority);

// Создание пункта меню Reset.
JMenuItem jmiReset = new JMenuItem("Reset");
jmOptions.addSeparator();
jmOptions.add(jmiReset);

// Связывание меню Options со строкой меню.
jmb.add(jmOptions);

// Создание меню Help и связывание его со строкой меню.
JMenu jmHelp = new JMenu("Help");
JMenuItem jmiAbout = new JMenuItem("About");
jmHelp.add(jmiAbout);
jmb.add(jmHelp);
```

```
// Связывание обработчика событий с пунктами меню.
jmiOpen.addActionListener(this);
jmiClose.addActionListener(this);
jmiSave.addActionListener(this);
jmiExit.addActionListener(this);
jmiRed.addActionListener(this);
jmiGreen.addActionListener(this);
jmiBlue.addActionListener(this);
jmiHigh.addActionListener(this);
jmiLow.addActionListener(this);
jmiReset.addActionListener(this);
jmiAbout.addActionListener(this);

// Связывание обработчиков событий с пунктами меню,
// соответствующих дополнительным цветам.
jmiMoreLess.addActionListener(this);
jmiYellow.addActionListener(this);
jmiPurple.addActionListener(this);
jmiOrange.addActionListener(this);

// Включение метки на панель содержимого.
jfrm.getContentPane().add(jlab);

// Включение меню в состав фрейма.
jfrm.setJMenuBar(jmb);

// Отображение фрейма.
jfrm.setVisible(true);
}

// Обработка событий действий, соответствующих пунктам меню.
public void actionPerformed(ActionEvent ae) {
    // Получение команды действия для выбранного
    // пункта меню.
    String comStr = ae.getActionCommand();

    // Если пользователь выбрал пункт Exit,
    // выполнение программы должно завершиться.

    if(comStr.equals("Exit"))
        System.exit(0);
    else if(comStr.equals("More Colors")) {
```



```

        jmColors.add(jmiYellow);
        jmColors.add(jmiPurple);
        jmColors.add(jmiOrange);
        JMenuItem mi = (JMenuItem) ae.getSource();
        mi.setText("Less Colors");
    } else if (comStr.equals("Less Colors")) {
        jmColors.remove(jmiYellow);
        jmColors.remove(jmiPurple);
        jmColors.remove(jmiOrange);
        JMenuItem mi = (JMenuItem) ae.getSource();
        mi.setText("More Colors");
    }

    // В противном случае выводится информация о
    // выборе пользователя.
    jlab.setText(comStr + " Selected");
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new DynMenuDemo();
        }
    });
}
}

```

ВАЖНО!

7.6

## Использование объектов

### JRadioButtonMenuItem и JCheckBoxMenuItem

Несмотря на то что чаще всего в приложениях используются пункты меню тех типов, которые были рассмотрены ранее, в Swing также поддерживаются пункты, представляющие собой флажки и переключатели опций. В результате появляется возможность упростить интерфейс, реализовав с помощью меню те функции, для которых потребовались бы дополнительные компоненты. В некоторых случаях флажками и переключателями опций удобнее пользоваться тогда, когда они присутствуют в составе меню. Независимо от того, по каким причинам возникла необходимость включать в меню пункты специальных типов, Swing предоставляет средства, позволяющие решить эту задачу.

Для того чтобы включить в состав меню флажок опции, надо создать экземпляр класса `JCheckBoxMenuItem`. Конструкторы этого класса описаны в табл. 7.4. Класс `JCheckBoxMenuItem` является подклассом класса `JMenuItem`. Флажки опций в составе меню работают так же, как и независимые компоненты данного типа. В частности, при изменении состояния они генерируют события действия и события элементов. Флажки опций в меню наиболее полезны тогда, когда приходится устанавливать некоторые свойства и желательно, чтобы их состояние отображалось на экране.

Таблица 7.4. Конструкторы класса `JCheckBoxMenuItem`

| Конструктор                                                           | Описание                                                                                                                                                                                                                                                                                                                       |
|-----------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JCheckBoxMenuItem()</code>                                      | Создает пункт меню в виде флажка опции. Для данного пункта не определяется ни имя, ни изображение                                                                                                                                                                                                                              |
| <code>JCheckBoxMenuItem(String name)</code>                           | Создает пункт меню в виде флажка опции. Для данного пункта определяется имя, указанное посредством параметра <code>name</code>                                                                                                                                                                                                 |
| <code>JCheckBoxMenuItem(Icon icon)</code>                             | Создает пункт меню в виде флажка опции. С пунктом связывается изображение, заданное посредством параметра <code>icon</code>                                                                                                                                                                                                    |
| <code>JCheckBoxMenuItem(String name, boolean state)</code>            | Создает пункт меню в виде флажка опции и связывает с ним имя, указанное посредством параметра <code>name</code> . Если значение параметра <code>state</code> равно <code>true</code> , флажок изначально установлен, в противном случае он сброшен                                                                             |
| <code>JCheckBoxMenuItem(String name, Icon icon)</code>                | Создает пункт меню в виде флажка опции. С данным пунктом связывается имя, указанное с помощью параметра <code>name</code> , и изображение, заданное посредством параметра <code>icon</code>                                                                                                                                    |
| <code>JCheckBoxMenuItem(String name, Icon icon, boolean state)</code> | Создает пункт меню в виде флажка опции. С данным пунктом связывается имя, указанное с помощью параметра <code>name</code> , и изображение, заданное посредством параметра <code>icon</code> . Если значение параметра <code>state</code> равно <code>true</code> , флажок изначально установлен, в противном случае он сброшен |
| <code>JCheckBoxMenuItem(Action act)</code>                            | Создает пункт меню в виде флажка опции, для которого имя, изображение и другие свойства определяет объект, переданный посредством параметра <code>act</code>                                                                                                                                                                   |

Для того чтобы включить в меню кнопку переключателя опций, надо создать объект `JRadioButtonMenuItem`. Конструкторы этого класса описаны в табл. 7.5. Класс `JRadioButtonMenuItem` является подклассом класса `JMenuItem`. Пункты данного типа ведут себя как зависимые кнопки переключателей опций, в частности, они генерируют события действий и события элементов. Подобно обычным переключателям опций, пункты данного типа, находящиеся в составе меню, можно объединить в группу, чтобы они работали подобно кнопкам с зависимой фиксацией.

Таблица 7.5. Конструкторы класса `JRadioButtonMenuItem`

| Конструктор                                                              | Описание                                                                                                                                                                                                                                                                                            |
|--------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JRadioButtonMenuItem()</code>                                      | Создает пункт меню в виде кнопки переключателя опций. Для данного пункта не определяется ни имя, ни изображение                                                                                                                                                                                     |
| <code>JRadioButtonMenuItem(String name)</code>                           | Создает пункт меню в виде кнопки переключателя опций. Для данного пункта определяется имя, заданное с помощью параметра <code>name</code>                                                                                                                                                           |
| <code>JRadioButtonMenuItem(Icon icon)</code>                             | Создает пункт меню в виде кнопки переключателя опций. С пунктом связывается изображение, заданное посредством параметра <code>icon</code>                                                                                                                                                           |
| <code>JRadioButtonMenuItem(String name, boolean state)</code>            | Создает пункт меню в виде кнопки переключателя опций и связывает с ним имя, заданное с помощью параметра <code>name</code> . Если значение параметра <code>state</code> равно <code>true</code> , кнопка изначально выбрана                                                                         |
| <code>JRadioButtonMenuItem(Icon icon, boolean state)</code>              | Создает пункт меню в виде кнопки переключателя опций. С пунктом связывается изображение, заданное посредством параметра <code>icon</code> . Если значение параметра <code>state</code> равно <code>true</code> , кнопка изначально выбрана                                                          |
| <code>JRadioButtonMenuItem(String name, Icon icon)</code>                | Создает пункт меню в виде кнопки переключателя опций. С пунктом связывается имя, указанное с помощью параметра <code>name</code> , и изображение, заданное посредством параметра <code>icon</code>                                                                                                  |
| <code>JRadioButtonMenuItem(String name, Icon icon, boolean state)</code> | Создает пункт меню в виде кнопки переключателя опций. С пунктом связывается имя, указанное с помощью параметра <code>name</code> , и изображение, заданное посредством параметра <code>icon</code> . Если значение параметра <code>state</code> равно <code>true</code> , кнопка изначально выбрана |
| <code>JRadioButtonMenuItem(Action act)</code>                            | Создает пункт меню в виде кнопки переключателя опций, для которого имя, изображение и другие свойства определяет объект, переданный посредством параметра <code>act</code>                                                                                                                          |

Для того чтобы реализовать флажки и переключатели опций в составе меню, удалите из рассмотренной ранее программы MenuDemo код, предназначенный для создания меню Options. Включите вместо него приведенный ниже код; с его помощью в подменю Colors реализуются флажки опций, а в подменю Priority — переключатели опций. В результате внесенных изменений меню Options должно выглядеть так, как показано на рис. 7.4.

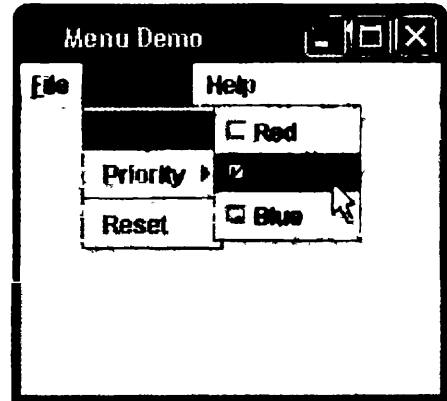
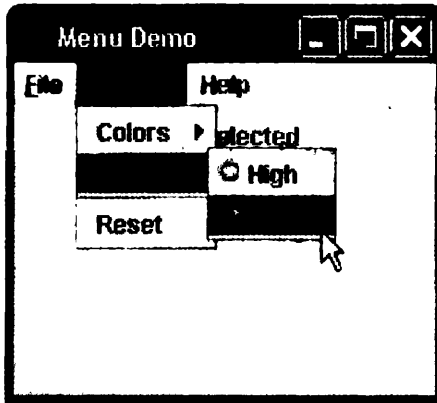


Рис. 7.4. Пункты меню, реализующие флажки и переключатели опций

```
// Создание меню Options.  
JMenu jmOptions = new JMenu("Options");  
  
// Создание подменю Colors.  
JMenu jmColors = new JMenu("Colors");  
  
// Использование флажков опций для выбора цвета. Данный  
// подход позволяет пользователю выбирать несколько цветов.  
JCheckBoxMenuItem jmiRed = new JCheckBoxMenuItem("Red");  
JCheckBoxMenuItem jmiGreen = new JCheckBoxMenuItem("Green");  
JCheckBoxMenuItem jmiBlue = new JCheckBoxMenuItem("Blue");  
  
jmColors.add(jmiRed);  
jmColors.add(jmiGreen);  
jmColors.add(jmiBlue);  
jmOptions.add(jmColors);  
  
// Создание подменю Priority.  
JMenu jmPriority = new JMenu("Priority");
```

```
// Для выбора приоритета применяется переключатель опций.  
// Благодаря использованию пунктов данного типа  
// пользователь видит, какой приоритет установлен,  
// и может выбирать в каждый момент времени только одно значение.  
// Заметьте, что кнопка High изначально выбрана.  
JRadioButtonMenuItem jmiHigh =  
    new JRadioButtonMenuItem("High", true);  
JRadioButtonMenuItem jmiLow =  
    new JRadioButtonMenuItem("Low");  
jmPriority.add(jmiHigh);  
jmPriority.add(jmiLow);  
jmOptions.add(jmPriority);  
  
// Объединение кнопок переключателя опций в группу.  
ButtonGroup bg = new ButtonGroup();  
bg.add(jmiHigh);  
bg.add(jmiLow);  
  
// Создание пункта Reset.  
JMenuItem jmiReset = new JMenuItem("Reset");  
jmOptions.addSeparator();  
jmOptions.add(jmiReset);  
  
// Связывание меню Options со строкой меню.  
jmb.add(jmOptions);
```



## Вопросы для текущего контроля

1. Какой класс используется для создания пункта меню в виде флажка опции?
2. Когда объекты `JRadioButtonMenuItem` включаются в состав меню, они автоматически объединяются в группу. Да или нет?

- 
1. `JCheckBoxMenuItem`.
  2. Нет. Для объединения пунктов меню, созданных в виде кнопок переключателя опций, в группу надо принять специальные меры.

ВАЖНО!

## 7.7. Создание контекстного меню

Помимо строки меню, в программе можно создать также контекстное меню. Обычно такое меню активизируется щелчком правой кнопкой мыши на том компоненте, для которого оно было создано. Для поддержки контекстных меню в библиотеке Swing был предусмотрен класс `JPopupMenu`. Конструкторы, определенные в классе `JPopupMenu`, описаны в табл. 7.6.

Таблица 7.6. Конструкторы класса `JPopupMenu`

| Конструктор                           | Описание                                                                                                                                                        |
|---------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JPopupMenu ()</code>            | Создает контекстное меню по умолчанию                                                                                                                           |
| <code>JPopupMenu (String name)</code> | Создает контекстное меню, заголовок которого определяется параметром <code>name</code> . Будет ли отображаться данный заголовок, зависит от используемого стиля |

В большинстве случаев контекстные меню создаются подобно обычным меню. В первую очередь для этого надо создать объект `JPopupMenu`, а затем заполнить его конкретными пунктами. На действия пользователя по выбору пунктов меню реагируют обработчики соответствующих событий. Основным отличием между контекстным и обычным меню является процедура активизации.

Процесс активизации контекстного меню включает три этапа.

1. Необходимо зарегистрировать обработчик событий, связанных с мышью.
2. При выполнении обработчика следует проверять состояние переключателя контекстного меню.
3. Если переключатель контекстного меню установлен, надо отобразить меню путем вызова метода `show ()`.

Рассмотрим подробно действия, выполняемые на каждом из этих этапов.

Как правило, контекстное меню активизируется по щелчку правой кнопкой мыши; при этом курсор мыши должен находиться на компоненте, для которого данное меню было создано. Таким образом, переключатель контекстного меню проверяется в обработчике событий, соответствующих щелчку правой кнопкой мыши на компоненте. Для этого в программе должен присутствовать класс, реализующий интерфейс `MouseListener`; экземпляр этого класса следует зарегистрировать путем вызова метода `addMouseListener ()`. В классе, реализующем интерфейс `MouseListener`, должны быть определены следующие методы:

```
void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me)
void mouseReleased(MouseEvent me)
```

Только два из них, `mousePressed()` и `mouseReleased()`, имеют отношение к контекстному меню. В зависимости от используемого стиля, одно из этих событий может установить переключатель контекстного меню. Поскольку для работы с контекстным меню три остальных метода не важны, мы будем использовать класс `MouseAdapater`, реализующий интерфейс `MouseListener`. В подклассе адаптера переопределим методы `mousePressed()` и `mouseReleased()`.

В классе `MouseEvent` определено несколько методов, но лишь четыре из них важны при работе с контекстными меню. Эти методы перечислены ниже.

```
int getX()
int getY()
boolean isPopupTrigger()
Component getComponent()
```

Текущие координаты курсора мыши относительно источника события возвращают методы `getX()` и `getY()`. Они определяют верхний левый угол отображаемого контекстного меню. Метод `isPopupTrigger()`, возвращает значение `true`, если событие мыши представляет переключатель контекстного меню, и `false` — в противном случае. Этот метод позволяет определить, когда следует отображать меню. Получить ссылку на компонент, сгенерировавший событие мыши, дает возможность метод `getComponent()`.

Для того чтобы отобразить контекстное меню, надо вызвать метод `show()`, определенный в классе `JPopupMenu`. Заголовок этого метода имеет следующий вид:

```
void show(Component invoker, int upperX, int upperY)
```

Здесь параметр `invoker` определяет компонент, для которого отображается контекстное меню. Значения `upperX` и `upperY` задают координаты верхнего левого угла меню относительно компонента `invoker`. Получить ссылку на компонент вызывающего меню проще всего, вызвав метод `getComponent()` объекта события. Этот объект передается методу обработчика событий мыши.

Используем приведенные выше сведения на практике. Включите в состав программы `MenuDemo`, рассмотренной в начале данного модуля, контекстное меню

**Edit.** Данное меню содержит три пункта: **Cut**, **Copy** и **Paste**. Для этого прежде всего определите в классе `MenuDemo` следующую переменную экземпляра:

```
JPopupMenu jpu;
```

Переменная `jpu` будет содержать ссылку на контекстное меню.

Включите в конструктор `MenuDemo` следующий фрагмент кода:

```
// Создание контекстного меню Edit.
jpu = new JPopupMenu();

// Создание пунктов контекстного меню.
JMenuItem jmiCut = new JMenuItem("Cut");
JMenuItem jmiCopy = new JMenuItem("Copy");
JMenuItem jmiPaste = new JMenuItem("Paste");

// Включение пунктов в состав контекстного меню.
jpu.add(jmiCut);
jpu.add(jmiCopy);
jpu.add(jmiPaste);

// Связывание обработчика событий мыши с панелью содержимого.
jfrm.getContentPane().addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent me) {
        // Проверка переключателя контекстного меню.
        if(me.isPopupTrigger())
            jpu.show(me.getComponent(), me.getX(), me.getY());
    }
    public void mouseReleased(MouseEvent me) {
        // Проверка переключателя контекстного меню.
        if(me.isPopupTrigger())
            jpu.show(me.getComponent(), me.getX(), me.getY());
    }
});
```

В начале данного фрагмента создается экземпляр класса `JPopupMenu`, ссылка на который сохраняется в переменной `jpu`. Затем создаются три пункта меню: **Cut**, **Copy** и **Paste**. Данные пункты формируются обычным способом, после чего включаются в меню `jpu`. Этим заканчивается процесс формирования меню **Edit**. Контекстные меню не связываются со строкой меню и меню других типов.

Далее в программе создается обработчик событий `MouseListener`, который реализуется посредством анонимного внутреннего класса. Суперклассом



этого класса является `MouseAdapter`. Следовательно, в обработчике надо переопределить лишь методы, имеющие отношение к работе с контекстным меню: `mousePressed()` и `mouseReleased()`. Для остальных методов, объявленных в интерфейсе `MouseListener`, принимаются реализации по умолчанию, предоставляемые классом адаптера. Заметьте, что обработчик событий мыши связывается с панелью содержимого объекта `jfrm`. Это означает, что по щелчку мышью в любой части панели отобразится контекстное меню.

В теле методов `mousePressed()` и `mouseReleased()` вызывается метод `isPopupTrigger()`. Это позволяет определить, должно ли событие мыши сопровождаться отображением контекстного меню. Если данный метод возвращает значение `true`, меню отображается путем вызова метода `show()`. Ссылку на объект, для которого вызывается меню, возвращает метод `getComponent()` объекта события. В данном примере вызывающим объектом является панель содержимого. Координаты верхнего левого угла меню предоставляют методы `getX()` и `getY()`. В результате левый верхний угол контекстного меню совпадает с курсором мыши.

Затем надо включить в программу обработчик событий действия. Такое событие возникает тогда, когда пользователь выбирает пункт контекстного меню.

```
jmiCut.addActionListener(this);  
jmiCopy.addActionListener(this);  
jmiPaste.addActionListener(this);
```

После того как вы измените программу описанным выше способом, вы сможете вызвать контекстное меню, щелкая правой кнопкой мыши в любой части панели содержимого. Окно с контекстным меню показано на рис. 7.5.

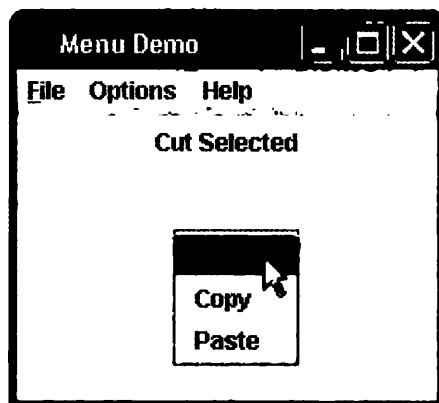


Рис. 7.5. Контекстное меню Edit

Вопросы для текущего контроля

1. Для использования контекстного меню в программе должны обрабатываться события \_\_\_\_\_.
2. Какие действия выполняет метод `isPopupTrigger()`?
3. Какой метод класса `JPopupMenu` необходимо вызвать, чтобы отобразить контекстное меню?

## Спросим у опытного программиста

**Вопрос.** Для того чтобы получить ссылку на компонент, для которого должно быть отображено контекстное меню, вызывается метод `getComponent()`. Однако в рассмотренном выше примере таким компонентом всегда является панель содержимого. Можно ли явным образом использовать ссылку на нее?

**Ответ.** Да. Если вы объявите `jfrm` как переменную экземпляра для класса `MenuDemo`, она будет доступна и во внутреннем классе. В этом случае вы сможете использовать вызов метода `show()` для того, чтобы отобразить контекстное меню.

```
jpu.show(jfrm.getContentPane(), me.getX(),  
        me.getY());
```

Преимущество использования метода `getComponent()` состоит в том, что он позволяет автоматически отобразить контекстное меню. Таким образом, если вынести метод `checkPU()` за пределы неименованного внутреннего класса, его можно использовать для вывода любого контекстного меню, которое соответствует вызываемому объекту.

1. Мыши.
2. Метод `isPopupTrigger()` определяет, должно ли отображаться контекстное меню.
3. `show()`.

ВАЖНО!

## 7.8. Создание панели инструментов

Панель инструментов — это компонент, который предоставляет альтернативные средства обращения к функциям, доступным посредством меню. Панель инструментов содержит набор кнопок (или других компонентов), которые позволяют пользователю обращаться к различным функциям приложения. Например, на панели инструментов могут располагаться кнопки для установки характеристик шрифта (полужирного, курсива с подчеркиванием и т.д.). Как правило, на кнопках, расположенных на панели инструментов, располагаются изображения, однако они могут дополняться текстом. Часто с кнопками на панели инструментов связываются строки подсказки. Путем перетаскивания панель инструментов можно разместить в любой позиции окна и даже вынести за его пределы. В Swing панель инструментов поддерживается с помощью класса `JToolBar`. Конструкторы, используемые для создания объекта `JToolBar`, описаны в табл. 7.7. Панель инструментов обычно используется в составе окна, с которым связан диспетчер компоновки `BorderLayout`. Существуют два аргумента в пользу такого решения. Во-первых, благодаря ему панель инструментов можно разместить вдоль любой границы окна (обычно она располагается вдоль верхней границы). Во-вторых, в этом случае панель можно перемещать путем перетаскивания.

Таблица 7.7. Конструкторы класса `JToolBar`

| Конструктор                                  | Описание                                                                                                                                                                                                                                                                                             |
|----------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JToolBar()</code>                      | Создает горизонтальную панель инструментов без заголовка                                                                                                                                                                                                                                             |
| <code>JToolBar(String title)</code>          | Создает горизонтальную панель инструментов с заголовком, заданным посредством параметра <code>title</code> . Заголовок отображается только в том случае, если панель инструментов перетаскивается за пределы окна                                                                                    |
| <code>JToolBar(int how)</code>               | Создает панель инструментов, ориентация которой определяется параметром <code>how</code> . Значение этого параметра может быть либо <code>JToolBar.VERTICAL</code> , либо <code>JToolBar.HORIZONTAL</code>                                                                                           |
| <code>JToolBar(String title, int how)</code> | Создает горизонтальную панель инструментов с заголовком, заданным посредством параметра <code>title</code> . Ориентация панели определяется параметром <code>how</code> . Значение параметра <code>how</code> может быть либо <code>JToolBar.VERTICAL</code> , либо <code>JToolBar.HORIZONTAL</code> |

Панель инструментов можно не только перетаскивать в пределах окна, но и размещать за его границами. В этом случае формируется *плавающая* панель (в отличие от *пришвартованной*). Если при создании панели инструментов вы укажете заголовок, он будет отображаться в тот момент, когда панель превратится в плавающую.

Добавление кнопок (или других компонентов) к панели осуществляется так же, как и включение пунктов в состав меню. Для этого достаточно вызвать метод `add()`. Компоненты разместятся на панели инструментов в том порядке, в котором они были в нее включены.

Готовую панель инструментов не следует связывать с полосой меню. Вместо этого панель включается в состав контейнера. Обычно она размещается в северной области контейнера, определяемой диспетчером компоновки `BorderLayout`, т.е. в верхней части окна. Компонент, на который воздействуют средства, доступные с помощью полосы прокрутки, обычно включается в центральную область окна. Даже если панель инструментов будет перенесена в другую часть окна, основной компонент по-прежнему остается в центральной области.

Чтобы продемонстрировать особенности работы с панелью инструментов, включим данный компонент в программу `MenuDemo`. Посредством данной панели организуем поддержку следующих функций, необходимых при отладке: установку точки останова, удаление точки останова и продолжение выполнения программы. Для того чтобы реализовать панель инструментов, следует выполнить описанные ниже действия.

Во-первых, удалите из программы следующую строку кода:

```
jfrm.getContentPane().setLayout(new BorderLayout());
```

В результате `JFrame` будет автоматически использовать диспетчер компоновки `BorderLayout`, предусмотренный по умолчанию. Во-вторых, учитывая тот факт, что с контейнером связан диспетчер `BorderLayout`, надо изменить строку кода, включающую метку `jlab`, в состав фрейма, следующим образом:

```
jfrm.getContentPane().add(jlab, BorderLayout.CENTER);
```

С помощью данной строки `jlab` явным образом помещается в центральную область контейнера, управляемого диспетчером `BorderLayout`. (Явно указывать центральную область не обязательно. Если с контейнером связан диспетчер компоновки `BorderLayout`, компонент по умолчанию помещается именно в нее. Однако такая запись делает программу более удобочитаемой.)

Включите в программу следующий фрагмент кода, предназначенный для создания панели инструментов `Debug`:

```
// Создание панели инструментов Debug.
JToolBar jtb = new JToolBar("Debug");

// Загрузка изображений для кнопок панели инструментов.
ImageIcon set = new ImageIcon("setBP.gif");
ImageIcon clear = new ImageIcon("clearBP.gif");
ImageIcon resume = new ImageIcon("resume.gif");

// Создание кнопок панели инструментов с использованием
// изображений. С кнопками связываются команды действия
// и строки подсказки.
JButton jbbtnSet = new JButton(set);
jbbtnSet.setActionCommand("Set Breakpoint");
jbbtnSet.setToolTipText("Set Breakpoint");

JButton jbbtnClear = new JButton(clear);
jbbtnClear.setActionCommand("Clear Breakpoint");
jbbtnClear.setToolTipText("Clear Breakpoint");

JButton jbbtnResume = new JButton(resume);
jbbtnResume.setActionCommand("Resume");
jbbtnResume.setToolTipText("Resume");

// Включение кнопок в состав панели инструментов.
jtb.add(jbbtnSet);
jtb.add(jbbtnClear);
jtb.add(jbbtnResume);

// Панель инструментов помещается в северную область
// панели содержимого.
jfrm.getContentPane().add(jtb, BorderLayout.NORTH);
```

Рассмотрим подробнее представленный выше код. В первую очередь в данном фрагменте создается объект `JToolBar`; для него указывается заголовок `Debug`. Затем создается набор объектов `ImageIcon`, содержащих изображения для кнопок панели инструментов. Далее формируются три кнопки. Заметьте, что на кнопках имеются изображения, но не текст. Для каждой из них задается команда действия и строка подсказки. Устанавливать команды действия необходимо, так как при создании кнопкам не присваиваются имена. Строка подсказки — полезный инструмент, помогающий в работе с кнопками, содержащими только изображения. В ряде случаев бывает трудно подобрать картинку, которая была бы интуитивно понятна каждому пользователю. Кнопки включатся

в состав панели инструментов, а сама панель помещается в северную область фрейма, управляемого диспетчером компоновки `BorderLayout`.

В завершение работы надо связать с кнопками панели инструментов обработчики событий.

```
// Установка обработчиков событий.
jbtnSet.addActionListener(this);
jbtnClear.addActionListener(this);
jbtnResume.addActionListener(this);
```

Каждый раз, когда пользователь щелкает на кнопке, расположенной на панели инструментов, генерируется событие действия. Это событие обрабатывается так же, как и события, имеющие отношение к меню. Готовая панель инструментов показана на рис. 7.6.

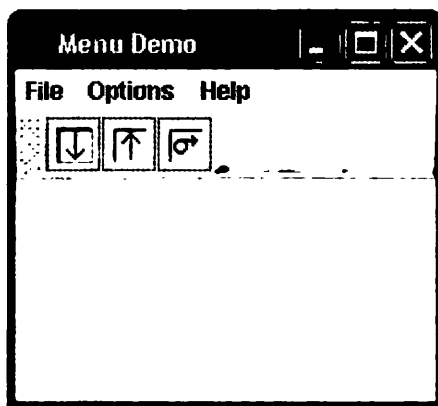


Рис. 7.6. Панель инструментов *Debug* в окне приложения

## Дополнительные возможности панелей инструментов

Компонент `JToolBar` предоставляет разработчику ряд специальных возможностей. Например, в панель инструментов можно включать разделители. Для этой цели предусмотрен метод `addSeparator()`. Два варианта данного метода приведены ниже.

```
void addSeparator()
```

```
void addSeparator(Dimension dim)
```

Первый вариант метода включает разделитель, размеры которого определяются используемым стилем. Второй позволяет явно задавать размер.

Запретить перемещение панели инструментов можно, вызвав метод `setFloatable()`.

```
void setFloatable(boolean canFloat)
```

Если значение параметра `canFloat` равно `true`, панель можно перемещать. Значение `false` данного параметра фиксирует позицию панели. По умолчанию перемещение панели инструментов разрешено.

Кнопка панели инструментов может отслеживать факт помещения на нее курсора мыши. Для этой цели предусмотрен метод `setRollover()`.

```
void setRollover(boolean on)
```

Если значение параметра `on` равно `true`, внешний вид кнопки на панели инструментов изменяется в том случае, если на нее наведен курсор мыши. Значение `false` данного параметра отменяет данный эффект. При использовании некоторых стилей компоненты никогда не реагируют на наведение на них курсора мыши.



## Вопросы для текущего контроля

1. Панели инструментов никогда не дублируют функциональные возможности меню. Да или нет?
2. С каким диспетчером компоновки обычно связывается фрейм, содержащий панель инструментов?
3. Как предотвратить перемещение панели инструментов в новую позицию?

**ВАЖНО!**

## 7.9 Действия

Очень часто кнопки панели инструментов и пункты меню выполняют одинаковые действия. Например, в рассмотренном ранее примере можно создать пункты меню, позволяющие обращаться к функциям, доступным посредством панели инструментов **Debug**. В этом случае активизация кнопки и выбор соответствующего пункта дадут один и тот же результат (например, установят точ-

1. Нет. Кнопки на панели инструментов, как правило, дублируют пункты меню.
2. `BorderLayout`. При этом панель инструментов обычно помещается в северную область (`NORTH`).
3. С помощью вызова `setFloatable(false)`.

ки останова). Часто для кнопки панели инструментов и для пункта меню используется одна и та же пиктограмма. Более того, когда доступ к кнопке панели инструментов запрещается, соответствующий пункт меню также должен быть недоступным. Чтобы учесть подобные условия, необходимо создать большой объем дублирующегося кода, который наверняка будет далек от оптимального. Решением подобной проблемы является специальное средство, предоставляемое Swing, которое называется *действием*.

Действие — это экземпляр класса, реализующего интерфейс `Action`. Интерфейс `Action` расширяет интерфейс `ActionListener` и предоставляет средства для объединения информации о состоянии с обработчиком событий `actionPerformed()`. Это позволяет управлять несколькими компонентами посредством одного действия. Например, действие дает возможность реализовать централизованный контроль над кнопкой панели инструментов и пунктом меню. Вместо того чтобы дублировать код, достаточно создать в программе действие, которое будет автоматически управлять обоими компонентами.

Действие инкапсулирует следующие возможности.

- Управление клавишами быстрого доступа.
- Управление мнемоническими обозначениями.
- Поддержка имени.
- Поддержка пиктограммы.
- Управление строкой подсказки.
- Поддержка описания.
- Поддержка команды действия.
- Управление состоянием, соответствующим разрешению или запрету доступа.

Эти свойства представляют собой информацию о состоянии, которая может быть использована в процессе выполнения программы.

Как было сказано выше, интерфейс `Action` расширяет интерфейс `ActionListener`. Таким образом, объект действия должен реализовывать метод `actionPerformed()`. Данный обработчик обрабатывает события действия, генерируемые объектами, связанными с действием.

Кроме унаследованного метода `actionPerformed()`, в интерфейсе `Action` также определено несколько дополнительных методов. Они перечислены в табл. 7.8. Обратите внимание на методы `putValue()` и `getValue()`. Эти методы позволяют устанавливать или получать значения различных свойств, связанных с действием. Ключ, передаваемый им с помощью параметра `key`, представляет требуемое свойство. Ключевые значения показаны в табл. 7.9.



Например, для того, чтобы установить в качестве мнемонического обозначения букву X, надо использовать следующий вызов `putValue()`:

```
actionOb.putValue(MNEMONIC_KEY, new Integer(KeyEvent.VK_X));
```

Таблица 7.8. Методы, объявленные в интерфейсе `Action`

| Метод                                                                      | Описание                                                                                                                                                    |
|----------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void addPropertyChangeListener(PropertyChangeListener pcl)</code>    | Устанавливает обработчик изменения свойств, заданный с помощью параметра <code>pcl</code>                                                                   |
| <code>Object getValue(String key)</code>                                   | Возвращает ссылку на свойство, заданную посредством параметра <code>key</code>                                                                              |
| <code>boolean isEnabled()</code>                                           | Возвращает значение <code>true</code> , если действия разрешено, и <code>false</code> — в противном случае                                                  |
| <code>void putValue(String key, Object val)</code>                         | Связывает объект <code>val</code> со свойством, заданным с помощью параметра <code>key</code>                                                               |
| <code>void removePropertyChangeListener(PropertyChangeListener pcl)</code> | Удаляет обработчик изменения свойств, заданный с помощью параметра <code>pcl</code>                                                                         |
| <code>void setEnabled(boolean enabled)</code>                              | Если значение параметра <code>enabled</code> равно <code>true</code> , действие разрешено. Значение <code>false</code> данного параметра запрещает действие |

Таблица 7.9. Ключевые значения, определенные в классе `Action`

| Ключ                                          | Описание                                                                                                                                      |
|-----------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| <code>static String ACCELERATOR_KEY</code>    | Представляет свойство, соответствующее клавишам быстрого доступа. Клавиши быстрого доступа задаются с помощью объектов <code>KeyStroke</code> |
| <code>static String ACTION_COMMAND_KEY</code> | Представляет свойство, соответствующее команде действия. Команда действия задается в виде строки                                              |
| <code>static String DEFAULT</code>            | Не используется                                                                                                                               |
| <code>static String LONG_DESCRIPTION</code>   | Представляет длинное описание действия. Описание задается в виде строки                                                                       |
| <code>static String MNEMONIC_KEY</code>       | Представляет свойство, соответствующее мнемоническому обозначению. Мнемоническое обозначение задается в виде константы <code>KeyEvent</code>  |
| <code>static String NAME</code>               | Представляет имя действия (оно становится именем кнопки и пункта меню, связанных с действием). Имя задается в виде строки                     |

## Окончание таблицы 7.9

| Ключ                                         | Описание                                                                                               |
|----------------------------------------------|--------------------------------------------------------------------------------------------------------|
| <code>static String SHORT_DESCRIPTION</code> | Представляет подсказку, связанную с действием. Текст подсказки задается в виде строки                  |
| <code>static String SMALL_ICON</code>        | Представляет пиктограмму, связанную с действием. Пиктограмма задается в виде объекта <code>Icon</code> |

Среди свойств `Action` есть одно, не доступное посредством методов `putValue()` и `getValue()`. Это свойство разрешает или запрещает доступ к соответствующему элементу. Для этой цели предусмотрены методы `setEnabled()` и `isEnabled()`. Заметьте, что объект `Action` поддерживает событие `PropertyChangeEvent`. Любой класс, реализующий интерфейс `Action`, может генерировать событие данного типа при изменении свойств.

Интерфейс `Action` можно реализовать вручную, однако в большинстве случаев в этом нет необходимости. Swing предоставляет частичную реализацию данного интерфейса — класс `AbstractAction`. Свой класс вы можете объявить как подкласс данного класса. Расширяя `AbstractAction`, следует объявить только метод `actionPerformed()`. Остальные методы, объявленные в интерфейсе `Action`, реализуются суперклассом. В классе `AbstractAction` определены конструкторы, перечисленные в табл. 7.10.

Таблица 7.10. Конструкторы класса `AbstractAction`

| Конструктор                                          | Описание                                                                                                                                                                         |
|------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>AbstractAction()</code>                        | Создает объект по умолчанию                                                                                                                                                      |
| <code>AbstractAction(String name)</code>             | Создает объект <code>AbstractAction</code> с именем, заданным посредством параметра <code>name</code>                                                                            |
| <code>AbstractAction(String name, Icon image)</code> | Создает объект <code>AbstractAction</code> . С ним связывается имя, заданное посредством параметра <code>name</code> , и изображение, определяемое параметром <code>image</code> |

Созданное действие надо включить в состав `JToolBar` и использовать при формировании `JMenuItem`. Для добавления действия к `JToolBar` используется следующий вариант метода `add()`:

```
void add(Action actObj)
```

Здесь параметр `actObj` определяет действие, добавляемое к панели инструментов. Свойства, определенные посредством `actObj`, используются для

создания кнопки, располагаемой на панели инструментов. Для формирования пункта меню на основе действия применяется конструктор

```
JMenuItem(Action actObj)
```

Здесь параметр `actObj` — это действие, используемое для создания пункта, учитывающего его свойства.

На заметку



Помимо `JToolBar` и `JMenuItem`, действия также поддерживаются некоторыми другими компонентами Swing. К их числу относятся `JPopupMenu`, `JButton`, `JRadioButton` и `JCheckBox`. Объекты `JRadioButtonMenuItem` и `JCheckBoxMenuItem` также поддерживают действия.

Чтобы проиллюстрировать преимущества действий, используем их для управления панелью инструментов `Debug`, созданной в предыдущем разделе. Мы также включим в состав меню `Options` подменю `Debug`. Данное подменю будет содержать пункты `Set Breakpoint`, `Clear Breakpoint` и `Resume`, выполняющие те же действия, что и соответствующие кнопки панели инструментов. Вместо того чтобы дублировать один и тот же код для панели инструментов и для меню, мы используем действия.

Начните с создания внутреннего класса `DebugAction`, являющегося подклассом класса `AbstractAction`.

```
// Класс для создания действия, управляющего меню Debug
// и панелью инструментов.
class DebugAction extends AbstractAction {
    public DebugAction(String name, Icon image, int mnem,
                       int accel, String tTip) {
        super(name, image);
        putValue(ACCELERATOR_KEY,
                 KeyStroke.getKeyStroke(accel,
                                         InputEvent.CTRL_MASK));
        putValue(MNEMONIC_KEY, new Integer(mnem));
        putValue(SHORT_DESCRIPTION, tTip);
    }

    // Обработка событий панели инструментов и меню Debug.
    public void actionPerformed(ActionEvent ae) {
        String comStr = ae.getActionCommand();

        jlab.setText(comStr + " Selected");

        // Разрешение и запрет доступа к средствам
```

```
// установки и удаления точек останова.  
if (comStr.equals("Set Breakpoint")) {  
    clearAct.setEnabled(true);  
    setAct.setEnabled(false);  
} else if (comStr.equals("Clear Breakpoint")) {  
    clearAct.setEnabled(false);  
    setAct.setEnabled(true);  
}  
}  
}
```

Класс `DebugAction` расширяет `AbstractAction`. С его помощью создается класс действия, который используется для определения свойств, связанных с меню `Debug` и панелью инструментов. Его конструктору передается пять параметров, позволяющих определять следующие характеристики:

- имя;
- пиктограмму;
- мнемоническое обозначение;
- клавиши быстрого доступа;
- строку подсказки.

Первые два параметра передаются конструктору `AbstractAction` посредством ключевого слова `super`. Остальные три используются для установки свойств путем вызова метода `putValue()`.

Метод `actionPerformed()` класса `DebugAction` обрабатывает события для действия. Это означает, что если экземпляр `DebugAction` был использован для создания кнопки панели инструментов и пункта меню, то события, генерируемые любым из этих компонентов, обрабатываются методом `actionPerformed()`, определенным в классе `DebugAction`. Заметьте, что посредством метки `jlab` данный обработчик отображает выбор пользователя. Если выбран пункт меню или активизирована кнопка установки точки останова (`Set Breakpoint`), кнопка и пункт меню, соответствующие сбросу точки останова (`Clear Breakpoint`), доступны, а обращение к `Set Breakpoint` запрещено. Если выбрана кнопка или пункт сброса точки останова (`Clear Breakpoint`), разрешается доступ к `Set Breakpoint` и запрещается обращение к `Clear Breakpoint`. Этим демонстрируется возможность разрешать и запрещать доступ к компонентам посредством действий. Когда действие запрещено, недоступны все соответствующие ему элементы. В данном случае запрет `Set Breakpoint` делает недоступными и пункт меню, и кнопку.

Включите в состав MenuDemo следующие переменные экземпляра:

```
DebugAction setAct;  
DebugAction clearAct;  
DebugAction resumeAct;
```

Создайте три объекта ImageIcons, представляющие средства отладки.

```
// Загрузка трех изображений для действий.  
ImageIcon setIcon = new ImageIcon("setBP.gif");  
ImageIcon clearIcon = new ImageIcon("clearBP.gif");  
ImageIcon resumeIcon = new ImageIcon("resume.gif");
```

Теперь создайте действия, которые будут управлять средствами отладки.

```
// Создание действий, используемых панелью инструментов  
// Debug и меню.
```

```
setAct =  
    new DebugAction("Set Breakpoint",  
                    setIcon,  
                    KeyEvent.VK_S,  
                    KeyEvent.VK_B,  
                    "Set a breakpoint.");  
  
clearAct =  
    new DebugAction("Clear Breakpoint",  
                    clearIcon,  
                    KeyEvent.VK_C,  
                    KeyEvent.VK_L,  
                    "Clear a breakpoint.");  
  
resumeAct =  
    new DebugAction("Resume",  
                    resumeIcon,  
                    KeyEvent.VK_R,  
                    KeyEvent.VK_R,  
                    "Resume execution after breakpoint.");
```

```
// Первоначально обращение к Clear Breakpoint запрещено.  
clearAct.setEnabled(false);
```

Заметьте, что для Set Breakpoint задана клавиша быстрого доступа <B>, а для Clear Breakpoint — <L>. Эти клавиши выбраны вместо <S> и <C> потому, что <S> и <C> уже поставлены в соответствие пунктам Save и Close меню File. Однако они могут быть использованы в качестве мнемонического обозначения,

так как мнемонические обозначения действуют только в пределах одного меню. Заметьте также, что действие, представляющее Clear Breakpoint, изначально запрещено. Оно станет доступным только после установки точки останова.

Используйте подготовленные действия для создания кнопок, а затем включите их в состав панели инструментов.

```
// Создание кнопок панели инструментов с использованием действий.  
JButton jbbtnSet = new JButton(setAct);  
JButton jbbtnClear = new JButton(clearAct);  
JButton jbbtnResume = new JButton(resumeAct);
```

```
// Создание панели инструментов Debug.  
JToolBar jtb = new JToolBar("Breakpoints");
```

```
// Включение кнопок в состав панели инструментов.  
jtb.add(jbbtnSet);  
jtb.add(jbbtnClear);  
jtb.add(jbbtnResume);
```

```
// Размещение панели инструментов в северной области  
// панели содержимого.  
jfrm.getContentPane().add(jtb, BorderLayout.NORTH);
```

Создайте меню Debug с помощью приведенного ниже фрагмента кода.

```
// Создание меню Debug, которое вызывается из меню Options.  
// Для создания пунктов меню используются действия.  
JMenu jmDebug = new JMenu("Debug");  
JMenuItem jmISetBP = new JMenuItem(setAct);  
JMenuItem jmIClearBP = new JMenuItem(clearAct);  
JMenuItem jmIResume = new JMenuItem(resumeAct);  
jmDebug.add(jmISetBP);  
jmDebug.add(jmIClearBP);  
jmDebug.add(jmIResume);  
jmOptions.add(jmDebug);
```

После внесения указанных изменений и дополнений созданные вами действия будут управлять меню Debug и одноименной панелью инструментов. Таким образом, изменение свойства в составе действия (например, его запрет) воздействует на все компоненты, использующие данное действие. Теперь окно программы будет выглядеть так, как показано на рис. 7.7.

## Вопросы для текущего контроля

1. В каких случаях оправдано использование действий?
2. Какой класс предоставляет частичную реализацию интерфейса Action?
3. Если вы запретите действие, станут ли все связанные с ним компоненты недоступными?

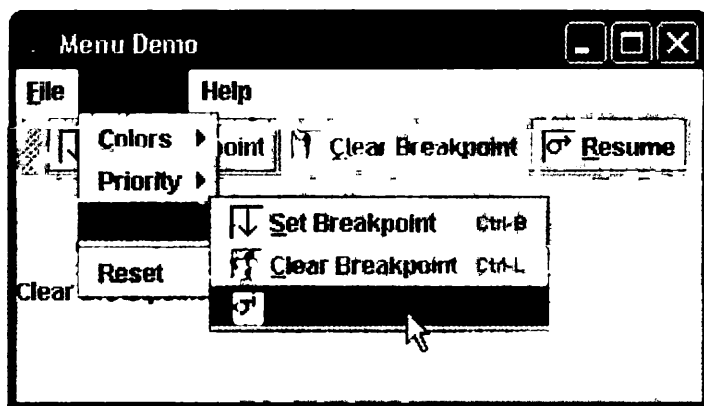


Рис. 7.7. Использование действий для управления панелью инструментов *Debug* и меню с тем же именем

**ВАЖНО!**

## 7.10. Окончательный вариант программы MenuDemo

В данном модуле мы рассмотрели программу MenuDemo, а затем обсудили много изменений и дополнений к ней. Перед тем как завершить модуль, желательно объединить все созданные в нем фрагменты кода. Это следует сделать не только для того, чтобы исключить возможные ошибки, но и для того, чтобы создать готовую демонстрационную программу, с которой можно было бы экспериментировать.

1. Действия имеет смысл использовать тогда, когда несколько компонентов, например меню и панель инструментов, предоставляют доступ к одним и тем же функциям приложения.
2. *AbstractAction*.
3. Да.

Приведенный ниже вариант класса MenuDemo включает все дополнения, описанные в данном модуле. Структура программы несколько изменена, в частности, для создания различных меню и панели инструментов используются отдельные методы. Обратите внимание на то, что некоторые переменные, имеющие отношение к меню, например `jmb`, `jmFile` и `jtb`, объявлены как переменные экземпляра.

```
// Окончательный вариант программы MenuDemo.
```

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

class MenuDemo implements ActionListener {

    JLabel jlab;

    JMenuBar jmb;

    JToolBar jtb;

    JPopupMenu jpu;

    DebugAction setAct;
    DebugAction clearAct;
    DebugAction resumeAct;

    MenuDemo() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Complete Menu Demo");

        // В данном случае используется диспетчер компоновки
        // по умолчанию.

        // Установка начальных размеров фрейма.
        jfrm.setSize(360, 200);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание метки, с помощью которой отображается
        // информация о выборе пользователя.
        jlab = new JLabel();
```



```
// Создание строки меню.
jmb = new JMenuBar();

// Создание меню File.
makeFileMenu();

// Создание действий для управления меню и
// панелью инструментов Debug.
makeActions();

// Создание панели инструментов.
makeToolBar();

// Создание меню Options.
makeOptionsMenu();

// Создание меню Help.
makeHelpMenu();

// Создание контекстного меню Edit.
makeEditPUMenu();

// Создание обработчика событий мыши, в котором будет
// проверяться признак, разрешающий вывод контекстного меню.
jfrm.getContentPane().addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent me) {
        if(me.isPopupTrigger())
            jpu.show(me.getComponent(), me.getX(), me.getY());
    }
    public void mouseReleased(MouseEvent me) {
        if(me.isPopupTrigger())
            jpu.show(me.getComponent(), me.getX(), me.getY());
    }
});

// Размещение метки в центре панели содержимого.
jfrm.getContentPane().add(jlab, SwingConstants.CENTER);

// Размещение панели инструментов в северной области
// панели содержимого.
jfrm.getContentPane().add(jtb, BorderLayout.NORTH);
```

```
// Включение строки меню в состав фрейма.
jfrm.setJMenuBar(jmb);

// Отображение фрейма.
jfrm.setVisible(true);
}

// Обработка событий действия, связанных с пунктом меню.
// Данный обработчик не поддерживает событий, генерируемых
// меню Debug и одноименной панелью инструментов.
public void actionPerformed(ActionEvent ae) {
    // Получение команды действия, определяющей
    // выбор пользователя.
    String comStr = ae.getActionCommand();

    // Если пользователь выбирает пункт меню Exit,
    // выполнение программы завершается.
    if(comStr.equals("Exit")) System.exit(0);

    // В противном случае на экране отображается информация
    // о выборе пользователя.
    jlab.setText(comStr + " Selected");
}

// Класс действия для меню Debug и одноименной
// панели инструментов.
class DebugAction extends AbstractAction {
    public DebugAction(String name, Icon image, int mnem,
                       int accel, String tTip) {
        super(name, image);
        putValue(ACCELERATOR_KEY,
                 KeyStroke.getKeyStroke(accel,
                                         InputEvent.CTRL_MASK));
        putValue(MNEMONIC_KEY, new Integer(mnem));
        putValue(SHORT_DESCRIPTION, tTip);
    }

    // Обработка событий, связанных с панелью инструментов
    // и меню Debug.
    public void actionPerformed(ActionEvent ae) {
        String comStr = ae.getActionCommand();

        jlab.setText(comStr + " Selected");
    }
}
```



```
jmFile.add(jmiOpen);
jmFile.add(jmiClose);
jmFile.add(jmiSave);
jmFile.addSeparator();
jmFile.add(jmiExit);
jmb.add(jmFile);

// Связывание обработчиков событий действия с меню File.
jmiOpen.addActionListener(this);
jmiClose.addActionListener(this);
jmiSave.addActionListener(this);
jmiExit.addActionListener(this);
}

// Создание меню Options.
void makeOptionsMenu() {
    JMenu jmOptions = new JMenu("Options");

    // Создание подменю Colors.
    JMenu jmColors = new JMenu("Colors");

    // Использование флажков опций для выбора цвета. Данный
    // подход позволяет пользователю выбирать несколько цветов.
    JCheckBoxMenuItem jmiRed = new JCheckBoxMenuItem("Red");
    JCheckBoxMenuItem jmiGreen = new JCheckBoxMenuItem("Green");
    JCheckBoxMenuItem jmiBlue = new JCheckBoxMenuItem("Blue");

    // Включение пунктов в состав меню Colors.
    jmColors.add(jmiRed);
    jmColors.add(jmiGreen);
    jmColors.add(jmiBlue);
    jmOptions.add(jmColors);

    // Создание подменю Priority.
    JMenu jmPriority = new JMenu("Priority");

    // Для выбора свойств применяется переключатель опций.
    // Благодаря использованию пунктов данного типа
    // пользователь видит, какое свойство установлено,
    // и может выбирать в каждый момент времени только
    // одно значение. Заметьте, что кнопка High
    // изначально выбрана.
    JRadioButtonMenuItem jmiHigh =
```

```
new JRadioButtonMenuItem("High", true);
JRadioButtonMenuItem jmiLow =
    new JRadioButtonMenuItem("Low");

// Включение пунктов в состав меню Priority.
jmPriority.add(jmiHigh);
jmPriority.add(jmiLow);
jmOptions.add(jmPriority);

// Объединение кнопок переключателя опций в группу
ButtonGroup bg = new ButtonGroup();
bg.add(jmiHigh);
bg.add(jmiLow);

// Создание подменю Debug, входящего в состав
// меню Options. Для создания пунктов меню используются
// действия.
JMenu jmDebug = new JMenu("Debug");
JMenuItem jmiSetBP = new JMenuItem(setAct);
JMenuItem jmiClearBP = new JMenuItem(clearAct);
JMenuItem jmiResume = new JMenuItem(resumeAct);

// Включение пунктов в состав меню Debug.
jmDebug.add(jmiSetBP);
jmDebug.add(jmiClearBP);
jmDebug.add(jmiResume);
jmOptions.add(jmDebug);

// Создание пункта меню Reset.
JMenuItem jmiReset = new JMenuItem("Reset");
jmOptions.addSeparator();
jmOptions.add(jmiReset);

// Связывание меню Options со строкой меню.
jmb.add(jmOptions);

// Связывание обработчиков событий действия
// с меню Options. Эти обработчики не поддерживают
// событий меню Debug.
jmiRed.addActionListener(this);
jmiGreen.addActionListener(this);
jmiBlue.addActionListener(this);
jmiHigh.addActionListener(this);
```

```
jmiLow.addActionListener(this);
jmiReset.addActionListener(this);
}

// Создание меню Help.
void makeHelpMenu() {
    JMenu jmHelp = new JMenu("Help");

    // Связывание пиктограммы с пунктом меню About.
    ImageIcon icon = new ImageIcon("AboutIcon.gif");

    JMenuItem jmiAbout = new JMenuItem("About", icon);
    jmiAbout.setToolTipText("Info about the MenuDemo program.");
    jmHelp.add(jmiAbout);
    jmb.add(jmHelp);

    // Определение обработчика событий для About.
    jmiAbout.addActionListener(this);
}

// Формирование действий для меню Debug и
// панели инструментов.
void makeActions() {
    // Загрузка изображений для действий.
    ImageIcon setIcon = new ImageIcon("setBP.gif");
    ImageIcon clearIcon = new ImageIcon("clearBP.gif");
    ImageIcon resumeIcon = new ImageIcon("resume.gif");

    // Создание действий.
    setAct =
        new DebugAction("Set Breakpoint",
                        setIcon,
                        KeyEvent.VK_S,
                        KeyEvent.VK_B,
                        "Set a breakpoint.");

    clearAct =
        new DebugAction("Clear Breakpoint",
                        clearIcon,
                        KeyEvent.VK_C,
                        KeyEvent.VK_L,
                        "Clear a breakpoint.");
```

```
resumeAct =
    new DebugAction("Resume",
                    resumeIcon,
                    KeyEvent.VK_R,
                    KeyEvent.VK_R,
                    "Resume execution after breakpoint.");

// Первоначально обращение к Clear Breakpoint запрещено.
clearAct.setEnabled(false);
}

// Создание панели инструментов Debug.
void makeToolBar() {
    // Создание кнопок панели инструментов
    // с использованием действий.
    JButton jbbtnSet = new JButton(setAct);
    JButton jbbtnClear = new JButton(clearAct);
    JButton jbbtnResume = new JButton(resumeAct);

    // Создание панели инструментов Debug.
    jtb = new JToolBar("Breakpoints");

    // Включение кнопок в состав панели инструментов.
    jtb.add(jbbtnSet);
    jtb.add(jbbtnClear);
    jtb.add(jbbtnResume);
}

// Создание контекстного меню Edit.
void makeEditPUMenu() {
    jpu = new JPopupMenu();

    // Создание пунктов контекстного меню
    JMenuItem jmiCut = new JMenuItem("Cut");
    JMenuItem jmiCopy = new JMenuItem("Copy");
    JMenuItem jmiPaste = new JMenuItem("Paste");

    // Включение пунктов в состав контекстного меню.
    jpu.add(jmiCut);
    jpu.add(jmiCopy);
    jpu.add(jmiPaste);
}
```

```
// Связывание обработчика событий с пунктами
// контекстного меню Edit.
jmiCut.addActionListener(this);
jmiCopy.addActionListener(this);
jmiPaste.addActionListener(this);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new MenuDemo();
        }
    });
}
```



## Тест для самоконтроля по модулю 7

1. Назовите основные классы Swing, используемые для создания меню.
2. С помощью какого класса создается меню? Какой класс позволяет создать контекстное меню? Какой класс следует использовать для создания строки меню?
3. Какое событие генерируется при выборе пользователем пункта меню?
4. В состав пунктов меню нельзя включать изображения. Да или нет?
5. Какой метод включает строку меню в окно?
6. С помощью какого метода можно определить мнемоническое обозначение для пункта меню?
7. Может ли пиктограмма использоваться в качестве пункта меню? Если да, запрещает ли она использовать имя пункта?
8. Какой класс создает пункт меню в виде кнопки переключателя опций?
9. В составе меню могут присутствовать флажки опций, однако они ухудшают интерфейс, так как при этом меню выглядит непривычно для пользователя. Да или нет?
10. Контекстное меню обычно вызывается по \_\_\_\_\_ мышью.
11. Какой метод, объявленный в интерфейсе `MouseListener`, следует переопределить для вызова контекстного меню? Какой метод класса



`MouseEvent` следует вызвать, чтобы определить, следует ли отображать контекстное меню?

12. Какой метод надо вызвать, чтобы получить ссылку на компонент, вызывающий контекстное меню?
13. Экземпляр какого класса реализует панель инструментов?
14. Можно ли связать панель инструментов со строкой меню?
15. Какой интерфейс используется для определения действия?
16. Перечислите свойства, определенные посредством действия?
17. Какие действия выполняет метод `System.exit()`?
18. Определите самостоятельно мнемонические обозначения и клавиши быстрого доступа для остальных пунктов меню в окончательной версии программы `MenuDemo`.
19. Создайте самостоятельно меню `View` и в окончательной версии программы `MenuDemo` свяжите его со строкой меню. Данное меню должно содержать три пункта: **Full Screen**, **Normal** и **Thumbnail**. Добавьте к панели инструментов кнопки, вызывающие те же функции. Используйте действия для управления кнопками панели инструментов и пунктами меню.

# Модуль 8

## Таблицы и деревья

- 8.1.** Общие сведения о компоненте `JTable`
- 8.2.** Режимы выбора
- 8.3.** Выбор столбцов и ячеек таблицы
- 8.4.** Обработка событий выбора
- 8.5.** Обработка событий модели таблицы
- 8.6.** Изменение размеров таблицы
- 8.7.** Модель таблицы, определяемая разработчиком
- 8.8.** Дополнительные возможности, предоставляемые таблицами
- 8.9.** Общие сведения о компоненте `JTree`
- 8.10.** Обработка событий, связанных с деревьями
- 8.11.** Дополнительные возможности, предоставляемые древовидными структурами

Данная глава посвящена двум наиболее сложным компонентам Swing: `JTable` и `JTree`. Компонент `JTable` позволяет отображать данные в виде двумерной таблицы и управлять ими, а компонент `JTree` дает возможность организовать данные в виде дерева. Оба компонента предоставляют мощные средства управления информацией; их применение позволит существенно повысить качество интерфейса прикладной программы. Для того чтобы подробно описать каждый из этих компонентов, потребовалась бы отдельная книга, поэтому мы ограничимся лишь поверхностным их рассмотрением.

**ВАЖНО!**

## **8.1. Общие сведения о компоненте `JTable`**

Из всех компонентов библиотеки Swing самым мощным, наверное, является `JTable`. Он позволяет создавать таблицы, отображать их и управлять ими. Таблицы чрезвычайно важны для корпоративных приложений, поскольку часто используются для отображения содержимого баз данных. В виде таблицы можно представлять самые разнообразные сведения. Например, такой формат удобен для отображения списка почтовых сообщений, записей из адресной книги, информации о состоянии заказа и т.д. Независимо от особенностей использования, таблицы — важная часть графического интерфейса, создаваемого средствами Swing.

Подобно другим компонентам Swing, `JTable` принадлежит пакету `javax.swing`. Однако многие из классов и интерфейсов, используемых для поддержки таблиц, находятся в пакете `javax.swing.table`. Поскольку количество таких классов и интерфейсов довольно велико, для них выделен отдельный пакет. Не следует, однако, считать, что для работы с таблицами необходимо владеть всеми вспомогательными средствами. Как вы вскоре увидите, создать простую таблицу и управлять ею достаточно просто, не сложнее, чем работать с другими компонентами Swing. По мере усложнения таблицы все большую часть работы по управлению ею приходится производить вручную. Но, к счастью, основные действия, связанные с настройкой таблиц, не требуют дополнительных усилий.

Базовые средства компонента `JTable` достаточно просты. Этот компонент организует информацию в виде одного или нескольких столбцов. В верхней части каждого столбца располагается заголовок. Помимо описания данных, заголовки выполняют еще одну функцию: с их помощью можно изменять размеры столбцов, переставлять столбцы в таблице и отображать строки подсказки.

Информация, отображаемая с помощью таблицы, содержится в ее ячейках. С ячейками связываются *объект воспроизведения*, который определяет, как должны отображаться данные, а также *редактор ячейки*, который задает правила

редактирования информации пользователем. Компонент `JTable` предоставляет объект воспроизведения и редактор по умолчанию, которые могут в неизменном виде использоваться во многих приложениях. При необходимости можно также связать с компонентом собственный объект воспроизведения и редактор.

Компонент `JTable` не поддерживает прокрутку. Вместо этого таблицы обычно включаются в состав компонента `JScrollPane`. Несмотря на то что содержимое таблицы может прокручиваться вверх и вниз, заголовки столбцов автоматически отображаются на экране. Если вы не включите `JTable` в контейнер `JScrollPane`, вам придется позаботиться об отображении как таблицы, так и ее заголовков.

Поместив таблицу в панель с прокруткой, желательно установить предпочтительный размер *прокручиваемой области просмотра* таблицы. Область просмотра определяет пространство в пределах таблицы, в которых данные отображаются и прокручиваются. Она не включает заголовков таблиц. Если вы не установите размер области просмотра, будет принято значение по умолчанию, которое, вероятнее всего, не удовлетворит требованиям вашего приложения. Для того чтобы задать размер прокручиваемой области просмотра, надо вызвать метод `setPreferredScrollableViewportSize()`:

```
setPreferredScrollableViewportSize(Dimension dim)
```

где параметр `dim` задает требуемый размер прокручиваемой области.

Компонент `JTable` базируется на трех моделях. Первая — это модель таблицы, определяемая с помощью интерфейса `TableModel`. Эта модель управляет отображением данных в виде двумерной структуры. Вторая — модель столбца, представляемая с помощью интерфейса `TableColumnModel`. Модель столбца непосредственно влияет на структуру таблицы. Эти две модели принадлежат пакету `javax.swing.table`.

Третья модель определяет порядок выбора пунктов и задается с помощью интерфейса `ListSelectionModel` (см. модуль 5), который расположен в пакете `javax.swing`. По умолчанию `JTable` позволяет пользователю выбирать одну или несколько строк. Это поведение можно изменить двумя способами. Во-первых, можно предоставить пользователю возможность выделять столбцы или отдельные ячейки. Во-вторых, можно ограничить возможности пользователя выбором одного объекта. Необходимые для этого действия будут рассмотрены далее в данном модуле.

Таблицы генерируют ряд событий. Наиболее важными являются событие `ListSelectionEvent`, которое возникает при выборе строки столбца или ячейки таблицы, и событие `TableModelEvent`, генерируемое при изменении данных. Оба события будут подробно рассмотрены в следующем разделе. Ком-

пункт `JTable` также генерирует событие `TableColumnModelEvent`, которое соответствует изменению модели столбца.

Конструкторы, предоставляемые `JTable`, перечислены в табл. 8.1. Наиболее сложные таблицы требуют явного указания одной или нескольких моделей, но необходимость в этом возникает не всегда. Два конструктора класса `JTable` автоматизируют процесс создания простых таблиц. Первый из них выглядит следующим образом:

```
JTable(Object[][] data, Object[] headerNames)
```

Таблица 8.1. Конструкторы класса `JTable`

| Конструктор                                                                      | Описание                                                                                                                                                                                                                                                                                                                                                                                                                         |
|----------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JTable()</code>                                                            | Создает пустую таблицу, использующую модели по умолчанию                                                                                                                                                                                                                                                                                                                                                                         |
| <code>JTable(Object[][] data, Object[] headerNames)</code>                       | Создает таблицу, которая содержит данные, указанные посредством параметра <code>data</code> , и заголовки столбцов, заданные посредством параметра <code>headerNames</code> . С таблицей связываются модель выбора и модель столбца по умолчанию                                                                                                                                                                                 |
| <code>JTable(Vector data, Vector headerNames)</code>                             | Создает таблицу, которая содержит данные, указанные посредством параметра <code>data</code> , и заголовки столбцов, заданные посредством параметра <code>headerNames</code> . Параметр <code>data</code> должен представлять собой вектор, составленный из элементов типа <code>Vector</code> . В этих элементах содержатся данные для отображения в ячейках. С таблицей связываются модель выбора и модель столбца по умолчанию |
| <code>JTable(int rows, int cols)</code>                                          | Создает пустую таблицу, число строк которой определяется с помощью параметра <code>rows</code> , а число столбцов — посредством параметра <code>cols</code> . С таблицей связываются модель таблицы, модель выбора и модель столбца по умолчанию                                                                                                                                                                                 |
| <code>JTable(TableModel tm)</code>                                               | Создает таблицу, которая использует модель таблицы, заданную посредством параметра <code>tm</code> . С таблицей связываются модель выбора и модель столбца по умолчанию                                                                                                                                                                                                                                                          |
| <code>JTable(TableModel tm, TableColumnModel tcm)</code>                         | Создает таблицу, которая использует модель таблицы, заданную посредством параметра <code>tm</code> , и модель столбца, заданную с помощью <code>tcm</code> . С таблицей связываются модель выбора по умолчанию                                                                                                                                                                                                                   |
| <code>JTable(TableModel tm, TableColumnModel tcm, ListSelectionModel lsm)</code> | Создает таблицу, которая использует модель таблицы, заданную посредством параметра <code>tm</code> , и модель столбца, заданную с помощью <code>tcm</code> , и модель выбора, определяемую параметром <code>lsm</code>                                                                                                                                                                                                           |

Этот конструктор автоматически создает таблицу для данных, указанных посредством параметра `data`; заголовки столбцов задаются с помощью параметра `headerNames`. С таблицей связывается объект воспроизведения и редактор по умолчанию. Для большинства приложений возможности, предоставляемые ими, оказываются достаточными. Размерность двумерного массива `data` определяется числом строк таблицы и числом элементов в каждой строке. Например, приведенный ниже массив содержит данные для таблицы, состоящей из семи строк, каждая из которых содержит по девять элементов.

```
Object[][] info = new Object[7][9];
```

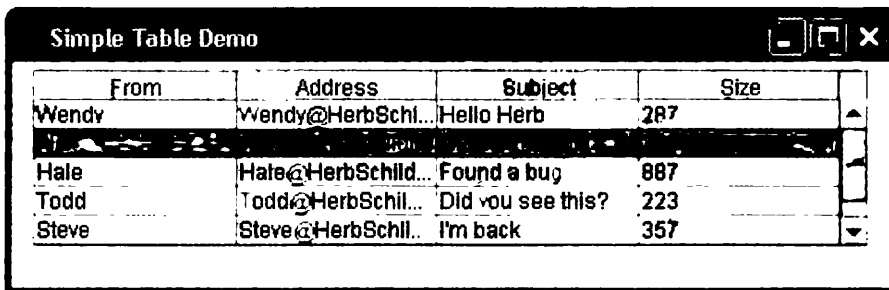
Число элементов в каждой строке таблицы должна быть равна длине массива `headerNames`. Описанный выше конструктор — самый простой инструмент для создания таблицы, содержащей фиксированный набор данных. Если структура данных сравнительно проста, применение данного конструктора является наилучшим решением.

В некоторых ситуациях, в особенности тогда, когда данные извлекаются из набора, проще использовать при создании компонента `JTable` вектор, а не массив. В этом случае следует воспользоваться следующим конструктором:

```
JTable(Vector data, Vector headerNames)
```

где параметр `data` — это вектор, составленный из элементов `Vector`. Каждый вектор может содержать число элементов, равное числу заголовков, заданных посредством параметра `headerNames`. Данный конструктор отличается от рассмотренного ранее лишь типом параметров. Эти два конструктора дают возможность достаточно легко создавать таблицы, пригодные для использования во многих приложениях.

Применим изложенный материал при создании примера. Ниже приведен код программы, которая создает простую таблицу, предназначенную для отображения списка почтовых сообщений. Эта программа не реагирует на действия



| From  | Address            | Subject           | Size |
|-------|--------------------|-------------------|------|
| Wendy | Wendy@HerbSchil... | Hello Herb        | 287  |
| Hale  | Hale@HerbSchil...  | Found a bug       | 887  |
| Todd  | Todd@HerbSchil...  | Did you see this? | 223  |
| Steve | Steve@HerbSchil..  | I'm back          | 357  |

Рис. 8.1. Таблица, сформированная при выполнении программы `TableDemo`

пользователя, а лишь отображает данные. (События, связанные с таблицами, будут рассмотрены в следующем разделе.) Окно, создаваемое при работе программы, показано на рис. 8.1.

// Пример формирования простой таблицы.

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

class TableDemo {

    // Массив, содержащий заголовки таблицы.
    String[] headings = { "From", "Address", "Subject", "Size" };

    // Массив, содержащий данные таблицы.
    Object[][] data = {
        { "Wendy", "Wendy@HerbSchildt.com",
          "Hello Herb", new Integer(287) },
        { "Alex", "Alex@HerbSchildt.com",
          "Check this out!", new Integer(308) },
        { "Hale", "Hale@HerbSchildt.com",
          "Found a bug", new Integer(887) },
        { "Todd", "Todd@HerbSchildt.com",
          "Did you see this?", new Integer(223) },
        { "Steve", "Steve@HerbSchildt.com",
          "I'm back", new Integer(357) },
        { "Ken", "Ken@HerbSchildt.com",
          "Arrival time change", new Integer(512) }
    };

    JTable jtabEmail;

    TableDemo() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Simple Table Demo");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
```

```

jfrm.setSize(500, 160);

// Завершение программы при закрытии окна пользователем.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Создание таблицы для отображения почтовых сообщений.
// Заголовки и данные задаются посредством массивов.
jtabEmail = new JTable(data, headings);

// Включение таблицы в состав панели с прокруткой.
JScrollPane jsclrp = new JScrollPane(jtabEmail);

// Установка размеров прокручиваемой области просмотра.
jtabEmail.setPreferredScrollableViewportSize(
    new Dimension(450, 80));

// Включение таблицы в состав панели содержимого.
jfrm.getContentPane().add(jsclrp);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new TableDemo();
        }
    });
}
}

```

Несмотря на то что объем кода невелик (и в нем не предусмотрена обработка событий), в программе реализуется полнофункциональная таблица, которая позволяет изменять ширину столбцов и их взаимное расположение. Например, на рис. 8.2 показано, как будет выглядеть эта таблица, после того, как ширина столбца Address будет уменьшена, а порядок следования столбцов изменен. Также таблица дает возможность редактировать содержимое ячеек.

Рассмотрим подробнее процесс создания таблицы. Во-первых, обратите внимание на то, что в начале программы создаются и инициализируются два массива. Первый, heading, представляет собой массив строк, определяющих



| Address  | Size | Subject           | From  |
|----------|------|-------------------|-------|
| Wendy..  | 87   | Hello Herb        | Wendy |
| Alex@H.  | 308  | Check this out!   | Alex  |
| Hale@... | 887  | Found a bug       | Hale  |
| Todd@    | 223  | Did you see this? | Todd  |
| Steve@   | 357  | I'm back          | Steve |

Рис. 8.2. Таблица после изменения ширины столбца Address и перестановки столбцов

заголовки таблицы. Поскольку этот массив состоит из четырех элементов, таблица содержит четыре столбца. Второй массив, `data`, двумерный. Его элементы типа `Object` содержат данные, отображаемые в составе таблицы. Каждая строка массива преобразуется в строку данных в таблице. Как было сказано ранее, в каждой строке таблицы должно содержаться точно такое же количество элементов, как и в массиве заголовков.

Для формирования таблицы массивы `data` и `heading` передаются в качестве параметров конструктору `JTable()`.

```
JTable(Object[][] data, Object[] headerNames)
```

С полученной в результате таблицей автоматически связывается модель таблицы. Кроме того, в ней используются модель столбца и модель выбора по умолчанию, а также объект воспроизведения ячеек и редактор. Данные в каждой ячейке отображаются в формате строки, принятом по умолчанию. Приведенный выше конструктор хорошо подходит для сравнительно простых таблиц, для которых не требуются сложные средства форматирования.

После создания таблицы она помещается в состав панели содержимого, и для нее устанавливаются размеры прокручиваемой области просмотра. Таким образом, при наличии массивов с данными и заголовками для формирования готовой таблицы требуется выполнить лишь три действия. Это впечатляет, в особенности, если учесть богатые функциональные возможности компонента `JTable`. Как вы узнаете из следующих разделов, в некоторых случаях для настройки таблицы приходится применять значительные усилия, однако основные действия по ее созданию остаются минимальными.

## Вопросы для текущего контроля

1. Компонент `JTable` базируется на трех моделях. Что это за модели?
2. Обеспечивает ли компонент `JTable` автоматическую прокрутку данных?
3. С ячейками таблицы связаны \_\_\_\_\_ и \_\_\_\_\_.

## Спросим у опытного программиста

**Вопрос.** Вы говорили о том, что если не включить компонент `JTable` в состав `JScrollPane`, заголовки не будут отображаться автоматически. Можно ли рассмотреть этот вопрос более подробно?

**Ответ.** `JTable` — очень гибкий компонент, его можно настроить так, чтобы он удовлетворял самым различным требованиям. Обычно он помещается в состав панели с прокруткой, но такое решение нельзя рассматривать как единственно возможное. Если вы включите компонент `JTable` непосредственно в панель содержимого, таблица будет отображаться без заголовков. Чтобы вывести заголовки, их надо явным образом включить в панель содержимого. Такой подход обеспечивает необходимую гибкость, например, позволяет использовать таблицу без заголовков (для таблиц небольшого размера назначение столбцов часто бывает очевидным). Ниже показано, как выглядит таблица с почтовыми сообщениями, не включенная в панель с прокруткой.

Simple Table Demo Without Scroll Pane

|       |               |                |     |
|-------|---------------|----------------|-----|
| Wendy | Wendy@Herb... | Hello Herb     | 287 |
| Alex  | Alex@Herb...  | Check this ... | 308 |
| Hale  | Hale@Herb...  | Found a bug    | 887 |
| Todd  | Todd@Herb...  | Did you se...  | 223 |
| Steve | Steve@Herb... | I'm back       | 357 |
| Ken   | Ken@Herb...   | Arrival time   | 512 |

1. `TableModel`, `ListSelectionModel` и `TableColumnModel`.
2. Нет. Если прокрутка необходима, надо включить `JTable` в состав `JScrollPane`.
3. С ячейками таблицы связываются объект воспроизведения и редактор ячейки.

Чтобы добавить заголовки, их надо сначала извлечь, вызвав метод `getTableHeader()`.

```
JTableHeader getTableHeader()
```

Этот метод возвращает ссылку на компонент `JTableHeader`, управляющий заголовками таблицы.

Если вы отказываетесь от использования панели с прокруткой, целесообразнее всего поместить таблицу и заголовок в контейнер, с которым связан диспетчер компоновки `BorderLayout`. При этом таблицу надо включить в центральную, а заголовки — в северную область. Однако поступать таким образом вовсе не обязательно. Отделив заголовки столбцов от таблицы, вы открываете для себя широкие возможности по их использованию. Так, например, вы можете поместить заголовки ниже таблицы. Для того чтобы сделать это, модифицируйте программу `TableDemo`, рассмотренную в начале данного модуля. Вначале удалите выражения, ответственные за включение `jtabEmail` в панель с прокруткой, а затем добавьте следующую строку:

```
JTableHeader jth = jtabEmail.getTableHeader();
```

После этого поместите в панель содержимого заголовки столбцов. Сделать это позволяет следующий фрагмент кода:

```
// Включение таблицы и заголовков в панель
// содержимого.
```

```
jfrm.getContentPane().add(jtabEmail);
```

```
jfrm.getContentPane().add(jth);
```

В результате данных изменений таблица будет выглядеть так, как показано ниже.

| Table Header Below |              |                  |      |
|--------------------|--------------|------------------|------|
| Wendy              | Wendy@H...   | Hello Herb       | 287  |
| Alex               | Alex@Herb... | Check this ...   | 308  |
| Hale               | Hale@Her...  | Found a bug      | 887  |
| Todd               | Todd@Her...  | Did you se...    | 223  |
| Steve              | Steve@Her... | I'm back         | 357  |
| Ken                | Ken@Herb...  | Arrival time ... | 512  |
| From               | Address      | Subject          | Size |

Поэкспериментируйте с взаимным расположением таблицы и заголовков. Компонент `JTable` предоставляет возможности для этого.

**ВАЖНО!**

## 8.2. Режимы выбора

Как видно из предыдущего примера, по умолчанию компонент `JTable` позволяет пользователю выбирать одну или несколько строк. Однако можно ограничить возможности пользователя выбором лишь одной строки. Чтобы сделать это, надо изменить режим выбора, вызвав метод `setSelectionMode()`, определенный в классе `JTable`. Заголовок этого метода имеет следующий вид:

```
void setSelectionMode(int mode)
```

Здесь параметр `mode` задает режим выбора. Его значение должно быть равно одной из следующих констант, объявленных в интерфейсе `ListSelectionModel`.

|                                          |                                                                                                                       |
|------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| <code>SINGLE_SELECTION</code>            | Допускается выбор одной строки, столбца или ячейки                                                                    |
| <code>SINGLE_INTERVAL_SELECTION</code>   | Допускается выбор строк, столбцов или ячеек, лежащих в одном диапазоне                                                |
| <code>MULTIPLE_INTERVAL_SELECTION</code> | Допускается выбор строк, столбцов или ячеек, принадлежащих разным диапазонам. Этот режим устанавливается по умолчанию |

Следует помнить, что вновь установленный режим влияет на выбор как строк, так и столбцов.

Чтобы увидеть, как проявляется установка режима выбора `SINGLE_SELECTION`, включите в программу `TableDemo` следующую строку кода:

```
jtabEmail.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

После того как указанные изменения будут внесены, вы сможете выбирать в каждый момент времени лишь одну строку.

**ВАЖНО!**

## 8.3. Выбор столбцов и ячеек

В ряде случаев желательно предоставить пользователю возможность выбирать не строки, а столбцы таблицы или ее отдельные ячейки. Например, необходимость в выборе столбца может возникнуть для того, чтобы одновременно

преобразовывать все содержащиеся в этом столбце данные. Механизмы выбора столбцов и выбора отдельных ячеек очень похожи. В то же время они не идентичны, поэтому будут рассматриваться отдельно.

## Выбор столбцов

Процедура, призванная разрешить выбор столбцов, выполняется в два этапа. В первую очередь надо вызвать метод `setColumnSelectionAllowed()` и передать ему параметр `true`. Этот метод определяется следующим образом:

```
void setColumnSelectionAllowed(boolean enabled)
```

Если значение параметра `enabled` равно `true`, выбор столбцов разрешен. Значение `false` отменяет эту возможность.

Второй этап состоит в отключении режима выбора строк. Для этого надо передать значение `false` методу `setRowSelectionAllowed()`, заголовок которого приведен ниже.

```
void setRowSelectionAllowed(boolean enabled)
```

Если значение параметра `enabled` равно `true`, выбор строк разрешен. Значение `false` запрещает выбирать строки.

После вызова этих двух методов по щелчку на таблице будет выбрана не строка, а столбец. Чтобы проверить, так ли это, добавьте к предыдущей программе приведенные ниже строки.

```
jtabEmail.setColumnSelectionAllowed(true);
jtabEmail.setRowSelectionAllowed(false);
```

В результате вы сможете выбирать столбцы, как показано на рис. 8.3.

## Выбор ячеек

Разрешить выбор отдельных ячеек можно двумя способами. Первый способ — разрешить выбор и столбца, и строки. Поскольку выбор строк разрешен

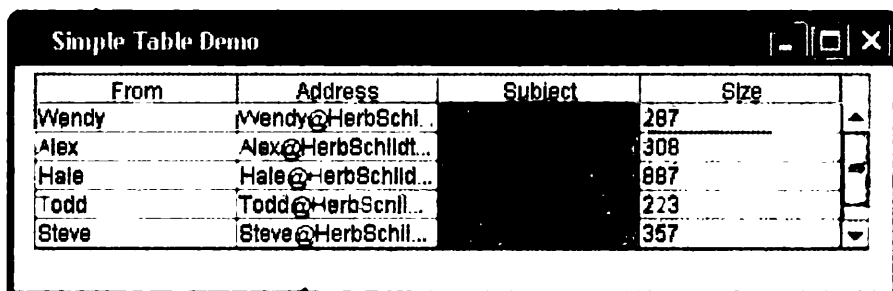


Рис. 8.3. Выбор столбца таблицы

по умолчанию, достаточно разрешить выбор столбцов. В результате пользователь сможет выбирать отдельные ячейки.

Второй способ — вызов метода `setCellSelectionEnabled()`.

```
void setCellSelectionEnabled(boolean enabled)
```

Если значение параметра `enabled` равно `true`, устанавливается режим выбора ячеек. Значение `false` отменяет этот режим. В теле данного метода вызываются методы `setRowSelectionAllowed()` и `setColumnSelectionAllowed()`, которым передается параметр, полученный от вызывающего метода. Поэтому, используя данный метод для запрета выбора ячеек, надо соблюдать осторожность. Передав этому методу значение `false`, вы запретите выбор как строк, так и столбцов. Таким образом, наилучший способ запретить выбор ячеек — отключить либо режим выбора строк, либо режим выбора столбцов.

Для того чтобы реализовать в программе выбор ячеек, удалите две строки, которые вы ранее включили в состав кода, и добавьте к программе следующее выражение:

```
jtabEmail.setCellSelectionEnabled(true);
```

После внесения изменений вы сможете выбрать ячейку так, как показано на рис. 8.4.

## Вопросы для текущего контроля . .

1. Какой метод класса `JTable` позволяет изменить режим выбора?
2. Какой метод надо использовать для того, чтобы разрешить выбор столбцов?
3. Выбор ячеек разрешен тогда, когда разрешен и выбор строк, и выбор столбцов. Да или нет?

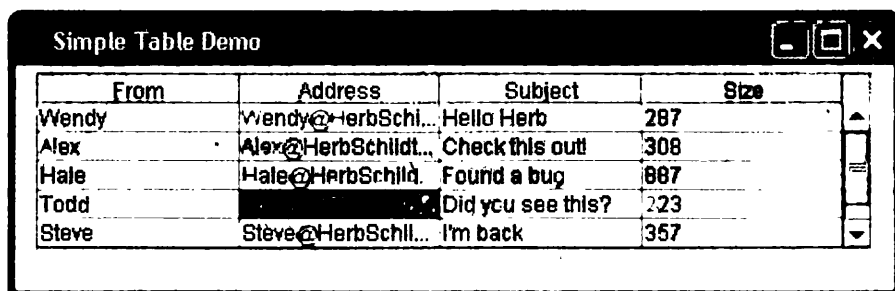


Рис. 8.4. Выбор ячеек таблицы

1. `setSelectionMode()`.
2. `setColumnSelectionAllowed()`.
3. Да.

ВАЖНО!

## 8.4. Обработка событий выбора

До сих пор таблицы, рассмотренные в этом модуле, использовались только для отображения данных. События, возникающие в процессе работы пользователя с программой, попросту игнорировались. Компонент `JTable` может генерировать несколько различных событий. Два основных события, возникающих при работе с таблицами, описываются классами `ListSelectionEvent` и `TableModelEvent`. Первое событие генерируется, когда пользователь выбирает данные из таблицы. Второе возникает при изменении таблицы. В данном разделе рассматривается обработка событий выбора. Следующий раздел будет посвящен событиям модели таблицы.

Выбрать данные из таблицы можно тремя способами: выбрать строку, столбец или ячейку. По умолчанию `JTable` позволяет выбирать одну или несколько строк, но при необходимости вы можете изменить поведение таблицы и предоставить пользователю возможность выбирать один или несколько столбцов, а также одну или несколько отдельных ячеек. Несмотря на то что общий принцип обработки событий остается постоянным, каждая из описанных ситуаций имеет свою специфику. Поэтому мы рассмотрим различные режимы выбора по отдельности. Начнем с общих принципов обработки событий выбора, применимых в любом случае.

### Общие сведения о событиях выбора

При выборе строки, столбца или ячейки таблицы генерируется событие `ListSelectionEvent`. Для поддержки подобных событий надо зарегистрировать обработчик `ListSelectionListener`. Класс `ListSelectionEvent` и интерфейс `ListSelectionListener` уже знакомы вам; они использовались в модуле 5 для обработки событий выбора, генерируемых компонентом `JList`. И `ListSelectionEvent`, и `ListSelectionListener` принадлежат пакету `javax.swing.event`.

Как вы, вероятно, помните, в интерфейсе `ListSelectionListener` объявлен только один метод: `valueChanged()`.

```
void valueChanged(ListSelectionEvent le)
```

где параметр `le` — это ссылка на объект события. В классе `ListSelectionEvent` определен ряд методов, но в большинстве случаев обращаться к ним не приходится. Достаточную информацию о происходящем позволяет получить объект `JTable`.

## Обработка событий выбора строк

Для поддержки событий, связанных с выбором строк, надо зарегистрировать обработчик `ListSelectionListener`. Экземпляр класса, реализующего этот интерфейс, связывается не с экземпляром `JTable`, а с моделью выбора. Ссылку на эту модель можно получить, вызвав метод `getSelectionModel()` объекта `JTable`. Заголовок данного метода имеет следующий вид:

```
ListSelectionModel getSelectionModel()
```

В классе `ListSelectionModel` объявлены методы, позволяющие определить состояние модели. Многие функции модели доступны непосредственно из класса `JTable`. Обращаться к `ListSelectionModel` чаще всего приходится для того, чтобы вызвать метод `getValueIsAdjusting()`. Он возвращает значение `true`, если процесс выбора продолжается, и `false`, если выбор уже сделан. (Метод `getValueIsAdjusting()` действует так же, как и соответствующие методы классов `JList`, `JScrollBar` и `JSlider`, описанные ранее.)

Если событие `ListSelectionEvent` было сгенерировано, определить выбранные строки позволит метод `getSelectedRow()` или `getSelectedRows()`. Заголовки этих методов приведены ниже.

```
int getSelectedRow()
```

```
int [] getSelectedRows()
```

Метод `getSelectedRow()` возвращает индекс первой выбранной строки. Если установлен режим `SINGLE_SELECTION`, эта строка будет единственной. Если ни одна строка не была выбрана, возвращается значение `-1`. Если установленный режим допускает выбор нескольких строк (такой режим принимается по умолчанию), то для получения списка индексов всех выбранных строк надо вызвать метод `getSelectedRows()`. Если ни одна строка не была выбрана, возвращается массив нулевого размера. Если была выбрана одна строка, массив содержит один

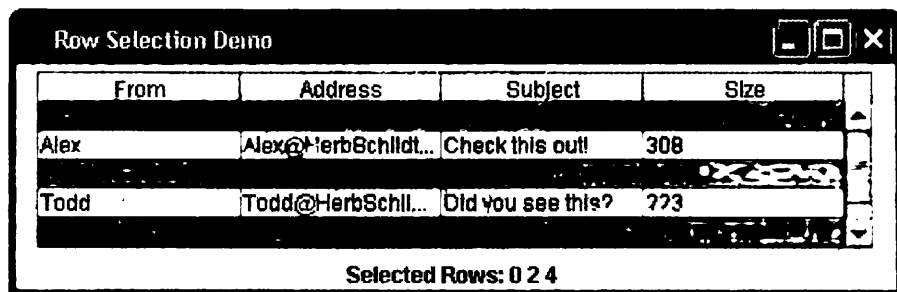


Рис. 8.5. Данные, которые отображаются программой, демонстрирующей обработку событий выбора строк



элемент. Таким образом, метод `getSelectedRows()` можно использовать даже в том случае, если выбор пользователя ограничен одной строкой.

Ниже приведен код программы, демонстрирующей обработку событий выбора. Эта программа создана на основе предыдущей. Индексы строк, выбранных пользователем, отображаются посредством метки. Окно, создаваемое при работе программы, показано на рис. 8.5.

// Обработка событий выбора строк таблицы

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

class RowSelDemo {

    String[] headings = { "From", "Address", "Subject", "Size" };

    Object[][] data = {
        { "Wendy", "Wendy@HerbSchildt.com",
          "Hello Herb", new Integer(287) },
        { "Alex", "Alex@HerbSchildt.com",
          "Check this out!", new Integer(308) },
        { "Hale", "Hale@HerbSchildt.com",
          "Found a bug", new Integer(887) },
        { "Todd", "Todd@HerbSchildt.com",
          "Did you see this?", new Integer(223) },
        { "Steve", "Steve@HerbSchildt.com",
          "I'm back", new Integer(357) },
        { "Ken", "Ken@HerbSchildt.com",
          "Arrival time change", new Integer(512) }
    };

    JTable jtabEmail;

    JLabel jlab;

    RowSelDemo() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Row Selection Demo");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());
    }
}
```

```
// Установка начальных размеров фрейма.
jfrm.setSize(500, 160);

// Завершение программы при закрытии окна пользователем.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Создание метки для отображения информации о выбранных
// строках таблицы.
jlab = new JLabel();

// Создание таблицы для отображения почтовых сообщений.
jtabEmail = new JTable(data, headings);

// Включение данных в состав панели с прокруткой.
JScrollPane jscrlp = new JScrollPane(jtabEmail);

// Установка размеров прокручиваемой области просмотра.
jtabEmail.setPreferredScrollableViewportSize(
    new Dimension(450, 80));

// Получение модели выбора. С ней связывается
// обработчик событий выбора.
ListSelectionModel lsmRow = jtabEmail.getSelectionModel();

// Связывание обработчика событий с моделью выбора.
// Обработчик реагирует на события выбора строк.
lsmRow.addListSelectionListener(new ListSelectionListener() {
    public void valueChanged(ListSelectionEvent le) {
        String str = "Selected Rows: ";

        // Получение индексов всех выбранных строк.
        int[] rows = jtabEmail.getSelectedRows();

        // Создание строки для представления индексов.
        for(int i=0; i < rows.length; i++)
            str += rows[i] + " ";

        // Отображение индексов выбранных строк.
        jlab.setText(str);
    }
});
```

```
// Включение таблицы и метки в состав панели содержимого.
jfrm.getContentPane().add(jscrlp);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new RowSelDemo();
        }
    });
}
```

В данной программе, по сравнению с предыдущей, добавлены три фрагмента кода. Во-первых, в ней определена метка `jlab`. Она предназначена для отображения выбранных строк. Во-вторых, в программе извлекается ссылка на модель выбора. Для этого используется следующее выражение:

```
ListSelectionModel lsmRow = jTableEmail.getSelectionModel();
```

В-третьих, с моделью связывается обработчик событий выбора. Он реализован с помощью неименованного внутреннего класса. В методе `valueChanged()` определяются индексы выбранных строк. Для этого вызывается метод `getSelectedRows()` объекта таблицы. Если была выбрана только одна строка, массив содержит один элемент. Индексы выбранных строк отображаются на экране.

## Обработка событий выбора столбцов

Подобно событиям выбора строк, для поддержки событий выбора столбцов следует зарегистрировать обработчик `ListSelectionListener`. Однако этот обработчик не связывается с моделью выбора, предоставляемой `JTable`. Вместо этого его следует связать с моделью выбора, которая используется моделью столбца таблицы. (Как вы помните, моделью столбца является реализация `TableColumnModel`.) Ссылку на модель столбца можно получить, вызвав метод `getColumnModel()` объекта `JTable`. Заголовок этого метода имеет такой вид:

```
TableColumnModel getColumnModel()
```

Возвращаемая ссылка позволяет найти модель `ListSelectionModel`, используемую столбцами. Если предположить, что объект `JTable` содержится в переменной `jtable`, то для получения модели выбора для столбца надо выполнить следующие действия:

```
TableColumnModel tcm = jtable.getColumnModel();
ListSelectionModel lsmCol = tcm.getSelectionModel();
```

Если событие `ListSelectionEvent` было сгенерировано, то для определения выбранных столбцов надо вызвать метод `getSelectedColumn()` или `getSelectedColumns()` объекта `JTable`. Эти методы приведены ниже.

```
int getSelectedColumn()
int[] getSelectedColumns()
```

Метод `getSelectedColumn()` возвращает индекс первого выбранного столбца. Если установлен режим, ограничивающий выбор пользователя одним столбцом, то этот столбец будет единственным. Если текст не был выделен, возвращается значение `-1`. Если установлен режим выбора нескольких столбцов (он принимается по умолчанию), то для получения списка индексов всех выбранных столбцов надо вызвать метод `getSelectedColumns()`. Если ни один столбец не был выбран, возвращается массив нулевого размера. Если был выбран один столбец, то массив содержит один элемент. Таким образом, метод `getSelectedColumns()` можно использовать даже в том случае, если выбор пользователя ограничен одним столбцом. Следует помнить, что индексы, возвращаемые описанными методами, соответствуют представлению; другими словами, они определяют номера столбцов, отображаемых на экране. Поскольку пользователь может изменить порядок следования столбцов, то при одних и тех же выбранных столбцах их индексы могут различаться.

Ниже приведена модифицированная версия предыдущей программы. Она отличается тем, что в ней разрешен выбор столбцов. Обратите внимание

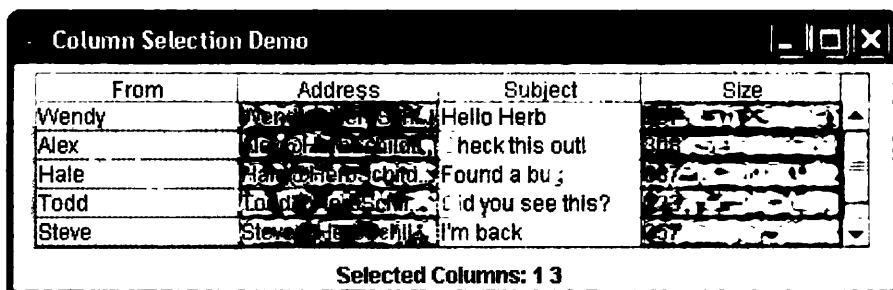


Рис. 8.6. Выходные данные программы, демонстрирующей выбор столбцов

## 452    Модуль 8. Таблицы и деревья

---

на обработчик событий выбора. Окно, создаваемое при работе программы, показано на рис. 8.6.

// Обработка событий выбора столбцов таблицы.

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

class ColSelDemo {

    String[] headings = { "From", "Address", "Subject", "Size" };

    Object[][] data = {
        { "Wendy", "Wendy@HerbSchildt.com",
          "Hello Herb", new Integer(287) },
        { "Alex", "Alex@HerbSchildt.com",
          "Check this out!", new Integer(308) },
        { "Hale", "Hale@HerbSchildt.com",
          "Found a bug", new Integer(887) },
        { "Todd", "Todd@HerbSchildt.com",
          "Did you see this?", new Integer(223) },
        { "Steve", "Steve@HerbSchildt.com",
          "I'm back", new Integer(357) },
        { "Ken", "Ken@HerbSchildt.com",
          "Arrival time change", new Integer(512) }
    };

    JTable jtabEmail;

    JLabel jlab;

    ColSelDemo() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Column Selection Demo");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(500, 160);
    }
}
```

```
// Завершение программы при закрытии окна пользователем.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Создание метки для отображения информации о выборе
// пользователя.
jlab = new JLabel();

// Создание таблицы для отображения почтовых сообщений.
jtabEmail = new JTable(data, headings);

// Включение данных в состав панели с прокруткой.
JScrollPane jscrlp = new JScrollPane(jtabEmail);

// Установка размеров прокручиваемой области просмотра.
jtabEmail.setPreferredScrollableViewportSize(
    new Dimension(450, 80));

// Разрешение выбора столбцов и запрет выбора строк.
jtabEmail.setColumnSelectionAllowed(true);
jtabEmail.setRowSelectionAllowed(false);

// Получение модели выбора для модели столбца.
// С моделью выбора связывается обработчик событий.
TableColumnModel tcm = jtabEmail.getColumnModel()
ListSelectionModel lsmCol = tcm.getSelectionModel();

// Обработка событий выбора.
lsmCol.addListSelectionListener(new ListSelectionListener() {
    public void valueChanged(ListSelectionEvent le) {
        String str = "Selected Columns: ";

        // Получение индексов выбранных столбцов.
        int[] cols = jtabEmail.getSelectedColumns();

        // Формирование строки, содержащей индексы.
        for(int i=0; i < cols.length; i++)
            str += cols[i] + " ";

        // Отображение индексов выбранных столбцов.
        jlab.setText(str);
    }
});
```

```

// Включение таблицы и метки в состав панели содержимого.
jfrm.getContentPane().add(jscrlp);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new ColSelDemo();
        }
    });
}
}

```

Перед тем как продолжить разговор об обработке событий таблицы, следует заметить, что, имея индексы столбцов на экране, можно получить соответствующие им индексы модели. Часто это бывает необходимо, так как индексы модели не изменяются. Для определения индексов модели надо выполнить ряд действий. Вначале следует получить ссылку на модель столбцов таблицы, вызвав метод `getColumnModel()`. Затем надо извлечь ссылку на требуемый столбец, вызвав метод `getColumn()` модели столбца. Заголовок метода `getColumn()` имеет следующий вид:

```
TableColumn getColumn(int idx)
```

где параметр `idx` — это индекс требуемого столбца (отсчет индексов начинается с нуля), полученный путем вызова метода `getSelectedColumns()`. Столбец инкапсулирован в объекте `TableColumn`. В нем определено несколько методов. Для получения индекса модели используется метод `getModelIndex()`.

Заголовок этого метода имеет вид

```
int getModelIndex()
```

Он возвращает индекс столбца относительно модели. Таким образом, для того, чтобы в предыдущей программе получить индекс модели, надо использовать следующее выражение (предполагается, что индекс выбранного столбца относительно экранного представления равен 1):

```
jtabEmail.getColumnModel().getColumn(1).getModelIndex();
```

## Обработка событий выбора ячеек

Если разрешен выбор ячеек, то, получив оповещение о событии `ListSelectionEvent`, вы можете определить номер выбранной ячейки по индексу строки и индексу столбца. Чтобы это было возможно, необходимо зарегистрировать как обработчик выбора строк, так и обработчик выбора столбцов. Использование таких обработчиков было описано в предыдущем разделе.



### Вопросы для текущего контроля.

1. Связывается ли обработчик событий выбора строк с экземпляром класса `JTable`?
2. Какой метод позволяет получать список выбранных строк?
3. С каким объектом необходимо связать обработчик, чтобы обеспечить обработку событий выбора столбцов?

## Спросим у опытного программиста

**Вопрос.** Всегда ли доступны методы `getSelectedRow()`, `getSelectedRows()`, `getSelectedColumn()` и `getSelectedColumns()`, или обращаться к ним можно только при возникновении соответствующих событий?

**Ответ.** Различные методы `getSelectedX` можно вызывать в любой момент, когда вам надо определить, выбраны ли строки, столбцы или ячейки таблицы. Например, если выбрана строка, метод `getSelectedColumn()` вернет индекс столбца, соответствующего событию мыши или клавиатуры, которое было сгенерировано при выборе строки. Более того, если разрешен выбор ячеек, индекс строки и столбца можно определить как в обработчике событий выбора строк, так и в обработчике событий выбора столбцов.

1. Нет. Он связывается с моделью `ListSelectionModel`, используемой таблицей.
2. `getSelectedRows()`.
3. Для обработки событий выбора столбца обработчик должен быть связан с объектом `ListSelectionModel`, используемым `TableColumnModel`.



**ВАЖНО!**

## 8.5. Обработка событий модели таблицы

Для того чтобы отслеживать изменения данных таблицы (например, содержимого ячейки), надо зарегистрировать в модели таблицы обработчик `TableModelListener`. Для получения модели таблицы вызывается метод `getModel()` объекта `JTable`. Заголовок этого метода имеет следующий вид:

```
TableModel getModel()
```

В `TableModel` определен метод `addTableModelListener()`, который используется для регистрации обработчика событий `TableModelEvent`. Этот метод объявлен следующим образом:

```
void addTableModelListener(TableModelListener tml)
```

В интерфейсе `TableModelListener` определен только один метод: `tableChanged()`. Заголовок этого метода приведен ниже,

```
void tableChanged(TableModelEvent tme)
```

Этот метод получает управление при каждом изменении модели таблицы: редактировании данных, заголовков, включении или удалении столбцов.

В классе `TableModelEvent` определены методы, показанные в табл. 8.2. Возвращаемыми значениями методов `getFirstRow()`, `getLastRow()` и `getColumn()` являются соответственно индексы строки и столбца, выбранные пользователем. Также информация может быть получена посредством методов класса `JTable`, которые были описаны ранее, однако в ряде случаев удобнее обращаться за этими сведениями к объекту события. Пользуясь информацией, предоставляемой объектом события, следует соблюдать осторожность, так как метод `getColumn()` возвращает индекс столбца, поддерживаемый моделью таблицы. Если пользователь изменит порядок следования столбцов, этот индекс будет отличаться от индекса, предоставляемого методом `getSelectedColumn()` класса `JTable` (как известно, этот индекс отражает позицию столбца на экране). Таким образом, необходимо четко представлять себе различия между индексами, полученными разными способами.

Следует обратить на метод `getType()`. Он возвращает значение, которое указывает на тип изменений, имевших место. Значения, возвращаемые данным методом, приведены ниже (они определены в классе `TableModelEvent`).

DELETE

Удаление строки или столбца

INSERT

Добавление строки или столбца

UPDATE

Изменение данных ячейки

Таблица 8.2. Методы, определенные в классе `TableModelEvent`

| Метод                          | Описание                                                                                                                                                                                                                          |
|--------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>int getColumn()</code>   | Возвращает индекс столбца, с которым связано событие. Индексы отсчитываются начиная с нуля. Возвращаемое значение <code>TableModelEvent.ALL_COLUMNS</code> указывает на то, что событие имеет отношение ко всем столбцам в строке |
| <code>int getFirstRow()</code> | Возвращает индекс первой из строк, с которыми связано событие. Индексы отсчитываются начиная с нуля. Возвращаемое значение <code>TableModelEvent.HEADER_ROW</code> указывает на изменение заголовка                               |
| <code>int getLastRow()</code>  | Возвращает индекс первой из строк, с которыми связано событие. Индексы отсчитываются начиная с нуля                                                                                                                               |
| <code>int getType()</code>     | Возвращает значение, описывающее тип произошедших изменений. Одно должно быть равно <code>TableModelEvent.DELETE</code> , <code>TableModelEvent.INSERT</code> или <code>TableModelEvent.UPDATE</code>                             |

Обычно при изменении данных ячейки внутреннее хранилище информации, используемое приложением, должно быть обновлено так, чтобы оно отражало внесенные изменения. Таким образом, если в таблице разрешено редактировать ячейки, важно позаботиться об обработке событий модели типа `UPDATE`.

При изменении содержимого ячейки новые данные можно получить путем вызова метода `getValueAt()` экземпляра модели. Этот метод определен следующим образом:

```
Object getValueAt(int row, int column)
```

где параметры `row` и `column` определяют координаты ячейки, подвергшейся изменениям. При необходимости можно задавать данные ячейки из программы, для этой цели предназначен метод `setValueAt()`:

```
void setValueAt(Object val, int row, int column)
```

В результате выполнения данного метода для ячейки, расположенной на пересечении строки `row` и столбца `column`, устанавливается значение, равное `val`. В обоих описанных выше методах `row` и `column` задают координаты ячейки относительно модели, а не относительно представления.

Ниже приведен код программы, демонстрирующей обработку событий модели таблицы. В ней также предусмотрена поддержка событий выбора строк. В результате, работая с программой, можно следить за тем, при каких условиях генерируются события этих типов. Если вы выберете строку, возникнет событие выбора и на экране отобразится индекс строки. При изменении значения

ячейки будет сгенерировано событие модели таблицы. В этом случае будут выведены индексы строки и столбца, а также обновленные данные. На рис. 8.7 показан пример выходных данных программы.

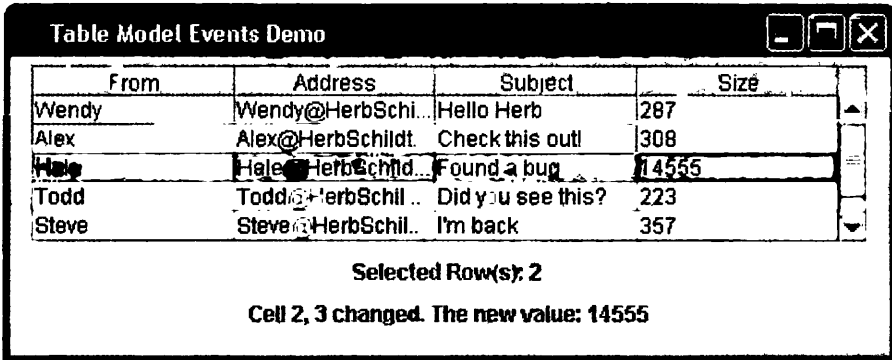


Рис. 8.7. Окно, отображаемое при работе программы TabModEventDemo

// Отслеживание изменений данных таблицы.

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

class TabModEventDemo {

    String[] headings = { "From", "Address", "Subject", "Size" };

    Object[][] data = {
        { "Wendy", "Wendy@HerbSchildt.com",
          "Hello Herb", new Integer(287) },
        { "Alex", "Alex@HerbSchildt.com",
          "Check this out!", new Integer(308) },
        { "Hale", "Hale@HerbSchildt.com",
          "Found a bug", new Integer(887) },
        { "Todd", "Todd@HerbSchildt.com",
          "Did you see this?", new Integer(223) },
        { "Steve", "Steve@HerbSchildt.com",
          "I'm back", new Integer(357) },
        { "Ken", "Ken@HerbSchildt.com",
          "Arrival time change", new Integer(512) }
    };
};
```

```
JTable jtabEmail;

JLabel jlab;
JLabel jlab2;

TableModel tm;

TabModEventDemo() {
    // Создание нового контейнера JFrame.
    JFrame jfrm = new JFrame("Table Model Events Demo");

    // Установка диспетчера компоновки FlowLayout.
    jfrm.getContentPane().setLayout(new FlowLayout());

    // Установка начальных размеров фрейма.
    jfrm.setSize(500, 200);

    // Завершение программы при закрытии окна пользователем.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Создание метки для отображения информации о выборе
    // пользователя.
    jlab = new JLabel();
    jlab.setPreferredSize(new Dimension(400, 20));
    jlab.setHorizontalAlignment(SwingConstants.CENTER);

    // Создание метки для отображения информации
    // об изменении данных.
    jlab2 = new JLabel();

    // Создание таблицы для отображения почтовых сообщений.
    jtabEmail = new JTable(data, headings);

    // Включение данных в панель с прокруткой.
    JScrollPane jscrlp = new JScrollPane(jtabEmail);

    // Установка размеров прокручиваемой области просмотра.
    jtabEmail.setPreferredScrollableViewportSize(
        new Dimension(450, 80));

    // Получение модели выбора.
    ListSelectionModel listSelMod = jtabEmail.
```

```

getSelectionModel();

// Обработка событий выбора строк.
listSelMod.addListener(
    new ListSelectionListener() {
        public void valueChanged(ListSelectionEvent le) {
            String str = "Selected Row(s): ";

            // Получение индексов выбранных строк.
            int[] rows = jtabEmail.getSelectedRows();

            // Создание строки для представления индексов.
            for(int i=0; i < rows.length; i++)
                str += rows[i] + " ";

            // Отображение индексов выбранных строк.
            jlab.setText(str);
        }
    });

// Получение модели таблицы.
tm = jtabEmail.getModel();

// Обработка событий изменения модели таблицы.
tm.addTableModelListener(new TableModelListener() {
    public void tableChanged(TableModelEvent tme) {
        if(tme.getType() == TableModelEvent.UPDATE) {
            // Отображение координат ячейки и нового значения.
            jlab2.setText("Cell " + tme.getFirstRow() + ", " + tme.getColumn() + " changed." +
                " The new value: " + tm.getValueAt(tme.getFirstRow(),
                    tme.getColumn()));
        }
    }
});

// Включение таблицы и метки в состав панели содержимого.
jfrm.getContentPane().add(jscrlp);
jfrm.getContentPane().add(jlab);
jfrm.getContentPane().add(jlab2);

// Отображение фрейма.

```

```
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new TabModEventDemo();
        }
    });
}
```

Рассмотрим более подробно часть программы, ответственную за обработку событий модели таблицы. В первую очередь заметьте, что в программе присутствует переменная `TableModel` с именем `tm`. Далее в программе извлекается ссылка на модель таблицы, для этого вызывается метод `getModel()` объекта `jtabEmail` (экземпляра класса `JTable`). Полученное значение сохраняется в переменной `tm`.

```
tm = jtabEmail.getModel();
```

Затем переменная `tm` используется для регистрации обработчика событий модели.

В обработчике событий модели таблицы обратите внимание на метод `tableChanged()`, который получает управление при каждом изменении модели. В данной программе изменения вносит пользователь посредством ввода данных в ячейку. (В программе не разрешается вставлять и удалять столбцы, а также изменять заголовки.) Для получения координат ячейки, претерпевшей изменения, используются методы `getFirstRow()` и `getColumn()`. Координаты ячейки применяются при извлечении данных из модели столбца. Для этой цели вызывается метод `getValueAt()` модели таблицы (переменная `tm`); ему передаются значения, возвращенные методами `getFirstRow()` и `getColumn()`. Таким образом, независимо от того, изменил ли пользователь порядок следования столбцов, данные будут извлекаться из нужной ячейки, поскольку индексы в модели остаются неизменными.

Обратите внимание, что в методе `tableChanged()` сначала вызывается метод `getType()` и возвращаемое им значение сравнивается с константной `UPDATE`. Это делается для того, чтобы убедиться, что причиной события стало изменение данных ячейки. Такой подход не является совершенно необходимым, так как в данной программе могут генерироваться события модели только одного типа. Вызов метода `getType()` использован здесь лишь для того, чтобы продемонстрировать проверку типа события.

## Вопросы для текущего контроля

1. Какое событие инкапсулирует изменения данных таблицы?
2. Обработчик событий модели таблицы связывается непосредственно с таблицей. Да или нет?
3. Какие значения может возвращать метод `getType()` класса `TableModelEvent`?

**ВАЖНО!**

## 8.6. Изменение размеров таблицы

Компонент `JTable` поддерживает несколько режимов изменения размеров таблицы. Они определяют поведение столбцов при перетаскивании границы между заголовками. По умолчанию при изменении ширины столбца размеры всех последующих столбцов (т.е. расположенных справа от столбца, размеры которого изменяются) также изменяются с тем, чтобы общая ширина таблицы оставалась прежней. Ширина столбцов, расположенных слева, остается неизменной. Однако такое поведение, реализуемое по умолчанию, соответствует лишь одному из пяти возможных режимов.

Для того чтобы изменить режим, надо вызвать метод `setAutoResizeMode()`:

```
void setAutoResizeMode(int how)
```

где параметр `how` должен иметь значение, равное одной из пяти констант, определенных в классе `JTable`. Эти константы описаны в табл. 8.3. Желательно поэкспериментировать, устанавливая различные режимы изменения размеров, чтобы лучше понять, как они влияют на поведение таблицы. Один из режимов, `AUTO_RESIZE_OFF`, заслуживает детального обсуждения.

Если автоматическое изменение размеров отключено, т.е. если установлен режим `AUTO_RESIZE_OFF`, то при перетаскивании границы между столбцами размеры остальных столбцов автоматически не изменяются. Если в результате действий пользователя размеры таблицы превосходят размеры прокручиваемой области, то отображается горизонтальная полоса прокрутки, позволяющая сдвигать содержимое таблицы вправо и влево. Если же размеры таблицы становятся меньше области прокрутки, попытки заполнить освободившееся пространство не предпринимаются. В результате в составе таблицы формируется

1. `TableModelEvent`.
2. Нет. Обработчик событий модели таблицы связывается с моделью.
3. `INSERT`, `DELETE` и `UPDATE`.

пустая область. При создании таблицы размеры столбцов и строк выбираются без учета ширины всей таблицы. Таким образом, если вы отключите автоматическое изменение размеров, вам придется указывать ширину столбцов вручную.

Таблица 8.3. Режимы изменения размеров, определенные в классе `JTable`

| Режим                                       | Описание                                                                                                                                                                                                                                                                                                                                                                                     |
|---------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>AUTO_RESIZE_ALL_COLUMNS</code>        | Изменение ширины одного столбца влияет на размеры всех столбцов                                                                                                                                                                                                                                                                                                                              |
| <code>AUTO_RESIZE_LAST_COLUMN</code>        | Изменение ширины столбца влияет только на размер самого правого столбца                                                                                                                                                                                                                                                                                                                      |
| <code>AUTO_RESIZE_NEXT_COLUMN</code>        | Изменение ширины столбца влияет только на размер следующего столбца                                                                                                                                                                                                                                                                                                                          |
| <code>AUTO_RESIZE_OFF</code>                | При изменении ширины столбца размеры остальных столбцов остаются неизменными, но изменяется размер всей таблицы. Если таблица включена в панель с прокруткой и ее ширина станет больше, чем ширина прокручиваемой области просмотра, отображается горизонтальная полоса прокрутки, позволяющая сдвигать содержимое таблицы влево и вправо, обеспечивая тем самым доступ к остальным столбцам |
| <code>AUTO_RESIZE_SUBSEQUENT_COLUMNS</code> | Изменение ширины столбца влияет на размеры столбцов, расположенных справа от него                                                                                                                                                                                                                                                                                                            |

Для того чтобы убедиться, как ведет себя таблица при отсутствии автоматического изменения размеров, включите в рассмотренную ранее программу следующую строку:

```
jtabEmail.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
```

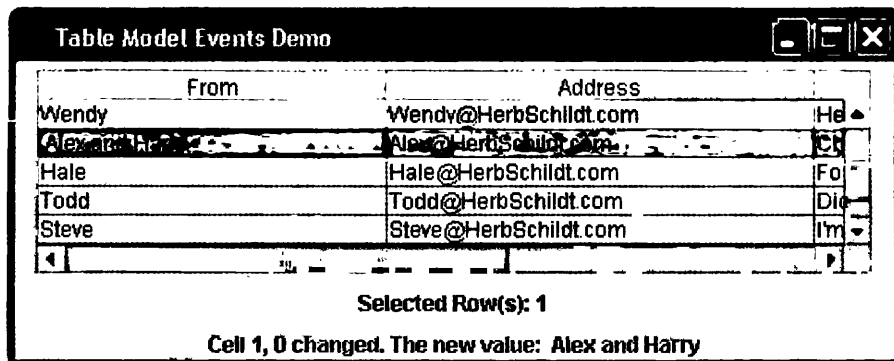


Рис. 8.8. Вид таблицы при отключении автоматического изменения размеров



На рис. 8.8 показано состояние таблицы после изменения размеров столбцов. Обратите внимание, что, для того, чтобы предоставить пользователю доступ к скрытой части таблицы, отображается горизонтальная полоса прокрутки.

## Спросим у опытного программиста

**Вопрос.** Можно ли установить размер столбца из программы? Можно ли получить программными средствами информацию о текущей ширине столбца.

**Ответ.** На оба вопроса можно ответить положительно. Для того чтобы задать ширину столбца, надо выполнить процедуру, состоящую из трех этапов. Сначала следует получить ссылку на модель столбцов таблицы, вызвав метод `getColumnModel()`. Далее надо извлечь ссылку на требуемый столбец, вызвав метод `getColumn()` модели столбца. Как было сказано ранее, метод `getColumn()` определяется следующим образом:

```
TableColumn getColumn(int idx)
```

Значение параметра `idx` определяет индекс требуемого столбца (индексы отсчитываются начиная с нуля). В классе `TableColumn` определены методы, позволяющие получать или задавать ширину столбца. Для установки ширины используется метод `setPreferredWidth()`. Заголовок этого метода имеет следующий вид:

```
void setPreferredWidth(int w)
```

где параметр `w` указывает предпочтительную ширину. Таким образом, третий этап рассматриваемой процедуры представляет собой установку требуемой ширины столбца путем вызова метода `setPreferredWidth()`.

Для того чтобы реализовать данный подход на практике, включите в рассмотренную ранее программу приведенные ниже строки кода.

```
TableColumnModel tcm =  
jtabEmail.getColumnModel();  
TableColumn tc = tcm.getColumnModel(1);  
tc.setPreferredWidth(200);
```

После выполнения данных выражений ширина второго столбца (его индекс равен 1) станет равной 200.

Помимо установки предпочтительных размеров, вы также можете задать минимальную и максимальную ширину столбца. Для этой цели предназначены методы `setMinWidth()` и `setMaxWidth()`.

```
void setMinWidth(int w)
```

```
void setMaxWidth(int w)
```

Параметр `w` задает требуемую ширину. Определить текущую ширину столбца позволяет метод `getWidth()`, который определен следующим образом.

```
int getWidth()
```

Он возвращает текущую ширину столбца. Это значение изменяется каждый раз, когда пользователь перетаскивает границу между заголовками.

Определить предпочтительную ширину столбца, а также его минимальную и максимальную ширину позволяют приведенные ниже методы.

```
int getPreferredWidth()
```

```
int getMinWidth()
```

```
int getMaxWidth()
```

Назначение каждого из этих методов ясно из его имени.

## Вопросы для текущего контроля

1. Какой режим изменения ширины столбца используется по умолчанию?
2. Предположим, что для таблицы установлен режим `AUTO_RESIZE_OFF`. Что произойдет, если ширина таблицы превысит ширину панели с прокруткой, в составе которой находится данная таблица?

1. `AUTO_RESIZE_SUBSEQUENT_COLUMNS`.

2. Отобразится горизонтальная полоса прокрутки, позволяющая пользователю обращаться к столбцам, оказавшимся за пределами области просмотра.

ВАЖНО!

## 8.7. Модель таблицы, определяемая разработчиком

Во всех рассмотренных ранее примерах с помощью таблицы отображались данные, заданные посредством массива. Такой подход вполне удовлетворяет потребностям большинства программ, но для некоторых типов приложений он неприемлем. Например, если данные, отображаемые в таблице, получаются путем вычислений или извлекаются из источника, внешнего по отношению к программе (например, из базы данных), помещать информацию в массив неудобно и неэффективно. Swing позволяет решить эту проблему. Вы можете создать собственную модель таблицы, реализовав в ней требуемую обработку данных, предназначенных для отображения. Это, в частности, позволяет легко поддерживать динамически вычисляемые данные.

Для формирования модели таблицы необходимо создать экземпляр класса, реализующего интерфейс `TableModel`. В данном интерфейсе определены методы, показанные в табл. 8.4. Несмотря на то что все методы `TableModel` можно реализовать самостоятельно, проще создать модель как подкласс класса `AbstractTableModel`.

Таблица 8.4. Методы, объявленные в интерфейсе `TableModel`

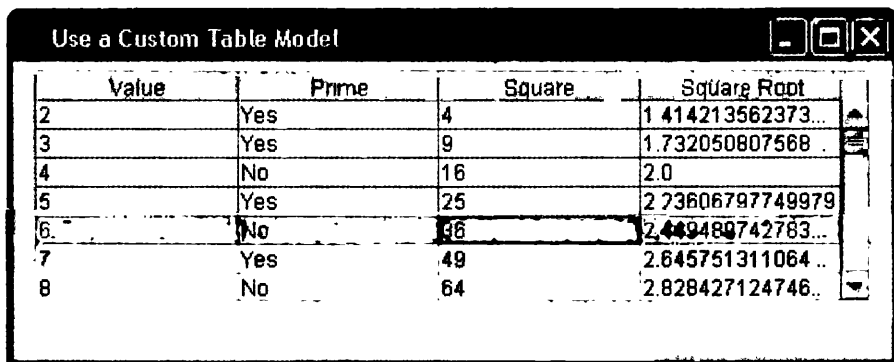
| Метод                                                           | Описание                                                                                                                                                                                                                                |
|-----------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void addTableModelListener(TableModelListener tml)</code> | Регистрирует объект <code>tml</code> в качестве обработчика событий модели таблицы                                                                                                                                                      |
| <code>Class&lt;?&gt; getColumnClass(int idx)</code>             | Возвращает тип данных, которые содержатся в столбце, определяемом параметром <code>idx</code>                                                                                                                                           |
| <code>int getColumnCount()</code>                               | Возвращает число столбцов                                                                                                                                                                                                               |
| <code>String getColumnName(int idx)</code>                      | Возвращает имя столбца, заданного посредством параметра <code>idx</code>                                                                                                                                                                |
| <code>int getRowCount()</code>                                  | Возвращает число строк                                                                                                                                                                                                                  |
| <code>Object getValueAt(int rIdx, int cIdx)</code>              | Возвращает данные, содержащиеся в ячейке, координаты которой задаются с помощью параметров <code>rIdx</code> и <code>cIdx</code>                                                                                                        |
| <code>boolean isCellEditable(int rIdx, int cIdx)</code>         | Если ячейка с координатами, указанными с помощью параметров <code>rIdx</code> и <code>cIdx</code> , допускает редактирование, возвращается значение <code>true</code> . В противном случае метод возвращает значение <code>false</code> |

Окончание табл. 8.4

| Метод                                                               | Описание                                                                                                                                        |
|---------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void removeTableModelListener (TableModelListener tml)</code> | Удаляет обработчик событий модели таблицы, указанный с помощью параметра <code>tml</code>                                                       |
| <code>void setValueAt (Object v, int rIdx, int cIdx)</code>         | Устанавливает <code>v</code> в качестве значения ячейки, координаты которой задаются с помощью параметров <code>rIdx</code> и <code>cIdx</code> |

В классе `AbstractTableModel` содержится реализация всех методов, за исключением `getValueAt()`, `getRowCount()` и `getColumnCount()`. Однако если вы хотите использовать имена столбцов, отличные от принятых по умолчанию (буквы A–Z), вам надо также переопределить метод `getColumnName()`.

Ниже приведен код программы, показывающей, насколько просто реализуется простая модель таблицы. В программе создается модель `NumInfoModel`, предназначенная для отображения в первой строке таблицы целых чисел (начиная с 2). Во втором столбце выводятся сведения о том, является ли число в первом столбце простым. В следующем столбце представляется квадрат числа, а в последнем — его квадратный корень. Все данные, за исключением имен столбцов, вычисляются динамически; необходимость в массиве для хранения информации не возникает. Окно, создаваемое при работе программы, показано на рис. 8.9.



| Value | Prime | Square | Square Root         |
|-------|-------|--------|---------------------|
| 2     | Yes   | 4      | 1.414213562373...   |
| 3     | Yes   | 9      | 1.732050807568...   |
| 4     | No    | 16     | 2.0                 |
| 5     | Yes   | 25     | 2.23606797749979... |
| 6     | No    | 36     | 2.449489742783...   |
| 7     | Yes   | 49     | 2.645751311064...   |
| 8     | No    | 64     | 2.828427124746...   |

Рис. 8.9. Данные, отображаемые программой, которая использует модель таблицы, определяемую разработчиком

```
// Использование модели таблицы, определяемой разработчиком
// Согласно модели, в первом столбце таблицы отображаются
// целые числа от 2 до N. В следующий трюх столбцах выводится
// сообщение о том, является ли соответствующее число
```

```
// простым, а также квадрат и квадратный корень этого числа.
```

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

```
// Определение модели для динамической генерации данных.
```

```
class NumInfoModel extends AbstractTableModel {
    int numRows;

    String colNames[] = { "Value", "Prime",
                          "Square", "Square Root" };

```

```
    NumInfoModel(int len) {
        super();
        numRows = len;
    }

```

```
    public int getRowCount() { return numRows; }
    public int getColumnCount() { return 4; }

```

```
    // Метод возвращает имя столбца по заданному индексу.
```

```
    // Имя извлекается из массива colNames.
```

```
    public String getColumnName(int c) {
        return colNames[c];
    }

```

```
    // Вычисление данных.
```

```
    // Для столбца 0 возвращается номер строки плюс 2.
```

```
    // Для столбца 1 возвращается значение "Yes",
```

```
    // если число в столбце 0 является простым.
```

```
    // Для столбца 2 возвращается квадрат числа,
```

```
    // содержащегося в столбце 0.
```

```
    // Для столбца 3 возвращается квадратный корень числа,
```

```
    // содержащегося в столбце 0.
```

```
    public Object getValueAt(int r, int c) {

```

```
        if(c==0) return new Integer(r+2);
```

```
        else if(c==1) {
```

```
            if(isPrime(r+2)) return "Yes";
```

```
            else return "No";
```

```
        }
```

```
        else if(c==2) return new Integer((r+2) * (r+2));
```

```
        else return new Double(Math.sqrt(r+2));
    }

    // Возвращает значение true, если параметр v
    // представляет собой простое число.
    boolean isPrime(int v) {
        int i;

        for(i=2; i <= v/i; i++)
            if((v%i) == 0) return false;

        return true;
    }
}

class NumInfoTable {

    JTable jtabNumInfo;

    NumInfoTable() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Use a Custom Table Model");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(500, 200);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание таблицы, использующей модель NumInfoModel
        // для отображения значений от 2 до 100.
        jtabNumInfo = new JTable(new NumInfoModel(99));

        // Включение данных в панель с прокруткой.
        JScrollPane jscrip = new JScrollPane(jtabNumInfo);

        // Установка размеров прокручиваемой области просмотра.
        jtabNumInfo.setPreferredScrollableViewportSize(
            new Dimension(450, 110));
    }
}
```

```

// Включение таблицы в состав панели содержимого.
jfrm.getContentPane().add(jscrlp);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new NumInfoTable();
        }
    });
}
}

```

При написании программы использован подход, ранее незнакомый вам, поэтому рассмотрим код более подробно. Обратите внимание на импортирование пакета `javax.swing.table`. Как было сказано ранее, в этом пакете содержатся все интерфейсы и классы, используемые компонентом `JTable`. В данном случае нам необходим класс `AbstractTableModel`.

В программе определен класс `NumInfoModel`, являющийся подклассом `AbstractTableModel`. В классе `NumInfoModel` определены два поля. Первое, `numRows`, содержит число строк модели. Как видно из кода программы, конструктор модели позволяет создавать таблицу с любым количеством строк. Информация о числе строк хранится в `numRows`. Второе поле, `colNames`, представляет собой массив строк, содержащих заголовки столбцов таблицы.

Далее в программе определен конструктор класса `NumInfoModel`. В качестве параметра конструктору передается информация о числе строк. В теле конструктора присутствует выражение `super()`, посредством которого вызывается конструктор класса `AbstractTableModel`. После обращения к конструктору суперкласса число строк, заданное посредством параметра `len`, записывается в переменную `numRows`.

В классе `NumInfoModel` переопределяются методы `getRowCount()` (он возвращает значение `numRows`) и `getColumnCount()` (он возвращает значение 4). Метод `getColumnCount()` всегда возвращает одно и то же значение, поскольку в данном примере число столбцов таблицы не изменяется. Далее в классе `NumInfoModel` переопределяется метод `getColumnName()`, который по индексу столбца возвращает его имя. В данном примере требуемое имя извлекается из массива `colNames`.

Наиболее интересен метод `getValueAt()`, переопределяемый в данном классе. Он вычисляет и возвращает значения для всех четырех столбцов. Для первого столбца он лишь возвращает номер строки плюс 2. Для второго столбца вызывается метод `isPrime()`. Он определяет, является ли число в первом столбце простым. Если число простое, метод возвращает значение "Yes", в противном случае — значение "No". Для третьего и четвертого столбцов метод возвращает соответственно квадрат и квадратный корень значения, присутствующего в первом столбце. Таким образом, значения, предназначенные для отображения в каждом столбце, формируются динамически, и массив для их хранения не нужен. Подобный подход можно применить, если вы хотите создать таблицу, которая содержала бы данные, извлекаемые из базы или из файла либо полученные по сети.

Помимо `NumInfoModel`, в программе создается класс `NumInfoTable`. Он выполняет те же действия, что и в предыдущем примере, но при создании компонента `JTable` передает ему вместо массива экземпляр класса `NumInfoModel`.

```
jtabNumInfo = new JTable(new NumInfoModel(99));
```

Таким образом, вместо того, чтобы создавать таблицу на основе массива или вектора, она формируется с использованием модели.

В данном примере запрещено редактирование содержимого ячеек. Если вы хотите предоставить пользователю эту возможность, вам надо переопределить методы `isCellEditable()` и `setValueAt()`. Значение, возвращаемое методом `isCellEditable()`, определяет, может ли пользователь изменять данные, содержащиеся в ячейках. Реализация `isCellEditable()`, предоставляемая по умолчанию классом `AbstractTableModel()`, возвращает значение `false`, т.е. редактирование ячеек запрещено. По этой причине метод `isCellEditable()` нуждается в переопределении. Однако метод `isCellEditable()`, возвращающий значение `true`, еще не решает проблему, поскольку реализация метода `setValueAt()`, принятая по умолчанию, не выполняет никаких действий. Таким образом, даже если `isCellEditable()` вернет для некоторой ячейки значение `true`, отредактированное значение не будет отражено в модели. Чтобы реализовать эту возможность, надо переопределить метод `setValueAt()` так, чтобы он поддерживал обновленные данные. Этот метод должен также генерировать событие `TableModelEvent`, информирующее обработчиков о том, что данные в составе ячейки изменились. Самый простой способ сделать это — вызвать метод `fireTableCellUpdated()`, определенный в классе `AbstractTableModel`. (В качестве примера решения данной задачи можно использовать ответ на вопрос 16 теста для самоконтроля, приведенного в приложении.)



Заметьте также, что при отображении значений в столбце Square Root выводится большое число знаков после десятичной точки. Такой формат поддерживает объект воспроизведения, связанный с ячейкой по умолчанию. Для того чтобы уменьшить длину дробной части (или задать любой другой формат данных, представляемых в таблице), вам надо создать новый объект воспроизведения.

**ВАЖНО!**

## **8.8. Дополнительные возможности таблиц**

В данном модуле мы ограничились лишь поверхностным обсуждением возможностей, предоставляемых компонентом `JTable`. В этом разделе будут кратко рассмотрены еще некоторые средства.

В ряде случаев может оказаться полезной способность `JTable` выводить таблицу на печать. Данную задачу решает метод `print()`. Существует несколько вариантов этого метода. Наиболее простой из них приведен ниже.

```
boolean print() throws java.awt.print.PrinterException
```

Он возвращает значение `true`, если таблица была успешно выведена на печать, или значение `false`, если процесс печати был прерван. Данный метод также вызывает отображение стандартного диалогового окна `Print`. Например, для того, чтобы напечатать таблицу с почтовыми сообщениями, рассмотренную ранее в данном модуле, надо использовать следующий фрагмент кода:

```
try {
    jtabEmail.print();
} catch (java.awt.print.PrinterException exc) { // ... }
```

Этот фрагмент можно включить в один из предыдущих примеров и убедиться, что таблица будет действительно напечатана.

Класс `JTable` предоставляет также свойства для управления разделительными линиями в составе таблицы. Вы можете задать цвет линий, вызвав метод `setGridColor()`. Можно также указать, должны ли разделительные линии отображаться на экране; сделать это позволяет метод `setShowGrid()`. По умолчанию линии выводятся. Если при вызове данного метода вы передадите ему в качестве параметра значение `false`, они не будут отображаться. Можно также по отдельности управлять отображением горизонтальных и вертикальных линий сетки. Для этой цели предназначены методы `setShowVerticalLines()` и `setShowHorizontalLines()`.

Добавить столбец к таблице позволяет метод `addColumn()`. Для удаления столбца следует вызвать `removeColumn()`. Метод `moveColumn()` дает возможность переместить столбец.

При работе с таблицей очень важно иметь возможность установить требуемый объект воспроизведения содержимого ячеек. Как было сказано ранее, объект воспроизведения определяет, как данные, содержащиеся в таблице, будут представлены на экране. Если вы не зададите объект воспроизведения явно, будет использован объект по умолчанию, действия которого ограничиваются вызовом метода `toString()`. Такое решение подходит для простых таблиц, но может быть неприменимо и в более сложных случаях. Например, для отображения финансовой информации требуется специальный формат, принятый в конкретном регионе. Объект воспроизведения представляет собой экземпляр класса, реализующего интерфейс `TableCellRenderer`. Для того чтобы установить объект воспроизведения, надо вызвать метод `setDefaultRenderer()`. Установленный объект воспроизведения будет автоматически применяться для отображения содержимого ячеек. (Пример реализации объекта воспроизведения будет приведен в проекте 8.1.)

При необходимости можно также определить собственный редактор ячеек. Редактор ячеек — это компонент, который активизируется в том случае, когда пользователь предпринимает попытку отредактировать данные в ячейке. Часто возможностей, предоставляемых редактором по умолчанию, вполне достаточно, но в некоторых случаях приходится определять собственный редактор. В роли редактора может выступать объект, представляющий собой экземпляр класса, который реализует интерфейс `TableCellEditor`. Для того чтобы установить редактор, надо вызвать метод `setDefaultEditor()`. Установленный редактор будет вызываться автоматически. Вопрос создания редакторов ячеек не входит в круг задач данной книги. Если вы захотите глубже ознакомиться с возможностями, предоставляемыми Swing, вам придется изучить этот вопрос самостоятельно.

## Вопросы для текущего контроля

1. Какой интерфейс надо реализовать при создании модели таблицы, определяемой разработчиком?
2. Какие методы надо определить в классе, являющемся подклассом `AbstractTableModel`?
3. Какой метод используется для вывода таблицы на печать?

- 
1. `TableModel`.
  2. `getValueAt()`, `getRowCount()` и `getColumnCount()`.
  3. `print()`.

## Проект 8.1. Создание объекта воспроизведения содержимого ячеек

CellRendererDemo.java

В примерах, приведенных в данном модуле, использовался объект воспроизведения содержимого ячеек, принимаемый по умолчанию. Он вполне подходит для многих приложений, однако его возможности ограничены отображением строк или целочисленных значений. Если же, например, вам надо отформатировать значение с плавающей точкой, представить финансовые данные либо отобразить дату и время, вам потребуется связать с таблицей специальный объект воспроизведения. В примере, демонстрирующем применение модели, определяемой разработчиком, квадратный корень числа выводился с помощью объекта воспроизведения по умолчанию. В результате на экране были представлены числа с шестнадцатью цифрами после десятичной точки. Как правило, такая точность совершенно не нужна. Создав свой объект воспроизведения, вы сможете контролировать число цифр дробной части.

Создать объект воспроизведения содержимого ячеек достаточно просто. Как было сказано ранее, он представляет собой экземпляр класса, реализующего интерфейс `TableCellRenderer`. В составе данного интерфейса объявлен только один метод, представленный ниже.

```
Component getTableCellRendererComponent(JTable jtab, Object val,
                                         boolean selected,
                                         boolean focus,
                                         int r, int c);
```

Здесь параметр `jtab` представляет собой ссылку на таблицу, генерирующую запрос. Воспроизводимый объект передается посредством параметра `val`. Если значение параметра `selected` равно `true`, ячейка должна воспроизводиться как выбранная. Если значение параметра `focus` равно `true`, ячейка должна воспроизводиться как имеющая фокус ввода. Координаты ячейки (строка и столбец) задаются посредством параметров `r` и `c`. Метод возвращает объект `Component`, предназначенный для воспроизведения значения.

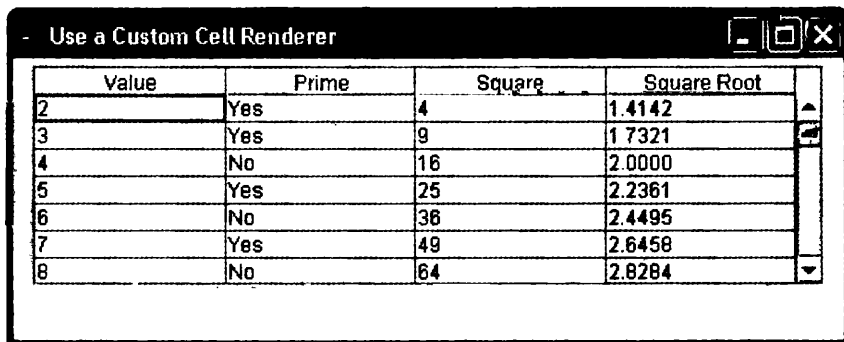
Несмотря на то что методы, объявленные в составе интерфейса `TableCellRenderer`, можно реализовать самостоятельно, проще использовать в качестве базового объект, реализуемый классом `DefaultTableCellRenderer`. Класс `DefaultTableCellRenderer` представляет собой объект воспроизведения, используемый компонентом `JTable` по умолчанию. Данный класс является подклассом класса `JLabel`. Таким образом, для воспроизведения данных применяются средства `JLabel`. При необходимости вы можете

использовать любые свойства и методы класса `JLabel`. Например, для воспроизведения форматированного значения достаточно вызвать метод `setText()` объекта `DefaultTableCellRenderer`.

Для установки объекта воспроизведения, определенного разработчиком, надо обратиться к методу `setDefaultRenderer()` компонента `JTable`.

```
void setDefaultRenderer(Class<?> cl, TableCellRenderer tcr)
```

Здесь `cl` — это тип данных, предназначенных для воспроизведения. Например, для отображения чисел с двойной точностью надо передать методу значение `Double.class`. Объект воспроизведения задается посредством параметра `tcr`. Таким образом, после выполнения метода `setDefaultRenderer()` объект воспроизведения, созданный разработчиком, будет автоматически использоваться каждый раз, когда в составе таблицы необходимо будет отобразить значение указанного типа. В данном проекте создается объект воспроизведения, который будет использован классом `NumInfoTable` для того, чтобы в столбце `Square Root` отображались значения, содержащие четыре цифры после десятичной точки. После установки объекта воспроизведения, определяемого пользователем, таблица будет выглядеть так, как показано ниже.



| Value | Prime | Square | Square Root |
|-------|-------|--------|-------------|
| 2     | Yes   | 4      | 1.4142      |
| 3     | Yes   | 9      | 1.7321      |
| 4     | No    | 16     | 2.0000      |
| 5     | Yes   | 25     | 2.2361      |
| 6     | No    | 36     | 2.4495      |
| 7     | Yes   | 49     | 2.6458      |
| 8     | No    | 64     | 2.8284      |

## Последовательность действий

1. Скопируйте код программы `NumInfoTable`, рассмотренной ранее, в файл `CellRendererDemo.java`.
2. Для форматирования значений типа `double` в столбце `Square Root` будет использован объект `NumberFormat`. Класс `NumberFormat` содержится в пакете `java.text`, поэтому в начало программы следует включить следующее выражение:

```
import java.text.*;
```

3. Начните класс `MyRenderer`, предназначенный для воспроизведения данных таблицы, следующим кодом:

```
// Простой объект воспроизведения, ограничивающий точность
// значений double четырьмя знаками после десятичной точки.
class MyRenderer extends DefaultTableCellRenderer {
```

Обратите внимание на то, что класс `MyRenderer` является подклассом `DefaultTableCellRenderer`. Поскольку данный класс лишь расширяет базовые возможности объекта воспроизведения по умолчанию, вы избавлены от необходимости определять все методы.

4. Переопределите метод `getTableCellRendererComponent()`, начав его следующим образом:

```
// Переопределение метода getTableCellRendererComponent().
public Component getTableCellRendererComponent(
    JTable jtab, Object v,
    boolean selected, boolean focus,
    int r, int c) {
    // Получение компонента по умолчанию.
    JLabel rendComp =
        (JLabel) super.getTableCellRendererComponent(
            jtab, v, selected, focus, r, c);
```

Обратите внимание на вызов метода `getTableCellRendererComponent()` суперкласса. Ссылка на объект, возвращенный этим методом, сохраняется в переменной `rendComp`. Как вы помните, `DefaultTableCellRenderer` является подклассом `JLabel`. Это означает, что значение, возвращаемое методом `getTableCellRendererComponent()`, можно привести к типу `JLabel`.

5. Создайте объект форматирования, который будет форматировать значения `double` так, чтобы на экране отображались четыре знака после точки. Для этого надо прежде всего получить объект `NumberFormat`. Затем следует задать максимальное и минимальное число знаков в дробной части числа, равное четырем. Фрагмент программы, выполняющий описанные действия, выглядит следующим образом:

```
// Получение объекта форматирования.
NumberFormat nf = NumberFormat.getNumberInstance();
// Установки для отображения четырех знаков
// в дробной части.
nf.setMaximumFractionDigits(4);
nf.setMinimumFractionDigits(4);
```

6. Отформатируйте значение и поместите результат в объект `rendComp`.

```
// В качестве текста метки устанавливается
// отформатированное значение.
rendComp.setText(nf.format(v));
```

Поскольку переменная `rendComp` содержит ссылку на объект `JLabel`, данное выражение задаст текст для отображения. Теперь значение будет выводиться с четырьмя десятичными знаками.

7. Завершите код класса `MyRenderer`. Метод `getTableCellRendererComponent()` должен возвращать значение `rendComp`.

```
// Возвращение пользовательского объекта воспроизведения.
return rendComp;
}
}
```

8. В классе `NumInfoModel` вам следует переопределить метод `getColumnClass()`.

```
// Для столбца, отображающего квадратный корень,
// необходимо вернуть класс Double.
// Для остальных столбцов возвращается класс Object
public Class getColumnClass(int c) {
    if(c==3) return Double.class;
    else return Object.class;
}
```

Данный метод возвращает класс `Object` для всех столбцов, кроме `Square Root`. Для этого столбца возвращается класс `Double`. Средства воспроизведения содержимого ячейки используют для выбора объекта воспроизведения данных значение, предоставленное методом `getColumnClass()`. В этом случае данные в четвертом столбце будут отформатированы как значения `Double` (при условии, что объект воспроизведения для `Double` доступен).

9. После создания объекта `JTable` посредством `NumInfoModel` вам следует установить объект воспроизведения для значений `Double`. Сделать это можно, вызвав метод `setDefaultRenderer()`.

```
// Добавление объекта воспроизведения содержимого ячеек
jtabNumInfo.setDefaultRenderer(Double.class, new
MyRenderer());
```

После вызова данного метода в таблице, на которую ссылается переменная `jtabNumInfo`, все значения `double` будут отображаться посредством объекта `MyRenderer`.

10. Ниже представлен окончательный вариант программы, использующей объект воспроизведения содержимого ячеек, созданный разработчиком.

```
// Проект 8.1. Использование объекта воспроизведения,  
// определяемого разработчиком, с классом NumInfoModel.  
// Объект воспроизведения отображает значение  
// квадратного  
// корня с четырьмя знаками после десятичной точки.  
  
import java.awt.*;  
import javax.swing.*;  
import javax.swing.table.*;  
import java.text.*;  
  
// Простой объект воспроизведения, ограничивающий точность  
// значений double четырьмя знаками после десятичной точки.  
class MyRenderer extends DefaultTableCellRenderer {  
  
    // Переопределение метода getTableCellRendererComponent().  
    public Component getTableCellRendererComponent(  
        JTable jtab, Object v,  
        boolean selected, boolean focus,  
        int r, int c) {  
  
        // Получение компонента по умолчанию.  
        JLabel rendComp =  
            (JLabel) super.getTableCellRendererComponent(  
                jtab, v, selected, focus, r, c);  
  
        // Получение объекта форматирования.  
        NumberFormat nf = NumberFormat.getNumberInstance();  
        // Установки для отображения четырех знаков  
        // в дробной части.  
        nf.setMaximumFractionDigits(4);  
        nf.setMinimumFractionDigits(4);  
  
        // В качестве текста метки устанавливается  
        // отформатированное значение.  
        rendComp.setText(nf.format(v));  
    }  
}
```

```
// Возвращение пользовательского объекта
// воспроизведения.
return rendComp;
}
}

// Создание модели предоставляющей числовые данные.
class NumInfoModel extends AbstractTableModel {
    int numRows;

    String colNames[] = { "Value", "Prime",
        "Square", "Square Root" };

    NumInfoModel(int len) {
        super();
        numRows = len;
    }

    public int getRowCount() { return numRows; }
    public int getColumnCount() { return 4; }

    // Метод возвращает имя столбца по заданному индексу.
    // Имя извлекается из массива colNames.
    public String getColumnName(int c) {
        return colNames[c];
    }

    // Для столбца 0 возвращается номер строки плюс 2.
    // Для столбца 1 возвращается значение "Yes",
    // если число в столбце 0 является простым.
    // Для столбца 2 возвращается квадрат числа,
    // содержащегося в столбце 0.
    // Для столбца 3 возвращается квадратный корень числа,
    // содержащегося в столбце 0.
    public Object getValueAt(int r, int c) {

        if(c==0) return new Integer(r+2);
        else if(c==1) {
            if(isPrime(r+2)) return "Yes";
            else return "No";
        }
        else if(c==2) return new Integer((r+2) * (r+2));
        else return new Double(Math.sqrt(r+2));
    }
}
```



```
}

// Возвращает значение true, если параметр v
// представляет собой простое число.
boolean isPrime(int v) {
    int i;

    for(i=2; i <= v/i; i++)
        if((v%i) == 0) return false;

    return true;
}

// Для столбца, отображающего квадратный корень,
// необходимо вернуть класс Double.
// Для остальных столбцов возвращается класс Object
public Class getColumnClass(int c) {
    if(c==3) return Double.class;
    else return Object.class;
}
}

class CellRendererDemo {

    JTable jtabNumInfo;

    CellRendererDemo() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame(
            "Use a Custom Cell Renderer");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(500, 200);

        // Завершение программы при закрытии окна
        // пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание таблицы, использующей модель NumInfoModel
        // для отображения значений от 2 до 100.
```

```
jtabNumInfo = new JTable(new NumInfoModel(99));

// Добавление объекта воспроизведения, определенного
// разработчиком, ориентированного на обработку
// значений double.
jtabNumInfo.setDefaultRenderer(Double.class,
    new MyRenderer());

// Включение данных в панель с прокруткой.
JScrollPane jsclrp = new JScrollPane(jtabNumInfo);

// Установка размеров прокручиваемой области
// просмотра.
jtabNumInfo.setPreferredScrollableViewportSize(
    new Dimension(450, 110));

// Включение таблицы в состав панели содержимого.
jfrm.getContentPane().add(jsclrp);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new CellRendererDemo();
        }
    });
}
```

**ВАЖНО!****8.9.****Общие сведения о компоненте JTree**

JTree — один из самых интересных компонентов библиотеки Swing. Он представляет иерархическую структуру данных, отображая ее в виде дерева. Термин *иерархическая структура* означает, что одни ее элементы подчинены другим. Например, в виде дерева можно отобразить содержимое файловой системы. В этом случае отдельные файлы будут подчинены каталогам, в которых они находятся. Пользователь может свертывать и разворачивать ветви дерева, в ре-

зультате появляется возможность представлять иерархические данные в компактном виде и при необходимости переходить к их детальному отображению.

Подобно другим компонентам Swing, JTree принадлежит пакету `javax.swing`. Однако многие из классов и интерфейсов, используемых для поддержки древовидной структуры, находятся в пакете `javax.swing.tree`. Поскольку количество таких классов и интерфейсов довольно велико, для них выделен отдельный пакет. Подобно таблицам, деревья допускают настройку в соответствии с требованиями конкретных приложений. К счастью, во многих случаях можно довольствоваться возможностями, предоставляемыми деревом по умолчанию. Таким образом, несмотря на то, что компонент, реализующий дерево, имеет сложную организацию, он достаточно прост в использовании.

Компонент JTree поддерживает сравнительно простую древовидную структуру данных. Дерево берет свое начало в одном корневом узле, которому подчинены один или несколько дочерних узлов. Существуют два типа дочерних узлов: *листья* (их также называют *терминальными узлами*), которые не имеют дочерних узлов, и *узлы-ветви*, которые выступают в качестве корневых узлов для поддеревьев. *Поддерево* — это дерево, являющееся частью другой древовидной структуры. Последовательность узлов, по которым можно перейти от корневого узла к некоторому конкретному узлу, называется *путем*.

С каждым узлом дерева связаны *объект воспроизведения*, который определяет, как должны отображаться данные, а также *редактор*, который задает правила редактирования информации пользователем. (Аналогичные объекты используются и в компоненте JTable.) Компонент JTree предоставляет объект воспроизведения и редактор по умолчанию, которые могут в неизменном виде использоваться во многих приложениях. При необходимости можно также связать с компонентом собственный объект воспроизведения и редактор.

Компонент JTree не поддерживает прокрутку. При необходимости данный компонент помещается в состав JScrollPane. Поскольку обычно дерево состоит из большого количества узлов, панель с прокруткой, как правило, бывает необходимой. Несмотря на то что полностью свернутое дерево очень компактно, в развернутом виде его размеры достаточно велики.

Компонент JTree базируется на двух моделях. Первая — это модель дерева, определяемая с помощью интерфейса `TreeModel`. Эта модель управляет отображением данных в виде древовидной структуры. Swing предоставляет класс `DefaultTreeModel`, представляющий собой реализацию модели дерева по умолчанию. И `TreeModel`, и `DefaultTreeModel` принадлежат пакету `javax.swing.tree`.

Вторая модель определяет порядок выбора пунктов и задается с помощью интерфейса `TreeSelectionModel`. Она также расположена в пакете `javax.swing.tree`. Для дерева поддерживаются три режима выбора. Они задаются константами, определенными в интерфейсе `TreeSelectionModel`. Эти константы приведены ниже.

`CONTIGUOUS_TREE_SELECTION`

`DISCONTIGUOUS_TREE_SELECTION`

`SINGLE_TREE_SELECTION`

В режиме `CONTIGUOUS_TREE_SELECTION` можно выбирать несколько узлов, но они должны принадлежать одному пути. Режим `DISCONTIGUOUS_TREE_SELECTION` также допускает выбор нескольких узлов, но требование принадлежности к одному пути не предъявляется. (Этот режим установлен по умолчанию.) В режиме `SINGLE_TREE_SELECTION` в каждый момент времени может быть выбран лишь один узел. Изменить режим выбора позволяет метод `setSelectionMode()` модели дерева. Swing предоставляет класс `DefaultTreeSelectionModel`, реализующий интерфейс `TreeSelectionModel`.

Компонент `JTree` может генерировать различные события, но непосредственное отношение к древовидным структурам имеют лишь три из них: `TreeSelectionEvent`, `TreeExpansionEvent` и `TreeModelEvent`. Эти события будут описаны в следующем разделе.

В классе `JTree` определены конструкторы, описанные в табл. 8.5. Наиболее часто используется следующий конструктор:

```
JTree(TreeNode tn)
```

Таблица 8.5. Конструкторы класса `JTree`

| Конструктор                                        | Описание                                                                                                                                                                                                                                                                                                                                            |
|----------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JTree()</code>                               | Создает древовидную структуру по умолчанию                                                                                                                                                                                                                                                                                                          |
| <code>JTree(TreeNode tn)</code>                    | Создает дерево, корневым элементом которого является <code>tn</code>                                                                                                                                                                                                                                                                                |
| <code>JTree(TreeNode tn, boolean checkLeaf)</code> | Создает дерево, корневым элементом которого является <code>tn</code> . Если значение параметра <code>checkLeaf</code> равно <code>true</code> , листьями считаются только те узлы, с которыми запрещено связывать дочерние узлы. Если значение параметра равно <code>false</code> , все узлы, не имеющие дочерних, автоматически считаются листьями |

| Конструктор                                   | Описание                                                                                                                                                                                                                                                                                                                                                           |
|-----------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JTree (Hashtable&lt;?, ?&gt; ht)</code> | Создает древовидную структуру на основе хэш-таблицы, заданной посредством параметра <code>ht</code> . В хэш-таблице содержатся пары "ключ–значение". Каждый ключ в <code>ht</code> определяет узел, дочерний по отношению к корневому. Если значение представляет собой массив, его элементы становятся дочерними по отношению к узлу, заданному посредством ключа |
| <code>JTree (Vector&lt;?&gt; v)</code>        | Создает древовидную структуру на основе вектора, заданного посредством параметра <code>v</code> . Каждый элемент вектора <code>v</code> определяет узел, дочерний по отношению к корневому. Если элемент <code>v</code> представляет собой вектор, его элементы рассматриваются как дочерние узлы в составе поддерева                                              |
| <code>JTree (Object obj[])</code>             | Создает древовидную структуру на основе массива, заданного посредством параметра <code>obj</code> . Каждый элемент массива <code>obj</code> определяет узел, дочерний по отношению к корневому. Если элемент <code>obj</code> представляет собой массив, его элементы рассматриваются как дочерние узлы в составе поддерева                                        |
| <code>JTree (TreeModel tm)</code>             | Создает древовидную структуру на основе модели, заданной посредством параметра <code>tm</code>                                                                                                                                                                                                                                                                     |

Параметр `tn` представляет корневой узел дерева. Несмотря на то что дерево можно создавать динамически, обычно оно полностью формируется перед передачей конструктору. Данный вариант конструктора мы будем использовать в примере, демонстрирующем работу компонента `JTree`.

В интерфейсе `TreeNode` объявлены методы, инкапсулирующие информацию об узле дерева. Например, есть возможность получить ссылку на родительский узел или набор дочерних узлов. Можно также выяснить, является ли узел листом. Интерфейс `MutableTreeNode` расширяет `TreeNode`. Он определяет узел, допускающий изменение данных, добавление и удаление дочерних узлов.

Реализацией интерфейса `MutableTreeNode` по умолчанию является класс `DefaultMutableTreeNode`. Это класс может быть использован для формирования узлов, применимых в различных приложениях. Мы используем его в примере, который будет рассмотрен далее в данном модуле. В этом классе объявлены три конструктора. Мы будем использовать следующий:

```
DefaultMutableTreeNode (Object obj)
```

Здесь параметр `obj` представляет собой ссылку на объект, связанный с данным узлом.

Для создания иерархической структуры узлов дерева надо иметь возможность присоединять одни узлы к другим. Эту возможность реализует метод `add()` класса `DefaultMutableTreeNode`. Данный метод определен следующим образом:

```
void add(MutableTreeNode child)
```

Здесь параметр `child` задает изменяемый узел дерева, который добавляется в качестве дочернего к узлу, для которого был вызван метод `add()`. Таким образом, для формирования дерева надо создать корневой узел, а затем добавлять к нему подчиненные узлы.

Программа, приведенная ниже в качестве примера, иллюстрирует процесс формирования дерева. Дерево представляет собой иерархическую структуру, описывающую пищевые продукты. Корневой узел, `Food`, имеет два непосредственных потомка: узлы `Fruit` и `Vegetables`. Дочерними узлами `Fruit` являются `Apples` и `Pears`. В составе `Apples`, `Pears` и `Vegetables` присутствуют листья, например `Winesap` и `Corn`. Окно, создаваемое при работе программы, показано на рис. 8.10.

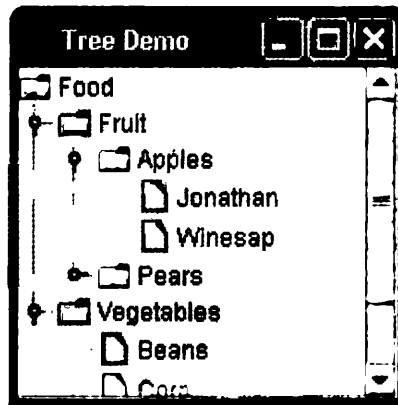


Рис. 8.10. Данные, отображаемые программой `TreeDemo`

```
// Пример, демонстрирующий использование древовидной структуры.
//
// В данной программе создается и отображается дерево.
// События, связанные с ним, не обрабатываются.
```

```
import java.awt.*;
```

```
import javax.swing.*;
import javax.swing.tree.*;

class TreeDemo {

    TreeDemo() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Tree Demo");

        // В данной программе используется диспетчер компоновки
        // BorderLayout, принимаемый по умолчанию.

        // Установка начальных размеров фрейма.
        jfrm.setSize(200, 200);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Процесс формирования дерева начинается с определения
        // узлов и связей между ними.

        // Создание корневого узла дерева.
        DefaultMutableTreeNode root =
            new DefaultMutableTreeNode("Food");

        // Создание двух поддеревьев. Одно из них содержит
        // информацию о фруктах, другое - об овощах.

        // Создание корневого узла поддерева Fruit.
        // Этот узел должен быть дочерним по отношению к корневому.
        DefaultMutableTreeNode fruit =
            new DefaultMutableTreeNode("Fruit");
        root.add(fruit); // Добавление узла Fruit к дереву.

        // В состав поддерева Fruit входят два других поддерева:
        // Apples и Pears.

        // Создание поддерева Apples.
        DefaultMutableTreeNode apples =
            new DefaultMutableTreeNode("Apples");
        fruit.add(apples); // Добавление узла Apples к Fruit.

        // Заполнение поддерева Apples путем добавления к нему
```

```
// узлов, являющихся листьями.
apples.add(new DefaultMutableTreeNode("Jonathan"));
apples.add(new DefaultMutableTreeNode("Winesap"));

// Создание поддерева Pears.
DefaultMutableTreeNode pears =
    new DefaultMutableTreeNode("Pears");
fruit.add(pears); // Добавление узла Pears к Fruit.

// Заполнение поддерева Pears.
pears.add(new DefaultMutableTreeNode("Bartlett"));

// Создание поддерева Vegetables.
// Создание корневого узла поддерева.
DefaultMutableTreeNode veg =
    new DefaultMutableTreeNode("Vegetables");
root.add(veg); // Добавление узла Vegetables к дереву.

// Заполнение поддерева Vegetables.
veg.add(new DefaultMutableTreeNode("Beans"));
veg.add(new DefaultMutableTreeNode("Corn"));
veg.add(new DefaultMutableTreeNode("Potatoes"));
veg.add(new DefaultMutableTreeNode("Rice"));

// Создание компонента JTree на основе сформированной
// ранее структуры.
JTree jtree = new JTree(root);

// Включение дерева в состав панели с прокруткой.
JScrollPane jscrlp = new JScrollPane(jtree);

// Включение дерева в панель содержимого.
jfrm.getContentPane().add(jscrlp, BorderLayout.CENTER);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new TreeDemo();
        }
    });
}
```



```

    }
    });
}
}

```

## Спросим у опытного программиста

**Вопрос.** Напоминают ли объект воспроизведения и редактор ячейки соответствующие объекты класса `JTable`?

**Ответ.** Да. Между ними много общего. Конечно, объекты воспроизведения и редакторы `JTree` ориентированы на работу с древовидной структурой. Для дерева объект воспроизведения создается на основе класса, реализующего интерфейс `TreeCellRenderer`. Редактор ячейки для дерева должен реализовывать интерфейс `TreeCellEditor`. Если вы создадите дерево, описанное в данном модуле, с ним будут связаны объект воспроизведения и редактор по умолчанию. Это объекты `DefaultTreeCellRenderer` и `DefaultTreeCellEditor`. Необходимо помнить, что по умолчанию узлы деревьев не допускают редактирование. Чтобы разрешить их редактирование, надо вызвать метод `setEditable(true)`.

Рассмотрим процесс создания дерева. Сначала создается корневой узел, и ссылка на него помещается в переменную `root`. С этим узлом связывается строка "Food". Далее формируется узел `fruit`, ему ставится в соответствие строка "Fruit", и он добавляется к корневому узлу. Узел `fruit` выступает в роли корневого узла для поддерева, описывающего различные типы фруктов. К узлу `fruit` добавляется узел `apples`, представляющий яблоки. Этот узел также становится корневым узлом поддерева. К узлу `apples` добавляются листья, представляющие различные сорта яблок. Таким же образом создается поддерево `pears`. После формирования поддеревьев `apples` и `pears` получается структура, в которой узел `fruit` имеет два дочерних узла; каждый из них выступает в качестве корневого для отдельного поддерева. Далее создается узел `veg` и добавляется к узлу `root`. Он становится корнем для поддерева `Vegetables`. К узлу `veg` добавляются листья `Beans`, `Corn`, `Potatoes` и `Rice`.

После выполнения описанных действий структура дерева оказывается сформированной и появляется возможность создать компонент `JTree` посредством следующего выражения:

```
JTree jtree = new JTree(root);
```

Здесь конструктору `JTree()` передается параметр `root`. В результате `jtree` будет отображать дерево, "растущее" из узла `root`. Далее компонент `jtree` помещается в панель с прокруткой, которая помещается в центральную область панели содержимого. (Как вы помните, по умолчанию с панелью содержимого связывается диспетчер компоновки `BorderLayout`.)



### Вопросы для текущего контроля

1. Какие две модели использует компонент `JTree`?
2. Что такое `TreeNode`?
3. Какой метод надо вызвать, чтобы связать с узлом дочерний узел?

**ВАЖНО!**

## 8.10. Обработка событий компонента `JTree`

С деревьями непосредственно связаны три события: `TreeSelectionEvent`, `TreeExpansionEvent` и `TreeModelEvent`. Событие `TreeSelectionEvent` возникает при выборе или отмене выбора узла. Событие `TreeExpansionEvent` генерируется в результате свертывания или развертывания дерева. Событие `TreeModelEvent` соответствует изменению данных или структуры дерева. Рассмотрим их подробнее.

### Событие `TreeSelectionEvent`

Для обработки события `TreeSelectionEvent` надо создать класс, реализующий интерфейс `TreeSelectionListener`, и связать его с экземпляром класса `JTree`. В интерфейсе `TreeSelectionListener` объявлен следующий метод:

```
void valueChanged(TreeSelectionEvent tse)
```

1. `TreeModel` и `TreeSelectionModel`.
2. `TreeNode` — это интерфейс, описывающий узел дерева.
3. `add()`.

Параметр `tse` описывает событие выбора.

В классе `TreeSelectionEvent` определено несколько методов. Самый важный из них — метод `getPath()`, объявленный следующим образом:

```
TreePath getPath()
```

Путь к выбранному узлу возвращается в виде объекта `TreePath`.

Класс `TreePath` принадлежит пакету `javax.swing.tree`. Он инкапсулирует путь от корневого узла дерева к выбранному узлу. В данном классе определено несколько методов. Наиболее интересны следующие:

```
Object[] getPath()
```

```
Object getLastPathComponent()
```

Метод `getPath()` возвращает массив объектов, представляющих все узлы в составе пути. Метод `getLastPathComponent()` возвращает ссылку на последний узел пути.

## Событие `TreeExpansionEvent`

Для обработки события `TreeExpansionEvent` надо создать класс, реализующий интерфейс `TreeExpansionListener`, и связать его с экземпляром класса `JTree`. В интерфейсе `TreeExpansionListener` объявлены следующие два метода:

```
void treeCollapsed(TreeExpansionEvent tee)
```

```
void treeExpanded(TreeExpansionEvent tee)
```

где параметр `tee` представляет собой ссылку на событие дерева. Первый метод вызывается при свертывании поддерева, а второй — при его развертывании. Компонент `JTree` позволяет также более детально отслеживать процесс свертывания и развертывания дерева. Можно получать оповещение непосредственно перед событием развертывания. Для этой цели надо зарегистрировать объект `TreeWillExpandListener`. Для получения оповещения после развертывания применяется обработчик `TreeExpansionListener`. Обычно разработчики ограничиваются использованием `TreeExpansionListener`.

В классе `TreeExpansionEvent` определен только один метод, `getPath()`.

```
TreePath getPath()
```

Этот метод возвращает объект `TreePath`, содержащий путь к узлу, над которым была выполнена операция свертывания и развертывания или над которым соответствующие действия должны быть выполнены.

## Событие `TreeModelEvent`

Для обработки события `TreeModelEvent` надо создать класс, реализующий интерфейс `TreeModelListener`, и связать его с моделью дерева. В интерфейсе `TreeModelEvent` объявлены такие методы:

```
void treeNodesChanged(TreeModelEvent tme)

void treeStructureChanged(TreeModelEvent tme)

void treeNodesInserted(TreeModelEvent tme)

void treeNodesRemoved(TreeModelEvent tme)
```

где параметр `tme` представляет собой ссылку на событие модели дерева. Имена методов отражают типы событий модели.

В классе `TreeModelEvent` определено несколько методов. Наиболее интересен метод `getTreePath()`, объявленный следующим образом:

```
TreePath getTreePath()
```

Этот метод возвращает путь к родительскому узлу, потомок которого претерпел изменения.

Как было сказано ранее, обработчик событий модели должен быть связан не с экземпляром класса `JTree`, а с объектом модели. Для получения модели дерева вызывается метод `getModel()` объекта `JTree`. Этот метод определен следующим образом:

```
TreeModel getModel()
```

## Программа, демонстрирующая обработку событий дерева

В данном разделе рассматривается программа, представляющая собой модификацию предыдущего примера. В ней предусмотрена поддержка всех трех событий `JTree`: `TreeSelectionEvent`, `TreeExpansionEvent` и `TreeModelEvent`. Каждый раз, когда возникает одно из этих событий, сообщение, описывающее его, отображается на экране. Окно, создаваемое при работе программы, показано на рис. 8.11.

// Демонстрация обработки событий дерева.

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



*Рис. 8.11. Данные, отображаемые программой TreeEventDemo*

```
class TreeEventDemo {

    JLabel jlab;

    TreeEventDemo() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Tree Event Demo");

        // В данном случае используется диспетчер компоновки
        // BorderLayout, принимаемый по умолчанию.

        // Установка начальных размеров фрейма.
        jfrm.setSize(200, 200);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание метки, отображающей выбор пользователя.
        jlab = new JLabel();

        // Процесс формирования дерева начинается с определения
        // узлов и связей между ними.

        // Создание корневого узла дерева.
        DefaultMutableTreeNode root =
            new DefaultMutableTreeNode("Food");
```

```
// Создание двух поддеревьев. Одно из них содержит
// информацию о фруктах, другое - об овощах.

// Создание корневого узла поддерева Fruit.
DefaultMutableTreeNode fruit =
    new DefaultMutableTreeNode("Fruit");
root.add(fruit); // Добавление узла Fruit к дереву.

// В состав поддерева Fruit входят два других поддерева:
// Apples и Pears.

// Создание поддерева Apples.
DefaultMutableTreeNode apples =
    new DefaultMutableTreeNode("Apples");
fruit.add(apples); // Добавление узла Apples к Fruit.

// Заполнение поддерева Apples путем добавления к нему
// узлов, являющихся листьями.
apples.add(new DefaultMutableTreeNode("Jonathan"));
apples.add(new DefaultMutableTreeNode("Winesap"));

// Создание поддерева Pears.
DefaultMutableTreeNode pears =
    new DefaultMutableTreeNode("Pears");
fruit.add(pears); // Добавление узла Pears к Fruit.

// Заполнение поддерева Pears.
pears.add(new DefaultMutableTreeNode("Bartlett"));

// Создание корневого узла поддерева Vegetables.
DefaultMutableTreeNode veg =
    new DefaultMutableTreeNode("Vegetables");
root.add(veg); // Добавление узла Vegetables к дереву.

// Заполнение поддерева Vegetables.
veg.add(new DefaultMutableTreeNode("Beans"));
veg.add(new DefaultMutableTreeNode("Corn"));
veg.add(new DefaultMutableTreeNode("Potatoes"));
veg.add(new DefaultMutableTreeNode("Rice"));

// Создание компонента JTree на основе сформированной
// ранее структуры.
JTree jtree = new JTree(root);
```

```
// Разрешение редактирования дерева с тем, чтобы
// события могли генерироваться.
jtree.setEditable(true);

// Установка режима выбора одного узла дерева.
TreeSelectionModel tsm = jtree.getSelectionModel();
tsm.setSelectionMode(
    TreeSelectionModel.SINGLE_TREE_SELECTION);

// Включение дерева в состав панели с прокруткой.
JScrollPane jscrlp = new JScrollPane(jtree);

// Обработка событий развертывания дерева.
jtree.addTreeExpansionListener(new TreeExpansionListener() {
    public void treeExpanded(TreeExpansionEvent tse) {
        // Получение пути к узлу.
        TreePath tp = tse.getPath();

        // Отображение узла,
        jlab.setText("Expansion: " +
            tp.getLastPathComponent());
    }

    public void treeCollapsed (TreeExpansionEvent tse) {
        // Получение пути к узлу.
        TreePath tp = tse.getPath();

        // Отображение узла.
        jlab.setText("Collapse: " +
            tp.getLastPathComponent());
    }
});

// Обработка событий выбора.
jtree.addTreeSelectionListener(new TreeSelectionListener() {
    public void valueChanged(TreeSelectionEvent tse) {
        // Получение пути к узлу.
        TreePath tp = tse.getPath();

        // Отображение выбранного узла.
        jlab.setText("Selection event: " +
            tp.getLastPathComponent());
    }
});
```

```
    }  
    });  
  
    // Обработка событий модели дерева. Заметьте, что  
    // обработчик связывается с моделью.  
    jtree.getModel().addTreeModelListener(  
        new TreeModelListener() {  
            public void treeNodesChanged(TreeModelEvent tse) {  
                // Получение пути к узлу, претерпевшему изменения.  
                TreePath tp = tse.getTreePath();  
  
                // Отображение пути.  
                jlab.setText("Model change path: " + tp);  
            }  
  
            // реализация остальных методов TreeModelEvent.  
            // В данном случае методы не выполняют никаких действий.  
            public void treeNodesInserted(TreeModelEvent tse) {}  
            public void treeNodesRemoved(TreeModelEvent tse) {}  
            public void treeStructureChanged(TreeModelEvent tse) {}  
        });  
  
    // Включение дерева и метки в панель содержимого.  
    jfrm.getContentPane().add(jscrlp, BorderLayout.CENTER);  
    jfrm.getContentPane().add(jlab, BorderLayout.SOUTH);  
  
    // Отображение фрейма.  
    jfrm.setVisible(true);  
}  
  
public static void main(String args[]) {  
    // Фрейм создается в потоке обработки событий.  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            new TreeEventDemo();  
        }  
    });  
}
```



В данной программе используется дерево, имеющее ту же структуру, что и в предыдущем примере. Порядок формирования дерева также не изменился. Для обработки событий в программу были включены обработчики, а также добавлены некоторые фрагменты кода. Рассмотрим внесенные изменения более подробно. Обратите внимание на новую метку `jlab`. Она используется для отображения информации о событиях. Эта метка размещается в южной области панели содержимого.

Для того чтобы события модели могли генерироваться, разрешается редактирование данных. С этой целью вызывается метод `setEditable(true)` компонента `jtree`. По умолчанию редактирование запрещено. Заголовок метода `setEditable()` имеет следующий вид:

```
void setEditable(boolean canEdit)
```

Если значение параметра `canEdit` равно `true`, узлы дерева допускают редактирование. Значение `false` данного параметра запрещает редактирование содержимого узлов.

По умолчанию дерево допускает выбор нескольких узлов. В программе режим выбора изменен так, чтобы пользователь мог в каждый момент времени выбрать только один узел. Для этого служат следующие выражения:

```
TreeSelectionMode tsm = jtree.getSelectionModel();  
tsm.setSelectionMode(TreeSelectionMode.SINGLE_TREE_SELECTION);
```

Как было сказано ранее, режим выбора управляется моделью. Следовательно, для того, чтобы вызвать метод `setSelectionMode()`, надо получить доступ к модели.

Далее в программе регистрируются три обработчика событий. В обработчике, реализующем интерфейс `TreeExpansionListener`, определяются методы `treeExpanded()` и `treeCollapsed()`. В теле этих методов содержится код, предназначенный для отображения сведений о разворачивании или сворачивании узла. В обработчике, созданном на базе интерфейса `TreeSelectionListener`, определен метод `valueChanged()`, который отображает сведения о выбранном узле. В обработчике `TreeModelListener` реализован метод `treeNodesChanged()`, который выводит путь к узлу, претерпевшему изменения. Этот метод получает управление при редактировании узла. Реализации остальных трех методов, объявленных в `TreeModelListener`, не выполняют никаких действий, поскольку эти методы не используются программой.

}

ВАЖНО!

## 8.11. Дополнительные возможности компонента JTree

Подобно таблицам, деревья предоставляют разработчикам ряд дополнительных возможностей. Например, вы можете определить модель, используемую древовидной структурой. Также можно связывать с деревом объекты воспроизведения и редакторы. Однако чаще всего модель, объект воспроизведения и редактор, связанные по умолчанию с деревом, удовлетворяют требованиям приложения, и чтобы изменять их, нужны веские причины.

В ряде случаев при работе с компонентом JTree может оказаться полезным метод `setRootVisible()`. Он определяет, должен ли корневой узел отображаться на экране.

```
void setRootVisible(boolean visible)
```

Если значение параметра `visible` равно `true`, корневой узел отображается на экране. В противном случае он скрыт от пользователя. Скрывать корневой узел имеет смысл тогда, когда он не несет полезной информации. В этом случае дерево занимает меньше места на экране.

Часто полезной оказывается возможность отключения маркера корневого узла. При использовании некоторых стилей с корневым узлом может быть связан маркер — небольшая пиктограмма, указывающая на то, развернута или свернута соответствующая ветвь. Включить или отключить отображение маркера позволяет метод `setShowsRootHandles()`, показанный ниже.

```
void setShowsRootHandles(boolean on)
```

Если значение параметра `on` равно `true`, маркер отображается на экране, в противном случае его вывод запрещается. Класс `DefaultMutableTreeNode` также предоставляет ряд интересных возможностей. Ниже описаны некоторые из них.

Изменить содержимое узла позволяет метод `setUserObject()`.

```
void setUserObject(Object obj)
```

Здесь `obj` — это новый объект, связываемый с узлом, которому принадлежит данный метод.

Удалить узел можно, вызвав метод `remove()` родительского узла. Один из вариантов этого метода показан ниже.

```
void remove(MutableTreeNode node)
```

Здесь `node` должен быть узлом, дочерним по отношению к текущему.

Определить, является ли узел листом, можно, вызвав метод `isLeaf()`, показанный ниже.

```
boolean isLeaf()
```

Он возвращает значение `true`, если текущий узел является терминальным (т.е. не имеет дочерних узлов), и значение `false` в противном случае.

В классе `DefaultMutableTreeNode` определены методы, каждый из которых возвращает объект `Enumeration`, представляющий узлы дерева. В этих методах используются различные способы обхода дерева, влияющие на порядок следования узлов.

```
Enumeration breadthFirstEnumeration()
```

```
Enumeration depthFirstEnumeration()
```

```
Enumeration postorderEnumeration()
```

```
Enumeration preorderEnumeration()
```

Обход дерева в ширину (метод `breadthFirstEnumeration()`) начинается с корневого узла; перебираются все узлы текущего уровня, и лишь потом осуществляется переход на следующий уровень. При обходе дерева в глубину в обратном направлении (это принцип реализуют методы `depthFirstEnumeration()` и `postorderEnumeration()`) сначала посещаются все листья, а затем корневой узел поддерева. При обходе в глубину в прямом направлении (метод `preorderEnumeration()`) сначала обрабатывается корень поддерева, а затем его листья.

### Вопросы для текущего контроля

1. При развертывании узла генерируется событие \_\_\_\_\_.
2. Какое событие возникает при изменении данных в составе узла?
3. Что такое `TreePath`?
4. Какой метод реализует обход в глубину в прямом направлении?

---

1. `TreeExpansionEvent`.

2. `TreeModelEvent`.

3. `TreePath` — это класс, содержащий узлы на пути от корневого узла к заданному узлу.

4. `preorderEnumeration()`.

**Проект 8.2. Обход дерева**

`TreeEnum.java` Данный проект демонстрирует принцип получения перечня узлов дерева, рассматриваемого в этом модуле в качестве примера. В данном случае осуществляется обход в глубину в прямом направлении, однако тот же принцип может применяться при любом другом способе обхода дерева.

**Последовательность действий**

1. Скопируйте программу `TreeDemo` в файл `TreeEnum.java`.
2. Добавьте следующее выражение `import`:

```
import java.util.*;
```

Данный пакет необходим, поскольку в программе используется класс `Enumeration`.

3. После создания дерева включите следующие строки кода:

```
Enumeration preorder = root.preorderEnumeration();
while(preorder.hasMoreElements())
    System.out.println(preorder.nextElement());
```

При выполнении данных команд осуществляется обход дерева вдоль каждой ветви, начиная с корневого узла. Для простоты результаты выводятся на консоль, но вы можете организовать отображение информации с помощью метки или вывод в текстовый файл. Выходные данные показаны ниже.

```
Food
Fruit
Apples
Jonathan
Winesap
Pears
Bartlett
Vegetables
Beans
Corn
Potatoes
Rice
```

4. Как видно из данного проекта, получить список всех узлов дерева очень просто. При необходимости вы можете выбрать другой порядок обхода. Поэкспериментируйте с различными методами, чтобы вам стало ясно, чем разные способы обхода отличаются друг от друга.



## Тест для самоконтроля по модулю 8

1. Для обеспечения работы `JTable` используются несколько классов и интерфейсов. В каком пакете они находятся?
2. Какой метод позволяет задать размеры прокручиваемой области просмотра для таблицы?
3. Как называется модель, определяющая таблицу?
4. Если компонент `JTable` не помещен в панель с прокруткой, для отображения заголовка таблицы необходимо предпринять специальные действия. Да или нет?
5. Предположим, вы хотите организовать обработку событий выбора для таблицы. С каким объектом надо связать обработчик `ListSelectionListener`?
6. Для обработки событий модели таблицы надо зарегистрировать обработчик \_\_\_\_\_.
7. Какие действия выполняет метод `setValueAt()` для модели `TableModel`.
8. Какой метод класса `JTable` изменяет режим автоматического изменения размеров?
9. Какой класс следует расширить для того, чтобы реализовать модель таблицы, определяемую разработчиком?

| A Tip Calculator |         |         |         |           |
|------------------|---------|---------|---------|-----------|
| Price            | 10% Tip | 15% Tip | 20% Tip | Suggested |
| \$47.00          | \$4.70  | \$7.05  | \$9.40  |           |
| \$48.00          | \$4.80  | \$7.20  | \$9.60  |           |
| \$49.00          | \$4.90  | \$7.35  | \$9.80  | \$8.00    |
| \$50.00          | \$5.00  | \$7.50  | \$10.00 |           |
| \$51.00          | \$5.10  | \$7.65  | \$10.20 |           |

New suggested tax: \$8.00

10. Какой интерфейс определяет объект воспроизведения для таблицы?
11. Компонент `JTree` применим только для отображения бинарных деревьев. Да или нет?
12. Что такое лист?
13. Какие действия надо выполнить для создания дерева из объектов `DefaultMutableTreeNode`?
14. Какое событие возникает при изменении данных в составе дерева?
15. Какой метод надо вызвать для формирования объекта `Enumeration`, содержащего узлы дерева, посредством обхода в глубину в обратном порядке?
16. Создайте программу, которая выводит таблицу, помогающую рассчитывать стоимость товара. В первом столбце должна отображаться цена изделий с шагом 1. В последующих трех столбцах отображаются величины 10%, 20% и 30% от этой суммы. В последнем столбце пользователь должен иметь возможность ввести свое предложение. Используйте объект воспроизведения, отображающий значения в формате денежных единиц. Предусмотрите в программе возможность указания числа строк при создании таблицы. Готовая таблица должна выглядеть так, как показано ниже.
17. Какое выражение надо включить в программу `TreeDemo`, чтобы корневой узел не отображался на экране?
18. Измените проект 8.2 так, чтобы программа при посещении узла сообщала, является ли он листом или имеет дочерние узлы.
19. Используя код `TreeDemo` в качестве примера, создайте программу, в которой древовидная структура использовалась бы для отображения вашего собрания книг. Поместите каждую книгу в соответствующую категорию. Например, данная книга будет являться листом, а путь к ней будет иметь вид `Programming | Java | Swing`.



# Модуль 9

## Диалоговые окна

- 9.1. Общие сведения о классе `JOptionPane`
- 9.2. Создание диалоговых окон с помощью метода `showMessageDialog()`
- 9.3. Создание диалоговых окон с помощью метода `showConfirmDialog()`
- 9.4. Создание диалоговых окон с помощью метода `showInputDialog()`
- 9.5. Создание диалоговых окон с помощью метода `showOptionDialog()`
- 9.6. Использование класса `JDialog` для создания диалоговых окон
- 9.7. Создание немодальных диалоговых окон
- 9.8. Выбор файлов с помощью окна `JFileChooser`
- 9.9. Использование фильтров, взаимодействующих с компонентом `JFileChooser`
- 9.10. Дополнительные возможности `JFileChooser`
- 9.11. Выбор цвета с помощью окна `JColorChooser`



Управляющие элементы, предоставляемые Swing, например поля редактирования, кнопки и списки, удовлетворяют основным требованиям разработчиков графических пользовательских интерфейсов. Однако бывают случаи, когда необходимо объединить несколько элементов и предоставить тем самым возможность выполнять более сложные операции ввода. Например, если в программе для получения доступа к сети надо ввести пользовательское имя и пароль, желательно оформить два поля редактирования и кнопку подтверждения в виде единого модуля. Сделать это позволяют *диалоговые окна*.

Диалоговые окна специально предназначены для получения данных от пользователей. Минимальный набор элементов в диалоговом окне — метка и кнопка, но в большинстве случаев пользователю предоставляется более богатый набор элементов. Диалоговые окна позволяют решать две важные задачи. Во-первых, с их помощью можно организовывать достаточно сложные структуры для ввода данных пользователем. Эти структуры предоставляют гораздо более обширные возможности, чем отдельный компонент. Например, в текстовом процессоре посредством диалогового окна пользователь может устанавливать характеристики шрифта. Такое диалоговое окно включает в себя набор отдельных компонентов. Во-вторых, диалоговое окно позволяет приостановить работу программы до тех пор, пока пользователь не введет требуемую информацию. Так, посредством диалогового окна можно запросить у пользователя пароль и дождаться его получения. Независимо от особенностей использования, диалоговые окна — важная часть графического интерфейса, создаваемого средствами Swing.

Swing предоставляет мощные средства для поддержки диалоговых окон. Основным классом, предназначенным для этой цели, является `JDialog`, однако в большинстве случаев необходимость в непосредственном его использовании не возникает. Вместо этого применяются другие классы, например `JOptionPane`, предоставляющий большое количество встроенных средств. Кроме того, в составе Swing имеются предопределенные диалоговые окна, реализуемые посредством классов `JFileChooser` и `JColorChooser`. Они решают довольно сложные и в то же время часто возникающие задачи. В данном модуле мы рассмотрим три указанных выше класса, но основное внимание уделим классу `JOptionPane`.

**ВАЖНО!**

## **9.1. Класс `JOptionPane`**

В современных средах, предназначенных для создания графических интерфейсов, используются две разновидности диалоговых окон. Первый вариант окна по сути представляет собой составной компонент, предназначенный для ввода информации пользователем. В качестве примера можно привести диалоговое окно, применяемое для настройки модема. Компоненты, содержащиеся в

этом окне, позволяют пользователю задать скорость и протокол, указать инициализационную команду и т.д. Для того чтобы создать такое сложное диалоговое окно, приходится прибегать к помощи класса `JDialog`, который обсуждается в этом модуле.

Диалоговые окна второго типа намного проще. Они лишь выводят сообщения и ожидают реакции пользователя. Одно из самых простых диалоговых окон запрашивает подтверждение на завершение работы программы ("Exit? Yes/No"). Поскольку простые диалоговые окна, подобные упомянутому выше, широко применяются, в библиотеке Swing предусмотрены специальные средства для их поддержки — класс `JOptionPane`.

Класс `JOptionPane` прост в использовании и позволяет решать многие проблемы. Он поддерживает четыре основных типа диалоговых окон, перечисленные ниже.

- Окна для вывода сообщений.
- Окна для получения подтверждения от пользователя.
- Окна для ввода данных.
- Окна для установки опций.

Диалоговое окно с сообщением отображается на экране до тех пор, пока пользователь не щелкнет на кнопке **ОК**. Это окно — простое и эффективное средство, позволяющее убедиться в том, что пользователь получил определенную информацию. Например, вы можете использовать диалоговое окно данного типа, чтобы сообщить пользователю о том, что сетевое соединение разорвано (сообщение "Connection Lost").

В диалоговом окне, предназначенном для получения подтверждения от пользователя, отображается вопрос типа "да/нет?" и ожидается ответ. Например, окно с сообщением "Exit without saving changes?" может быть использовано в тех случаях, когда надо убедиться, что пользователь действительно хочет закончить работу с программой и не собирается сохранять изменения, внесенные в документ.

Окно для ввода данных предоставляет возможность ввести строку или выбрать пункт из списка. В отличие от предыдущих окон в этом окне действия пользователя не ограничиваются щелчком на кнопке; пользователь может сам формировать ответ на вопрос, заданный программой. Этот тип диалогового окна можно применить, например, для получения URL требуемого ресурса.

Диалоговое окно для установки опций, как правило, отображает набор опций, позволяющий выбирать нужные значения. Таким образом, это окно позволяет выполнять действия, недоступные посредством окон других типов.

Как ни странно, класс `JOptionPane` не является потомком `JDialog`. Окно `JOptionPane` представляет собой контейнер для компонентов, которые должны присутствовать в нем. В процессе работы `JOptionPane` динамически формирует объект `JDialog`, устанавливает характеристики отображаемого диалогового окна, получает ответ и закрывает окно. Одним словом, объект `JOptionPane` обеспечивает создание простого диалогового окна и управление им.

Все диалоговые окна, создаваемые с помощью класса `JOptionPane`, являются *модальными*. Модальное окно выполняется в том же потоке, что и вызывающий его метод, поэтому выполнение потока приостанавливается до получения ответа от пользователя. Для того чтобы передать фокус вводу другому элементу приложения, надо сначала закрыть окно. Таким образом, с точки зрения пользователя, выполнение программы прекращается до тех пор, пока он не выполнит предлагаемые действия. Несмотря на то что в приложениях чаще всего используются модальные диалоговые окна, можно также создавать *немодальные* окна. Немодальное диалоговое окно выполняется в отдельном потоке, поэтому, в то время, когда оно отображается на экране, пользователь может работать с другими элементами приложения. Класс `JOptionPane` не позволяет создавать немодальные окна. Для этого надо воспользоваться классом `JDialog`, который будет описан ниже.

Создать диалоговое окно `JOptionPane` можно, используя конструктор данного класса, однако в большинстве случаев разработчики действуют по-другому. Они используют один из фабричных методов `show...()`. Эти методы автоматически конструируют диалоговое окно одного из четырех описанных выше типов. Например, для того, чтобы создать простое окно для отображения сообщений, надо использовать метод `showMessageDialog()`. Он формирует диалоговое окно, которое выводит на экран сообщение и ожидает щелчка на кнопке ОК. Как вы вскоре увидите, фабричные методы существенно упрощают работу с классом `JOptionPane`.

Класс `JOptionPane` поддерживает две основные разновидности метода `show...()`. Методы первой категории используют для формирования диалогового окна класс `JDialog`. Именно такие окна, используемые чаще других, будут описаны в данной книге. Методы второй категории используют при создании диалогового окна класс `JInternalFrame`. Окна, сформированные таким образом, встречаются гораздо реже. Их рассмотрение выходит за рамки данной книги.

В классе `JOptionPane` определены четыре фабричных метода, позволяющих создавать стандартные диалоговые окна на базе класса `JDialog`: `showConfirmDialog()`, `showInputDialog()`, `showMessageDialog()` и `showOptionDialog()`. Имя метода отражает тип создаваемого окна. Все четыре метода являются статическими. Для первых трех методов существует не-

сколько вариантов. Более подробно каждый метод будет рассмотрен при обсуждении диалогового окна соответствующего типа.

На заметку



Для формирования диалоговых окон на основе класса `JInternalFrame` также используются четыре фабричных метода `show...()`. Их имена отличаются от имен методов, рассмотренных выше, наличием слова `Internal`. Так, если для создания окна на базе `JDialog`, предназначенного для вывода сообщений, используется метод `showMessageDialog()`, то аналогичное окно на основе `JInternalFrame` создается посредством метода `showInternalMessageDialog()`.

Диалоговые окна



## Вопросы для текущего контроля

1. Назовите четыре типа диалоговых окон, которые позволяет создать класс `JOptionPane`.
2. В чем различие между модальным и немодальным диалоговым окном.
3. Является ли класс `JOptionPane` подклассом `JDialog`?

**ВАЖНО!**

**9.2.**

## Метод `showMessageDialog()`

Метод `showMessageDialog()` создает самый простой тип диалогового окна. Это окно отображает сообщение и присутствует на экране до тех пор, пока пользователь не щелкнет на кнопке ОК. Несмотря на простоту создаваемого окна, предусмотрены три варианта метода `showMessageDialog()`. Наиболее простой из них приведен ниже.

```
static void showMessageDialog(Component parent, Object msg)
    throws HeadlessException
```

1. Диалоговые окна для вывода сообщений, для получения подтверждения от пользователя, для ввода данных и установки опций.
2. Модальное диалоговое окно выполняется в том же потоке, что и вызывающий метод, поэтому приостанавливает работу программы. Немодальное окно выполняется в отдельном потоке, поэтому остальные компоненты программы остаются активизированными.
3. Нет.

Здесь параметр `parent` определяет компонент, которому принадлежит отображаемое окно, т.е. компонент, выступающий в роли родительского. Если в качестве первого параметра вы зададите значение `null`, то окно будет размещено по центру экрана. Сообщение, которое должно быть выведено, передается с помощью параметра `msg`. Это не обязательно должна быть строка текста. В качестве сообщения можно, например, задать компонент `JLabel`. Однако для простых диалоговых окон чаще всего используется объект `String`. В состав отображаемого окна включается кнопка **ОК**. Окно отображается на экране до тех пор, пока пользователь не щелкнет на этой кнопке. Таким образом, `showMessageDialog()` не вернет управление вызывающему методу до тех пор, пока пользователь не щелкнет на кнопке **ОК** или не закроет диалоговое окно щелчком на соответствующей кнопке в его верхнем правом углу. Как было сказано ранее, все диалоговые окна, создаваемые методами `show...()`, являются модальными. Это означает, что поток, из которого они вызываются, приостанавливает работу до тех пор, пока управление не будет возвращено вызывающему методу. Так как пользователь может выполнить только одно действие, а именно закрыть окно, возвращаемое значение не нужно, поэтому для данного метода указан тип `void`.

Заметьте, что метод `showMessageDialog()` может генерировать исключение `HeadlessException`. Это исключение возникает при попытке отобразить диалоговое окно в неинтерактивном окружении, например, в системе, в которой отсутствует экран, клавиатура или мышь. Поскольку использование библиотеки `Swing` обычно предполагает наличие интерактивного графического окружения, обрабатывать это исключение не обязательно. Однако, если вы напишете код, ко-

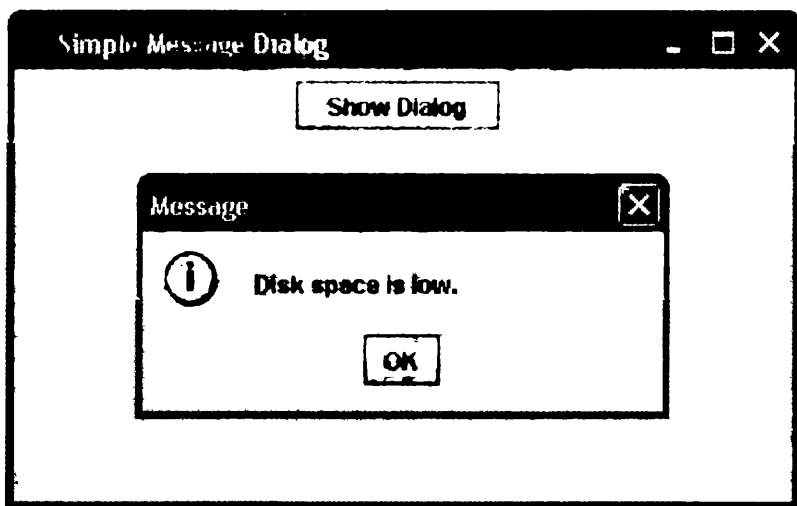


Рис. 9.1. Окно, отображаемое при работе программы `MsgDialogDemo`

торый может выполняться в среде, не поддерживающей графический интерфейс, желательно предусмотреть в нем обработку `HeadlessException` и принять меры на тот случай, если диалоговое окно не может быть выведено на экран.

Перед тем как продолжить разговор о диалоговых окнах, имеет смысл рассмотреть пример использования наиболее простого из них. В программе, код которой приведен ниже, применяется метод `showMessageDialog()`. С его помощью отображается диалоговое окно, которое сообщает пользователю о том, что на диске осталось слишком мало места. Окно, создаваемое при работе программы, показано на рис. 9.1.

// Пример использования компонента `JOptionPane`.

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

class MsgDialogDemo {

    JLabel jlab;
    JButton jbtnShow;
    JFrame jfrm;

    MsgDialogDemo() {
        // Создание нового контейнера JFrame.
        jfrm = new JFrame("Simple Message Dialog");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(400, 250);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание метки, отображающейся после закрытия
        // диалогового окна.
        jlab = new JLabel();

        // Создание кнопки, вызывающей отображение диалогового окна.
        jbtnShow = new JButton("Show Dialog");

        // Связывание с кнопкой обработчика событий.
```

```
jbtnShow.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        // Создание диалогового окна, отображающего сообщение.
        JOptionPane.showMessageDialog(jfrm,
                                     "Disk space is low.");

        // Данное выражение не будет выполнено до тех пор, пока
        // метод showMessageDialog() не завершит работу.
        jlab.setText("Dialog Closed");
    }
});

// Включение кнопки и метки в панель содержимого.
jfrm.getContentPane().add(jbtnShow);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new MsgDialogDemo();
        }
    });
}
```

Работает программа следующим образом. В основном окне отображается кнопка Show Dialog. По щелчку на данной кнопке обработчик событий действия, связанный с ней, выводит диалоговое окно, вызывая метод `showMessageDialog()`. Считается, что `jfrm` принадлежит главному окну программы. В диалоговом окне отображается сообщение "Disk space is low". Окно появляется на экране в результате вызова метода `showMessageDialog()`. Диалоговое окно обладает фокусом ввода, и, поскольку оно является модальным, фокус не может быть передан главному окну. Таким образом, главное окно будет неактивизированным до тех пор, пока диалоговое окно не закроется, а это произойдет, если пользователь щелкнет на кнопке ОК либо на кнопке в верхнем правом углу окна. После закрытия диалогового окна метод

`showMessageDialog()` вернет управление вызывающему методу, а в главном окне посредством метки `jlab` будет выведено соответствующее сообщение.

Существуют два других варианта метода `showMessageDialog()`, позволяющие задавать некоторые характеристики диалогового окна. Один из этих методов показан ниже.

```
static void showMessageDialog(Component parent, Object msg,
                             String title, int msgT)
                             throws HeadlessException
```

Первые два параметра уже знакомы вам. Параметр `title` позволяет задавать заголовок диалогового окна. По умолчанию для окна принимается заголовок `Message` (см. рис. 9.1), однако желательно указать заголовок, отражающий тип сообщения в окне. Параметр `msgT` задает тип сообщения. Значение данного параметра должно быть равно одной из следующих констант, объявленных в классе `JOptionPane`.

|                                  |                                                                                                                                                                   |
|----------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ERROR_MESSAGE</code>       | Отображение сообщения о наличии ошибки. В окне выводится стандартная пиктограмма ошибки                                                                           |
| <code>INFORMATION_MESSAGE</code> | Отображение информационного сообщения. В окне выводится стандартная пиктограмма, обозначающая сообщение данного типа. Этот тип сообщения принимается по умолчанию |
| <code>PLAIN_MESSAGE</code>       | Отображение сообщения, не сопровождаемого пиктограммой                                                                                                            |
| <code>QUESTION_MESSAGE</code>    | Отображение сообщения-вопроса. В окне выводится пиктограмма в виде вопросительного знака                                                                          |
| <code>WARNING_MESSAGE</code>     | Отображение предупреждающего сообщения. В окне выводится стандартная пиктограмма, обозначающая предупреждение                                                     |

Подобно многим другим характеристикам компонентов Swing, конкретное действие параметра `msgT` зависит от используемого стиля.

Для того чтобы увидеть, как влияет указание заголовка и типа сообщения, замените в приведенной выше программе вызов метода `showMessageDialog()` следующим выражением:

```
JOptionPane.showMessageDialog(jfrm,
                               "Disk space is low.",
                               "Warning",
                               JOptionPane.WARNING_MESSAGE);
```

Внешний вид диалогового окна изменится, и оно будет выглядеть так, как показано на рис. 9.2.



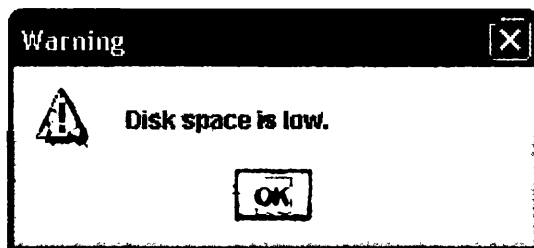


Рис. 9.2. Диалоговое окно, для которого задан заголовок окна и тип сообщения

По умолчанию в диалоговом окне рассматриваемого здесь типа отображается стандартная пиктограмма. Заменить ее собственной можно, используя следующий вариант метода `showMessageDialog()`:

```
static void showMessageDialog(Component parent, Object msg,  
                             String title, int msgT, Icon image)  
    throws HeadlessException
```

Здесь пиктограмма для вывода в составе окна задается посредством параметра `image`. Однако если ваше сообщение относится к одной из определенных категорий, то лучше, если в окне будет отображаться стандартная пиктограмма, поскольку большинство пользователей привыкли к ним.

Для того чтобы использовать в рассмотренной ранее программе собственную пиктограмму, создайте изображение, сохраните его в файле `myIcon.gif` и замените вызов метода `showMessageDialog()` приведенным ниже выражением.

```
JOptionPane.showMessageDialog(jfrm,  
                              "Disk space is low.",  
                              "Warning",  
                              JOptionPane.WARNING_MESSAGE,  
                              new ImageIcon("myIcon.gif"));
```

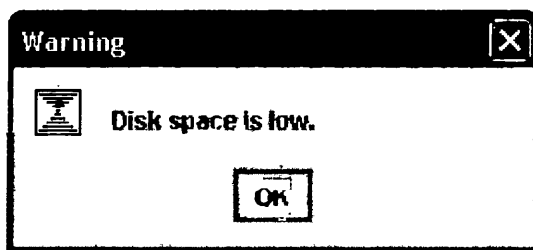


Рис. 9.3. Отображение в диалоговом окне пиктограммы, определенной разработчиком

После внесения изменений диалоговое окно будет выглядеть так, как показано на рис. 9.3.



## Вопросы для текущего контроля

1. Какая кнопка отображается в диалоговом окне, создаваемом с помощью метода `showMessageDialog()`?
2. Какая константа используется для обозначения предупреждающего сообщения?
3. Обязательно ли сообщение, отображаемое посредством метода `showMessageDialog()`, должно быть текстовой строкой?

**ВАЖНО!**

**9.3**

## Метод `showConfirmDialog()`

Помимо окна, предназначенного для вывода сообщения, очень распространено диалоговое окно, в котором пользователю предлагается вопрос, предполагающий ответ "да/нет". В Swing для создания окна данного типа предусмотрен метод `showConfirmDialog()`. Существует несколько вариантов этого метода. Самый простой из них имеет следующий вид:

```
static int showConfirmDialog(Component parent, Object msg)
    throws HeadlessException
```

Здесь параметр `parent` определяет компонент, которому принадлежит отображаемое окно. Если в качестве первого параметра вы зададите значение `null`, то окно разместится по центру экрана. Сообщение, которое должно быть выведено, передается с помощью параметра `msg`. Посредством данного параметра можно задать любой тип объекта, но обычно разработчики указывают строку текста. В диалоговом окне автоматически отображаются три кнопки: `Yes`, `No` и `Cancel`. В качестве заголовка окна выводится строка "Select an Option".

Метод возвращает целочисленное значение, отражающее выбор пользователя (кнопку, на которой он щелкнул мышью). Возвращаемое значение должно быть равно одной из следующих констант, определенных в классе `JOptionPane`.

1. `OK`.
2. `WARNING_MESSAGE`.
3. Нет. Оно может представлять собой объект любого типа.

|               |                                                                                 |
|---------------|---------------------------------------------------------------------------------|
| CANCEL_OPTION | Данное значение возвращается, если пользователь щелкнул на кнопке <b>Cancel</b> |
| CLOSED_OPTION | Данное значение возвращается, если пользователь закрыл окно, не сделав выбора   |
| NO_OPTION     | Данное значение возвращается, если пользователь щелкнул на кнопке <b>No</b>     |
| YES_OPTION    | Данное значение возвращается, если пользователь щелкнул на кнопке <b>Yes</b>    |

Обратите внимание на значение `CLOSED_OPTION`. Оно возвращается в том случае, если пользователь не выбрал ни один из предложенных ответов, а закрыл диалоговое окно, щелкнув на кнопке в его верхнем правом углу. В большинстве приложений значение `CLOSED_OPTION` обрабатывается так же, как и `NO_OPTION`.

Ниже приведен код программы, в которой создается диалоговое окно для получения подтверждения от пользователя. Внешний вид окна показан на рис. 9.4.

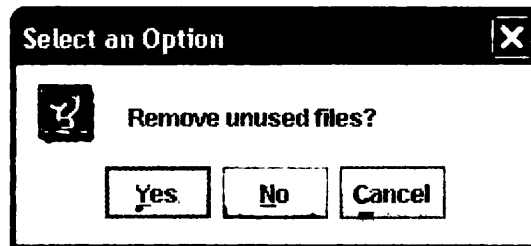


*Рис. 9.4. Диалоговое окно, отображаемое программой `ConfirmDialogDemo`*

```
// Использование диалогового окна, предназначенного для получения
// подтверждения от пользователя.
```

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

```
class ConfirmDialogDemo {
```

```
    JLabel jlab;
    JButton jbtnShow;
    JFrame jfrm;
```

```
ConfirmDialogDemo() {
    // Создание нового контейнера JFrame.
    jfrm = new JFrame("A Confirmation Dialog");

    // Установка диспетчера компоновки FlowLayout.
    jfrm.getContentPane().setLayout(new FlowLayout());

    // Установка начальных размеров фрейма.
    jfrm.setSize(400, 250);

    // Завершение программы при закрытии окна пользователем.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Создание метки, отображающей выбор пользователя.
    jlab = new JLabel();

    // Создание кнопки, вызывающей отображение
    // диалогового окна.
    jbtnShow = new JButton("Show Dialog");

    // Связывание с кнопкой обработчика событий.
    jbtnShow.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent le) {
            // Создание диалогового окна для получения
            // подтверждения от пользователя.
            int response = JOptionPane.showConfirmDialog(
                jfrm,
                "Remove unused files?");

            // Проверка ответа пользователя.
            switch(response) {
                case JOptionPane.YES_OPTION:
                    jlab.setText("You answered Yes.");
                    break;
                case JOptionPane.NO_OPTION:
                    jlab.setText("You answered No.");
                    break;
                case JOptionPane.CANCEL_OPTION:
                    jlab.setText("Cancel pressed.");
                    break;
                case JOptionPane.CLOSED_OPTION:
                    jlab.setText("Dialog closed without response.");
            }
        }
    });
}
```

```

        break;
    }
}
});

// Включение кнопки и метки в состав панели содержимого.
jfrm.getContentPane().add(jbtnShow);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new ConfirmDialogDemo();
        }
    });
}
}

```

Как видите, код, предназначенный для обработки ответа пользователя, очень простой. Поскольку все диалоговые окна, созданные методами `show...()` класса `JOptionPane`, являются модальными, метод `showConfirmDialog()` вернет управление лишь после того, как пользователь сделает свой выбор или закроет диалоговое окно щелчком на кнопке в его правом верхнем углу. Значение, возвращаемое методом `showConfirmDialog()`, присваивается переменной `response`. Эта переменная затем используется в выражении `switch`, посредством которого анализируется выбор пользователя.

Несмотря на то что рассмотренный вариант метода `showConfirmDialog()` очень прост в использовании, для приложений он, как правило, не подходит, так как в окне по умолчанию отображается заголовок **Select an Option**. Желательно, чтобы заголовок окна был более информативным, например **Disk Space Is Low**. Заголовок и другие характеристики диалогового окна несложно изменить, используя один из приведенных ниже вариантов метода `showConfirmDialog()`.

[illegible]

```

        throws HeadlessException
static int showConfirmDialog(Component parent, Object msg,
        String title, int optT, int msgT,
        Icon image)
        throws HeadlessException

```

Назначение параметров `parent` и `msg` было описано ранее. Заголовок окна определяется с помощью параметра `title`. Варианты выбора пользователя (т.е. кнопки) задаются посредством параметра `optT`. Этот параметр должен принимать значение, равное одной из следующих констант, объявленных в классе `JOptionPane`.

|                                   |                                                                              |
|-----------------------------------|------------------------------------------------------------------------------|
| <code>YES_NO_OPTION</code>        | В диалоговое окно помещаются кнопки <b>Yes</b> и <b>No</b>                   |
| <code>YES_NO_CANCEL_OPTION</code> | В диалоговое окно помещаются кнопки <b>Yes</b> , <b>No</b> и <b>Cancel</b> . |

По умолчанию в окне отображаются кнопки **Yes**, **No** и **Cancel**. Однако в ряде случаев кнопка **Cancel** не нужна. Если в окне должны присутствовать только кнопки **Yes** и **No**, следует задать значение параметра `optT`, равное `YES_NO_OPTION`.

Тип диалогового окна задается с помощью параметра `msgT`. Его значение должно быть равно одной из следующих констант (их назначение было описано ранее):

|                               |                                  |                            |
|-------------------------------|----------------------------------|----------------------------|
| <code>ERROR_MESSAGE</code>    | <code>INFORMATION_MESSAGE</code> | <code>PLAIN_MESSAGE</code> |
| <code>QUESTION_MESSAGE</code> | <code>WARNING_MESSAGE</code>     |                            |

По умолчанию формируется окно типа `QUESTION_MESSAGE`.

Пиктограмма, отображаемая в диалоговом окне, задается с помощью параметра `image`. По умолчанию принимается графическое изображение вопросительного знака.

Несложно выполнить дополнительную настройку диалогового окна, созданного в предыдущем примере. Например, можно изменить его заголовок и удалить кнопку **Cancel**. Сделать это позволяет следующий фрагмент кода, реализующий обработчик событий действия:

```

jbtnShow.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        // Создание диалогового окна, отображающего сообщение.
        int response = JOptionPane.showConfirmDialog(
            jfrm,
            "Remove unused files?", // Сообщение.
            "Disk Space Is Low",    // Заголовок
            JOptionPane.YES_NO_OPTION); // Отображаются только кнопки
        // Yes и No
    }
});

```

```
switch(response) {  
    case JOptionPane.YES_OPTION:  
        jlab.setText("You answered Yes.");  
        break;  
    case JOptionPane.NO_OPTION:  
        jlab.setText("You answered No.");  
        break;  
    case JOptionPane.CLOSED_OPTION:  
        jlab.setText("Dialog closed without response.");  
        break;  
}  
}  
});
```

Внешний вид диалогового окна изменится — оно будет выглядеть так, как показано на рис. 9.5. Заметьте, что теперь в окне отображается заголовок **Disk Space Is Low**, а кнопка **Cancel** отсутствует.

### Вопросы для текущего контроля

1. По умолчанию метод `showConfirmDialog()` формирует окно, содержащее кнопки **Yes**, **No** и **Cancel**. Да или нет?
2. Каким будет внешний вид окна, если в качестве значения параметра `optT` метода `showConfirmDialog()` задать `YES_NO_OPTION`?

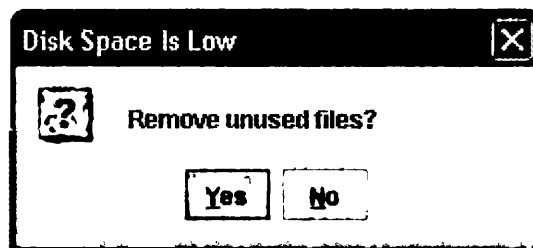


Рис. 9.5. Модифицированное диалоговое окно подтверждения

1. Да.
2. В окне будут присутствовать только кнопки **Yes** и **No**.

ВАЖНО!

## 9.4. Метод `showInputDialog()`

Несмотря на то что ответ “да/нет” часто бывает достаточным, в некоторых случаях желательно, чтобы он был более подробным. В подобных ситуациях нужны другие типы диалоговых окон. Окна, создаваемые с помощью метода `showOptionDialog()`, будут рассматриваться в следующем разделе, а здесь речь пойдет об окнах, которые формируются при выполнении метода `showInputDialog()`.

Существует несколько вариантов метода `showInputDialog()`. Самый простой из них создает окно с полем редактирования, в котором пользователь может ввести текстовую строку. Этот метод объявляется следующим образом:

```
static String showInputDialog(Object msg) throws  
HeadlessException
```

Сообщение, которое должно быть выведено, передается с помощью параметра `msg`. Для диалогового окна используется фрейм по умолчанию; окно располагается по центру экрана. Указанный метод возвращает строку, введенную пользователем. В окне отображаются кнопки `OK` и `Cancel`. Щелчок на кнопке `OK` означает, что введенная строка должна быть возвращена вызывающему методу. При активизации кнопки `Cancel` (или в случае, если окно закрывается щелчком на кнопке в его верхнем правом углу) введенная строка игнорируется и метод возвращает значение `null`. Если щелкнуть на кнопке `OK`, не введя строку, метод вернет строку нулевой длины.

Ниже приведен код примера, который демонстрирует использование простейшего варианта метода `showInputDialog()`. В окне предлагается ввести имя; внешний вид окна показан на рис. 9.6.

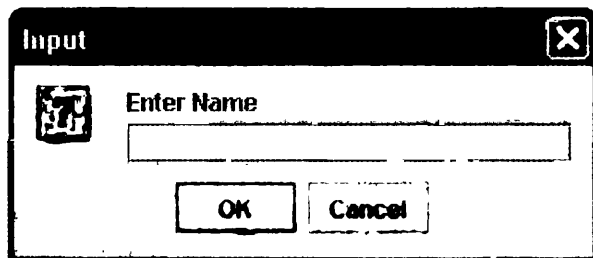


Рис. 9.6. Диалоговое окно, отображаемое при выполнении программы `InputDialogDemo`



```
// Простое диалоговое окно для ввода строки текста.

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

class InputDialogDemo {

    JLabel jlab;

    JButton jbtnShow;

    JFrame jfrm;

    InputDialogDemo() {
        // Создание нового контейнера JFrame.
        jfrm = new JFrame("A Simple Input Dialog");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(400, 250);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание метки, с помощью которой отображается
        // ответ пользователя.
        jlab = new JLabel();

        // Создание кнопки, вызывающей отображение диалогового окна.
        jbtnShow = new JButton("Show Dialog");

        // Связывание с кнопкой обработчика событий.
        jbtnShow.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent le) {
                // Создание диалогового окна для ввода строки.
                String response = JOptionPane.showInputDialog(
                    "Enter Name");

                // Если метод возвращает значение null, это означает,
                // что пользователь щелкнул на кнопке Cancel или закрыл
```

```

// окно с помощью кнопки в правом верхнем углу.
// Метод возвращает строку нулевой длины в случае, если
// пользователь не ввел данные. В других случаях
// возвращается строка, введенная пользователем.
if(response == null)
    jlab.setText("Dialog cancelled or closed");
else if(response.length() == 0)
    jlab.setText("No string entered");
else
    jlab.setText("Hi there " + response);
}
});

```

```

// Включение кнопки и метки в панель содержимого.
jfrm.getContentPane().add(jbtnShow);
jfrm.getContentPane().add(jlab);

```

```

// Отображение фрейма.
jfrm.setVisible(true);
}

```

```

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new InputDialogDemo();
        }
    });
}
}

```

Несмотря на то что данная программа создает полнофункциональное диалоговое окно, его применимость ограничена. В частности, заголовок `Input` предопределен. Окно всегда располагается по центру экрана, причем положение окна приложения не учитывается. Кроме того, данная программа не позволяет инициализировать поле редактирования в составе диалогового окна начальным значением. Все эти недостатки могут быть устранены путем использования вариантов метода `showInputDialog()`, показанных ниже.

```

static String showInputDialog(Object msg, Object initVal)
    throws HeadlessException
static String showInputDialog(Component parent, Object msg)
    throws HeadlessException

```

```

static String showInputDialog(Component parent, Object msg,
                             Object initVal)
    throws HeadlessException
static String showInputDialog(Component parent, Object msg,
                             String title, int msgT)
    throws HeadlessException

```

Здесь параметр `parent` определяет компонент, которому принадлежит отображаемое окно. Для поля редактирования можно задать начальное значение посредством параметра `initVal`. Параметр `title` задает заголовок окна. Тип диалогового окна определяется параметром `msgT`. Его значение должно быть равно одной из следующих констант:

```

ERROR_MESSAGE           INFORMATION_MESSAGE   PLAIN_MESSAGE
QUESTION_MESSAGE       WARNING_MESSAGE

```

По умолчанию формируется окно типа `QUESTION_MESSAGE`.

Например, в результате выполнения приведенного ниже выражения создается диалоговое окно, позиция которого определяется расположением основного окна приложения, а поле редактирования инициализируется строкой "Bob Smith".

```

String response = JOptionPane.showInputDialog(
    jfrm,
    "Enter Name",
    "Bob Smith");

```

Если в предыдущей программе вы замените вызов метода `showInputDialog()` приведенным выше выражением, то диалоговое окно будет выглядеть так, как показано на рис. 9.7.

Существует также вариант метода `showInputDialog()`, который позволяет задать список, из которого пользователь может выбрать нужный пункт,

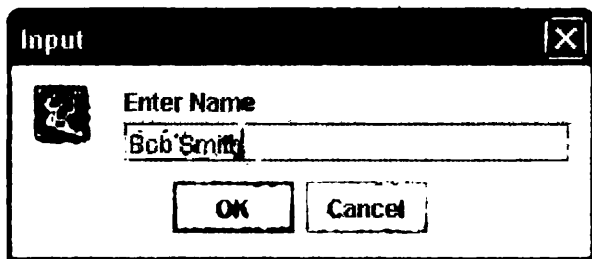


Рис. 9.7. Диалоговое окно, в котором поле редактирования инициализировано строкой текста

а также указать пиктограмму для отображения в составе окна. Этот метод объявляется следующим образом:

```
static Object showInputDialog(Component parent, Object msg,  
                             String title, int msgT, Icon image,  
                             Object[] vals, Object initVal)  
    throws HeadlessException
```

Назначение параметров `parent`, `title` и `msgT` было описано ранее. Пиктограмма для вывода в составе окна задается посредством параметра `image`. Если значение данного параметра равно `null`, отображается стандартная пиктограмма, соответствующая типу сообщения. Список пунктов для выбора передается посредством параметра `vals`. Они оформляются в виде компонента-списка. Если значение `vals` равно `null`, то вместо списка отображается поле редактирования, в котором пользователь может ввести нужное значение. Начальное значение задается посредством параметра `initVal`.

Диалоговое окно, создаваемое этим вариантом метода `showInputDialog()`, особенно полезно в тех случаях, когда вам надо ограничить выбор пользователя набором predetermined ответов. Для того чтобы увидеть, как диалоговое окно данного типа будет отображаться на экране, замените в предыдущей программе обработчик события приведенным ниже фрагментом кода. После внесения изменений диалоговое окно будет выглядеть так, как показано на рис. 9.8.

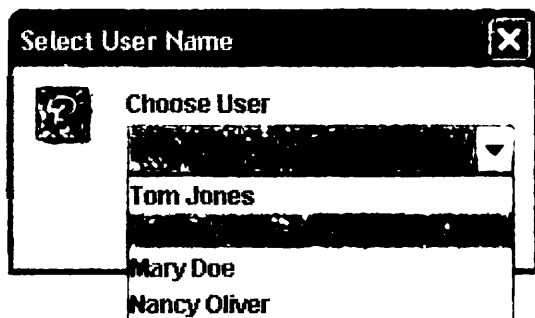


Рис. 9.8. Диалоговое окно, обеспечивающее выбор из списка

```
jbtnShow.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent le) {  
        String[] names = { "Tom Jones", "Bob Smith",  
                           "Mary Doe", "Nancy Oliver" };  
    }  
});
```

// Создание диалогового окна, позволяющего пользователю

```
// выбирать имя из списка.
String response =
    (String) JOptionPane.showInputDialog(
        jfrm,
        "Choose User",
        "Select User Name",
        JOptionPane.QUESTION_MESSAGE,
        null,
        names,
        "Bob Smith");

if(response == null)
    jlab.setText("Dialog cancelled or closed");
else if(response.length() == 0)
    jlab.setText("No string entered");
else
    jlab.setText("Hi there " + response);
}
});
```



## Вопросы для текущего контроля

1. Какие данные может вводить пользователь в диалоговом окне, созданном с помощью метода `showInputDialog()`?
2. Позволяет ли метод `showInputDialog()` инициализировать начальным значением компонент, предназначенный для ввода данных пользователем?

**ВАЖНО!**

## 2.5. Метод `showOptionDialog()`

Несмотря на то что диалоговые окна, созданные с помощью методов `showMessageDialog()`, `showConfirmDialog()` и `showInputDialog()`, удовлетворяют потребностям большинства приложений, бывают случаи, когда возможности, предоставляемые ими, недостаточны. В классе `JOptionPane` предусмотрен еще один метод, `showOptionDialog()`, который создает диало-

1. В зависимости от параметров метод `showInputDialog()` позволяет пользователю либо вводить строку, либо выбирать пункт списка.
2. Да.

говое окно, содержащее указанные элементы. Таким образом, с помощью метода `showOptionDialog()` вы можете формировать диалоговые окна, наиболее соответствующие вашим потребностям.

Заголовок метода `showOptionDialog()` имеет следующий вид:

```
static int showOptionDialog(Component parent, Object msg,  
                           String title, int optT, int msgT, Icon image,  
                           Object[] options, Object initVal)  
                           throws HeadlessException
```

Большинство из параметров данного метода уже знакомо вам. Так, параметры `parent`, `msg` и `title` задают соответственно владельца диалогового окна (если этот параметр имеет значение `null`, используется фрейм по умолчанию), сообщение, отображаемое в окне, и заголовок окна. Опции для выбора (т.е. кнопки) задаются посредством параметра `optT`. Этот параметр должен принимать одно из следующих значений, определенных в классе `JOptionPane`:

`YES_NO_OPTION`

`YES_NO_CANCEL_OPTION`

Заметьте, что параметр `optT` используется только в том случае, если значение параметра `options` равно `null`. В противном случае он игнорируется. В подобных ситуациях некоторые программисты указывают константу `DEFAULT_OPTION`. Тип диалогового окна задается с помощью параметра `msgT`. Его значение должно быть равно одной из следующих констант:

`ERROR_MESSAGE`

`INFORMATION_MESSAGE`

`PLAIN_MESSAGE`

`QUESTION_MESSAGE`

`WARNING_MESSAGE`

Пиктограмма, отображаемая в диалоговом окне, задается с помощью параметра `image`. Если вы хотите, чтобы в окне присутствовала стандартная пиктограмма, в качестве значения этого параметра надо установить `null`. Начальное значение задается посредством параметра `initVal`.

В данном методе присутствует новый параметр — `options`. Он представляет собой массив объектов `Object`, содержащий элементы, которые должны выводиться в окне. Обычно посредством данного параметра методу передается массив строк. В этом случае каждая строка интерпретируется как надпись на кнопке. По щелчку на соответствующей кнопке диалоговое окно закрывается и метод возвращает индекс строки в массиве. Можно также передать методу массив пиктограмм и даже массив, в котором будут присутствовать и строки, и пиктограммы. Если в массиве присутствует пиктограмма, то создается кнопка, на которой располагается указанное изображение. Методу можно передавать и другие объекты (этот вопрос будет рассмотрен несколько позже).

Ниже приведен код программы, в которой метод `showOptionDialog()` используется для создания окна, позволяющего пользователю выбирать способ соединения с сетью. Внешний вид диалогового окна показан на рис. 9.9.

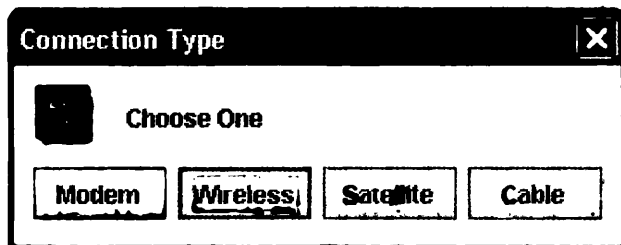


*Рис. 9.9. Диалоговое окно, отображаемое при выполнении программы `OptionDialogDemo`*

```
// Использование метода showOptionDialog()
// для создания диалогового окна.

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

class OptionDialogDemo {

    JLabel jlab;
    JButton jbtnShow;
    JFrame jfrm;

    OptionDialogDemo() {
        // Создание нового контейнера JFrame.
        jfrm = new JFrame("A Simple Option Dialog");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(400, 250);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание метки, отображающей выбор пользователя.
```

```
jlab = new JLabel();

// Создание кнопки, вызывающей отображение
// диалогового окна.
jbtnShow = new JButton("Show Dialog");

// Связывание с кнопкой обработчика событий действия.
jbtnShow.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        // Определение опций для отображения в окне.
        String[] connectOpts = { "Modem", "Wireless",
                                   "Satellite", "Cable" };

        // Создание диалогового окна, позволяющего пользователю
        // выбирать способ подключения к сети.
        // Каждая опция оформляется в виде отдельной кнопки.
        int response = JOptionPane.showOptionDialog(
            jfrm,
            "Choose One",
            "Connection Type",
            JOptionPane.DEFAULT_OPTION,
            JOptionPane.QUESTION_MESSAGE,
            null,
            connectOpts,
            "Wireless");

        // Отображение информации о выборе пользователя.
        // В переменной response содержится индекс
        // кнопки, на которой щелкнул пользователь.
        switch(response) {
            case 0:
                jlab.setText("Connect via modem.");
                break;
            case 1:
                jlab.setText("Connect via wireless.");
                break;
            case 2:
                jlab.setText("Connect via satellite.");
                break;
            case 3:
                jlab.setText("Connect via cable.");
                break;
            case JOptionPane.CLOSED_OPTION:
```



```

        jlab.setText("Dialog cancelled.");
        break;
    }
}
});

// Включение компонентов в состав панели содержимого.
jfrm.getContentPane().add(jbtnShow);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new OptionDialogDemo();
        }
    });
}
}
}

```

Рассмотрим, как активизируется диалоговое окно в методе `actionPerformed()`. Прежде всего, заметьте, что методу `showOptionDialog()` передается массив `names`, в котором содержатся строки, описывающие различные способы соединения с сетью. При формировании диалогового окна строки интерпретируются как надписи на кнопках. По щелчку на одной из этих кнопок метод `showOptionDialog()` возвращает целочисленный индекс, соответствующий строке в массиве. Так, если вы щелкнете на кнопке `Modem`, будет возвращено значение 0, а если щелкнете на кнопке `Wireless`, то метод вернет значение 1.

Создавая окно данного типа, необходимо помнить, что содержимое массива `options` изменяет некоторые стандартные характеристики окна. Например, если вы зададите в качестве элемента массива строку `"Cancel"`, то при активизации соответствующей кнопки метод вернет индекс элемента массива, а не значение `CANCEL_OPTION`. Единственным стандартным значением является `CLOSED_OPTION`, поскольку оно генерируется по щелчку на кнопке в верхнем правом углу окна.

## Спросим у опытного программиста

**Вопрос.** Я собираюсь использовать для создания диалогового окна метод `showOptionDialog()`. Могу ли я передать ему в качестве параметра `options` массив объектов, отличных от строк и пиктограмм?

**Ответ** Да. Данному методу можно передавать объекты любого типа, в том числе компоненты Swing. Компоненты, содержащиеся в массиве, будут включены в диалоговое окно. Так, например, можно создать окно, содержащее поле редактирования, два флажка опций и кнопку ОК. Для этого надо передать методу ссылки на данные компоненты. Если вы передадите объект, не имеющий графического представления, будет создана кнопка с надписью в виде строки, полученной путем вызова метода `toString()` объекта.

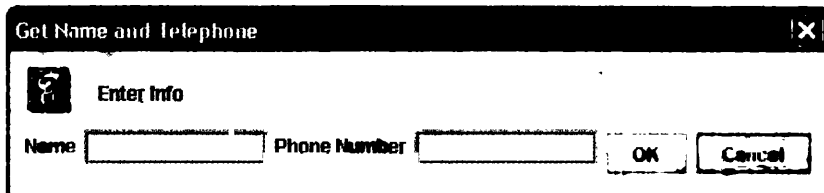
К сожалению, вопросы компоновки объектов в диалоговом окне не решены, поэтому обычно разработчики передают методу `showOptionDialog()` либо строки, либо пиктограммы. При использовании других компонентов внешний вид окна будет неприемлемым. Предположим, например, что вы использовали для создания диалогового окна следующие выражения:

```
Object[] ops = { new JLabel("Name"),
                  new JTextField(10),
                  new JLabel("Phone Number"),
                  new JTextField(10),
                  "OK", "Cancel" };

int response = JOptionPane.showOptionDialog(
    jfrm,
    "Enter Info",
    "Get Name and Telephone",
    JOptionPane.OK_CANCEL_OPTION,
    JOptionPane.QUESTION_MESSAGE,
    null,
    ops,
    "Cancel");
```

Полученное в результате окно показано ниже.

Как видите, расположение компонентов нельзя назвать удачным.



Несмотря на то что метод `showOptionDialog()` обеспечивает большую гибкость, используется он реже, чем могло бы показаться на первый взгляд. Причина в том, что разработчик не имеет возможности управлять размещением элементов в диалоговом окне. Методы `show...()`, в том числе `showOptionDialog()`, используют для создания окон одинаковые шаблоны. Окна содержат заголовок, сообщение в виде одной строки, пиктограмму, расположенную слева от сообщения, и элементы для выбора, расположенные в один ряд. Изменить взаимное расположение элементов нет возможности.

### Вопросы для текущего контроля

1. Метод `showOptionDialog()` имеет ограниченное применение, поскольку в создаваемом диалоговом окне отображается стандартный набор компонентов, изменить который невозможно. Да или нет?
2. Почему метод `showOptionDialog()` используется сравнительно редко?

## ВАЖНО! 9.6.

### Класс `JDialog`

Класс `JOptionPane` позволяет очень просто создавать диалоговые окна, но в некоторых случаях его возможностей оказывается недостаточно. Если вам нужно окно, в котором содержится несколько полей, или требуется специаль-

1. Нет.
2. Метод `showOptionDialog()` используется сравнительно редко, потому что он не обеспечивает контроля над расположением элементов.

ная обработка данных, то следует воспользоваться классом `JDialog`. Класс `JDialog` представляет собой контейнер верхнего уровня, который не является подклассом `JComponent`. Следовательно, `JDialog` — тяжеловесный компонент. Как было сказано ранее, `JOptionPane` использует класс `JDialog` для создания диалоговых окон. Следовательно, окна, сформированные посредством подкласса `JOptionPane`, представляют собой экземпляры класса `JDialog`.

Создание объекта `JDialog` и управление им похоже на работу с `JFrame`. Например, вы можете включать компоненты в состав панели содержимого, устанавливать диспетчер компоновки и задавать размер окна. Метод `setVisible()` позволяет отображать окно и удалять его с экрана. Можно также создавать меню. Объект `JDialog` наследует возможности нескольких классов AWT: `Container`, `Component`, `Window` и `Dialog`. Таким образом, он поддерживает все функции, типичные для компонентов AWT.

Класс `JDialog` позволяет создавать как модальные, так и немодальные диалоговые окна. Как было сказано ранее, модальное диалоговое окно выполняется в том же потоке, что и вызывающий метод, поэтому выполнение приложения приостанавливается до тех пор, пока модальное окно не будет закрыто. Немодальное окно выполняется в собственном потоке, а это означает, что остальные элементы приложения остаются активизированными. Возможность создания немодальных диалоговых окон — одна из причин, по которым имеет смысл отказаться от `JOptionPane` и использовать класс `JDialog`. Как вы вскоре увидите, данный класс предоставляет возможность создать диалоговое окно любого типа.

В классе `JDialog` определено большое количество конструкторов, которые показаны в табл. 9.1. Разнообразие конструкторов обусловлено тем, что владельцем `JDialog` может быть как фрейм, так и другое диалоговое окно. По этой причине необходимы два набора конструкторов, отличающиеся друг от друга только владельцем.

Таблица 9.1. Конструкторы класса `JDialog`

| Конструктор                        | Описание                                                                                                                                                                               |
|------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JDialog()</code>             | Создает немодальное диалоговое окно, в качестве владельца которого выступает разделяемое скрытое окно. Диалоговое окно, создаваемое с помощью данного конструктора, не имеет заголовка |
| <code>JDialog(Frame parent)</code> | Создает немодальное диалоговое окно, владелец которого определяется значением параметра <code>parent</code> . Окно, создаваемое с помощью данного конструктора, не имеет заголовка     |

Продолжение табл. 9.1

| Конструктор                                                                                          | Описание                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JDialog(Frame parent, String title)</code>                                                     | Создает немодальное диалоговое окно, владелец которого определяется значением параметра <code>parent</code> . Заголовок окна задается посредством параметра <code>title</code>                                                                                                                                                                                                                                                                                                                                           |
| <code>JDialog(Frame parent, boolean isModal)</code>                                                  | Создает диалоговое окно, владелец которого определяется значением параметра <code>parent</code> . Если значение параметра <code>isModal</code> равно <code>true</code> , диалоговое окно является модальным. Значение <code>false</code> данного параметра указывает на то, что должно быть создано немодальное окно. Диалоговое окно, создаваемое с помощью данного конструктора, не имеет заголовка                                                                                                                    |
| <code>JDialog(Frame parent, String title, boolean isModal)</code>                                    | Создает диалоговое окно, владелец которого определяется значением параметра <code>parent</code> . Если значение параметра <code>isModal</code> равно <code>true</code> , диалоговое окно является модальным. Значение <code>false</code> данного параметра указывает на то, что должно быть создано немодальное окно. Заголовок окна задается посредством параметра <code>title</code>                                                                                                                                   |
| <code>JDialog(Frame parent, String title, boolean isModal, GraphicsConfiguration graphConfig)</code> | Создает диалоговое окно, владелец которого определяется значением параметра <code>parent</code> . Если значение параметра <code>isModal</code> равно <code>true</code> , диалоговое окно является модальным. Значение <code>false</code> данного параметра указывает на то, что должно быть создано немодальное окно. Заголовок окна задается посредством параметра <code>title</code> . Для данного окна используется объект <code>GraphicsConfiguration</code> , заданный с помощью параметра <code>graphConfig</code> |
| <code>JDialog(Dialog parent)</code>                                                                  | Создает немодальное диалоговое окно, владелец которого определяется значением параметра <code>parent</code> . Диалоговое окно, создаваемое с помощью данного конструктора, не имеет заголовка                                                                                                                                                                                                                                                                                                                            |
| <code>JDialog(Dialog parent, String title)</code>                                                    | Создает немодальное диалоговое окно, владелец которого определяется значением параметра <code>parent</code> . Заголовок окна задается посредством параметра <code>title</code>                                                                                                                                                                                                                                                                                                                                           |
| <code>JDialog(Dialog parent, boolean isModal)</code>                                                 | Создает диалоговое окно, владелец которого определяется значением параметра <code>parent</code> . Если значение параметра <code>isModal</code> равно <code>true</code> , диалоговое окно является модальным. Значение <code>false</code> данного параметра указывает на то, что должно быть создано немодальное окно. Диалоговое окно, создаваемое с помощью данного конструктора, не имеет заголовка                                                                                                                    |

Окончание табл. 9.1

| Конструктор                                                                                                                                                            | Описание                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JDialog(</code><br><code>Dialog parent, String</code><br><code>title, boolean isModal)</code>                                                                    | Создает диалоговое окно, владелец которого определяется значением параметра <code>parent</code> . Если значение параметра <code>isModal</code> равно <code>true</code> , диалоговое окно является модальным. Значение <code>false</code> данного параметра указывает на то, что должно быть создано немодальное окно. Заголовок окна задается посредством параметра <code>title</code>                                                                                                                                   |
| <code>JDialog(</code><br><code>Dialog parent, String</code><br><code>title, boolean isModal,</code><br><code>GraphicsConfiguration</code><br><code>graphConfig)</code> | Создает диалоговое окно, владелец которого определяется значением параметра <code>parent</code> . Если значение параметра <code>isModal</code> равно <code>true</code> , диалоговое окно является модальным. Значение <code>false</code> данного параметра указывает на то, что должно быть создано немодальное окно. Заголовок окна задается посредством параметра <code>title</code> . Для данного окна используется объект <code>GraphicsConfiguration</code> , заданный с помощью параметра <code>graphConfig</code> |

Чтобы создать и отобразить диалоговое окно на базе класса `JDialog`, необходимо выполнить перечисленные ниже действия.

1. Создать объект `JDialog`.
2. Определить диспетчер компоновки, размер и, возможно, политику закрытия диалогового окна.
3. Включить требуемые компоненты в панель содержимого.
4. Отобразить окно, вызвав метод `setVisible(true)`.

Для того чтобы удалить диалоговое окно с экрана, надо вызвать либо `setVisible(false)`, либо `dispose()`. Выражение `setVisible(false)` следует использовать в том случае, если в процессе работы приложения надо часто отображать данное окно. Если же вы не предполагаете в дальнейшем выводить это диалоговое окно на экран, лучше вызвать метод `dispose()`. Этот метод освобождает ресурсы, необходимые для поддержки диалогового окна. Метод `setVisible(false)` лишь делает окно невидимым.

Ниже приведен код программы, который демонстрирует создание простого модального диалогового окна с помощью класса `JDialog`. Окно с заголовком `Direction` позволяет пользователю выбирать направление. В нем отображаются две кнопки: `Up` и `Down`. По щелчку на кнопке диалоговое окно закрывается. В основном окне приложения расположены две кнопки и метка. Метка отображает выбранное направление. По щелчку на кнопке `Show Dialog` на экране

появляется диалоговое окно **Direction**. Кнопка **Reset Direction** сбрасывает направление, отображаемое посредством метки. Окно, создаваемое при работе программы, показано на рис. 9.10.

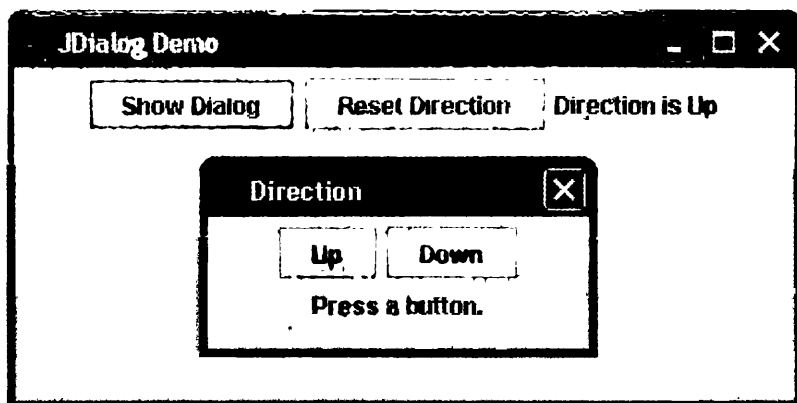


*Рис. 9.10. Выходные данные программы JDIALOGDemo*

```
// Пример создания простого диалогового окна
// с помощью класса JDialog.

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

class JDIALOGDemo {

    JLabel jlab;

    JButton jbtnShow;
    JButton jbtnReset;

    // Кнопки, содержащиеся в диалоговом окне
    JButton jbtnUp;
    JButton jbtnDown;

    JDialog jdlg;

    JDIALOGDemo() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("JDIALOG Demo");
```

```
// Установка диспетчера компоновки FlowLayout.
jfrm.getContentPane().setLayout(new FlowLayout());

// Установка начальных размеров фрейма.
jfrm.setSize(400, 200);

// Завершение программы при закрытии окна пользователем.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Создание метки, отображающей выбранное направление.
jlab = new JLabel("Direction is pending.");

// Создание кнопки, вызывающей отображение
// диалогового окна.
jbtnShow = new JButton("Show Dialog");

// Создание кнопки, сбрасывающей направление.
jbtnReset = new JButton("Reset Direction");

// Создание и настройка простого
// модального диалогового окна.
jdlg = new JDialog(jfrm, "Direction", true);
jdlg.setSize(200, 100);
jdlg.getContentPane().setLayout(new FlowLayout());

// Создание кнопок, используемых в диалоговом окне.
jbtnUp = new JButton("Up");
jbtnDown = new JButton("Down");

// Включение кнопок в состав диалогового окна.
jdlg.getContentPane().add(jbtnUp);
jdlg.getContentPane().add(jbtnDown);

// Включение метки в состав диалогового окна.
jdlg.getContentPane().add(new JLabel("Press a button.));

// Отображение диалогового окна по щелчку
// на кнопке Show Dialog.
jbtnShow.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        jdlg.setVisible(true);
    }
});
```



```
// Сброс направления по щелчку на кнопке Reset Direction.
jbtnReset.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        jlab.setText("Direction is Pending.");
    }
});

// Реакция на активизацию кнопки Up в диалоговом окне.
jbtnUp.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        jlab.setText("Direction is Up");

        // Удаление диалогового окна с экрана после выбора
        // направления пользователем.
        jdlg.setVisible(false);
    }
});

// Реакция на активизацию кнопки Down в диалоговом окне.
jbtnDown.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        jlab.setText("Direction is Down");

        // Удаление диалогового окна с экрана после выбора
        // направления пользователем.
        jdlg.setVisible(false);
    }
});

// Включение кнопки Show Dialog и метки в состав
// панели содержимого.
jfrm.getContentPane().add(jbtnShow);
jfrm.getContentPane().add(jbtnReset);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
```

```
        public void run() {  
            new JDialogDemo();  
        }  
    });  
}  
}
```

Следует отметить некоторые важные особенности данной программы. Прежде всего, обратите внимание на то, как создается объект `JDialog` для модального диалогового окна.

```
// Создание и настройка простого  
// модального диалогового окна.  
jdlg = new JDialog(jfrm, "Direction", true);  
jdlg.setSize(200, 100);  
jdlg.getContentPane().setLayout(new FlowLayout());
```

Сначала создается объект `JDialog`, ссылка на который помещается в переменную `jdlg`. При создании диалогового окна определяется заголовок `Direction` и принадлежность его к основному окну приложения, `jfrm`. На то, что окно является модальным, указывает значение `true`, передаваемое конструктору `JDialog` в качестве третьего параметра. Затем задается размер окна и диспетчер компоновки `FlowLayout`. На этом этапе формирование окна завершается, но оно пока невидимо и не содержит компонентов.

Далее в состав окна `jdlg` включаются две кнопки и одна метка.

```
// Создание кнопок, используемых в диалоговом окне.  
jbtnUp = new JButton("Up");  
jbtnDown = new JButton("Down");  
  
// Включение кнопок в состав диалогового окна.  
jdlg.getContentPane().add(jbtnUp);  
jdlg.getContentPane().add(jbtnDown);  
  
// Включение метки в состав диалогового окна.  
jdlg.getContentPane().add(new JLabel("Press a button."));
```

Нетрудно заметить, что компоненты включаются в диалоговое окно так же, как и в основное окно приложения. После выполнения указанных команд формирование диалогового окна завершается, но оно по-прежнему невидимо. Окно остается невидимым до тех пор, пока не будет активизирована кнопка `Show Dialog`, находящаяся в главном окне приложения. Для отображения диалогового окна в обработчике события, связанном с кнопкой `Show Dialog`, вызывается метод `setVisible(true)` объекта `jdlg`.

Обратите также внимание на то, что обработчики событий для кнопок Up и Down содержатся в составе `jdlg`. Каждый из них устанавливает текст метки `jlab`, а затем вызывает метод `setVisible(false)`, в результате чего диалоговое окно удаляется с экрана. Выражение `setVisible(false)` — самый эффективный способ скрыть окно, которое может потребоваться в дальнейшем.

**ВАЖНО!**

## **9.7. Создание немодального диалогового окна**

Используя класс `JDialog`, можно создавать немодальные диалоговые окна. Для этого надо либо вызвать один из конструкторов с двумя параметрами (тот, который автоматически создает немодальное окно), либо явным образом передать значение `false` конструктору с тремя параметрами. Преимущество немодального диалогового окна состоит в том, что остальные компоненты приложения остаются активизированными. Очевидно, что создавать немодальное окно имеет смысл только в том случае, если другие части приложения не зависят от данных, введенных пользователем в окне. В качестве примера можно привести приложение для обработки фотоснимков, в котором фильтр выбирается посредством диалогового окна. При использовании немодального окна пользователь может выбирать фильтры и оценивать их действие; при этом ему нет необходимости закрывать диалоговое окно и вновь открывать его. Как правило, в программах применяются модальные окна, но если использование немодального окна оправдано, оно, как правило оказывается очень полезным.

Для того чтобы продемонстрировать работу с немодальными диалоговыми окнами, изменим предыдущий пример и сделаем окно `Direction` немодальным. Перед тем как приступить к модификации кода, запустите программу `JDialogDemo` и активизируйте окно `Direction`. Затем щелкните на кнопке `Reset Direction`, расположенной в основном окне приложения. Вы увидите, что, до тех пор, пока модальное окно `Direction` отображается на экране, кнопка `Reset Direction` недоступна. Если окно `Direction` станет немодальным, оба окна будут активизированными, и вы сможете сбросить активизированную кнопку `Reset Direction` в любой момент.

Для того чтобы сделать окно `Direction` модальным, надо внести небольшие изменения в программу. Во-первых, измените обращение к конструктору `JDialog`, удалив третий параметр.

```
jdlg = new JDialog(jfrm, "Direction");
```

Такой конструктор автоматически создаст немодальное окно.

## Спросим у опытного программиста

**Вопрос.** В модуле 1 говорилось о том, что, начиная с версии JDK 5, при включении компонента в контейнер `JFrame` нет необходимости вызывать метод `getContentPane()`. Можно ли поступать так же, помещая компоненты в состав `JDialog`?

**Ответ.** Да. Начиная с версии JDK 5, можно включать компоненты в контейнер `JDialog`, непосредственно вызывая метод `add()` экземпляра `JDialog`. Компонент будет автоматически добавлен к панели содержимого. Следовательно, если вы работаете с JDK 5 или более поздними версиями, в программе `JDialogDemo` для включения компонентов в состав `jdlg` можно использовать приведенные ниже выражения.

```
// Включение кнопок в состав диалогового окна.
```

```
jdlg.add(jbtnUp);
```

```
jdlg.add(jbtnDown);
```

```
// Включение метки в состав диалогового окна.
```

```
jdlg.add(new JLabel("Press a button."));
```

При написании данной книги принято решение явным образом вызывать метод `getContentPane()`, чтобы примеры сохраняли работоспособность независимо от используемой версии Java, а также для того, чтобы при просмотре исходного кода было понятнее, какие действия выполняются на самом деле. Имея в своем распоряжении JDK 5 или более позднюю версию данного пакета, можно отказаться от вызова метода `getContentPane()`.

Далее удалите вызов `setVisible(false)` в обработчиках событий кнопок `Up` и `Down`. Обработчики должны принять следующий вид:

```
// Реакция на активизацию кнопки Up в диалоговом окне.
```

```
jbtnUp.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent le) {  
        jlab.setText("Direction is Up");  
    }  
});
```

```
// Реакция на активизацию кнопки Down в диалоговом окне.
```

```
jbtnDown.addActionListener(new ActionListener() {
```

```
public void actionPerformed(ActionEvent le) {
    jlab.setText("Direction is Down");
}
});
```

После того как вы измените код программы, окно **Direction** будет отображаться на экране до тех пор, пока вы не закроете его, щелкнув на кнопке в правом верхнем углу. В результате у вас появится возможность изменять направление, не закрывая окно. Вы также сможете активизировать кнопку **Reset Direction** основного окна в то время, когда диалоговое окно отображается на экране.

### Вопросы для текущего контроля

1. Является ли класс `JDialog` подклассом `JComponent`?
2. Компоненты, предназначенные для размещения в составе `JDialog`, следует включить в панель содержимого. Да или нет?
3. Какие действия выполняет метод `dispose()`?
4. Немодальное окно выполняется в отдельном \_\_\_\_\_.

**ВАЖНО!**  
**9.8.**

## Выбор файлов с помощью компонента `JFileChooser`

Одно из наиболее часто используемых диалоговых окон является в то же время наиболее сложным в реализации. Речь идет об окне, предназначенном для выбора файлов. К счастью, в составе `Swing` присутствует класс, позволяющий решить эту сложную задачу. Окно для выбора файлов представляет собой экземпляр класса `JFileChooser`.

Компонент `JFileChooser` имеет две положительные особенности. Во-первых, он обеспечивает согласованность интерфейсов различных программ. Выбирать файл приходится практически в каждом приложении. В приложениях, использующих `JFileChooser`, окна для выбора файлов выглядят одинаково. Таким образом, научившись работать с одной программой, пользователь смо-

1. Нет.
2. Да.
3. Метод `dispose()` удаляет диалоговое окно с экрана и освобождает используемые им ресурсы.
4. Потоке.

жет выбирать файлы в любом приложении, даже созданном другими разработчиками. Компонент `JFileChooser` реализует стандартный механизм, понятный пользователям. Во-вторых, компонент `JFileChooser` является чрезвычайно эффективным. Реализация диалогового окна, позволяющего выбрать файл, — сложная задача, требующая больших усилий. Стандартная реализация такого окна, включенная в библиотеку Swing, предотвращает ненужное повторение единожды выполненной работы. Таким образом, если вам нужно диалоговое окно, позволяющее выбирать файлы, используйте класс `JFileChooser`. Исключением являются лишь немногие приложения, предъявляющие специальные требования к процедуре выбора файлов.

Класс `JFileChooser` является подклассом `JComponent`. В нем определено несколько конструкторов, но наиболее часто используются три из них (табл. 9.2). Кроме того, в данном классе содержится много методов. Мы не можем рассмотреть их все, но, поняв основные средства `JFileChooser`, вы без труда сможете создать окно, соответствующее вашим требованиям.

Таблица 9.2. Часто используемые конструкторы класса `JFileChooser`

| Конструктор                           | Описание                                                                                                                                                                                                                              |
|---------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>JFileChooser()</code>           | Создает окно выбора файлов, в котором первоначально отображается содержимое текущего каталога                                                                                                                                         |
| <code>JFileChooser(File dir)</code>   | Создает окно выбора файлов, в котором первоначально отображается содержимое каталога, заданного с помощью параметра <code>dir</code> . Если значение этого параметра равно <code>null</code> , выводится содержимое текущего каталога |
| <code>JFileChooser(String dir)</code> | Создает окно выбора файлов, в котором первоначально отображается содержимое каталога, заданного с помощью параметра <code>dir</code> . Если значение этого параметра равно <code>null</code> , выводится содержимое текущего каталога |

После создания окна `JFileChooser` его надо отобразить на экране, вызвав один из следующих методов:

```
int showOpenDialog(Component parent) throws HeadlessException
int showSaveDialog(Component parent) throws HeadlessException
int showDialog(Component parent, String name)
    throws HeadlessException
```

В каждом из них параметр `parent` определяет компонент, относительно которого позиционируется окно. Если значение `parent` равно `null`, окно выбора файла будет выведено посередине экрана. Метод `showOpenDialog()` отображает стандартное окно `Open`. Метод `showSaveDialog()` выводит стандартное окно `Save`. Эти окна отличаются между собой заголовком и над-

писью на кнопке, подтверждающей выбор файла. Если окно создано с помощью метода `showOpenDialog()`, то эта кнопка называется **Open**. Метод `showSaveDialog()` формирует кнопку с надписью **Save**. Задать заголовок и надпись на кнопке явным образом позволяет метод `showDialog()`. Подобно предыдущим методам, он также создает стандартное окно выбора файла, отличающееся лишь заголовком и названием кнопки. Например, если вам нужно окно, посредством которого пользователь сможет выбирать файлы для удаления, вы можете задать значение `"Delete"` параметра `name`.

Каждый из перечисленных выше методов возвращает целочисленное значение, сообщающее о результатах выбора файла. Его значение равно одной из нижеприведенных констант, объявленных в классе `JFileChooser`.

|                             |                                                                                                                                    |
|-----------------------------|------------------------------------------------------------------------------------------------------------------------------------|
| <code>APPROVE_OPTION</code> | Пользователь выбрал файл                                                                                                           |
| <code>CANCEL_OPTION</code>  | Пользователь отказался от выбора, щелкнув на кнопке <code>Cancel</code> , либо на кнопке, расположенной в правом верхнем углу окна |
| <code>ERROR_OPTION</code>   | В процессе работы возникла ошибка                                                                                                  |

Следует помнить, что пользователь может ввести любое имя файла. Это имя не обязательно должно быть допустимым, а файл с этим именем может отсутствовать. Таким образом, значение `APPROVE_OPTION`, возвращенное одним из методов `show...()`, не означает, что файл доступен.

После того как пользователь сделал выбор, информацию о файле можно получить с помощью метода `getSelectedFile()`, определенного в классе `JFileChooser`. Заголовок этого метода выглядит следующим образом:

```
File getSelectedFile()
```

Данный метод возвращает объект `File`, представляющий выбранный файл. (Заметьте, что файл не открыт.)

Класс `File` инкапсулирует информацию о файле. Этот класс находится в пакете `java.io` и содержит несколько полезных методов. Например, получить имя файла позволяет метод `getName()`, показанный ниже.

```
String getName()
```

Имя возвращается в виде строки. Определить, существует ли файл, можно, вызвав метод `exists()`.

```
boolean exists()
```

Если файл существует, метод возвращает значение `true`, в противном случае — значение `false`.

С помощью рассматриваемого здесь окна можно выбирать как файлы, так и каталоги. (Как известно, каталоги представляют собой специальный тип файлов.) Определить, представляет ли объект `File`, возвращенный методом `getSelectedFile()`, файл или каталог, можно, вызвав метод `isFile()` или `isDirectory()`. Заголовки этих методов приведены ниже.

```
boolean isFile()
```

```
boolean isDirectory()
```

Метод `isFile()` возвращает значение `true`, если объект является файлом, и значение `false` в противном случае. Метод `isDirectory()` выполняет те же действия для каталогов. В классе `File` определено большое число других методов, которые могут быть полезны при работе над приложением. Желательно выбрать время и подробно изучить возможности, предоставляемые данным классом.

Ниже приведен код программы, демонстрирующей использование класса `JFileChooser`. В процессе работы программы отображается диалоговое окно `Open`, инициализированное для работы с каталогом по умолчанию (рис. 9.11).

```
// Пример, демонстрирующий использование класса JFileChooser.
```

```
import java.awt.*;
```

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

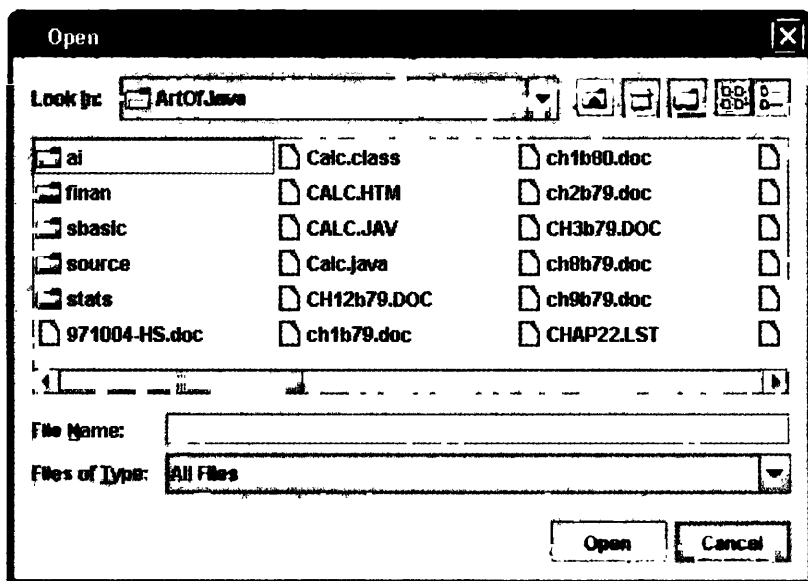


Рис. 9.11. Диалоговое окно, отображаемое в процессе работы программы `FileChooserDemo`



```
import javax.swing.*;

class FileChooserDemo {

    JLabel jlab;
    JButton jbtnShow;
    JFileChooser jfc;

    FileChooserDemo() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("JFileChooser Demo");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(400, 200);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание метки, отображающей выбранный файл.
        jlab = new JLabel();

        // Создание кнопки, вызывающей отображение
        // диалогового окна.
        jbtnShow = new JButton("Show File Chooser");

        // Создание выбора файла, первоначально отображающего
        // содержимое каталога по умолчанию.
        jfc = new JFileChooser();

        // Отображение окна выбора файла в ответ на активизацию
        // кнопки Show File Chooser.
        jbtnShow.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent le) {
                // Отображение окна выбора файла.
                // В качестве параметра, определяющего владельца окна,
                // задается значение null. В результате диалоговое
                // окно выводится по центру экрана.
                int result = jfc.showOpenDialog(null);

                // Если файл был выбран, отображается его имя.
            }
        });
    }
}
```

```
        if(result == JFileChooser.APPROVE_OPTION)
            jlab.setText("Selected file is: " +
                        jfc.getSelectedFile().getName());
        else
            jlab.setText("No file selected.");
    }
});

// Включение кнопки Show File Chooser и метки
// в состав панели содержимого.
jfrm.getContentPane().add(jbtnShow);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new FileChooserDemo();
        }
    });
}
}
```

Код программы достаточно прост. После ее запуска создается объект `JFileChooser`, ссылка на который помещается в переменную `jfc`. Поскольку каталог не указан, в окне отображаются файлы из текущего каталога. При активизации кнопки `Show File Chooser` получает управление метод `actionPerformed()`, который посредством обращения к методу `showOpenDialog()` отображает диалоговое окно выбора файла. Если метод возвращает значение `APPROVE_OPTION`, то имя выбранного файла отображается с помощью метки `jlab`. В противном случае выводится сообщение о том, что файл не был выбран. Следует заметить, что подход, согласно которому окно выбора файла формируется за пределами метода `actionPerformed()`, позволяет повторно использовать это окно, не создавая его каждый раз заново.

Перед тем как переходить к изучению следующего раздела, попробуйте вместо `showOpenDialog()` использовать метод `showSaveDialog()`. Как вы увидите, эти методы отличаются только заголовком окна и именем кнопки.

**ВАЖНО!****9.9. Использование фильтров**

По умолчанию `JFileChooser` отображает все файлы выбранного каталога, за исключением скрытых. Его поведение можно изменить, указав имена файлов, содержащие символы групповых операций. Эту же задачу можно решить из программы, используя *фильтр файлов*. Фильтр файлов — это экземпляр класса, расширяющего абстрактный класс `FileFilter`. (Класс `FileFilter` принадлежит пакету `javax.swing.filechooser`.) Необходимо учитывать одну важную особенность. В пакете `javax.swing.filechooser` имеется класс `FileFilter`, но интерфейс с таким же именем присутствует и в пакете `java.io`. При создании фильтра для выбора файлов часто бывает необходимо импортировать оба пакета (`java.io` и `javax.swing.filechooser`). Для того чтобы исключить конфликт имен, следует указывать при импортировании имя класса `FileFilter`:

```
import javax.swing.filechooser.FileFilter;
```

В классе `FileFilter` объявлены следующие методы:

```
abstract boolean accept(File file)
abstract String getDescription()
```

Оба они должны быть определены в классе, реализующем фильтр. Метод `accept()` должен возвращать значение `true`, чтобы файл, переданный посредством параметра `file`, был принят. Другими словами, если вы хотите, чтобы файл отображался в окне, необходимо создать метод `accept()` так, чтобы он возвращал для этого файла значение `true`. Отменить отображение файла можно, реализовав данный метод так, чтобы он возвращал для этого файла значение `false`. Метод `getDescription()` должен возвращать строку с описанием фильтра. Оно отображается в окне выбора файлов в списке `Files of Type`.

Следует помнить, что при использовании фильтра, созданного разработчиком, каталоги не отображаются автоматически. Если вы хотите разрешить вывод каталогов, то должны предусмотреть это в методе `accept()`.

Для того чтобы связать фильтр с окном выбора файлов, вам надо вызвать метод `setFileFilter()` экземпляра класса `JFileChooser`. Заголовок этого метода имеет следующий вид:

```
void setFileFilter(FileFilter ff)
```

где `ff` — это фильтр, который надо связать с окном.

Ниже приведен код программы, реализующей простой фильтр файлов. В диалоговом окне отображаются исходные файлы Java, т.е. файлы, имеющие

расширение .java. Кроме того, программа выводит в окне каталоги, что обеспечивает навигацию по файловой системе. Результаты использования фильтра показаны на рис. 9.12.

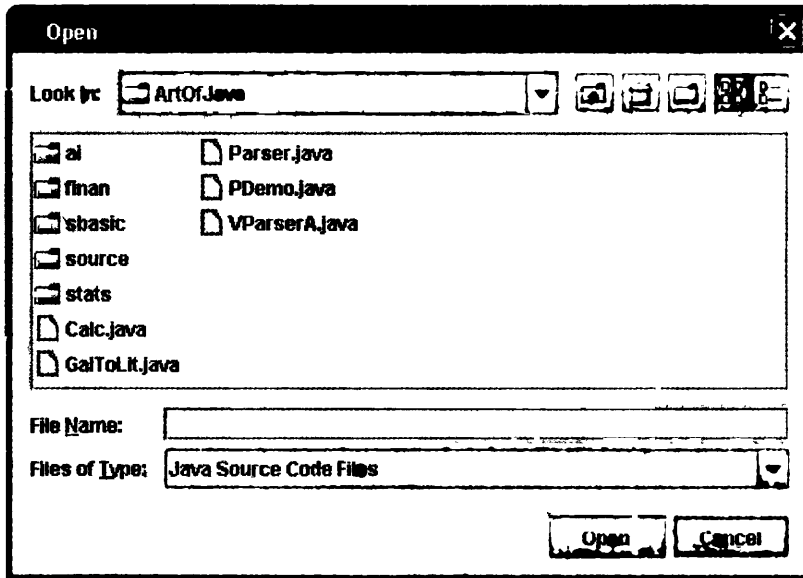


Рис. 9.12. Диалоговое окно, отображаемое в процессе работы программы *FileFilterDemo*

// Использование фильтра файлов.

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.filechooser.FileFilter;
```

// Создание фильтра файлов, обеспечивающего отображение  
// исходных файлов Java и каталогов.

```
class JavaFileFilter extends FileFilter {
    public boolean accept(File file) {
        // Метод возвращает значение true, если в качестве
        // параметра ему был передан исходный файл Java
        // или каталог.
        if (file.getName().endsWith(".java")) return true;
```

```
        if(file.isDirectory()) return true;

        // В противном случае метод возвращает значение false.
        return false;
    }

    public String getDescription() {
        return "Java Source Code Files";
    }
}

class FileFilterDemo {

    JLabel jlab;
    JButton jbtnShow;
    JFileChooser jfc;

    FileFilterDemo() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("File Filter Demo");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(400, 200);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание метки, отображающей имя выбранного файла.
        jlab = new JLabel();

        // Создание кнопки, вызывающей отображение
        // диалогового окна.
        jbtnShow = new JButton("Show File Chooser");

        // Создание окна выбора файлов.
        jfc = new JFileChooser();

        // Установка фильтра файлов.
        jfc.setFileFilter(new JavaFileFilter());
    }
}
```

```
// Отображение окна выбора файла при активизации
// кнопки Show File Chooser.
jbtnShow.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        // В качестве параметра, определяющего владельца окна,
        // задается значение null. В результате диалоговое
        // окно выводится по центру экрана.
        int result = jfc.showOpenDialog(null);

        if(result == JFileChooser.APPROVE_OPTION)
            jlab.setText("Selected file is: " +
                        jfc.getSelectedFile().getName());
        else
            jlab.setText("No file selected.");
    }
});

// Включение кнопки Show File Chooser и метки
// в состав панели содержимого.
jfrm.getContentPane().add(jbtnShow);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new FileFilterDemo();
        }
    });
}
}
```

Данная программа создана на основе предыдущего примера. Сначала в ней определяется класс файлового фильтра `JavaFileFilter`, который принимает либо каталоги, либо файлы, содержащие исходный текст на языке Java. Заметьте, что решение о том, должен ли файл отображаться в окне, принимает метод `accept()`, содержащий следующий фрагмент кода:

```
if(file.getName().endsWith(".java")) return true;
if(file.isDirectory()) return true;

// В противном случае возвращается значение false.
return false;
```

Если файл имеет расширение `.java`, он принимается. Проверка выполняется с помощью метода `endsWith()`, определенного в классе `String`. Если имя файла оканчивается символами `.java`, то метод `accept()` возвращает значение `true`. В противном случае вызывается метод `isDirectory()`, который проверяет, является ли файл каталогом; при положительном результате проверки метод `accept()` также возвращает значение `true`. В других случаях данный метод возвращает значение `false`. В результате в диалоговом окне отображаются только файлы `.java` и каталоги.

Для того чтобы фильтр файлов выполнял поставленную перед ним задачу, его надо связать с окном. Для этой цели сразу после создания фильтра вызывается метод `setFileFilter()` объекта `jfc`. Перед тем как продолжить изучение материала, желательно поэкспериментировать с файловыми фильтрами.

**ВАЖНО!**  
**9.10.**

## **Дополнительные возможности класса JFileChooser**

Класс `JFileChooser` предоставляет разработчику ряд дополнительных возможностей. В этом разделе мы рассмотрим некоторые из них, используемые наиболее часто. Иногда бывает необходимо ограничить возможности пользователя выбором только файлов или только каталогов; в других случаях пользователю следует предоставить доступ и к каталогам, и к файлам. По умолчанию класс `JFileChooser` позволяет пользователю выбирать только файлы. Чтобы проверить, так ли это, запустите рассмотренную выше программу и попытайтесь выбрать каталог. Вы увидите, что это невозможно. Для того чтобы разрешить выбор каталогов, надо вызвать метод `setFileSelectionMode()`.

```
void setFileSelectionMode(int fsm)
```

Здесь параметр `fsm` определяет режим выбора; его значение должно быть равно одной из перечисленных ниже констант, определенных в файле `JFileChooser`.

```
FILES_ONLY
```

```
DIRECTORIES_ONLY
```

```
FILES_AND_DIRECTORIES
```

Режим, задаваемый каждым из этих значений, понятен по имени константы.

Получить текущий каталог позволяет метод `getCurrentDirectory()`, который определен следующим образом:

```
File getCurrentDirectory()
```

Возвращаемый объект `File` инкапсулирует текущий каталог. С его помощью можно получить полный путь, вызвав метод `getPath()`.

Можно предоставить пользователю возможность выбирать несколько файлов. Сделать это позволяет метод `setMultiSelectionEnabled()`.

```
void setMultiSelectionEnabled(boolean on)
```

Если значение параметра `on` равно `true`, выбор нескольких файлов разрешен. Если задано значение `false`, пользователь может выбирать только один файл. По умолчанию разрешен выбор только одного файла.

Если установлен режим выбора нескольких файлов, то получить список файлов, выбранных пользователем, можно, вызвав метод `getSelectedFiles()`.

```
File[] getSelectedFiles()
```

Он возвращает массив объектов `File`, содержащих информацию о выбранных файлах (или каталогах).

По умолчанию скрытые файлы не отображаются в окне. Для того чтобы отменить данное ограничение, надо вызвать метод `setFileHidingEnabled()`.

```
void setFileHidingEnabled(boolean on)
```

Если значение параметра `on` равно `true`, скрытые файлы не отображаются. Значение `false` разрешает их вывод.

## Вопросы для текущего контроля

1. Какой метод отображает окно выбора файлов `Open`?
2. На базе какого класса создается файловый фильтр, предназначенный для работы с окном `JFileChooser`?
3. Какой метод устанавливает режим, позволяющий пользователю выбирать несколько файлов?

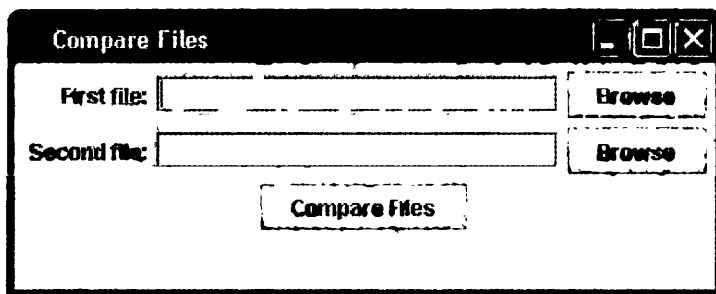
- 
1. `showOpenDialog()`.
  2. `FileFilter`.
  3. `setMultiSelectionEnabled()`.



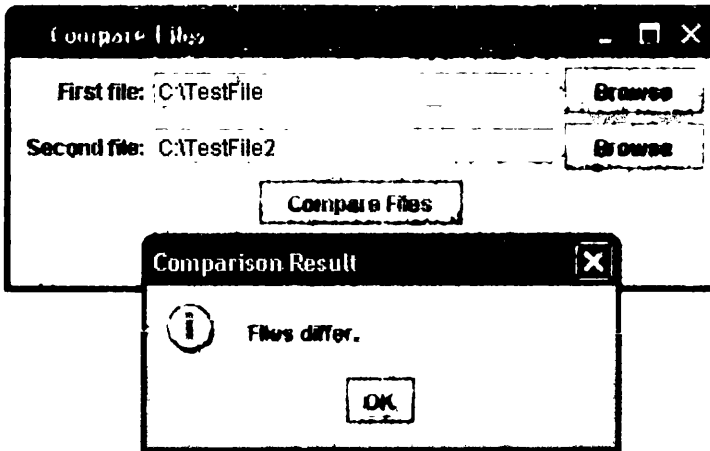
Поддержку компонента `JFileChooser` осуществляют три класса, содержащиеся в пакете `javax.swing.filechooser`. Один из них, `FileFilter`, уже знаком вам. Два других — `FileView` и `FileSystemView`. Класс `FileView` определяет, как должна отображаться информация о файле в окне. Класс `FileSystemView` инкапсулирует информацию о файловой системе. С его помощью можно легко получить данные о файлах, разделах, устройствах и т.д. Необходимость в использовании этих классов почти никогда не возникает. Исключение составляют специализированные приложения.

## Проект 9.1. Создание программы сравнения файлов

`CompareFiles.java`    В рамках данного проекта мы создадим реальное приложение, использующее два класса, предназначенных для создания диалоговых окон: `JOptionPane` и `JFileChooser`. Данное приложение будет выполнять сравнение файлов. Сначала программа получает от пользователя имена файлов, а затем сравнивает их и сообщает, совпадает ли их содержимое. Окно программы показано ниже.



Имена файлов для сравнения можно задавать двумя способами: вводить их в полях редактирования или выбирать, перемещаясь по каталогам с помощью специальных диалоговых окон. Во втором случае путь к выбранному файлу копируется в соответствующее поле редактирования. По щелчку на кнопке `Compare Files` производится сравнение указанных файлов. Результаты отображаются с помощью диалогового окна вывода сообщений. Внешний вид такого окна приведен ниже.



## Последовательность действий

1. Создайте файл `CompareFiles.java` и включите в него следующие комментарии и выражения `import`, приведенные ниже.

```
// Проект 9.1. Программа сравнения файлов.  
//  
// В данном проекте используются классы  
// JOptionPane и JFileChooser.
```

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

2. Определите класс `CompareFiles` и включите в него следующие переменные:

```
class CompareFiles {  
  
    JLabel jlabFirst;  
    JLabel jlabSecond;  
  
    JButton jbtnGetFirst;  
    JButton jbtnGetSecond;  
    JButton jbtnCompare;  
  
    JTextField jtfFirst;  
    JTextField jtfSecond;
```

```
JFileChooser jfc;
```

Поля редактирования `jtfFirst` и `jtfSecond` отображают имена сравниваемых файлов. Пользователь может непосредственно вводить их. Если пользователь выберет файлы с помощью диалогового окна, то имена файлов также отобразятся в составе данных компонентов.

- 3.** Начните код конструктора `CompareFiles` приведенными ниже выражениями.

```
CompareFiles() {  
    // Создание нового контейнера JFrame.  
    JFrame jfrm = new JFrame("Compare Files");  
  
    // Установка диспетчера компоновки FlowLayout.  
    jfrm.getContentPane().setLayout(new FlowLayout());  
  
    // Установка начальных размеров фрейма.  
    jfrm.setSize(400, 160);  
  
    // Завершение программы при закрытии окна пользователем.  
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
}
```

- 4.** Создайте метки, описывающие назначение полей редактирования и сами поля.

```
// Создание меток для полей редактирования.  
JLabel jlabFirst = new JLabel("First file:");  
jlabFirst.setPreferredSize(new Dimension(70, 20));  
jlabFirst.setHorizontalAlignment(SwingConstants.RIGHT);  
  
JLabel jlabSecond = new JLabel("Second file:");  
jlabSecond.setPreferredSize(new Dimension(70, 20));  
jlabSecond.setHorizontalAlignment(SwingConstants.RIGHT);  
  
// Создание полей редактирования для ввода имен файлов.  
JTextField jtfFirst = new JTextField(20);  
JTextField jtfSecond = new JTextField(20);
```

Заметьте, что для меток заданы одинаковые размеры и тип выравнивания. Параметры полей редактирования также совпадают. Это улучшает внешний вид окна.



## 5. Создайте две кнопки: Browse и Compare Files.

```
// Создание кнопок Browse.
jbtnGetFirst = new JButton("Browse");
jbtnGetSecond = new JButton("Browse");

// Создание кнопки, запускающей процесс сравнения файлов.
jbtnCompare = new JButton("Compare Files");
```

## 6. Создайте окно выбора файла, первоначально отображающее текущий каталог.

```
// Создание окна выбора файла, отображаемого по
// щелчку на кнопке Browse.
jfc = new JFileChooser();
```

## 7. Свяжите с кнопками Browse обработчики событий действия.

```
// Выбор первого файла.
jbtnGetFirst.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {

        int result = jfc.showDialog(null, "Select");

        if(result == JFileChooser.APPROVE_OPTION) {
            File f = jfc.getSelectedFile();
            jtfFirst.setText(f.getPath());
        }
    }
});

// Выбор второго файла.
jbtnGetSecond.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {

        int result = jfc.showDialog(null, "Select");

        if(result == JFileChooser.APPROVE_OPTION) {
            File f = jfc.getSelectedFile();
            jtfSecond.setText(f.getPath());
        }
    }
});
```

Каждый из методов `actionPerformed()` активизирует окно выбора файла **Select**. После выбора файла пользователем путь к нему копируется в соответствующее поле редактирования. Этот подход гарантирует, что в полях редактирования всегда будут отображаться имена файлов, предназначенных для сравнения.

8. Свяжите с кнопкой **Compare Files** обработчик событий, код которого приведен ниже. При активизации кнопки начнется сравнение выбранных файлов.

```
// Сравнение файлов.
jbtnCompare.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {

        // Проверить, введены ли имена файлов.
        if(jtfFirst.getText().length() == 0 ||
            jtfSecond.getText().length() == 0) {
            JOptionPane.showMessageDialog(null,
                "Please specify the files to compare.",
                "Filename Not Specified",
                JOptionPane.WARNING_MESSAGE);

            return;
        }

        // Создание объектов File на основе имен файлов.
        File f1 = new File(jtfFirst.getText());
        File f2 = new File(jtfSecond.getText());

        // Проверить, существуют ли указанные файлы.
        if(!f1.exists()) {
            JOptionPane.showMessageDialog(null,
                "The first file does not exist.",
                "File Not Found",
                JOptionPane.WARNING_MESSAGE);

            return;
        }
        if(!f2.exists()) {
            JOptionPane.showMessageDialog(null,
                "The second file does not exist.",
                "File Not Found",
                JOptionPane.WARNING_MESSAGE);

            return;
        }
    }
}
```

```
// Сравнение файлов.
if(compare(f1, f2))
    JOptionPane.showMessageDialog(null,
        "Files compare equal.",
        "Comparison Result",
        JOptionPane.INFORMATION_MESSAGE);
else
    JOptionPane.showMessageDialog(null,
        "Files differ.",
        "Comparison Result",
        JOptionPane.INFORMATION_MESSAGE);
}
});
```

В процессе обработки события сначала проверяется, содержит ли каждое поле редактирования строку символов. Если хотя бы одно из полей пусто, отображается сообщение, напоминающее пользователю о том, что необходимо ввести имена файлов для сравнения. Далее создаются два объекта `File`, инкапсулирующие имена файлов. Как вы знаете, создание объекта `File` еще не означает открытие соответствующего файла. Данный объект используется лишь для представления файла. Далее в программе проверяется, существуют ли указанные файлы. Если хотя бы один из файлов отсутствует, отображается сообщение об этом.

Содержимое существующих файлов сравнивается с помощью метода `compare()`. Этот метод возвращает значение `true`, если содержимое двух файлов совпадает. Результаты сравнения отображаются с помощью диалогового окна с сообщением.

Заметьте, что обрабатывать события, сгенерированные полями редактирования, не обязательно. Поэтому обработчики для данных событий в программе не предусмотрены. Имена файлов извлекаются из соответствующих компонентов после того, как пользователь щелкнет на кнопке `Compare Files`.

9. Завершите конструктор `CompareFiles()` строками кода, предназначенными для включения компонентов в панель содержимого и отображения окна.

```
// Добавление компонентов к панели содержимого.
jfrm.getContentPane().add(jlabFirst);
jfrm.getContentPane().add(jtfFirst);
jfrm.getContentPane().add(jbtnGetFirst);
jfrm.getContentPane().add(jlabSecond);
```

```

jfrm.getContentPane().add(jtfSecond);
jfrm.getContentPane().add(jbtnGetSecond);
jfrm.getContentPane().add(jbtnCompare);

// Отображение фрейма.
jfrm.setVisible(true);
}

```

**10.** Добавьте метод `compare()`, код которого показан ниже.

```

// Сравнение двух файлов.
boolean compare(File fileA, File fileB) {

    // Если файлы имеют разную длину, они не совпадают.
    if(fileA.length() != fileB.length()) return false;

    FileInputStream f1, f2;
    int i, j;
    byte buf1[] = new byte[1024];
    byte buf2[] = new byte[1024];

    try {
        f1 = new FileInputStream(fileA);
        f2 = new FileInputStream(fileB);

        do {
            // Данная версия метода read() возвращает число
            // прочитанных байтов или значение -1, если был
            // достигнут конец файла.
            i = f1.read(buf1, 0, 1024);
            j = f2.read(buf2, 0, 1024);

            // Если содержимое буферов не совпадает, файлы
            // различаются. В этом случае файлы закрываются и
            // возвращается значение false.
            if(!Arrays.equals(buf1, buf2)) {
                f1.close();
                f2.close();
                return false;
            }
        } while(i != -1 && j != -1);

        f1.close();
        f2.close();
    }
}

```

```
} catch(IOException exc) {
    JOptionPane.showMessageDialog(null, exc,
        "File Error!",
        JOptionPane.WARNING_MESSAGE);
    return false; // В случае ошибки возвращается
                  // значение false.
}

// Файлы совпадают; возвращается значение true.
return true;
}
```

Методу `compare()` передаются два объекта `File`. В теле метода сначала выполняется проверка на совпадение длины двух файлов. Если файлы имеют различную длину, метод `compare()` возвращает значение `false`, поскольку файлы очевидно различаются. Затем файлы открываются, данные читаются и сравниваются. Эта процедура продолжается до тех пор, пока не будут найдены различающиеся байты (в этом случае метод возвращает значение `false`) либо пока не будет достигнут конец каждого из файлов. Вариант метода `read()`, применяемый в данной программе, возвращает значение `-1` по достижении конца файла. Если в составе файлов не обнаружены различающиеся байты, метод `compare()` возвращает значение `true`.

Для повышения эффективности данные читаются блоками по 1024 байта. Чтение такого блока производится гораздо быстрее, чем 1024 операции чтения одного байта. Значение 1024 выбрано почти произвольно. Учитывался лишь тот факт, что желательно, чтобы размер буфера был кратным 256 и чтобы он был не слишком мал и не слишком велик. Вы можете поэкспериментировать с размерами буфера, в частности, оценить как повлияет на эффективность работы программы многократное увеличение или уменьшение этого размера.

**11.** Завершите код программы привычным методом `main()`.

```
public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new CompareFiles();
        }
    });
}
```



**12. Полностью код программы сравнения файлов приведен ниже.**

```
// Проект 9.1. Программа сравнения файлов.
//
// В данном проекте используются классы

import java.io.*;
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class CompareFiles {

    JLabel jlabFirst;
    JLabel jlabSecond;

    JButton jbtnGetFirst;
    JButton jbtnGetSecond;
    JButton jbtnCompare;

    JTextField jtfFirst;
    JTextField jtfSecond;

    JFileChooser jfc;

    CompareFiles() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Compare Files");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(400, 160);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание меток для полей редактирования.
        jlabFirst = new JLabel("First file:");
        jlabFirst.setPreferredSize(new Dimension(70, 20));
        jlabFirst.setHorizontalAlignment(SwingConstants.RIGHT);
```

```
jlabSecond = new JLabel("Second file:");
jlabSecond.setPreferredSize(new Dimension(70, 20));
jlabSecond.setHorizontalAlignment(SwingConstants.RIGHT);

// Создание полей редактирования для ввода имен файлов.
jtfFirst = new JTextField(20);
jtfSecond = new JTextField(20);

// Создание кнопок Browse.
jbtnGetFirst = new JButton("Browse");
jbtnGetSecond = new JButton("Browse");

// Создание кнопки, запускающей процесс сравнения файлов.
jbtnCompare = new JButton("Compare Files");

// Создание окна выбора файла, отображаемого по
// щелчку на кнопке Browse.
jfc = new JFileChooser();

// Выбор первого файла.
jbtnGetFirst.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {

        int result = jfc.showDialog(null, "Select");

        if(result == JFileChooser.APPROVE_OPTION) {
            File f = jfc.getSelectedFile();
            jtfFirst.setText(f.getPath());
        }
    }
});

// Выбор второго файла.
jbtnGetSecond.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {

        int result = jfc.showDialog(null, "Select");

        if(result == JFileChooser.APPROVE_OPTION) {
            File f = jfc.getSelectedFile();
            jtfSecond.setText(f.getPath());
        }
    }
});
```

```
    }  
    });  
  
    // Сравнение файлов.  
    jbtnCompare.addActionListener(new ActionListener() {  
        public void actionPerformed(ActionEvent le) {  
  
            // Проверить, введены ли имена файлов.  
            if(jtfFirst.getText().length() == 0 ||  
                jtfSecond.getText().length() == 0) {  
                JOptionPane.showMessageDialog(null,  
                    "Please specify the files to compare.",  
                    "Filename Not Specified",  
                    JOptionPane.WARNING_MESSAGE);  
  
                return;  
            }  
  
            // Создание объектов File на основе имен файлов.  
            File f1 = new File(jtfFirst.getText());  
            File f2 = new File(jtfSecond.getText());  
  
            // Проверить, существуют ли указанные файлы.  
            if(!f1.exists()) {  
                JOptionPane.showMessageDialog(null,  
                    "The first file does not exist.",  
                    "File Not Found",  
                    JOptionPane.WARNING_MESSAGE);  
  
                return;  
            }  
  
            if(!f2.exists()) {  
                JOptionPane.showMessageDialog(null,  
                    "The second file does not exist.",  
                    "File Not Found",  
                    JOptionPane.WARNING_MESSAGE);  
  
                return;  
            }  
  
            // Сравнение файлов.  
            if(compare(f1, f2))  
                JOptionPane.showMessageDialog(null,  
                    "Files compare equal.",  
                    "Comparison Result",
```

```
        JOptionPane.INFORMATION_MESSAGE);
    else
        JOptionPane.showMessageDialog(null,
            "Files differ.",
            "Comparison Result",
            JOptionPane.INFORMATION_MESSAGE);
    }
});

// Добавление компонентов к панели содержимого.
jfrm.getContentPane().add(jlabFirst);
jfrm.getContentPane().add(jtfFirst);
jfrm.getContentPane().add(jbtnGetFirst);
jfrm.getContentPane().add(jlabSecond);
jfrm.getContentPane().add(jtfSecond);
jfrm.getContentPane().add(jbtnGetSecond);
jfrm.getContentPane().add(jbtnCompare);

// Отображение фрейма.
jfrm.setVisible(true);
}

// Сравнение двух файлов.
boolean compare(File fileA, File fileB) {

    // Если файлы имеют разную длину, они не совпадают.
    if(fileA.length() != fileB.length()) return false;

    FileInputStream f1, f2;
    int i, j;
    byte buf1[] = new byte[1024];
    byte buf2[] = new byte[1024];

    try {
        f1 = new FileInputStream(fileA);
        f2 = new FileInputStream(fileB);

        do {
            // Данная версия метода read() возвращает число
            // прочитанных байтов или значение -1, если был
            // достигнут конец файла.
            i = f1.read(buf1, 0, 1024);
            j = f2.read(buf2, 0, 1024);
```

```

        // Если содержимое буферов не совпадает, файлы
        // различаются. В этом случае файлы закрываются и
        // возвращается значение false.
        if(!Arrays.equals(buf1, buf2)) {
            f1.close();
            f2.close();
            return false;
        }
    } while(i != -1 && j != -1);

    f1.close();
    f2.close();
} catch(IOException exc) {
    JOptionPane.showMessageDialog(null, exc,
        "File Error!",
        JOptionPane.WARNING_MESSAGE);
    return false; // В случае ошибки возвращается
        // значение false.
}

// Файлы совпадают; возвращается значение true.
return true;
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new CompareFiles();
        }
    });
}
}

```

**ВАЖНО!**

## **9.11. Выбор цвета с помощью окна JColorChooser**

Помимо окон выбора файлов, Swing также содержит встроенное диалоговое окно `JColorChooser`, в котором пользователю предоставляется палитра для выбора цвета. Обычно выбирать цвет приходится гораздо реже, чем файлы, но,

несмотря на это, `JColorChooser` является важным компонентом, так как он автоматически формирует сложное диалоговое окно, которое иначе пришлось бы создавать вручную. В данном разделе мы рассмотрим только один вариант `JColorChooser`, наиболее простой в использовании. Для большинства приложений возможности, предоставляемые данным компонентом, оказываются вполне достаточными.

Класс `JColorChooser` является подклассом `JComponent`. В нем определены три конструктора, которые используются достаточно редко. Вместо них для создания окна выбора цвета обычно применяется статический метод `showDialog()`. Этот метод создает модальное диалоговое окно и отображает его на экране. Возвращает метод информацию о цвете, выбранном пользователем. Заголовок метода представлен ниже.

```
static Color showDialog(Component parent, String title,  
    Color initClr) throws HeadlessException
```

Здесь параметр `parent` определяет компонент, которому принадлежит отображаемое окно. Если в качестве первого параметра вы зададите значение `null`, окно будет выведено по центру экрана. Заголовок окна выбора цвета определяет параметр `title`. Обычно в качестве заголовка принимается строка типа "Choose Color", но при желании вы можете указать более информативный заголовок. Цвет, который должен быть выбран при отображении окна, задается посредством параметра `initClr`. Данный метод возвращает информацию о цвете, выбранном пользователем. Если пользователь отказался выбирать цвет, т.е. закрыл окно щелчком на кнопке `Cancel` или на кнопке, расположенной в верхнем правом углу окна, метод возвращает значение `null`.

Ниже приведен код программы, демонстрирующей использование класса `JColorChooser`. В процессе выполнения программы отображается диалоговое окно выбора цвета, показанное на рис. 9.13.

```
// Пример использования класса JColorChooser.
```

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

```
class ColorChooserDemo {
```

```
    JLabel jlab;  
    JButton btnShow;
```

```
    ColorChooserDemo() {
```

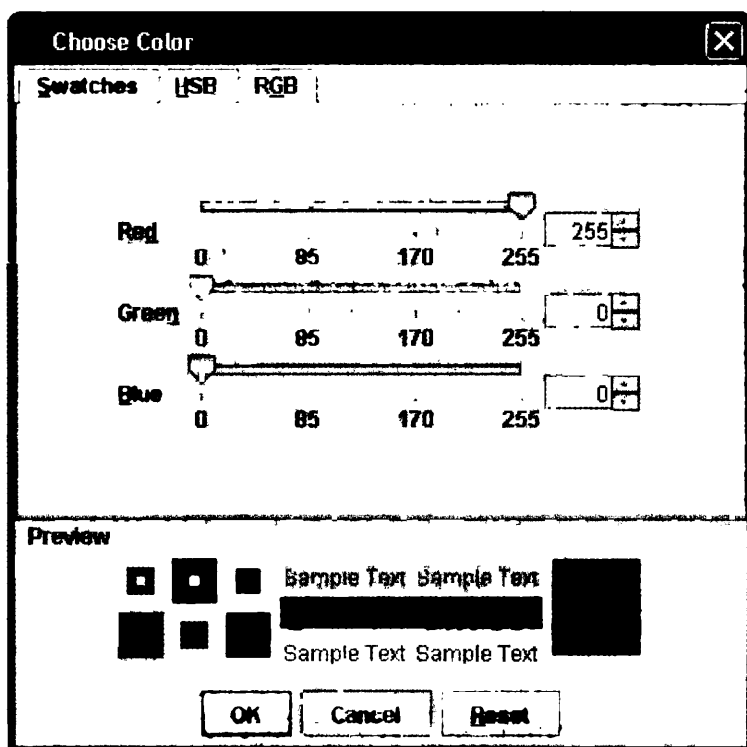


Рис. 9.13. Диалоговое окно JColorChooser

```
// Создание нового контейнера JFrame.
JFrame jfrm = new JFrame("Color Chooser Demo");

// Установка диспетчера компоновки FlowLayout.
jfrm.getContentPane().setLayout(new FlowLayout());

// Установка начальных размеров фрейма.
jfrm.setSize(400, 200);

// Завершение программы при закрытии окна пользователем.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Создание метки, отображающей выбор пользователя.
jlab = new JLabel();

// // Создание кнопки, вызывающей отображение
// диалогового окна.
jbtnShow = new JButton("Show Color Chooser");
```

```
// Отображение окна выбора цвета при активизации
// кнопки Show Color Chooser.
jbtnShow.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        // Создание окна выбора цвета. В качестве первого
        // параметра методу передается значение null.
        // В результате окно выводится по центру экрана.
        // При отображении окна первоначально выбирается
        // красный цвет
        Color color = JColorChooser.showDialog(null,
                                                "Choose Color",
                                                Color.RED);

        if(color != null)
            jlab.setText("Selected color is " +
                        color.toString());
        else
            jlab.setText("Color selection was cancelled.");
    }
});

// Включение кнопки Show Color Chooser и метки
// в состав панели содержимого.
jfrm.getContentPane().add(jbtnShow);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new ColorChooserDemo();
        }
    });
}
}
```



Код программы достаточно прост. После того как пользователь щелкнет на кнопке **Show Color Chooser**, отображается окно выбора цвета; для этого в обработчике событий кнопки `actionPerformed()` вызывается метод `showDialog()`.

```
Color color = JColorChooser.showDialog(null,
                                         "Choose Color",
                                         Color.RED);
```

В качестве значения параметра `parent` задается `null`, это означает, что окно выбора цвета должно отображаться посередине экрана. В качестве заголовка окна выбирается строка "Choose Color", а при отображении окна первоначально выбирается красный цвет.

Метод `showDialog()` возвращает цвет, выбранный пользователем, или значение `null`, если выбор не был сделан. Результаты отображаются посредством метки `jlab`. Вывод данных осуществляется посредством приведенного ниже фрагмента кода.

```
if(color != null)
    jlab.setText("Selected color is " +
                color.toString());
else
    jlab.setText("Color selection was cancelled.");
```

Очевидно, что в реальной программе информация о выборе пользователя должна использоваться для изменения характеристик компонента, например, для установки цвета фона. Как видно из приведенного примера, несмотря на то, что задача выбора цвета очень сложна, метод `JColorChooser` позволяет достаточно легко решить ее.



## **Тест для самоконтроля по модулю 9**

1. Диалоговое окно содержит два или несколько компонентов. Оно отображает сообщение для пользователя и ожидает ответа. Да или нет?
2. Какие классы, предназначенные для создания диалоговых окон, содержатся в составе `Swing`? Назовите два класса общего назначения.
3. Какой из методов класса `JOptionPane` создает диалоговое окно для ввода данных? Какой из них создает диалоговое окно с сообщением?
4. Какой метод класса `JOptionPane` обычно используется для создания диалогового окна, посредством которого запрашивается согласие пользователя на сохранение изменений, внесенных в документ? Как должен

выглядеть вызов этого метода?

5. Предположим, вы используете диалоговое окно для получения подтверждения от пользователя. Какое возвращаемое значение указывает на то, что пользователь щелкнул на кнопке **Yes**?
6. Какое значение указывает на то, что в диалоговом окне, предназначенном для получения подтверждения от пользователя, должны отображаться только кнопки **Yes** и **No**?
7. Предположим, вы хотите получить от пользователя ответ в виде строки. Какой метод класса `JOptionPane` следует для этого использовать?
8. Обязательно ли параметр метода `show...()` класса `JOptionPane`, определяющий сообщение, должен представлять собой строку символов? Обоснуйте свой ответ.
9. `JDialog` является контейнером верхнего уровня. Да или нет?
10. Какие четыре действия надо выполнить для создания и отображения диалогового окна на основе класса `JDialog`?
11. Можно ли с помощью класса `JDialog` создать немодальное диалоговое окно?
12. Чем отличаются вызовы методов `setVisible(false)` и `dispose()` для диалогового окна?
13. Какой метод класса `JFileChooser` создает диалоговое окно **Save** для выбора файла? Какой метод позволяет явно указывать заголовок создаваемого диалогового окна?
14. Какие два метода надо переопределить при реализации фильтра (`FileFilter`) для `JFileChooser`?
15. Можно ли с помощью окна выбора файла выбирать каталоги? Если да, то как это сделать?
16. Значения какого типа возвращает метод `showDialog()` класса `JColorChooser`?
17. Модуль 7 был посвящен меню. В нем рассматривался пример создания меню **File**, содержащего пункт **Exit**. Обработчик событий действия, соответствующий данному пункту, приведен ниже.

```
// Обработка событий действия для пункта меню.  
public void actionPerformed(ActionEvent ae) {  
    // Получение команды действия, соответствующей  
    // выбранному пункту.  
    String comStr = ae.getActionCommand();
```

```
// Если пользователь выбрал пункт Exit,  
// программа завершает выполнение.  
/  if(comStr.equals("Exit")) System.exit(0);  
    ...
```

Измените данный код так, чтобы при выборе пункта Exit отображалось диалоговое окно, запрашивающее у пользователя разрешение на завершение программы.

18. Поэкспериментируйте с программами, приведенными в данном модуле. Попробуйте изменять режимы их работы и оцените результаты.

# Модуль 10

## Потоки, апплеты рисование и компоновка

- 10.1.** Использование потоков
- 10.2.** Использование класса `Timer`
- 10.3.** Общие сведения об апплетах
- 10.4.** Основные элементы апплета
- 10.5.** Создание графического пользовательского интерфейса апплета
- 10.6.** Рисование в Java-программах
- 10.7.** Использование графического контекста
- 10.8.** Определение отображаемой области
- 10.9.** Запрос на рисование
- 10.10.** Использование диспетчера компоновки `GridBayLayout`
- 10.11.** Использование диспетчера компоновки `BoxLayout`

В предыдущих девяти модулях обсуждались различные компоненты Swing, например JButton, JMenu и JTable. Поскольку главным в библиотеке Swing являются компоненты, именно им посвящена большая часть данной книги. Однако существуют и другие вопросы, имеющие отношение к Swing, которые также требуют внимания. Они рассматриваются в последнем модуле книги. Несмотря на то что эти вопросы не имеют непосредственного отношения к визуальным компонентам, они влияют на организацию Swing-программы, ее выполнение и в конечном счете на внешний вид отображаемых окон. В данном модуле будут рассматриваться потоки, класс Timer, апплеты на базе Swing, средства рисования и два диспетчера компоновки. Изучив этот материал, вы будете готовы к тому, чтобы приступить к написанию собственных Swing-приложений. Вы также сможете продолжить изучение Swing в интересующих вас направлениях.

**ВАЖНО!**  
**10.1.**

## Многопоточность и Swing

Благодаря поддержке многопоточности у разработчика появляется возможность организовать одновременное выполнение различных частей Java-программ. Это позволяет создавать чрезвычайно эффективные приложения, максимально используя ресурс процессора. Многопоточность также обеспечивает реакцию приложения на действия пользователя даже в тот момент, когда оно выполняет длительные вычисления. При использовании Swing вычисления, требующие длительного времени, выполняются в отдельном потоке, в результате чего устраняется такой недостаток, как замедленный или вовсе отсутствующий отклик программы. Таким образом, поддержка многопоточности — важная особенность программ, созданных на базе Swing.

Каждый программист, применяющий Java, знает, что программа, написанная на этом языке, использует по крайней мере один поток (основной), а при необходимости могут быть созданы и другие потоки. В Swing-программе дополнительные потоки используются так же, как и в любом другом Java-приложении. Таким образом, все имеющиеся у вас знания о создании потоков и управлении ими применимы к Swing. В то же время поддержка потоков в Swing имеет важную особенность, которую мы рассмотрим в данном разделе.

На заметку



Предполагается, что читатель данной книги имеет хотя бы общее представление о работе с потоками. Желательно представлять себе возможности класса Thread и интерфейса Runnable. Объем данной книги не позволяет включить в нее обзор потоков. Если вы хотите получить общие сведения по этому вопросу, я могу порекомендовать вам свою книгу Java:

The Complete Reference, выпущенную издательством McGraw-Hill (русский перевод Полный справочник по Java, Издательский дом "Вильямс").

В модуле 1 было сказано, что любой код, предназначенный для взаимодействия с визуальным компонентом (в том числе с его моделью), должен выполняться в потоке обработки событий. Следование этому важному правилу позволит избежать многих проблем, например, он гарантирует, что не возникнет ситуация, при которой из двух различных потоков будет одновременно принята попытка обновить один и тот же компонент. По этой причине в примерах, приведенных в данной книге, для формирования и отображения пользовательского интерфейса в методе `main()` вызывается метод `invokeLater()`. Благодаря такому подходу элементы пользовательского интерфейса создаются в потоке обработки событий.

Тому же самому правилу необходимо следовать каждый раз, когда вы собираетесь обновить, изменить или опросить какой-либо компонент. Если некоторый фрагмент кода воздействует на компонент, то этот фрагмент должен быть выполнен в потоке обработки событий. Поскольку код обработчиков событий автоматически выполняется в этом потоке, ограничений по обращению к компонентам из обработчиков нет. Однако код, выполняющийся в другом потоке, не может делать это, хотя необходимость в подобных действиях возникает довольно часто. Предположим, например, что программа отображает посредством компонента `JLabel` информацию о температуре, которая должна обновляться каждую минуту. Для контроля температуры следует создать отдельный поток. Однако из этого потока нельзя обновлять метку, так как он не является потоком обработки событий. Каким же образом обновить метку из потока, отслеживающего изменения температуры? Или, если поставить вопрос шире, как из произвольного потока изменить состояние компонента пользовательского интерфейса?

Сделать это позволяет либо метод `invokeLater()`, либо `invokeAndWait()`. Оба метода определены в классе `SwingUtilities`. Принцип использования `invokeLater()` для создания и отображения пользовательского интерфейса уже знаком вам. Указанные методы упоминались в модуле 1, здесь же мы рассмотрим их более подробно. Заголовки данных методов приведены ниже.

```
static void invokeLater(Runnable obj)
```

```
static void invokeAndWait(Runnable obj)
```

```
throws InterruptedException, InvocationTargetException
```

Параметр `obj` — это объект `Runnable`, метод `run()` которого вызывается в потоке обработки событий. В метод `run()` помещается код, предназначенный для взаимодействия с компонентом Swing. Таким образом, если вам необходи-

мо обновить компонент, включите код в состав объекта `Runnable` и передайте этот объект методу `invokeLater()` или `invokeAndWait()`. В результате код будет выполнен в потоке обработки событий, а это значит, что при попытке изменить компонент проблемы не возникнут.

Различие между двумя указанными выше методами состоит в том, что метод `invokeLater()` сразу же возвращает управление вызывающему методу, а метод `invokeAndWait()` ожидает завершения метода `obj.run()`. В большинстве случаев удобнее использовать метод `invokeLater()`. Однако при создании интерфейса апплета лучше использовать `invokeAndWait()`. (Создание апплета, использующего компоненты `Swing`, будет описано далее в этом модуле.) В дальнейшем мы будем рассматривать только метод `invokeLater()`, однако те же самые принципы справедливы и для метода `invokeAndWait()`.

Ниже приведен пример, демонстрирующий постоянное обновление компонентов пользовательского интерфейса из отдельного потока. Программа представляет собой модифицированную версию класса `StopWatch`, созданного в рамках проекта 1.1. В данной версии при запущенном секундомере отсчитывается время и отображается на экране. Для этой цели создается отдельный поток, десять раз в секунду обновляющий метку с информацией о времени. Обратите внимание на использование метода `invokeLater()`. Окно, создаваемое при работе программы, показано на рис. 10.1.

```
// Модифицированная версия класса Stopwatch, созданного
// в рамках проекта 1.1. В этой версии для отображения
// времени, прошедшего с момента запуска секундомера,
// используется отдельный поток.
```

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

```
class ThreadStopWatch {
```

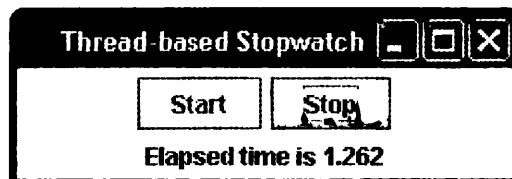


Рис. 10.1. Окно, отображаемое программой `ThreadStopWatch`

```
JLabel jlab; // Отображение времени, прошедшего
              // с момента запуска секундомера.

JButton jbtnStart; // Запуск секундомера.
JButton jbtnStop;  // Остановка секундомера.

long start; // Время запуска секундомера,
            // выраженное в миллисекундах.

boolean running=false; // Значение true, если секундомер запущен.

// Ссылка на поток, отслеживающий время, прошедшее
// с момента запуска секундомера.
Thread thrd;

ThreadStopWatch() {

    // Создание нового контейнера JFrame.
    JFrame jfrm = new JFrame("Thread-based Stopwatch");

    // Установка диспетчера компоновки FlowLayout.
    jfrm.getContentPane().setLayout(new FlowLayout());

    // Установка начальных размеров фрейма.
    jfrm.setSize(230, 90);

    // Завершение программы при закрытии окна пользователем.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Создание метки, отображающей время, прошедшее
    // с момента запуска секундомера.
    jlab = new JLabel("Press Start to begin timing.");

    // Создание меток Start и Stop.
    jbtnStart = new JButton("Start");
    jbtnStop = new JButton("Stop");

    // Первоначально доступ к кнопке Stop запрещен.
    jbtnStop.setEnabled(false);

    // Создание экземпляра класса Runnable, выполняющегося
    // в отдельном потоке.
    Runnable myThread = new Runnable() {
```



```
// Данный метод выполняется в отдельном потоке.
public void run() {
    try {
        // Отображение времени десять раз в секунду.
        for(;; ) {
            // Пауза длительностью в одну десятую секунды.
            Thread.sleep(100);

            // Вызов метода updateTime()
            // в потоке обработки событий.
            SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    // Обновление отображаемого значения. Метод
                    // updateTime() выполняется в потоке обработки
                    // событий.
                    updateTime();
                }
            });
        }
    } catch (InterruptedException exc) {
        System.out.println("Call to sleep was interrupted.");
        System.exit(1);
    }
}

// Создание нового потока.
thrd = new Thread(myThread);

// Запуск потока.
thrd.start();

// Связывание обработчиков событий с кнопками
// Start и Stop.
jbtnStart.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {

        // Запись времени запуска таймера.
        start = Calendar.getInstance().getTimeInMillis();

        // Изменение состояния кнопок на обратное.
        jbtnStop.setEnabled(true);
        jbtnStart.setEnabled(false);
    }
});
```

```
// Запуск секундомера.
running = true;
}
});

jbtnStop.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        long stop = Calendar.getInstance().getTimeInMillis();

        // Вычисление времени, прошедшего с момента запуска.
        jlab.setText("Elapsed time is "
            + (double) (stop - start)/1000);

        // Изменение состояния кнопок на обратное.
        jbtnStart.setEnabled(true);
        jbtnStop.setEnabled(false);

        // Остановка секундомера.
        running = false;
    }
});

// Включение кнопок и метки в состав панели содержимого.
jfrm.getContentPane().add(jbtnStart);
jfrm.getContentPane().add(jbtnStop);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

// Обновление значения времени, прошедшего с момента
// запуска секундомера.
// Данный метод выполняется в потоке обработки событий.
void updateTime() {
    if(!running) return;

    long temp = Calendar.getInstance().getTimeInMillis();
    jlab.setText("Elapsed time is " +
        (double) (temp - start)/1000);
}
```

```

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new ThreadStopWatch();
        }
    });
}
}

```

Рассмотрим код программы более подробно. В первую очередь обратите внимание на поля `running` и `thrd`. Переменная `running` инициализируется значением `false`. При запуске секундомера данной переменной присваивается значение `true`. В этом случае на экране отображается время, прошедшее с момента запуска секундомера. В поле `thrd` помещается ссылка на поток, отслеживающий время.

Заметьте также, как формируется второй поток. Сначала создается объект `Runnable` с именем `myThread`. Соответствующий фрагмент кода приведен ниже.

```

// Создание экземпляра класса Runnable, выполняющегося
// в отдельном потоке.
Runnable myThread = new Runnable() {
    // Данный метод выполняется в отдельном потоке.
    public void run() {
        try {
            // Отображение времени десять раз в секунду.
            for(;;) {
                // Пауза длительностью в одну десятую секунды.
                Thread.sleep(100);

                // Вызов метода updateTime()
                // в потоке обработки событий.
                SwingUtilities.invokeLater(new Runnable() {
                    public void run() {
                        updateTime();
                    }
                });
            }
        } catch (InterruptedException exc) {
            System.out.println("Call to sleep was interrupted.");
            System.exit(1);
        }
    }
};

```

В интерфейсе `Runnable` определен только один метод: `run()`. Этот метод выполняется в отдельном потоке. По сути, выполнение нового потока сводится к вызову метода `run()`. В теле этого метода находится бесконечный цикл, в котором выполнение программы приостанавливается на десятую часть секунды, а затем вызывается метод `invokeLater()`. Как вы уже знаете, тело цикла выполняется в отдельном потоке, отличном от потока обработки событий. (Если бы выполнение происходило в потоке обработки событий, пользователь не получал бы отклика от интерфейсных элементов, поскольку фрагменты программы вне цикла не получали бы управление.)

Обратите также внимание на параметр метода `invokeLater()`. Он представляет собой экземпляр класса `Runnable`, в методе `run()` которого присутствует вызов `updateTime()`. Заголовок метода `updateTime()` имеет следующий вид:

```
// Обновление значения времени, прошедшего с момента
// запуска секундомера.
void updateTime() {
    if(!running) return;

    long temp = Calendar.getInstance().getTimeInMillis();
    jlab.setText("Elapsed time is " +
                (double) (temp - start)/1000);
}
```

В теле метода сначала проверяется переменная `running`. Если ее значение равно `false`, выполнение метода прекращается, поскольку данное значение свидетельствует о том, что секундомер не используется. Если значение `running` равно `true`, в методе `updateTime()` определяется текущее время, из него вычитается время запуска секундомера, после чего полученное значение отображается с помощью метки `jlab`. Поскольку метод `updateTime()` изменяет текст в составе метки, он должен выполняться в потоке обработки событий. Для того чтобы решить эту задачу, используется метод `invokeLater()`; в качестве параметра ему передается объект `Runnable`, в методе `run()` которого присутствует вызов `updateTime()`. В результате метод `updateTime()` выполняется в потоке обработки событий, поэтому с его помощью можно обновлять значение времени в метке `jlab`.

После объявления объекта `myThread` формируется новый поток, использующий данный объект. Он запускается с помощью приведенного ниже фрагмента кода.

```
// Создание нового потока.
thrd = new Thread(myThread);
```

```
// Запуск потока.
thrd.start();
```

На данном этапе новый поток сформирован и выполняется. Однако реально в нем не предпринимаются никакие действия, так как секундомер еще не был запущен.

По щелчку на кнопке **Start** определяется текущее системное время и записывается в переменную `start`. Доступ к кнопке **Start** запрещается, вместо нее становится доступной кнопка **Stop**, и значение переменной `running` устанавливается равным `true`. По щелчку на кнопке **Stop** также определяется текущее системное время. Разность между текущим временем и временем запуска секундомера отображается посредством метки `jlab`. Затем состояние кнопок изменяется на обратное (доступная кнопка становится недоступной и наоборот), а в переменную `running` записывается значение `false`.

Как видно из данной программы, любой код, осуществляющий взаимодействие с компонентами **Swing**, должен выполняться в потоке обработки событий. Если у вас есть сомнение в том, в каком потоке выполняется тот или иной фрагмент кода, вы можете получить необходимую информацию, вызвав метод `isEventDispatchThread()`, определенный в классе `SwingUtilities`.

**ВАЖНО!**

## 10.2 Использование класса `Timer`

В предыдущем примере было продемонстрировано использование метода `invokeLater()` для организации взаимодействия с компонентами **Swing**. Как видите, код программы достаточно сложен, поэтому желательно упростить его. В некоторых случаях можно обойтись без явного создания отдельного потока, а использовать таймер, генерирующий событие по истечении определенного интервала времени. Например, таймер можно использовать для периодической перерисовки изображения с целью создания анимационного эффекта (см. проект 10.1).

Для поддержки таймера в **Swing** предусмотрен класс `Timer`, расположенный в пакете `javax.swing`. Следует отличать данный класс от класса `Timer`, расположенного в пакете `java.util`. Если в программе импортируются оба пакета, следует явным образом указывать, какой именно таймер необходим вам.

Класс `Timer` из библиотеки **Swing** автоматически генерирует событие действия, обслуживаемое специальным обработчиком. Обработчик события автоматически выполняется в нужном потоке. Следовательно, метод `actionPerformed()` объекта `ActionListener`, зарегистрированного для работы с таймером, будет выполняться в потоке обработки событий. Поэтому необходимость в использовании методов `invokeLater()` и `invokeAndWait()` не возникает. Таким образом, для работы с таймером вам достаточно создать

экземпляр класса `Timer` и определить обработчик, который будет реагировать на возникающие события.

В классе `Timer` определен только один конструктор, который имеет следующий вид:

```
Timer(int period, ActionListener al)
```

где параметр `period` определяет интервал времени между возникновением событий. Обработчик событий задается с помощью параметра `al`. Можно также зарегистрировать и другие обработчики, которые будут оповещаться о событиях таймера. Для этого надо вызвать метод `addActionListener()` класса таймера.

Для запуска таймера следует вызвать метод `start()`. Для остановки таймера служит метод `stop()`. Заголовки этих методов приведены ниже.

```
void start()
```

```
void stop()
```

По умолчанию таймер периодически генерирует события. При необходимости можно перевести таймер в режим, при котором он будет генерировать событие лишь один раз. Для этой цели служит метод `setRepeats()`.

```
void setRepeats(boolean repeats)
```

Если значение параметра `repeats` равно `true`, таймер периодически генерирует события. Значение `false` приводит к тому, что таймер сработает только один раз.

Как было сказано ранее, программа, реализующая секундомер, рассмотренная в предыдущем разделе, неоправданно сложная. Вместо того чтобы явно создавать отдельный поток, можно использовать таймер. Такой подход использован в очередном варианте программы.

```
// Версия программы-секундомера, использующая класс Timer.
```

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

```
class TimerStopWatch {
```

```
    JLabel jlab;      // Отображение времени, прошедшего
                      // с момента запуска секундомера.
```

```
    JButton jbtnStart; // Запуск секундомера.
```

```

JButton jbtnStop;    // Остановка секундомера.

long start;    // Время запуска секундомера,
               // выраженное в миллисекундах.

// Для обновления значения времени используется
// класс Timer.
Timer swTimer;

TimerStopWatch() {

    // Создание нового контейнера JFrame.
    JFrame jfrm = new JFrame("Timer-based Stopwatch");

    // Установка диспетчера компоновки FlowLayout.
    jfrm.getContentPane().setLayout(new FlowLayout());

    // Установка начальных размеров фрейма.
    jfrm.setSize(230, 90);

    // Завершение программы при закрытии окна пользователем.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Создание метки, отображающей время, прошедшее
    // с момента запуска секундомера.
    jlab = new JLabel("Press Start to begin timing.");

    // Создание меток Start и Stop.
    jbtnStart = new JButton("Start");
    jbtnStop = new JButton("Stop");
    jbtnStop.setEnabled(false);

    // Обработчик, получающий уведомление о событии,
    // сгенерированном таймером.
    ActionListener timerAL = new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            updateTime();
        }
    };

    // Создание таймера, генерирующего события
    // десять раз в секунду. Обработка событий осуществляется
    // обработчиком, указанным с помощью параметра timerAL.
```

```
swTimer = new Timer(100, timerAL);

// Связывание обработчиков событий с кнопками
// Start и stop.
jbtnStart.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {

        // Запись времени запуска таймера.
        start = Calendar.getInstance().getTimeInMillis();

        // Изменение состояния кнопок на обратное.
        jbtnStop.setEnabled(true);
        jbtnStart.setEnabled(false);

        // Запуск секундомера.
        swTimer.start();
    }
});

jbtnStop.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        long stop = Calendar.getInstance().getTimeInMillis();

        // Вычисление времени, прошедшего с момента запуска.
        jlab.setText("Elapsed time is "
            + (double) (stop - start)/1000);

        // Изменение состояния кнопок на обратное.
        jbtnStart.setEnabled(true);
        jbtnStop.setEnabled(false);

        // Остановка секундомера.
        swTimer.stop();
    }
});

// Включение кнопок и метки в состав панели содержимого.
jfrm.getContentPane().add(jbtnStart);
jfrm.getContentPane().add(jbtnStop);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
```



```

    }

    // Обновление отображаемого значения времени.
    // Обратите внимание на то, что переменная running
    // больше не нужна.
    void updateTime() {
        long temp = Calendar.getInstance().getTimeInMillis();
        jlab.setText("Elapsed time is " +
                     (double) (temp - start)/1000);
    }

    public static void main(String args[]) {
        // Фрейм создается в потоке обработки событий.
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new TimerStopWatch();
            }
        });
    }
}

```

Нетрудно заметить, что данная программа существенно короче предыдущей, поскольку в ней отсутствует код, предназначенный для создания потока. В ней даже не нужна переменная `running`. Использование класса `Timer` существенно упрощает решение задачи. Обратите внимание на то, насколько мал объем кода, используемого для организации работы с таймером. Сначала создается объект таймера. Для этого достаточно одной строки.

```
swTimer = new Timer(100, timerAL);
```

В результате выполнения данного выражения формируется таймер, который генерирует событие десять раз в секунду. Параметр `timerAL` определяет обработчик событий. Код обработчика показан ниже.

```

ActionListener timerAL = new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        updateTime();
    }
};

```

Действия метода `actionPerformed()` ограничиваются вызовом `updateTime()`. Поскольку данный код уже выполняется в потоке обработки событий, необходимости в использовании метода `invokeLater()` нет.

Таймер запускается по щелчку на кнопке **Start**. Для его остановки следует щелкнуть на кнопке **Stop**. Если отдельный поток нужен вам лишь для того, чтобы обновлять элементы пользовательского интерфейса, лучше всего использовать класс `javax.swing.Timer`. Если же отдельный поток необходим для других целей, вам придется создать его вручную.



## Вопросы для текущего контроля

1. Почему взаимодействие с компонентами следует осуществлять только из потока обработки событий?
2. Какие действия выполняет метод `invokeLater()`?
3. Какой метод класса `Timer` используется для запуска таймера?

## Создание апплетов на базе Swing

Предыдущие примеры из данной книги представляли собой независимые Java-приложения. Однако приложения — это не единственный возможный тип Java-программ. Существуют также *апплеты* и *сервлеты*. Апплеты — это небольшие программы, копируемые из Интернета и выполняющиеся в среде браузера. Сервлеты — это программы, выполняющиеся на стороне сервера. В апплетах часто применяются средства Swing. Вопросам их использования посвящен данный раздел. Несмотря на то что апплеты на базе Swing очень похожи на апплеты, созданные на основе AWT, у них есть ряд особенностей.

Рассмотрение общих вопросов создания апплетов выходит за рамки рассмотрения данной книги. Более подробное описание апплетов вы найдете в моей книге *Java: The Complete Reference*, выпущенной издательством McGraw-Hill в 2005 г. (русский перевод *Полный справочник по Java*, Издательский дом «Вильямс»). Мы лишь вкратце обсудим архитектуру апплетов и сосредоточимся на вопросах применения компонентов Swing.

1. В этом случае исключается конфликт между потоками, состоящий в том, что два потока пытаются одновременно обновить один и тот же компонент.
2. Он организует выполнение объекта `Runnable` в потоке обработки событий и немедленно возвращает управление, не дожидаясь завершения работы потока.
3. `start()`.

**ВАЖНО!****10.3. Общие сведения об апплетах**

Каждый апплет создается на базе класса `Applet`. Однако, для того, чтобы создать апплет, использующий компоненты `Swing`, вам надо выбрать в качестве базового класса `JApplet`, являющийся подклассом класса `Applet`. Класс `JApplet` представляет собой контейнер, который не является подклассом `JComponent`. Поскольку `JApplet` — это `Swing`-контейнер верхнего уровня, в нем присутствуют панели, рассмотренные в модуле 1. Это означает, что все компоненты должны добавляться к панели содержимого так же, как это делается при работе с компонентом `JFrame`. Взаимодействие с компонентами `Swing` также должно осуществляться в потоке обработки событий, как это было описано в предыдущем разделе.

Апплеты, в отличие от приложений, не выполняются непосредственно в операционной системе. Для них нужно специальное окружение, создаваемое либо браузерами, либо специальными программами просмотра апплетов. Подобная программа, `appletviewer`, имеется в составе `JDK`. В процессе разработки апплета удобнее тестировать его с помощью программы просмотра, чем загружать в браузер. Однако окончательный тест должен быть обязательно выполнен с помощью браузера.

Апплеты разрабатываются как составная часть `HTML`-документа. В настоящее время компания `Sun` рекомендует включать апплеты в состав документа с помощью дескриптора `<APPLET>`. Именно этот подход будет применяться в данной книге.

**ВАЖНО!****10.4. Основные элементы апплетов**

Все апплеты, независимо от того, используют ли они компоненты `Swing`, имеют одинаковую структуру и одинаковый жизненный цикл. Для реализации базового механизма, посредством которого браузер или специализированная программа просмотра взаимодействует с апплетами и контролирует их выполнение, при создании апплетов переопределяется ряд методов: `init()`, `start()`, `stop()` и `destroy()`. Они определены в классе `Applet` (и унаследованы от него классом `JApplet`) и влияют на жизненный цикл апплета. Следовательно, формально методы можно не переопределять. Однако апплет, использующий реализацию основных методов по умолчанию, не будет выполнять никаких действий. Перечисленные выше методы можно объединить в базовый апплет следующим образом:

```
// Элементарный апплет, использующий средства Swing.

import javax.swing.*;

/*
HTML-код, используемый для загрузки данного апплета,
имеет следующий вид:

<applet code="AppletSkel" width=300 height=100>
</applet>

*/

public class AppletSkel extends JApplet {

    // Четыре метода, определяющих жизненный цикл апплета.

    // Данный метод вызывается в первую очередь.
    public void init() {
        // Инициализация апплета и формирование
        // графического пользовательского интерфейса.
    }

    // Данный метод вызывается после init(). Также обращение
    // к нему производится при повторном запуске апплета.
    public void start() {
        // Запуск или возобновление работы апплета.
    }

    // Данный метод вызывается при остановке апплета.
    public void stop() {
        // Остановка апплета.
    }

    // Данный метод вызывается при завершении работы апплета.
    // Это последний выполняемый метод.
    public void destroy() {
        // Perform shutdown activities.
    }
}
```

Несмотря на то что представленный выше базовый апплет не выполняет никаких действий, его можно скомпилировать и запустить. Компилируется

апплет так же, как и любая другая Java-программа. Ниже представлена командная строка, используемая для компиляции `AppletSkel`.

```
javac AppletSkel.java
```

Однако, чтобы запустить апплет, вам надо создать HTML-файл и предусмотреть в нем дескриптор `<APPLET>`. Этот дескриптор оформлен в виде комментариев и включен в начало приведенного выше фрагмента. Если вы поместите данный код в файл `AppletSkel.html`, вы сможете выполнить апплет, загрузив его либо в браузер, либо в программу `appletviewer`.

```
appletviewer AppletSkel.html
```

При выполнении в среде `appletviewer` формируется окно, показанное на рис. 10.2.

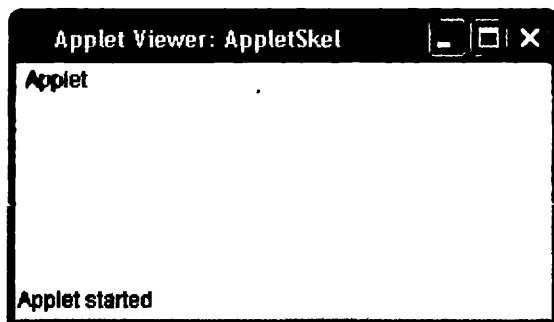


*Рис. 10.2. Окно апплета, отображаемое с помощью программы `appletviewer`*

Перед тем как обсуждать вопросы создания апплета, желательно подробнее рассмотреть методы, определяющие его жизненный цикл. При запуске апплета выполняются два приведенных ниже метода (в том же порядке, в котором они перечислены здесь).

1. `init()`
2. `start()`

При завершении апплета вызываются следующие методы:

1. `stop()`
2. `destroy()`

Рассмотрим их более подробно.

Метод `init()` вызывается первым. В него включается код, инициализирующий апплет. Этот метод вызывается лишь один раз. В методе `init()` следует выполнить действия по созданию пользовательского интерфейса, но нельзя не-

посредственно работать с компонентами. Вместо этого надо использовать метод `invokeAndWait()`, который будет описан в следующем разделе.

Метод `start()` вызывается после `init()`. Кроме того, к нему производится обращение для перезапуска апплета после того, как он был остановлен. В отличие от метода `init()` метод `start()` может быть вызван несколько раз — каждый раз, когда на экране восстанавливается изображение HTML-документа. Таким образом, если пользователь переходит к другому приложению, а затем возвращается к HTML-странице с апплетом, выполнение апплета возобновляется, при этом вызывается метод `start()`.

### Спросим у опытного программиста

**Вопрос.** В приложениях, приведенных в данной книге, для создания графического интерфейса используется метод `invokeLater()`. Почему в апплетах применяется метод `invokeAndWait()`?

**Ответ.** В апплетах следует использовать метод `invokeAndWait()`, потому что метод `init()` не должен возвращать управление до окончания процесса инициализации. В частности, метод `start()` не может быть вызван раньше, чем будет полностью сформирован пользовательский интерфейс.

Метод `stop()` вызывается перед тем, как браузер прекратит работу с HTML-документом, содержащим апплет, например, при переходе к другой странице. Предполагается, что в момент вызова метода `stop()` апплет выполняется. В методе `stop()` имеет смысл приостановить потоки, которые не нужны тогда, когда апплет невидим. Возобновить выполнение этих потоков можно в методе `start()`, который будет вызван, когда пользователь вернется к странице, отображаемой в браузере.

Метод `destroy()` вызывается тогда, когда апплет должен быть полностью удален из памяти. В этом случае следует освободить все ресурсы, используемые апплетом. Перед методом `destroy()` всегда вызывается метод `stop()`.

**ВАЖНО!**

## 10.5 Формирование графического пользовательского интерфейса для апплета

Несмотря на то что формально предыдущий апплет составлен правильно, он не выполняет никаких действий. В нем также отсутствуют компоненты. Как и Swing-приложения, апплеты должны взаимодействовать с компонентами пос-

редством потока обработки событий. Это означает, что сам метод `init()` нельзя использовать для создания графического пользовательского интерфейса. Вместо этого в теле `init()` надо вызвать метод `invokeAndWait()` и передать ему объект `Runnable`, метод `run()` которого будет выполняться в потоке обработки событий. С учетом данных требований базовая форма метода `init()` должна выглядеть так, как показано ниже.

```
// Данный метод вызывается в первую очередь.
public void init() {
    try {
        SwingUtilities.invokeAndWait(new Runnable () {
            // В апплете, создаваемом на базе Swing, графический
            // пользовательский интерфейс должен создаваться в
            // потоке обработки событий.
            public void run() {
                guiInit(); // Метод, инициализирующий компоненты Swing.
            }
        });
    } catch (Exception exc) {
        System.out.println("Can't create because of " + exc);
    }
}
```

В теле метода `run()` вызывается метод `guiInit()`, который используется для создания и инициализации компонентов `Swing`. Очевидно, что для этого метода можно выбрать любое другое имя.

## Простой апплет, использующий средства `Swing`

Ниже приведен код простого апплета, созданный с учетом изложенного ранее материала. В окне апплета отображаются две кнопки. После каждого щелчка на кнопке выводится сообщение о том, какая из кнопок была активизирована. Окно апплета, выполняемого с помощью программы `appletviewer`, показано на рис. 10.3.

```
// A simple Swing-based applet
```

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

```
/*
```

HTML-код, используемый для загрузки данного апплета, имеет следующий вид:

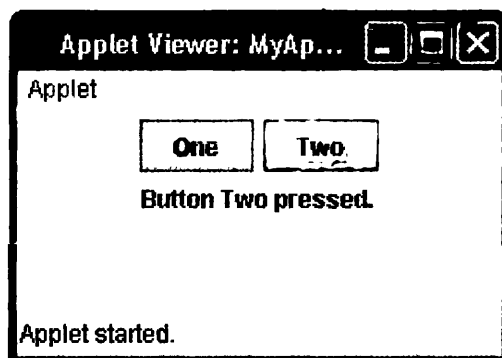


Рис. 10.3. Окно, отображаемое при работе  
апплета MyApplet

```
<object code="MyApplet" width=240 height=100>  
</object>
```

```
*/
```

```
public class MyApplet extends JApplet {  
    JButton jbtnOne;  
    JButton jbtnTwo;  
  
    JLabel jlab;  
  
    // Данный метод вызывается в первую очередь.  
    public void init() {  
        try {  
            SwingUtilities.invokeLater(new Runnable () {  
                public void run() {  
                    guiInit(); // Инициализация графического  
                               // пользовательского интерфейса.  
                }  
            });  
        } catch (Exception exc) {  
            System.out.println("Can't create because of "+ exc);  
        }  
    }  
  
    // Данный метод вызывается после init(). Также обращение  
    // к нему производится при повторном запуске апплета.  
    public void start() {
```



```
// В данном апплете этот метод не используется.
}

// Данный метод вызывается при остановке апплета.
public void stop() {
    // В данном апплете этот метод не используется.
}

// Данный метод вызывается при завершении работы апплета.
// Это последний выполняемый метод.
public void destroy() {
    // В данном апплете этот метод не используется.
}

// Формирование и инициализация пользовательского интерфейса.
private void guiInit() {
    // Установка диспетчера компоновки FlowLayout.
    setLayout(new FlowLayout());

    // Создание двух кнопок и метки.
    jbtnOne = new JButton("One");
    jbtnTwo = new JButton("Two");

    jlab = new JLabel("Press a button.");

    // Связывание с кнопками обработчиков событий.
    jbtnOne.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent le) {
            jlab.setText("Button One pressed.");
        }
    });

    jbtnTwo.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent le) {
            jlab.setText("Button Two pressed.");
        }
    });

    // Включение компонентов в состав панели содержимого апплета.
    getContentPane().add(jbtnOne);
    getContentPane().add(jbtnTwo);
    getContentPane().add(jlab);
}
}
```

Рассматривая данный апплет, необходимо обратить внимание на две его важные особенности. Во-первых, метод `init()` инициализирует графический интерфейс на базе Swing, вызывая метод `guiInit()`. Вызов данного метода организуется посредством метода `invokeAndWait()`. В теле `guiInit()` создаются две кнопки и метка. С каждой из кнопок связывается обработчик событий. Действие каждого обработчика сводится к установке текста метки, указывающей на то, какая из кнопок была активизирована. Несмотря на простоту примера, тот же самый подход должен быть использован при создании графического интерфейса любого апплета на базе Swing. Во-вторых, в данном апплете не используются методы `start()`, `stop()` и `destroy()`. Для простых апплетов такое решение можно считать типичным. Определять пустые методы нет необходимости, поскольку их реализации предоставляет класс `Applet`. Таким образом, если какой-то из методов, влияющих на жизненный цикл апплета, вам не нужен, переопределять его не обязательно.



## Вопросы для текущего контроля

1. Какой класс используется для создания апплетов на базе Swing?
2. Назовите четыре метода, определяющих жизненный цикл апплета.
3. Какой метод надо вызвать, чтобы в начале работы апплета создать графический пользовательский интерфейс?

## Проект 10.1. Перемещаемый текст в апплете

`ScrollText.java`

Данный проект демонстрирует создание апплетов на базе Swing и использование таймера. Таймер применяется для перемещения текстового сообщения, отображаемого с помощью компонента `JLabel`. Как вы увидите, объем кода, решающего данную задачу, на удивление мал. Этого позволяет добиться класс `Timer`, принадлежащий библиотеке Swing. Окно апплета показано ниже.

1. `JApplet`.
2. `init()`, `start()`, `stop()` и `destroy()`.
3. `invokeAndWait()`.

## Последовательность действий

1. Создайте файл `ScrollText.java` и включите в него следующие комментарии и выражения `import`:

```
// Проект 10.1: Апплет на базе Swing, в котором
// перемещается текст метки.

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

/*
HTML-код, используемый для загрузки данного апплета,
имеет следующий вид:

<object code="ScrollText" width=240 height=100>
</object>

*/
```

2. Создайте класс `ScrollText`, начав его следующими строками кода:

```
public class ScrollText extends JApplet {

    JLabel jlab;

    String msg = " Swing makes the GUI move! ";

    ActionListener scroller;

    Timer stTimer; // Таймер, определяющий скорость перемещения.
```

Поскольку `ScrollText` — это апплет на базе `Swing`, в качестве суперкласса для него выбирается `JApplet`. В классе `ScrollText` объявляются метка, используемая для отображения сообщения, текст сообщения, обработчик событий, выполняющий перемещение, и таймер, определяющий скорость перемещения текста.

3. Добавьте метод `init()`, код которого приведен ниже.

```
// Инициализация апплета.
public void init() {
    try {
        SwingUtilities.invokeAndWait(new Runnable () {
            public void run() {
```

```
        guiInit();
    }
});
} catch (Exception exc) {
    System.out.println("Can't create because of "+ exc);
}
}
```

Действия данного метода ограничиваются вызовом `guiInit()` в потоке обработки событий.

4. Добавьте код методов `start()`, `stop()` и `destroy()`, приведенный ниже.

```
// Активизация таймера при запуске апплета.
public void start() {
    stTimer.start();
}
```

```
// Остановка таймера при остановке апплета.
public void stop() {
    stTimer.stop();
}
```

```
// Остановка таймера при удалении апплета.
public void destroy() {
    stTimer.stop();
}
```

Несмотря на то что в теле каждого метода содержится лишь по одной команде, они выполняют важные функции. Каждый раз, когда апплет отображается на Web-странице, запускается таймер. При остановке апплета останавливается также и таймер. Кроме того, команда остановки таймера также выполняется и при удалении апплета.

5. Завершите написание апплета кодом метода `guiInit()`, приведенным ниже.

```
// Инициализация интерфейса.
private void guiInit() {

    // Создание метки для перемещения сообщения.
    jlab = new JLabel(msg);
    jlab.setHorizontalAlignment(SwingConstants.CENTER);

    // Создание обработчика событий для таймера.
    scroller = new ActionListener() {
```

```
public void actionPerformed(ActionEvent ae) {
    // Перемещение сообщения влево на один символ.
    char ch = msg.charAt(0);
    msg = msg.substring(1, msg.length());
    msg += ch;
    jlab.setText(msg);
}

};

// Создание таймера.
stTimer = new Timer(200, scroller);

// Включение метки в состав панели содержимого апплета.
getContentPane().add(jlab);
}
}
```

В теле данного метода создается метка, содержащая строку текста. Текст выравнивается по центру компонента. Это делается для того, чтобы улучшить внешний вид перемещаемого текста. При желании вы можете использовать выравнивание по умолчанию. Далее создается обработчик событий действия, который, получая управление, каждый раз перемещает текст на один символ влево. Ссылка на обработчик событий передается конструктору класса `Timer`. Для таймера устанавливается временной интервал, равный одной пятой секунды, но при желании вы можете изменить его. Очевидно, что, уменьшая период, вы увеличите скорость перемещения текста. Созданная метка включается в состав панели содержимого.

**6. Полностью код программы `ScrollText` приведен ниже.**

```
// Проект 10.1: Апплет на базе Swing, в котором
// перемещается текст метки.

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

/*
HTML-код, используемый для загрузки данного апплета,
имеет следующий вид:

<object code="ScrollText" width=240 height=100>
</object>
```

```
*/  
  
public class ScrollText extends JApplet {  
  
    JLabel jlab;  
  
    String msg = " Swing makes the GUI move! ";  
  
    ActionListener scroller;  
  
    Timer stTimer; // Таймер, определяющий скорость перемещения.  
  
    // Инициализация апплета.  
    public void init() {  
        try {  
            SwingUtilities.invokeLater(new Runnable () {  
                public void run() {  
                    guiInit();  
                }  
            });  
        } catch (Exception exc) {  
            System.out.println("Can't create because of "+ exc);  
        }  
    }  
  
    // Активизация таймера при запуске апплета.  
    public void start() {  
        stTimer.start();  
    }  
  
    // Остановка таймера при остановке апплета.  
    public void stop() {  
        stTimer.stop();  
    }  
  
    // Остановка таймера при удалении апплета.  
    public void destroy() {  
        stTimer.stop();  
    }  
  
    // Инициализация интерфейса.  
    private void guiInit() {
```

```
// Создание метки для перемещения сообщения.
jlab = new JLabel(msg);
jlab.setHorizontalAlignment(SwingConstants.CENTER);

// Создание обработчика событий для таймера.
scroller = new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        // Перемещение сообщения влево на один символ.
        char ch = msg.charAt(0);
        msg = msg.substring(1, msg.length());
        msg += ch;
        jlab.setText(msg);
    }
};

// Создание таймера.
stTimer = new Timer(200, scroller);

// Включение метки в состав панели содержимого апплета.
getContentPane().add(jlab);
}
```

## Рисование

Основная цель данной книги — показать компоненты Swing и особенности их использования. Тем не менее Swing позволяет создавать собственные визуальные объекты во фрейме, в панели или другом компоненте, например JLabel. Несмотря на то что многие задачи, решаемые средствами Swing (и даже большинство из них), не предполагают непосредственного рисования в составе компонентов, соответствующие инструменты доступны для разработчиков. Для того чтобы вывести информацию непосредственно на поверхности компонента, надо использовать специальные методы, например drawLine() или drawRect(). При этом необходимо осуществлять хотя бы минимальное управление процессом рисования. Детальное обсуждение подобных вопросов выходит за рамки данной книги. В этом разделе мы рассмотрим лишь базовые технологии, которые могут быть использованы для рисования.

**ВАЖНО!****10.6.**

## **Общие сведения о рисовании**

Планируя реализовать в приложении рисование средствами Swing, необходимо учитывать, что процесс этот достаточно сложен. В отличие от AWT, в библиотеке Swing отсутствует выделенная подсистема рисования. Перед тем как начать обсуждение деталей Swing-рисования, необходимо рассмотреть некоторые базовые механизмы.

Как вы помните, класс `JComponent` является подклассом AWT-класса `Component`. В этом классе определен метод `paint()`. Он вызывается тогда, когда возникает необходимость в отображении компонента на экране. За исключением отдельных, крайне редких случаев, метод `paint()` не вызывается непосредственно из программы. Обращение к этому методу осуществляет исполняющая система, занимающаяся воспроизведением компонентов. Такое обращение может происходить по нескольким причинам. Например, окно, в котором имеется компонент, может быть перекрыто другим окном, а затем снова переведено на передний план. Окно также может быть минимизировано, а затем восстановлены его обычные размеры. Метод `paint()` также вызывается в начале выполнения программы. При написании кода приложения на базе AWT, в котором необходимо выводить данные непосредственно в области отображения компонента, следует переопределить метод `paint()`.

Несмотря на то что легковесные компоненты Swing наследуют метод `paint()` от класса `Component`, для рисования в составе компонента не следует переопределять этот метод. В Swing используется другой подход, предполагающий использование трех методов: `paintComponent()`, `paintBorder()` и `paintChildren()`. Эти методы разделяют процесс рисования на три отдельных действия. В легковесном компоненте AWT-метод `paint()` лишь вызывает вышеперечисленные методы в нужном порядке.

Для рисования на поверхности компонента Swing следует создать подкласс компонента и переопределить метод `paintComponent()`. Этот метод осуществляет рисование внутри компонента. Остальные два метода в обычных условиях не переопределяются. Первое, что надо сделать, переопределяя метод `paintComponent()`, — это вызвать метод суперкласса `super.paintComponent()`, обеспечив тем самым выполнение стандартных действий. (Метод суперкласса можно не вызывать лишь в том случае, если вы собираетесь полностью контролировать процесс отображения компонента.) После вызова `super.paintComponent()` можно выполнять действия, необходимые



для отображения данных. Заголовок метода `paintComponent()` имеет следующий вид:

```
protected void paintComponent(Graphics g)
```

где параметр `g` — это графический контекст, посредством которого осуществляется вывод.

**ВАЖНО!**

## 10.7. Графический контекст

В языке Java с каждым компонентом связывается *графический контекст*. Контекст инкапсулирован в классе `Graphics` и хранит разнообразную информацию о среде отображения, например, цвет, используемый для рисования, и шрифт. Как было сказано выше, графический контекст передается различным методам рисования, в том числе `paintComponent()`. Рисование компонента осуществляется посредством методов, предоставляемых контекстом.

В классе `Graphics` предусмотрено много методов для вывода данных. Три из них приведены ниже.

```
void drawLine(int startX, int startY, int endX, int endY)
```

```
void drawRect(int left, int top, int width, int height)
```

```
void drawString(String str, int left, int top)
```

Метод `drawLine()` выводит линию текущим цветом. Линия начинается в точке с координатами `startX` и `startY` и оканчивается в точке `endX, endY`. Метод `drawRect()` выводит текущим цветом прямоугольник. При вызове данного метода указываются координаты верхнего левого угла прямоугольника, а также его высота и ширина. Метод `drawString()` выводит строку текста, заданную посредством параметра `str`. Базовая линия строки начинается в точке с координатами `left` и `top`.

Начало координат области отображения (точка 0,0) располагается в верхнем левом углу компонента. Расположение любого элемента отсчитывается от начала координат. Координаты задаются в пикселях. Таким образом, если вы хотите вывести точку в позиции 12,14, она будет сдвинута на 12 пикселей вправо и на 14 пикселей вниз относительно левого верхнего угла компонента.

**ВАЖНО!**

## 10.8. Определение области отображения

При рисовании на поверхности компонента необходимо ограничить вывод областью, находящейся внутри границ компонента. Несмотря на то что в Swing автоматически выполняется отсечение, точки, выходящие за пределы

области отображения, могут появиться на обрамлении, а затем исчезнуть после обновления границ. Для того чтобы избежать возникновения такого эффекта, необходимо вычислять размеры области отображения компонента и принимать меры для того, чтобы изображение не выходило за границы этой области. Размер области отображения равен текущему размеру компонента минус размер обрамления. Таким образом, перед тем, как начать рисование в составе компонента, вы должны определить толщину рамки и в дальнейшем учитывать ее при выводе.

Для определения толщины рамки надо вызвать метод `getInsets()`.

```
Insets getInsets()
```

Данный метод впервые определен в классе `Container` и переопределен в классе `JContainer`. Он возвращает объект `Insets`, содержащий размеры обрамления. Необходимая информация содержится в следующих полях:

```
int top;
```

```
int bottom;
```

```
int left;
```

```
int right;
```

Данные значения затем используются для вычисления размеров области отображения. Определить ширину и высоту компонента позволяют его методы `getWidth()` и `getHeight()`. Эти методы приведены ниже.

```
int getWidth()
```

```
int getHeight()
```

Вычисляя из полученных значений размеры обрамления, можно определить ширину и высоту области отображения компонента.

**ВАЖНО!**

## 10.9 Запрос на рисование

Как было сказано выше, рисование осуществляется методом `paint()`, обращение к которому приводит к вызову методов `paintComponent()`, `paintBorder()` и `paintChildren()`. В связи с этим возникает вопрос: как указать исполняющей системе на то, что компонент следует обновить? Предположим, например, что в программе данные, отображаемые в виде гистограммы, изменились. Каким образом инициировать перерисовку графика? Для этой цели следует использовать метод `repaint()`.

Данный метод определен в классе `Component`. Он сообщает системе о том, что метод `paint()` должен быть вызван при первой же возможности. Посколь-

ку операция рисования занимает достаточно долгое время, такой механизм позволяет отложить рисование на короткое время и завершить высокоприоритетную операцию. Как вы уже знаете, в Swing обращение к `paint()` приводит к вызову метода `paintComponent()`. Таким образом, чтобы отобразить данные на поверхности компонента, программа должна хранить их до вызова метода `paintComponent()`. В переопределенном методе `paintComponent()` должны быть предусмотрены действия по отображению хранящейся информации. Существует несколько вариантов метода `repaint()`, позволяющих контролировать процесс перерисовки. Однако в данном модуле будет использована лишь простейшая версия данного метода, приведенная ниже.

```
void repaint()
```

Этот метод вызывает перерисовку всего компонента.

## Пример программы, осуществляющей рисование

Ниже приведен код программы, в которой использованы описанные выше средства. В программе определен класс `PaintPanel`, являющийся подклассом `JPanel`. Экземпляр этого класса используется для отображения гистограммы, представляющей данные, сгенерированные по случайному закону. Помимо графика в окне содержатся две кнопки. Одна из них предназначена для смены отображаемого набора данных, вторая позволяет изменять размер рамки вокруг графика. Окно, создаваемое при работе программы, показано на рис. 10.4.

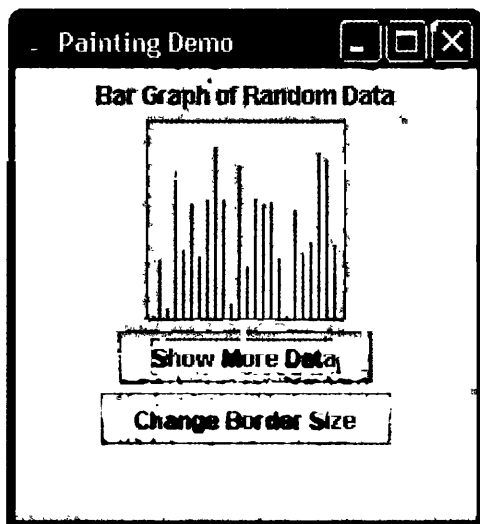


Рис. 10.4. Данные, отображаемые программой `PaintPanel`

## Спросим у опытного программиста

**Вопрос.** Известно, что графический контекст передается в качестве параметра таким методам, как `paintComponent()`. Существуют ли другие способы получения графического контекста?

**Ответ.** Да, графический контекст возвращает метод `getGraphics()`, определенный в классе `Component`. Как было сказано ранее, в большинстве случаев для вывода графических данных применяется метод `paintComponent()`, а другие средства не используются. Однако в некоторых, крайне редких случаях целесообразно получить контекст путем вызова метода `getGraphics()`, а затем использовать его для вывода данных на поверхности компонента.

// Рисование графика.

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

```
// Данный класс является подклассом JPanel. В нем
// переопределен метод paintComponent(), с помощью
// которого в составе панели отображаются данные,
// сгенерированные случайным образом.
```

```
class PaintPanel extends JPanel {
    Insets ins; // Размеры обрамления панели.
```

```
    Random rand; // Используется для генерации
                // псевдослучайных значений.
```

```
    PaintPanel(int w, int h) {
```

```
        // Панель должны быть непрозрачной.
        setOpaque(true);
```

```
        // Использование рамки в виде линии красного цвета.
        setBorder(
            BorderFactory.createLineBorder(Color.RED, 1));
```

```
// Установка предпочтительных размеров.
setPreferredSize(new Dimension(w, h));

rand = new Random();
}

// Переопределение метода paintComponent() для
// отображения данных на поверхности компонента.
protected void paintComponent(Graphics g) {
    // Тело метода начинается с вызова метода суперкласса.
    super.paintComponent(g);

    // Определение высоты и ширины компонента.
    int height = getHeight();
    int width = getWidth();

    // Получение размеров обрамления.
    ins = getInsets();

    // В составе панели выводятся псевдослучайные значения,
    // представленные в виде гистограммы.
    for(int i=ins.left+5; i <= width-ins.right-5; i += 4) {
        // Получение псевдослучайного значения в интервале от 0
        // до максимальной высоты области отображения.
        int h = rand.nextInt(height-ins.bottom);

        // Коррекция значения, располагающегося слишком
        // близко к рамке.
        if(h <= ins.top) h = ins.top+1;

        // Вывод линии, представляющей значение.
        g.drawLine(i, height-ins.bottom, i, h);
    }
}

// Изменение размера рамки.
public void changeBorderSize(int size) {
    setBorder(
        BorderFactory.createLineBorder(Color.RED, size));
}

// Демонстрация рисования на поверхности панели.
```

```
class PaintDemo {

    JButton jbtnMore;
    JButton jbtnSize;
    JLabel jlab;
    PaintPanel pp;

    boolean big; // Используется для изменения размеров панели.

    PaintDemo() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Painting Demo");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(240, 260);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание панели для вывода данных.
        pp = new PaintPanel(100, 100);

        // Создание кнопок.
        jbtnMore = new JButton("Show More Data");
        jbtnSize = new JButton("Change Border Size");

        // Описание графика.
        jlab = new JLabel("Bar Graph of Random Data");

        // Перерисовка панели по щелчку на кнопке
        // Show More Data.
        jbtnMore.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                // Запрос на перерисовку панели.
                pp.repaint();
            }
        });

        // Установка размеров обрамления по щелчку
        // на кнопке Change Border Size.
    }
}
```

```

// Изменение размеров обрамления автоматически
// приводит к перерисовке компонента.
jbtnSize.addActionListener(new ActionListener() {
    // Изменение размеров обрамления для графика.
    public void actionPerformed(ActionEvent ae) {
        if(!big) pp.changeBorderSize(5);
        else pp.changeBorderSize(1);
        big = !big;
    }
});

// Включение кнопок, метки и панели
// в состав панели содержимого.
jfrm.getContentPane().add(jlab);
jfrm.getContentPane().add(pp);
jfrm.getContentPane().add(jbtnMore);
jfrm.getContentPane().add(jbtnSize);

big = false;

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new PaintDemo();
        }
    });
}
}

```

Рассмотрим код программы более подробно. Класс `PaintPanel` расширяет класс `JPanel`; в нем определен метод `paintComponent()`. Это позволяет непосредственно отображать данные на поверхности компонента. Конструктору `PaintPanel` передаются два параметра, определяющие предпочтительные размеры (ширину и высоту) панели. В конструкторе также создается рамка толщиной в один пиксель.

В теле переопределенного метода `paintComponent()` прежде всего осуществляется вызов `super.paintComponent()`. Как уже говорилось ранее, это необходимо, чтобы обеспечить корректное отображение компонента. Далее

определяется ширина и высота панели (включая размеры обрамления). Эти значения используются для того, чтобы разместить график в области отображения панели и ограничить высоту его элементов. Размеры области отображения определяются как ширина и высота компонента минус толщина обрамления. Размеры вычисляются, чтобы обеспечить работу с различными объектами `PaintPanel` и рамками.

Метод `changeBorderSize()` изменяет размер обрамления для панели. Поскольку метод `paintComponent()` автоматически отображает данные, заполняющие область отображения текущего размера, при изменении толщины рамки не возникают проблемы.

Класс `PaintDemo` демонстрирует работу `PaintPanel`. Он создает объект `PaintPanel` с размерами  $100 \times 100$  пикселей и помещает ссылку на него в переменную `pp`. В нем также создаются две кнопки: `jbtnMore` и `jbtnSize`. По щелчку на кнопке `jbtnMore` вызывается метод `repaint()` объекта `pp`. Это приводит к вызову метода `paintComponent()` объекта `PaintPanel`. Таким образом, после каждого щелчка на кнопке `jbtnMore` отображается новый набор данных, сгенерированный случайным образом. Когда пользователь щелкает на кнопке `jbtnSize`, толщина рамки изменяется с 1 на 5 и наоборот. Поскольку метод `paintComponent()` динамически вычисляет границы области отображения, можно установить произвольные размеры обрамления.

## Спросим у опытного программиста ■ ■

**Вопрос.** В документации на API для класса `JComponent` упоминается метод `revalidate()`. Какие действия он выполняет?

**Ответ.** Метод `revalidate()` вызывает перекомпоновку компонента. В обычных условиях обращаться к этому методу не приходится, поскольку любые изменения, влияющие на внешний вид компонента, автоматически вызывают перекомпоновку. Однако в некоторых случаях прибегать к его помощи все же приходится. Предположим, например, что вы изменили предпочтительные размеры компонента, причем сделали это уже после того, как компонент был выведен на экран. Изменение предпочтительных размеров не вызывает перекомпоновку, поэтому, если вы хотите, чтобы эффект от выполненных действий проявился немедленно, вам надо вызвать метод `revalidate()`.



Как видно из предыдущего обсуждения, многие классы и методы, используемые для непосредственного рисования в области отображения компонента, предоставляет AWT, а не Swing. Создавая Swing-приложение, в котором значительная часть действий по выводу данных программируется вручную, вам надо хорошо разбираться в AWT. Эта библиотека является важным элементом Java.



### Вопросы для текущего контроля

1. Какой метод надо переопределить для того, чтобы обеспечить непосредственный вывод данных в область отображения компонента Swing?
2. Какой метод позволяет определить текущую толщину обрамления?
3. Какие действия надо предпринять для того, чтобы метод `paint()` получил управление?

## Диспетчеры компоновки с расширенными возможностями

В предыдущих модулях были рассмотрены три диспетчера компоновки: `BorderLayout`, `FlowLayout` и `GridLayout`. Как было сказано в модуле 1, существуют и другие диспетчеры. Некоторые из них, например `CardLayout` и `SpringLayout`, предоставляют специальные возможности и применяются относительно редко. Однако два модуля, обеспечивающие большую гибкость, нашли широкое распространение. Это диспетчеры `GridBagLayout` и `BoxLayout`. Они рассматриваются в данном разделе.

**ВАЖНО!**

### **10.10** Диспетчер компоновки `GridBagLayout`

Несмотря на то что диспетчеры компоновки `FlowLayout` и `BorderLayout` используются достаточно часто, многие приложения, созданные на базе Swing, требуют большей степени контроля над размещением компонентов в окне.

1. `paintComponent()`.
2. `getInsets()`.
3. Для того чтобы указать на необходимость вызова метода `paint()`, надо вызвать метод `repaint()`.

Подобные задачи позволяет решить диспетчер `GridBagLayout`. Положительная особенность данного диспетчера компоновки состоит в том, что вы можете указывать взаимное расположение компонентов, размещая их в ячейках таблицы. Компоненты могут иметь разные размеры, и в различных строках таблицы можно задать разное количество столбцов. Таким образом, таблица представляет собой объединение таблиц меньшего размера. Класс `GridBagLayout` принадлежит пакету `java.awt`.

Расположение и размер каждого компонента в таблице определяется набором ограничивающих условий. Ограничения задаются посредством объекта `GridBagConstraints`. Ограничения касаются высоты и ширины ячейки, расположения компонента, его выравнивания и “якоря” в ячейке.

Для использования диспетчера `GridBagLayout` надо создать объект данного типа и объявить его в качестве текущего диспетчера компоновки. Затем следует установить ограничения для каждого компонента. И наконец, надо включить компоненты в контейнер, с которым связан данный диспетчер. Несмотря на то что диспетчер `GridBagLayout` несколько сложнее, чем другие диспетчеры компоновки, понимая, как он работает, пользоваться им достаточно просто.

В классе `GridBagLayout` определен только один конструктор; он имеет следующий вид:

```
GridBagLayout ()
```

В классе `GridBagLayout` определено несколько методов. Многие из них объявлены как `protected`. Однако один метод, `setConstraints()`, использовать необходимо. Этот метод устанавливает ограничения, применяемые к компоненту, который помещается в таблицу. Заголовок этого метода определен следующим образом:

```
void setConstraints(Component comp, GridBagConstraints cons)
```

где параметр `comp` определяет компонент, для которого устанавливаются ограничения, заданные с помощью параметра `cons`.

Условием успешного применения диспетчера `GridBagLayout` является правильная установка ограничений, хранящихся в объекте `GridBagConstraints`. В классе `GridBagConstraints` определяется несколько полей, значения которых управляют размером, размещением компонентов и расстоянием между ними. Назначение полей описано в табл. 10.1. Некоторые из них будут подробно рассмотрены далее в этой главе.

В классе `GridBagConstraints` также определено несколько статических полей, содержащих стандартные ограничивающие значения, такие как `GridBagConstraints.CENTER` и `GridBagConstraints.VERTICAL`.

Таблица 10.1. Поля, определенные в классе GridBagConstraints

| Поле                        | Назначение                                                                                                                                                                                                                                                                                                                                                                                                                  |
|-----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>int anchor</code>     | Определяет расположение компонента в ячейке. По умолчанию принимается значение <code>GridBagConstraints.CENTER</code>                                                                                                                                                                                                                                                                                                       |
| <code>int fill</code>       | Определяет порядок изменения размеров компонента в случае, если компонент меньше ячейки. Допустимы следующие значения: <code>GridBagConstraints.NONE</code> (значение по умолчанию), <code>GridBagConstraints.HORIZONTAL</code> , <code>GridBagConstraints.VERTICAL</code> и <code>GridBagConstraints.BOTH</code>                                                                                                           |
| <code>int gridheight</code> | Определяет высоту компонента, представленную как количество ячеек. По умолчанию принимается значение 1                                                                                                                                                                                                                                                                                                                      |
| <code>int gridwidth</code>  | Определяет ширину компонента, представленную как количество ячеек. По умолчанию принимается значение 1                                                                                                                                                                                                                                                                                                                      |
| <code>int gridx</code>      | Задаёт координату X ячейки, в которую должен быть помещен компонент. По умолчанию принимается значение <code>GridBagConstraints.RELATIVE</code>                                                                                                                                                                                                                                                                             |
| <code>int gridy</code>      | Задаёт координату Y ячейки, в которую должен быть помещен компонент. По умолчанию принимается значение <code>GridBagConstraints.RELATIVE</code>                                                                                                                                                                                                                                                                             |
| <code>Insets insets</code>  | Определяет размеры обрамления. По умолчанию принимается значение 0                                                                                                                                                                                                                                                                                                                                                          |
| <code>int ipadx</code>      | Определяет горизонтальный размер дополнительного пространства, отображаемого вокруг компонента. По умолчанию принимается значение 0                                                                                                                                                                                                                                                                                         |
| <code>int ipady</code>      | Определяет вертикальный размер дополнительного пространства, отображаемого вокруг компонента. По умолчанию принимается значение 0                                                                                                                                                                                                                                                                                           |
| <code>double weightx</code> | Задаёт весовое значение, определяющее порядок распределения пространства по горизонтали между соседними ячейками, а также между ячейками и краями контейнера. По умолчанию принимается значение, равное 0.0. Чем больше вес, тем большая часть пространства выделяется для данного элемента. Если все значения равны 0.0, дополнительное пространство выделяется на краях контейнера и распределяется равномерно между ними |
| <code>double weighty</code> | Задаёт весовое значение, определяющее порядок распределения пространства по вертикали между соседними ячейками, а также между ячейками и краями контейнера. По умолчанию принимается значение, равное 0.0. Чем больше вес, тем большая часть пространства выделяется для данного элемента. Если все значения равны 0.0, дополнительное пространство выделяется на краях контейнера и распределяется равномерно между ними   |

Если размер компонента меньше размера ячейки, для определения положения компонента относительно верхнего левого угла ячейки можно использовать поле `anchor`. В переменную `anchor` можно поместить значения двух типов. Первый тип — абсолютные значения.

|                                           |                                           |
|-------------------------------------------|-------------------------------------------|
| <code>GridBagConstraints.CENTER</code>    | <code>GridBagConstraints.EAST</code>      |
| <code>GridBagConstraints.NORTH</code>     | <code>GridBagConstraints.NORTHEAST</code> |
| <code>GridBagConstraints.NORTHWEST</code> | <code>GridBagConstraints.SOUTH</code>     |
| <code>GridBagConstraints.SOUTHEAST</code> | <code>GridBagConstraints.SOUTHWEST</code> |
| <code>GridBagConstraints.WEST</code>      |                                           |

Позиция, которую занимает компонент в каждом случае, понятна из имени соответствующей константы.

Значения второго типа — относительные. Будучи помещенными в переменную `anchor`, они задают положение, зависящее от ориентации контейнера, которое может различаться для разных языков. Допустимые относительные значения приведены ниже.

|                                                |                                                  |
|------------------------------------------------|--------------------------------------------------|
| <code>GridBagConstraints.FIRST_LINE_END</code> | <code>GridBagConstraints.FIRST_LINE_START</code> |
| <code>GridBagConstraints.LAST_LINE_END</code>  | <code>GridBagConstraints.LAST_LINE_START</code>  |
| <code>GridBagConstraints.LINE_END</code>       | <code>GridBagConstraints.LINE_START</code>       |
| <code>GridBagConstraints.PAGE_END</code>       | <code>GridBagConstraints.PAGE_START</code>       |

В данном случае расположение компонента также понятно из имени соответствующей константы.

Поля `weightx` и `weighty` очень важны, хотя вначале их назначение может показаться не совсем понятным. Их значения определяют, какое дополнительное пространство в составе контейнера соответствует каждой строке и каждому столбцу. По умолчанию принимаются нулевые значения этих переменных. Если все значения в пределах строки или столбца равны 0, дополнительное пространство равномерно распределяется между границами контейнера и компонентов. При увеличении значения `weightx` или `weighty` пространство для строки или столбца увеличивается пропорционально. Лучший способ понять, как влияют значения данных переменных на внешний вид окна, — поэкспериментировать с ними.

Переменная `gridwidth` позволяет задать ширину ячейки. По умолчанию принимается значение, равное 1. Для того чтобы указать, что компонент должен использовать остальное пространство в строке, надо задать значение `GridBagConstraints.REMAINDER`. Чтобы указать, что компонент должен быть

предпоследним в строке, надо использовать значение `GridBagConstraints.RELATIVE`. Переменная `gridheight` действует так же, но в вертикальном направлении. При необходимости для увеличения минимального размера компонента можно использовать заполнение. За горизонтальное заполнение отвечает переменная `ipadx`. Чтобы задать вертикальное заполнение, надо использовать переменную `ipady`.

Ниже приведен пример программы, использующей диспетчер компоновки `GridBagLayout`. Эта программа демонстрирует некоторые из описанных выше свойств данного диспетчера. Окно, создаваемое при работе программы, показано на рис. 10.5.

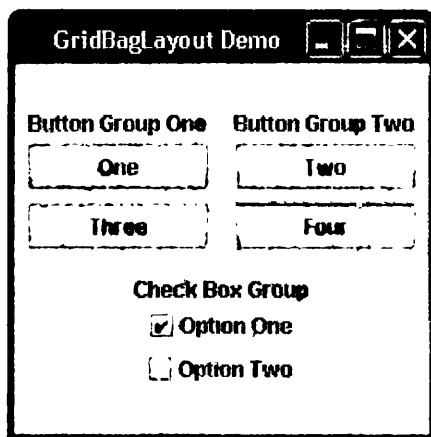


Рис. 10.5. Окно, отображаемое программой `GBDemo`

```
// Пример использования класса GridBagLayout.

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

class GBDemo {

    GBDemo() {

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("GridBagLayout Demo");

        // Создание пустых объектов GridBagLayout
        // и GridBagConstraints.
```

```
GridBagLayout gbag = new GridBagLayout();
GridBagConstraints gbc = new GridBagConstraints();

// Связывание диспетчера компоновки GridBagLayout
// с панелью содержимого фрейма.
jfrm.getContentPane().setLayout(gbag);

// Установка начальных размеров фрейма.
jfrm.setSize(240, 240);

// Завершение программы при закрытии окна пользователем.
jfrm.bsetDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Создание меток,
JLabel jlabOne = new JLabel("Button Group One");
JLabel jlabTwo = new JLabel("Button Group Two");
JLabel jlabThree = new JLabel("Check Box Group");

// Создание кнопок.
JButton jbtnOne = new JButton("One");
JButton jbtnTwo = new JButton("Two");
JButton jbtnThree = new JButton("Three");
JButton jbtnFour = new JButton("Four");

Dimension btnDim = new Dimension(100, 25);
jbtnOne.setPreferredSize(btnDim);
jbtnTwo.setPreferredSize(btnDim);
jbtnThree.setPreferredSize(btnDim);
jbtnFour.setPreferredSize(btnDim);

// Создание флажков опций.
JCheckBox jcbOne = new JCheckBox("Option One");
JCheckBox jcbTwo = new JCheckBox("Option Two");

// Определение таблицы для диспетчера компоновки.

// Значение 1.0 переменной weightx управляет
// распределением пространства в горизонтальном направлении.
// Поскольку для переменной weighty принято значение
// по умолчанию, равное 0.0, в вертикальном направлении
// компонент выравнивается по центру.
gbc.weightx = 1.0;
```

```
// Определение расположения каждого компонента в таблице.
```

```
// Метки для кнопок располагаются в ячейках 0,0 и 1,0.
```

```
gbc.gridx = 0;
gbc.gridy = 0;
gbag.setConstraints(jlabOne, gbc);
```

```
gbc.gridx = 1;
gbc.gridy = 0;
gbag.setConstraints(jlabTwo, gbc);
```

```
// Резервирование пространства между кнопками.
```

```
gbc.insets = new Insets(4, 4, 4, 4);
```

```
// Кнопки располагаются в ячейках 0,1, 1,1, и т.д.
```

```
gbc.gridx = 0;
gbc.gridy = 1;
gbag.setConstraints(jbtnOne, gbc);
```

```
gbc.gridx = 1;
gbc.gridy = 1;
gbag.setConstraints(jbtnTwo, gbc);
```

```
gbc.gridx = 0;
gbc.gridy = 2;
gbag.setConstraints(jbtnThree, gbc);
```

```
gbc.gridx = 1;
gbc.gridy = 2;
gbag.setConstraints(jbtnFour, gbc);
```

```
// Последняя метка и два флажка опций располагаются
```

```
// в оставшейся области и выравниваются по центру.
```

```
gbc.gridwidth = GridBagConstraints.REMAINDER;
```

```
// Выше метки резервируется свободное пространство
```

```
// в 10 пикселей.
```

```
gbc.insets = new Insets(10, 0, 0, 0);
gbc.gridx = 0;
gbc.gridy = 3;
gbag.setConstraints(jlabThree, gbc);
```

```
// Вокруг флажков опций пространство не резервируется.
```

```
gbc.insets = new Insets(0, 0, 0, 0);
gbc.gridx = 0;
gbc.gridy = 4;
gbag.setConstraints(jcbOne, gbc);

gbc.gridx = 0;
gbc.gridy = 5;
gbag.setConstraints(jcbTwo, gbc);

// Включение компонентов в состав панели содержимого.
jfrm.getContentPane().add(jlabOne);
jfrm.getContentPane().add(jlabTwo);
jfrm.getContentPane().add(jbtnOne);
jfrm.getContentPane().add(jbtnTwo);
jfrm.getContentPane().add(jbtnThree);
jfrm.getContentPane().add(jbtnFour);
jfrm.getContentPane().add(jlabThree);
jfrm.getContentPane().add(jcbOne);
jfrm.getContentPane().add(jcbTwo);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new GBDemo();
        }
    });
}
```

В данном случае диспетчер `GridBagLayout` используется для формирования трех групп компонентов. Каждая из двух первых групп состоит из метки и двух кнопок. Метка размещается над кнопками. Таким образом, кнопки размещаются в два столбца. В третью группу входит метка и два флажка опций. Данная группа расположена по центру под кнопками. Если вы измените размеры окна, то увидите, что относительное расположение компонентов осталось прежним. В этом состоит главное преимущество данного диспетчера компоновки. Он позволяет задать взаимное расположение элементов, которое не зависит от размеров окна.



В этой программе есть ряд интересных особенностей. Во-первых, обратите внимание, что переменной `weightx` присваивается значение 1.0. По умолчанию для переменной `weightx` принимается значение 0.0 и дополнительное пространство распределяется по краям контейнера, в результате чего компоненты размещаются рядом друг с другом. В этом нет ничего плохого, но, если задать значение, отличное от 0.0, можно улучшить внешний вид окна, более равномерно распределив компоненты по горизонтали. Для того чтобы проиллюстрировать эффект от другого решения, для переменной `weighty` принято значение по умолчанию, равное 0.0. Это означает, что верхняя граница нижнего компонента будет соприкасаться с нижней границей компонента, расположенного над ним. Желательно поэкспериментировать с обоими полями и оценить, как влияют их значения на внешний вид окна.

Далее метки над кнопками помещаются в верхнюю часть таблицы и задается зазор между кнопками, равный четырем пикселям. Перед добавлением метки над флажками опций и самих флажков устанавливается значение переменной `gridwidth`, равное `REMAINDER`. В результате компоненты занимают все оставшееся пространство в строке. Поскольку по умолчанию компоненты располагаются по центру ячейки, третья метка и флажки опций разместятся по центру оставшейся области.

`GridBagLayout` — мощный диспетчер компоновки. Стоит потратить время, чтобы изучить его подробнее и поэкспериментировать с ним. Как только вы поймете назначение его свойств, вы без труда сможете использовать его для позиционирования компонентов с высокой степенью точности.



## Вопросы для текущего контроля

1. Какие два класса совместно используются для реализации диспетчера компоновки `GridBagLayout`?
2. Какие поля класса `GridBagConstraints` определяют положение компонента в таблице?
3. Каково назначение переменных `weightx` и `weighty`?

- 
1. `GridBagLayout` и `GridBagConstraints`.
  2. `gridx` и `gridy`.
  3. Переменные `weightx` и `weighty` определяют, как должно распределяться свободное пространство в пределах контейнера.

ВАЖНО!

## 10.11 Диспетчер компоновки BorderLayout

Несмотря на то что диспетчер компоновки `GridBagLayout` широко применяется, для того, чтобы освоить его, придется приложить определенные усилия, причем добиться требуемого внешнего вида окна удастся не сразу. В ряде случаев необходимо лишь иметь возможность группировать компоненты, расположенные либо горизонтально, либо вертикально. Для подобных случаев хорошо подходит диспетчер компоновки `BoxLayout`. Класс `BoxLayout` расположен в пакете `javax.swing`.

Данный диспетчер компоновки позволяет без труда создавать группы элементов и организовывать их в виде блоков. И хотя диспетчер `BoxLayout` можно непосредственно связать с панелью содержимого, разработчики обычно так не поступают. Вместо этого они создают одну или несколько панелей и связывают их с диспетчером `BoxLayout`. В состав этих панелей включают компоненты, а сами панели размещают на панели содержимого. Таким способом можно без труда создавать группы компонентов. Хотя требуемые действия (создание компонентов `JPanel`, установка для них диспетчера `BoxLayout`, заполнение панелей компонентами и включение панелей в состав панели содержимого) можно выполнить вручную, Swing предоставляет более удобный способ решения данной задачи. Для создания контейнера автоматически использующего диспетчер компоновки `BoxLayout`, можно использовать класс `Box`. Класс `Box` также предоставляет вспомогательные методы, упрощающие позиционирование компонентов в блоке. Поскольку в большинстве приложений диспетчер `BoxLayout` используется посредством класса `Box`, рассмотрим данный подход подробнее.

Объект `Box` можно создать двумя способами. Во-первых, для этой цели подходит конструктор класса, который имеет следующий вид:

```
Box(int orientation)
```

где параметр `orientation` определяет ориентацию блока и должен принимать одно из приведенных ниже значений (данные константы определены в классе `BoxLayout`).

|                        |                                                                    |
|------------------------|--------------------------------------------------------------------|
| <code>X_AXIS</code>    | Компоненты располагаются слева направо                             |
| <code>Y_AXIS</code>    | Компоненты располагаются сверху вниз                               |
| <code>LINE_AXIS</code> | Компоненты располагаются по соглашениям, принятым в данном регионе |
| <code>PAGE_AXIS</code> | Компоненты располагаются по соглашениям, принятым в данном регионе |

Несмотря на то что подход, состоящий в создании объекта `Box` посредством конструктора класса, вполне приемлем, он используется редко. Вместо этого применяются фабричные методы, предоставляемые классом `Box`. Эти методы имеют следующий вид:

```
static Box createHorizontalBox()  
static Box createVerticalBox()
```

Первый метод создает горизонтальный блок, в котором компоненты располагаются в ряд слева направо. Второй метод создает вертикальный блок с расположением компонентов сверху вниз. Таким образом, эти два метода позволяют создать контейнер, использующий требуемый вариант диспетчера `BoxLayout`.

Имея класс `Box`, можно добавлять в него компоненты. Компоненты располагаются в том порядке, в котором они включаются в контейнер. Как вы уже знаете, блок имеет либо горизонтальную, либо вертикальную ориентацию. Соответственно компоненты располагаются либо в строку, либо в столбец.

По умолчанию компоненты размещаются непосредственно друг за другом. Чтобы оставить между ними пустое пространство, надо включить в контейнер объект, созданный с помощью метода `createRigidArea()`.

```
static Component createRigidArea(Dimension dim)
```

Здесь параметр `dim` задает размер создаваемой области. Метод возвращает компонент, имеющий указанные размеры. Созданный компонент можно добавлять в блок в том месте, где необходимо оставить пустую область между элементами.

Компонент, созданный с помощью метода `createRigidArea()`, называют “жесткой” областью, так как ее размеры фиксированы и остаются постоянными при изменении размеров блока. Класс `Box` предоставляет и другие методы для выделения пространства, однако чаще всего используются “жесткие” области. Необходимо помнить, что метод `createRigidArea()` возвращает объект `Component`. Каждый компонент можно включить в контейнер, управляемый диспетчером компоновки, только один раз. Следовательно, для формирования еще одного свободного пространства между элементами надо создать новую “жесткую” область.

Ниже приведен код программы, демонстрирующей использование диспетчера компоновки `BoxLayout` посредством класса `Box`. Данная программа представляет собой модификацию рассмотренного ранее примера, демонстрирующего работу с диспетчером `GridLayout`, но в ней каждая группа компонентов помещается в отдельный блок. Преимущество данного подхода состоит в том, что компоненты объединяются в группы, а каждая из групп может перемещаться при изменении размеров фрейма. Еще одна положительная особенность состоит в том, что код для создания блоков имеет меньший объем, чем

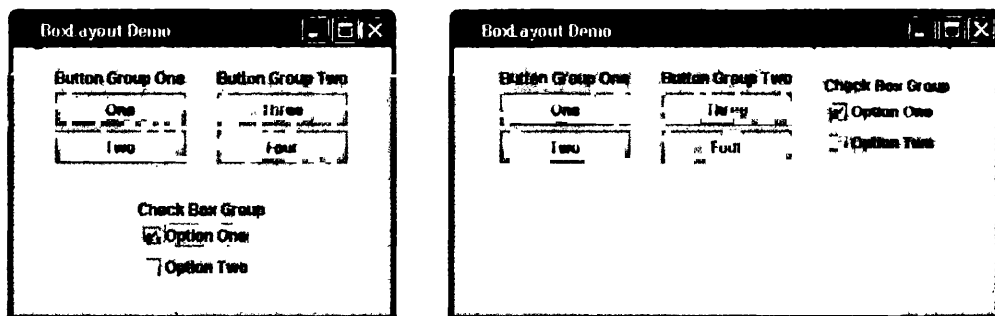


Рис. 10.6. Данные, отображаемое программой BoxDemo

аналогичный код, работающий с диспетчером GridLayout. Окно, создаваемое при работе программы, показано на рис. 10.6.

```
// Использование диспетчера компоновки BoxLayout
// посредством класса Box.

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

class BoxDemo {

    BoxDemo() {

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("BoxLayout Demo");

        // *** С панелью содержимого связывается
        // диспетчер компоновки FlowLayout. ***
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(300, 240);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание меток.
        JLabel jlabOne = new JLabel("Button Group One");
        JLabel jlabTwo = new JLabel("Button Group Two");
        JLabel jlabThree = new JLabel("Check Box Group");
```

```
// Создание кнопок.
JButton jbtnOne = new JButton("One");
JButton jbtnTwo = new JButton("Two");
JButton jbtnThree = new JButton("Three");
JButton jbtnFour = new JButton("Four");
Dimension btnDim = new Dimension(100, 25);

// Установка минимального и максимального размера кнопок.
jbtnOne.setMinimumSize(btnDim);
jbtnOne.setMaximumSize(btnDim);
jbtnTwo.setMinimumSize(btnDim);
jbtnTwo.setMaximumSize(btnDim);
jbtnThree.setMinimumSize(btnDim);
jbtnThree.setMaximumSize(btnDim);
jbtnFour.setMinimumSize(btnDim);
jbtnFour.setMaximumSize(btnDim);

// Создание флажков опций.
JCheckBox jcbOne = new JCheckBox("Option One");
JCheckBox jcbTwo = new JCheckBox("Option Two");

// Создание трех вертикальных блоков.
Box box1 = Box.createVerticalBox();
Box box2 = Box.createVerticalBox();
Box box3 = Box.createVerticalBox();

// Создание невидимых рамок для кнопок.
box1.setBorder(
    BorderFactory.createEmptyBorder(10, 10, 10, 10));
box2.setBorder(
    BorderFactory.createEmptyBorder(10, 10, 10, 10));
box3.setBorder(
    BorderFactory.createEmptyBorder(10, 10, 10, 10));

// Включение компонентов в блоки box1 и box2.
box1.add(jlabOne);
box1.add(Box.createRigidArea(new Dimension(0, 4)));
box1.add(jbtnOne);
box1.add(Box.createRigidArea(new Dimension(0, 4)));
box1.add(jbtnTwo);

box2.add(jlabTwo);
```

```
box2.add(Box.createRigidArea(new Dimension(0, 4)));
box2.add(jbtnThree);
box2.add(Box.createRigidArea(new Dimension(0, 4)));
box2.add(jbtnFour);

// Включение флажков опций в блок box3.
box3.add(jlabThree);
box3.add(jcbOne);
box3.add(jcbTwo);

// Включение блоков в состав панели содержимого.
jfrm.getContentPane().add(box1);
jfrm.getContentPane().add(box2);
jfrm.getContentPane().add(box3);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new BoxDemo();
        }
    });
}
```

Код программы достаточно прост. После создания всех компонентов в программе создаются три объекта `Box`. Для каждого блока задается невидимая рамка толщиной в 10 пикселей. Это не является абсолютно необходимым, но таким образом можно зарезервировать пространство между блоками. В каждый из блоков включается группа компонентов. Обратите внимание на использование “жестких” областей. С их помощью формируются зазоры между метками и кнопками. В группе флажков опций “жесткие” области не требуются, поскольку пустое пространство, предусмотренное по умолчанию, подходит для разделения данных компонентов.

Как уже говорилось ранее, в ряде случаев вместо диспетчера компоновки `GridBagLayout` удобнее использовать `BoxLayout`. В этом случае уменьшается объем кода и легче добиться требуемого размещения элементов. Диспетчер компоновки `BoxLayout` также позволяет визуально выделять группы элемен-

тов. При изменении размеров окна блоки изменяют свое взаимное расположение под управлением диспетчера компоновки, связанного с этим окном. Диспетчер `BoxLayout` желательно использовать в большинстве ситуаций, оставив `GridLayout` для тех случаев, когда он действительно необходим.



### Вопросы для текущего контроля

1. С помощью диспетчера компоновки `BoxLayout` компоненты можно располагать \_\_\_\_\_ или \_\_\_\_\_.
2. Какой класс создает контейнер, автоматически использующий диспетчер компоновки `BoxLayout`?
3. С помощью какого метода создается компонент фиксированного размера, используемый для резервирования пространства между компонентами в контейнере, управляемом диспетчером компоновки `BoxLayout`?

## Что далее

Примите мои поздравления! Если вы прочитали и поняли материал, изложенный в десяти модулях, значит, вы получили общее представление об основных средствах `Swing`. Теперь вы можете применять полученные знания при создании реальных приложений. Конечно, вам придется изучить еще многие средства `Swing`, но основа для дальнейшего накопления знаний и опыта у вас уже есть.

Вероятнее всего, вам потребуется изучить следующие темы:

- `JInternalPane`
- `JDesktopPane`
- `JLayeredPane`
- `JTextPane`
- Обеспечение доступа
- Отмена операций при работе с текстовыми компонентами
- Создание компонентов, определяемых разработчиком

---

5 1. Горизонтально или вертикально.

2. `Box`.

3. `createRigidArea()`.

## Спросим у опытного программиста

**Вопрос.** Предоставляет ли класс `Box` другие способы резервирования пространства между компонентами?

**Ответ.** Помимо “жестких” областей класс `Box` предоставляет “склейки” (`glue`) и “распорки” (`struts`). Название “склейка” выбрано явно неудачно. На самом деле этот механизм управляет расширением зазора между двумя компонентами. При наличии “склейки” появляющееся дополнительное пространство распределяется не между границами контейнера, а между “склеенными” компонентами. Для управления “склейками” используются методы `createGlue()`, `createHorizontalGlue()` и `createVerticalGlue()`. “Распорка” — это объект, имеющий один фиксированный и один переменный размер. Существуют два типа “распорок”: горизонтальные и вертикальные. Ширина горизонтальной “распорки” фиксирована, а высота может изменяться. Для вертикальной “распорки” фиксирована высота. Для создания “распорок” используются методы `createHorizontalStrut()` и `createVerticalStrut()`.

Вам также предстоит глубже изучить свойства, поля и методы, предоставляемые каждым компонентом Swing. В данной книге внимание сосредоточено лишь на тех средствах, которые чаще всего используются при создании программ. Однако при работе над приложениями пригодятся и другие механизмы. Возможно, вы заинтересуетесь реализацией перетаскивания объектов с помощью мыши. Тем же читателям, которые захотят достичь наивысшего уровня в Swing-программировании, возможно, потребуется изучить правила создания новых стилей.

В начале данной книги было сказано, что Swing — это большая и сложная среда. На ее изучение придется потратить время и усилия, однако они обязательно окупятся. Интерфейс программы — первое, с чем сталкивается пользователь. Всем известно, что первое впечатление часто остается навсегда. Освоив библиотеку Swing и грамотно применяя ее, вы создадите у пользователя благоприятное представление о своем продукте.



## Тест для самоконтроля по модулю 10

1. Код, взаимодействующий с компонентами Swing, должен выполняться в потоке \_\_\_\_\_.
2. Предположим, что в программе используется класс `javax.swing.Timer`. Какое событие генерируется таймером?
3. Какие методы используются для запуска и остановки таймера, реализуемого классом `javax.swing.Timer`?
4. Назовите четыре метода, определяющих жизненный цикл апплета. Когда вызываются эти методы?
5. Какой метод следует применять при создании пользовательского интерфейса апплета?
6. Можно ли организовать в программе рисование на поверхности компонента?
7. Какие три метода, связанные с рисованием, вызываются при отображении компонента Swing?
8. К какому методу следует обратиться, чтобы вызвать перерисовку содержимого компонента?
9. Какие методы позволяют определить текущую ширину и высоту компонента?
10. Диспетчер компоновки реализует объединение небольших \_\_\_\_\_.
11. Каждому компоненту в контейнере, связанном с диспетчером компоновки `GridBagLayout`, ставится в соответствие набор ограничений. В каком классе инкапсулированы эти ограничения?
12. Почему в ряде случаев диспетчер компоновки `BoxLayout` оказывается более эффективным, чем `GridBagLayout`?
13. Может ли в горизонтальном блоке, созданном с помощью класса `Box`, содержаться более одной строки компонентов?
14. Модифицируйте программу, отображающую гистограмму, так, чтобы цвет обрамления изменялся каждую секунду. В разные моменты времени должна отображаться красная, зеленая и синяя рамки. Кроме того, обеспечьте изменение размера рамки с одного на пять пикселей и обратно через одну пятую секунды. Для решения этой задачи используйте класс `javax.swing.Timer`.

15. Модифицируйте код апплета в проекте 1.10 так, чтобы направление перемещения текста периодически изменялось. Например, пусть текст некоторое время перемещается влево, затем вправо и т.д. Время, через которое направление перемещения будет изменяться, выберите сами. Программа, приведенная в приложении к данной книге, изменяет направление каждые 20 секунд.
16. Поэкспериментируйте с компонентами и контейнерами Swing. Попробуйте изменять их характеристики. Подобные эксперименты необходимы для глубокого понимания материала.



# Приложение

Ответы на вопросы  
для самоконтроля

## Модуль 1. Общие сведения о Swing

1. Большинство компонентов AWT преобразуются в платформенно-ориентированные элементы. Почему это считается проблемой и как она решается средствами Swing?

При использовании платформенно-ориентированных компонентов может возникнуть проблема, связанная с тем, что они по-разному отображаются на различных платформах и изменить порядок отображения достаточно сложно. На компоненты налагается ряд ограничений, в частности, они имеют прямоугольную форму и непрозрачны.

2. Большинство компонентов Swing полностью реализовано на языке Java. Да или нет?

Да.

3. Назовите четыре тяжелых контейнера верхнего уровня.

JFrame, JApplet, JDialog и JWindow.

4. Какой контейнер верхнего уровня чаще всего используется в приложениях?

JFrame.

5. Контейнер JFrame содержит несколько панелей. В какую панель помещаются компоненты?

Компоненты включаются в состав панели содержимого.

6. Чтобы обработчик получал оповещение о возникающих событиях, он должен быть \_\_\_\_\_ в источнике.

Зарегистрирован.

7. Какой интерфейс должен реализовывать класс, чтобы получать оповещение о событиях действий?

ActionListener.

8. Какой метод следует вызвать при работе с компонентом JButton или JTextField, чтобы установить команду действия?

setActionCommand().

9. Назовите три диспетчера компоновки.

FlowLayout, BorderLayout и GridLayout. Число существующих диспетчеров компоновки не ограничивается тремя. Применяются также GridBagLayout, BoxLayout и SpringLayout.

10. В секундомере, созданном в рамках проекта 1.1, используются две кнопки: одна для запуска секундомера, а другая — для его остановки. Однако можно использовать только одну кнопку, которая будет запускать остановленный секундомер и останавливать идущий. При использовании такого подхода имеет смысл заменять текст на кнопке (надпись Start менять на Stop и наоборот). Поскольку по умолчанию текст, отображаемый на кнопке, является в то же время командой действия, вы можете использовать одну кнопку для двух различных целей. Ваша задача — переписать проект 1.1 в соответствии с данным подходом.

Для решения этой задачи надо использовать метод `setText()`, предоставляемый классом `JButton`. Этот метод устанавливает текст на кнопке. Его заголовок выглядит следующим образом:

```
void setText(String msg).
```

Здесь параметр `msg` задает строку текста, которая должна выводиться на кнопке. Пользуясь этим методом, вы можете менять текст на кнопке в процессе выполнения программы.

Одно из возможных решений приведено ниже.

```
// Вариант секундомера, созданного в проекте 1.1,  
// в котором используется одна кнопка.
```

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

```
class Stopwatch implements ActionListener {
```

```
    JLabel jlab;  
    long start; // Содержит время запуска в миллисекундах.  
    JButton jbtnStartStop; // Кнопка для запуска или  
                           // остановки секундомера
```

```
    Stopwatch() {
```

```
        // Создание нового контейнера JFrame.  
        JFrame jfrm = new JFrame("A Simple Stopwatch");
```

```
        // Установка диспетчера компоновки FlowLayout.  
        jfrm.getContentPane().setLayout(new FlowLayout());
```

```
// Установка исходных размеров фрейма.
jfrm.setSize(230, 90);

// Завершение программы при закрытии окна пользователем.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Создание кнопки.
jbtnStartStop = new JButton("Start");

// Связывание обработчика событий с кнопкой.
jbtnStartStop.addActionListener(this);

// Включение кнопки в состав панели содержимого.
jfrm.getContentPane().add(jbtnStartStop);

// Создание текстовой метки.
jlab = new JLabel("Press Start to begin timing.");

// Включение метки в состав панели содержимого.
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

// Обработка событий, генерируемых кнопкой.
public void actionPerformed(ActionEvent ae) {
    // Получение текущего системного времени.
    Calendar cal = Calendar.getInstance();

    if(ae.getActionCommand().equals("Start")) {
        // Сохранение времени запуска.
        start = cal.getTimeInMillis();
        jlab.setText("Stopwatch is Running...");
        jbtnStartStop.setText("Stop");
    }
    else {
        // Определение времени, прошедшего с момента запуска.
        jlab.setText("Elapsed time is "
            + (double) (cal.getTimeInMillis() -
start)/1000);
        jbtnStartStop.setText("Start");
    }
}
```

```
}

public static void main(String args[]) {

    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new Stopwatch();
        }
    });
}
```

11. Если вы пользуетесь JDK 5 либо более поздней версией, перепишите код проекта 1.2 так, чтобы в нем не вызывался метод `getContentPane()`. Ниже приведен вариант программы из проекта 1.2, в которой отсутствуют обращения к методу `getContentPane()`.

```
// Вариант программы из проекта 1.2,
// в которой не используется метод getContentPane().

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

class Coder implements ActionListener {

    JTextField jtfPlaintext;
    JTextField jtfCiphertext;

    Coder() {

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("A Simple Code Machine");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(340, 120);

        // Завершение программы при закрытии окна
        // пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание двух меток.
```



```
JLabel jlabPlaintext = new JLabel(" Plain Text: ");
JLabel jlabCiphertext = new JLabel("Cipher Text: ");

// Создание двух полей редактирования.
jtfPlaintext = new JTextField(20);
jtfCiphertext = new JTextField(20);

// Установка команд действия для полей
// редактирования.
jtfPlaintext.setActionCommand("Encode");
jtfCiphertext.setActionCommand("Decode");

// Связывание обработчиков событий с полями
// редактирования.
jtfPlaintext.addActionListener(this);
jtfCiphertext.addActionListener(this);

// Добавление полей редактирования и меток
// к панели содержимого.
jfrm.add(jlabPlaintext);
jfrm.add(jtfPlaintext);
jfrm.add(jlabCiphertext);
jfrm.add(jtfCiphertext);

// Создание экземпляров кнопок.
JButton jbtnEncode = new JButton("Encode");
JButton jbtnDecode = new JButton("Decode");
JButton jbtnReset = new JButton("Reset");

// Связывание обработчиков событий с кнопками.
jbtnEncode.addActionListener(this);
jbtnDecode.addActionListener(this);
jbtnReset.addActionListener(this);

// Включение кнопок в состав панели содержимого.
jfrm.add(jbtnEncode);
jfrm.add(jbtnDecode);
jfrm.add(jbtnReset);

// Отображение фрейма.
jfrm.setVisible(true);
}

// Обработка событий действия.
public void actionPerformed(ActionEvent ae) {

    // Если команда действия равна "Encode",
```

```
// строка шифруется.
if(ae.getActionCommand().equals("Encode")) {

    // Получение текста и передача его
    // объекту StringBuilder.
    StringBuilder str =
        new StringBuilder(jtfPlaintext.getText());

    // Добавление единицы к коду каждого символа.
    for(int i=0; i<str.length(); i++)
        str.setCharAt(i, (char)(str.charAt(i) + 1));

    // Помещение зашифрованного текста в поле
    Cipher Text.
    jtfCiphertext.setText(str.toString());

}

// Если команда действия равна "Decod  ",
// строка декодируется.
else if(ae.getActionCommand().equals("Decode"))
{

    // Получение кодированного текста
    // и передача его объекту StringBuilder.
    StringBuilder str =
        new StringBuilder(jtfCiphertext.getText());

    // Вычитание единицы из кода каждого символа.
    for(int i=0; i<str.length(); i++)
        str.setCharAt(i, (char)(str.charAt(i) - 1));

    // Помещение декодированного текста
    // в поле Plain Text.
    jtfPlaintext.setText(str.toString());
}

// Последний вариант - команда Reset.
else {
    jtfPlaintext.setText("");
    jtfCiphertext.setText("");
}
}

public static void main(String args[]) {

    // Фрейм создается в потоке обработки событий.
```

```
SwingUtilities.invokeLater(new Runnable() {  
    public void run() {  
        new Coder();  
    }  
});  
}
```

## **Модуль 2. Метки, кнопки и обрамление**

1. Какой метод создает обрамление для компонента?

`setBorder()`.

2. По умолчанию содержимое метки выравнивается в вертикальном направлении по центру, а в горизонтальном направлении — по \_\_\_\_\_.  
Ведущему краю.

3. С помощью какого метода можно задать вертикальную позицию текста относительно изображения в составе метки?

`setVerticalTextPosition()`.

4. Зачем нужен класс `ImageIcon`?

Класс `ImageIcon` реализует интерфейс `Icon`. Помимо других возможностей, он позволяет загрузить изображение из файла.

5. По умолчанию деактивизированный элемент отображается тусклым цветом. Да или нет?

Да.

6. Что такое мнемоническое обозначение?

Мнемоническое обозначение определяет клавишу, которая, будучи нажатой в комбинации с клавишей `<Alt>`, вызывает передачу фокуса ввода данному компоненту.

7. Назовите четыре типа кнопок, предоставляемых Swing.

`JButton`, `JToggleButton`, `JCheckBox` и `JRadioButton`.

8. Какой метод надо определить при реализации интерфейса `ItemListener`?

`itemStateChanged()`.

9. Для каких кнопок с двумя состояниями суперклассом является `JToggleButton`?

`JCheckBox` и `JRadioButton`.

**10. Для чего используется метод `setDefaultButton()`?**

Метод `setDefaultButton()` определяет кнопку, которая будет активирована по нажатию клавиши `<Enter>`.

**11. В состав какого объекта надо включить объекты `JRadioButton`, чтобы выбор одной кнопки автоматически отменял выбранную ранее?**

Для того чтобы объекты `JRadioButton` вели себя как кнопки с зависимой фиксацией, их надо добавить к экземпляру класса `ButtonGroup`.

**12. Сложная задача. Модифицируйте программу `StopWatch`, созданную ранее в проекте 1.1, так, как показано ниже.**

- Вместо класса `Calendar` и метода `getTimeInMillis()` используйте для получения текущего времени информацию, предоставляемую классом, который описывает событие действия. Как вы помните, эти данные доступны посредством метода `getWhen()`.
- Сделайте так, чтобы до щелчка на кнопке `Start` кнопка `Stop` была недоступна. Также запретите доступ к кнопке `Start` до щелчка на кнопке `Stop`.
- Добавьте флажок опции, который будет указывать, должен ли отображаться протокол учтенного времени. Если флажок установлен, информация о времени должна включаться в журнал. При сбросе флажка опции содержимое журнала следует удалить. Хранить и отображать следует лишь последние три записи. При заполнении журнала (в случае, если в нем хранятся три записи) при каждой новой записи удаляйте одну старую из конца списка.

Одно из возможных решений данной задачи приведено ниже.

```
// Модифицированная версия программы, созданной  
// в рамках проекта 1.1.
```

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

```
class StopWatch {
```

```
    JLabel jlab;  
    JLabel jlabLog;  
    JCheckBox jcbKeepLog;
```

```
    JButton jbtnStart;  
    JButton jbtnStop;
```

```
long start; // Содержит время запуска в миллисекундах.

String[] etstr; // Содержит время, прошедшее с момента
                // запуска, представленное в виде строки.

static final int LOGMAX = 3; // Максимальное число
.....// записей в журнале.

StopWatch() {

    // Инициализация etstr строками нулевой длины.
    etstr = new String[LOGMAX];
    for(int i=0; i<LOGMAX; i++) etstr[i] = "";

    // Создание нового контейнера JFrame.
    JFrame jfrm = new JFrame("Improved Stopwatch");

    // Установка диспетчера компоновки FlowLayout.
    jfrm.getContentPane().setLayout(new FlowLayout());

    // Установка исходных размеров фрейма.
    jfrm.setSize(230, 170);

    // Завершение программы при закрытии окна пользователем.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Создание двух кнопок.
    jbtnStart = new JButton("Start");
    jbtnStop = new JButton("Stop");

    // Запрет доступа к кнопке Stop.
    jbtnStop.setEnabled(false);

    // Создание флажка опции Keep Time Log.
    jcbKeepLog = new JCheckBox("Keep Time Log");

    // Создание метки для журнала.
    jlabLog = new JLabel("<html>-Time Log-<br><br><br><br>");

    // Формирование рамки вокруг метки журнала.
    jlabLog.setBorder(
        BorderFactory.createLineBorder(Color.BLUE));
```

## Ответы на вопросы для самоконтроля

```

        // Обновление метки журнала.
        jlabLog.setText(logstr);
    }
}
});

// Связывание обработчика событий элемента с флажком
// Keep Time Log. При сбросе флажка строки удаляются.
jcbKeepLog.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent ie) {
        if(!jcbKeepLog.isSelected()) {
            // Удаление информации из строк журнала.
            for(int i=0; i<LOGMAX; i++)
                etstr[i] = "";

            // Очистка строки, отображающей
            // информацию из журнала.
            jlabLog.setText(
                "<html>-Time Log-<br><br><br><br>");
        }
    }
});

// Включение компонентов в состав панели содержимого.
jfrm.getContentPane().add(jbtnStart);
jfrm.getContentPane().add(jbtnStop);
jfrm.getContentPane().add(jlab);
jfrm.getContentPane().add(jcbKeepLog);
jfrm.getContentPane().add(jlabLog);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Создание фрейма в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new Stopwatch();
        }
    });
}
}
}

```

## Модуль 3. Полосы прокрутки, линейные регуляторы и индикаторы хода процесса

1. Полосы прокрутки, линейные регуляторы и индикаторы хода процесса используют модель \_\_\_\_\_.

`BoundedRangeModel`.

2. Опишите зависимость между текущим, минимальным, максимальным значением и расширением.

Текущее значение должно быть больше или равно минимальному значению и не должно превышать максимальное значение минус расширение.

3. Когда пользователь щелкает на полосе прокрутки, ползунок изменяет свое положение в соответствии с величиной приращения блока. Да или нет?

Да.

4. Какой интерфейс следует реализовать, чтобы обеспечить обработку событий, связанных с полосой прокрутки?

`AdjustmentListener`.

5. В чем различие между основными и вспомогательными маркерами линейного регулятора?

Основные маркеры делят диапазон линейного регулятора на большие интервалы, равные, например, 10. Вспомогательные маркеры располагаются между основными и делят каждый интервал на подынтервалы. Часто величина подынтервала выбирается равной одной единице измерения.

6. События какого типа генерирует линейный регулятор?

При перемещении ползунка пользователем линейный регулятор генерирует события изменения состояния.

7. Процесс установки маркеров и меток для линейного регулятора состоит из трех этапов. Опишите эти этапы.

Для отображения меток линейного регулятора необходимо, во-первых, создать эти метки, вызвав метод `createStandardLabels()`. Во-вторых, следует добавить метки к регулятору с помощью метода `setLabelTable()`. И наконец, надо обеспечить отображение меток путем вызова `setPaintLabels(true)`.

8. Может ли индикатор хода процесса генерировать события, связанные с действиями пользователя?

Нет.



9. Можно ли создать индикатор хода процесса, вокруг которого не будет отображаться рамка? Если можно, то как?

Да. Для того чтобы создать индикатор хода процесса без рамки, надо вызвать метод `setBorderPainted(false)`.

10. Какой интерфейс должен реализовывать обработчик событий, способный отслеживать состояние индикатора хода процесса?

Индикатор хода процесса генерирует событие изменения состояния при изменении свойства. Следовательно, обработчик должен реализовывать интерфейс `ChangeListener`.

11. Напишите программу, содержащую линейный регулятор и полосу прокрутки, взаимодействующие друг с другом. При каждом перемещении ползунка линейного регулятора должно соответствующим образом изменяться положение ползунка полосы прокрутки, а изменение состояния полосы прокрутки должно отражаться на линейном регуляторе.

Эта программа может иметь следующий вид:

```
// Взаимодействие линейного регулятора.
```

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

class SBSlider {

    JScrollBar jsb;
    JSlider jsldr;

    SBSlider() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Scroll Bar with Slider");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(260, 100);

        // Завершение программы
        // при закрытии приложения пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
// Создание полосы прокрутки и линейного регулятора.
jsb = new
    JScrollBar(Adjustable.HORIZONTAL, 0, 0, 0, 100);
jsldr = new JSlider(0, 100, 0);

// Увеличение отображаемого размера полосы прокрутки.
jsb.setPreferredSize(new Dimension(220, 20));

// Связывание с полосой прокрутки обработчика
// событий регулировки.
jsb.addAdjustmentListener(new AdjustmentListener() {
    public void adjustmentValueChanged(AdjustmentEvent ae)
        { jsldr.setValue(jsb.getValue()); }
});

// Связывание с линейным регулятором
// обработчика событий изменения состояния.
jsldr.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent ce) {
        jsb.setValue(jsldr.getValue());
    }
});

// Включение компонентов в состав панели содержимого.
jfrm.getContentPane().add(jsb);
jfrm.getContentPane().add(jsldr);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Создание фрейма в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new SBSlider();
        }
    });
}
```

12. В проекте 2.2 был создан простой телефонный справочник. Добавьте к нему полосу прокрутки, которая обеспечит перемещение по списку имен и номеров телефонов. Для этого добавьте метку, отображающую имя и номер телефона. При перемещении ползунок полосы прокрутки содержимое метки должно изменяться, т.е. в ней должны отображаться следующее имя и номер.

Пример решения данной задачи приведен ниже.'

```
// Реализация полосы прокрутки в программе,  
// созданной в рамках проекта 2.2.  
  
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
  
class Phonebook {  
  
    JTextField jtfName;  
    JTextField jtfNumber;  
  
    JRadioButton jrbExact;  
    JRadioButton jrbStartsWith;  
    JRadioButton jrbEndsWith;  
  
    JCheckBox jcbIgnoreCase;  
  
    JScrollBar jsb;  
  
    // Краткий список имен и номеров телефонов.  
    String[][] phonelist = {  
        {"Jon", "555-8765"},  
        {"Jessica", "555-5643"},  
        {"Adam", "555-1212"},  
        {"Rachel", "555-3435"},  
        {"Tom & Jerry", "555-1001"}  
    };  
  
    Phonebook() {  
  
        // Создание нового контейнера JFrame.  
        JFrame jfrm = new JFrame("A Simple Phone List");  
  
        // Установка диспетчера компоновки GridLayout.
```

```
jfrm.getContentPane().setLayout(new GridLayout(0, 1));

// Установка начальных размеров фрейма.
jfrm.setSize(240, 220);

// Завершение программы при закрытии
// приложения пользователем.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Создание меток.
JLabel jlabName = new JLabel("Name");
JLabel jlabNumber = new JLabel("Number");
JLabel jlabOptions = new JLabel("Search Options");

// Создание полей редактирования.
jtfName = new JTextField(10);
jtfNumber = new JTextField(10);

// Создание флажка опции Ignore Case.
jcbIgnoreCase = new JCheckBox("Ignore Case");

// Создание кнопок переключателей опций.
jrbExact = new JRadioButton("Exact Match", true);
jrbStartsWith = new JRadioButton("Starts With");
jrbEndsWith = new JRadioButton("Ends With");

// Добавление кнопок переключателей опций
// к группе.
ButtonGroup bg = new ButtonGroup();
bg.add(jrbExact);
bg.add(jrbStartsWith);
bg.add(jrbEndsWith);

// Создание полосы прокрутки.
jsb = new JScrollBar(Adjustable.HORIZONTAL, 0, 0, 0, 4);

// Связывание обработчика событий действия
// с полем редактирования Name.
jtfName.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        jtfNumber.setText(lookupName(jtfName.getText()));
    }
});
```

```
// Связывание обработчика событий действия
// с полем редактирования Number.
jtfNumber.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        jtfName.setText(lookupNumber(jtfNumber.getText()));
    }
});

// Обработка событий, генерируемых полосой
// прокрутки, и прокрутка списка телефонных номеров.
jsb.addAdjustmentListener(new AdjustmentListener() {
    {
        int i = jsb.getValue();
        jtfName.setText(phonelist[i][0]);
        jtfNumber.setText(phonelist[i][1]);
    }
});

// Включение компонентов в состав панели содержимого.
jfrm.getContentPane().add(jlabName);
jfrm.getContentPane().add(jtfName);
jfrm.getContentPane().add(jlabNumber);
jfrm.getContentPane().add(jtfNumber);
jfrm.getContentPane().add(new JLabel());
jfrm.getContentPane().add(jlabOptions);
jfrm.getContentPane().add(jcbIgnoreCase);
jfrm.getContentPane().add(new JLabel());
jfrm.getContentPane().add(jrbExact);
jfrm.getContentPane().add(jrbStartsWith);
jfrm.getContentPane().add(jrbEndsWith);
jfrm.getContentPane().add(jsb);

// Отображение фрейма.
jfrm.setVisible(true);
}

// Поиск по имени и возврат номера.
String lookupName(String n) {
    for(int i=0; i < phonelist.length; i++) {
        if(jrbStartsWith.isSelected()) {
            if(jcbIgnoreCase.isSelected()) {
```

```
        if(phonelist[i][0].toLowerCase().
           startsWith(n.toLowerCase()))
            return phonelist[i][1];
    } else {
        if(phonelist[i][0].startsWith(n))
            return phonelist[i][1];
    }
}
else if(jrbEndsWith.isSelected()) {
    if(jcbIgnoreCase.isSelected()) {
        if(phonelist[i][0].toLowerCase().
           endsWith(n.toLowerCase()))
            return phonelist[i][1];
    } else {
        if(phonelist[i][0].endsWith(n))
            return phonelist[i][1];
    }
}
else {
    if(jcbIgnoreCase.isSelected()) {
        if(phonelist[i][0].toLowerCase().
           equals(n.toLowerCase()))
            return phonelist[i][1];
    } else {
        if(phonelist[i][0].equals(n))
            return phonelist[i][1];
    }
}
}

return "Not Found";
}

// Поиск по номеру и возврат имени.
String lookupNumber(String n) {
    for(int i=0; i < phonelist.length; i++) {
        if(phonelist[i][1].equals(n))
            return phonelist[i][0];
    }
    return "Not Found";
}

public static void main(String args[]) {
```

```

// Создание фрейма в потоке обработки событий.
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        new Phonebook();
    }
});
}
}

```

## Модуль 4. Управление компонентами.

### Панели и строка подсказки

1. Что такое двойная буферизация и зачем она нужна?

Двойная буферизация — это механизм, обычно используемый для того, чтобы устранить нежелательные визуальные эффекты при обновлении экрана. Рисование каждого компонента непосредственно на экране может привести к возникновению эффекта мерцания, поэтому компоненты воспроизводятся в отдельном буфере. Когда процесс воспроизведения окончен, содержимое буфера копируется на экран посредством одной непрерывной операции. В результате новое содержимое панели выводится не по частям, а возникает на экране мгновенно.

2. Какой метод надо вызвать, чтобы компонент `JPanel` стал непрозрачным?

`setOpaque()`.

3. Можно ли использовать в качестве панели содержимого объект `JPanel`, созданный в программе?

Да.

4. Несмотря на обширные возможности, предоставляемые элементом `JScrollPane`, пользоваться им очень сложно. Да или нет?

Нет. Несмотря на обширные возможности, компонент `JScrollPane` прост в использовании.

5. Назовите девять областей в составе `JScrollPane`.

В составе `JScrollPane` содержатся область просмотра, вертикальная полоса прокрутки, горизонтальная полоса прокрутки, заголовок строки, заголовок столбца и четыре угловые области.

6. Какой метод надо вызвать, чтобы установить заголовок строки `JScrollPane`?

`setRowHeaderView()`.

7. Какой метод надо использовать для включения компонента в состав `JTabbedPane`?

`addTab()`.

8. Как установить строку подсказки для вкладки компонента `JTabbedPane`?  
Надо использовать вариант метода `addTab()`, в котором предусмотрен параметр, определяющий строку подсказки

9. Назовите возможные варианты ориентации панели `JSplitPane`?  
Вертикальная и горизонтальная.

10. Что такое средства мгновенного расширения `JSplitPane`? Какой метод надо использовать, чтобы включить их?

Средства мгновенного расширения — это две небольшие кнопки, помещаемые на разделитель в составе компонента `JTabbedPane`. По щелчку на такой кнопке одна часть разделяемой панели расширяется и занимает собой все окно, а вторая пропадает из области видимости. Скрытую часть можно отобразить, активизировав другую кнопку. Включить средства мгновенного расширения можно, вызвав метод `setOneTouchExpandable()`.

11. Правда ли, что строку подсказки можно задать для любого легковесного компонента Swing? Если можно, то как это сделать?

Да. Для этого надо вызвать метод `setToolTipText()`.

12. Желательно не использовать строки подсказки слишком часто, так как пользовательский интерфейс оказывается перегруженным ими. Да или нет?

Нет. Строки подсказки отображаются только тогда, когда пользователь задерживает курсор мыши на элементе, поэтому они не перегружают интерфейс

13. Добавьте строки подсказки для кнопок в программе `CustomCPDemo`, которая была рассмотрена в начале данной главы.

Одно из возможных решений данной задачи приведено ниже.

```
// Отображение строк подсказки при работе  
// с JPanel.
```

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

```
// Данный класс создает панель, расширяющую JPanel.  
// Она используется в качестве панели содержимого.  
// В этом классе не предусмотрены новые функциональные
```



```
// возможности, но его экземпляр может применяться
// во всех случаях, в которых допустимо использование JPanel.
class MyContentPanel extends JPanel {

    JLabel jlab;
    JButton jbtnRed;
    JButton jbtnBlue;

    MyContentPanel() {

        // Панель должна быть непрозрачной.
        setOpaque(true);

        // Установка рамки зеленого цвета толщиной в 5 пикселей.
        setBorder(
            BorderFactory.createLineBorder(Color.GREEN, 5));

        // Создание метки.
        jlab = new JLabel("Select Border Color");

        // Создание двух кнопок.
        jbtnRed = new JButton("Red");
        jbtnBlue = new JButton("Blue");

        // Связывание строк подсказки с кнопками.
        jbtnRed.setToolTipText("Makes the border red.");
        jbtnBlue.setToolTipText("Makes the border blue.");

        // Связывание обработчиков событий с кнопками.
        jbtnRed.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                setBorder(
                    BorderFactory.createLineBorder(Color.RED, 5));
            }
        });

        jbtnBlue.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                setBorder(
                    BorderFactory.createLineBorder(Color.BLUE, 5));
            }
        });

        // Включение кнопок и метки в состав панели.
```

```
        add(jbtnRed);
        add(jbtnBlue);
        add(jlab);
    }
}

// Создание контейнера верхнего уровня и использование
// панели, созданной с помощью MyContentPanel,
// в качестве панели содержимого
class CustomCPDemo {
    CustomCPDemo() {
        // Создание нового контейнера JFrame. По умолчанию
        // с этим контейнером связан диспетчер
        // компоновки BorderLayout.
        JFrame jfrm = new JFrame("Set the Content Pane");

        // Установка начальных размеров фрейма.
        jfrm.setSize(240, 150);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание экземпляра
        // пользовательской панели содержимого.
        MyContentPanel mcp = new MyContentPanel();

        // Установка mcp в качестве панели содержимого.
        jfrm.setContentPane(mcp);

        // Отображение фрейма.
        jfrm.setVisible(true);
    }

    public static void main(String args[]) {
        // Фрейм создается в потоке обработки событий.
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new CustomCPDemo();
            }
        });
    }
}
```

14. Модифицируйте пример, демонстрирующий работу с разделяемой панелью, так, чтобы в левой части этой панели отображалась панель с прокруткой, созданная в проекте 4.1.

Одно из возможных решений данной задачи приведено ниже.

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

class SplitPaneDemo {

    SplitPaneDemo() {
        // Создание нового контейнера JFrame. По умолчанию
        // с этим контейнером связан диспетчер
        // компоновки BorderLayout.
        JFrame jfrm = new JFrame("Split Pane Demo");

        // Установка начальных размеров фрейма.
        jfrm.setSize(380, 150);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание метки для правой части.
        JLabel jlab2 = new
            JLabel(" Right side: ABCDEFGHIJKLMNOPQRSTUVWXYZ");

        // Установка минимальных размеров метки.

        jlab2.setMinimumSize(new Dimension(90, 30));

        // Создание панели с прокруткой.

        // Создание метки.
        JLabel jlabOptions = new
            JLabel("Select one or more options: ");

        // Создание флажков опций.
        JCheckBox jcbOpt1 = new JCheckBox("Option One");
        JCheckBox jcbOpt2 = new JCheckBox("Option Two");
        JCheckBox jcbOpt3 = new JCheckBox("Option Three");
        JCheckBox jcbOpt4 = new JCheckBox("Option Four");
        JCheckBox jcbOpt5 = new JCheckBox("Option Five");
```

```
// В данном примере обработчики событий не используются.

// Создание объекта JPanel, в котором будут
// содержаться флажки опций.
JPanel jpnl = new JPanel();
jpnl.setLayout(new GridLayout(6, 1));
jpnl.setOpaque(true);

// Включение флажков опций и метки в состав JPanel.
jpnl.add(jlabOptions);
jpnl.add(jcbOpt1);
jpnl.add(jcbOpt2);
jpnl.add(jcbOpt3);
jpnl.add(jcbOpt4);
jpnl.add(jcbOpt5);

// Создание панели с прокруткой.
JScrollPane jscrlp = new JScrollPane(jpnl);
jscrlp.setMinimumSize(new Dimension(140, 140));

// Создание разделяемой панели.
JSplitPane jsp = new JSplitPane(
    JSplitPane.HORIZONTAL_SPLIT, true, jscrlp, jlab2);

// Включение разделяемой панели
// в состав панели содержимого.
jfrm.getContentPane().add(jsp);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new SplitPaneDemo();
        }
    });
}
}
```

## **Модуль 5. Списки**

- 1. Объект `JList` реализует средства, позволяющие пользователю выбирать один или несколько пунктов списка. Да или нет?**

Да.

- 2. Что надо сделать, чтобы обеспечить прокрутку содержимого объекта `JList`?**

Надо включить этот объект в состав контейнера `JScrollPane`.

- 3. Событие какого типа генерирует компонент `JList` при выборе пользователем пункта списка? Какой метод вызывается для получения индекса, соответствующего первому из выбранных пунктов?**

Когда пользователь выбирает пункт списка `JList`, генерируется событие `ListSelectionEvent`. Чтобы получить индекс первого выбранного пункта, надо вызвать метод `getSelectedIndex()`.

- 4. Какую модель использует компонент `JList`?**

`ListModel`.

- 5. Какой компонент `Swing` реализует раскрывающийся список?**

`JComboBox`.

- 6. Какой метод позволяет получить выбранный пункт компонента `JComboBox`?**

`getSelectedItem()`.

- 7. Какой метод надо вызвать, чтобы создать раскрывающийся список, поддерживающий редактирование?**

`setEditable(true)`.

- 8. Можно ли динамически добавлять пункты в раскрывающийся список в процессе выполнения программы? Если да, то какой метод надо для этого использовать?**

Можно. Для этого надо использовать метод `addItem()`.

- 9. Можно ли раскрывать и сворачивать раскрывающийся список из программы? Если да, то какой метод надо для этого использовать?**

Да. Для этого надо использовать метод `setPopupVisible()`.

- 10. Какой класс `Swing` поддерживает инкрементный регулятор?**

`JSpinner`.

11. Какой метод инкрементного регулятора позволяет получить текущее значение? Какой метод надо вызвать, чтобы определить предыдущее значение?

Для получения текущего значения инкрементного регулятора надо обратиться к методу `getValue()`. Получить предыдущее значение позволяет метод `getPrevious()`.

12. Позволяет ли инкрементный регулятор управлять списком строк?

Да. Инкрементный регулятор дает возможность управлять как списком строк, так и списком объектов любого другого типа.

13. Напишите программу, в которой перебирались бы значения типа `double` в диапазоне от 0,0 до 9,9. Величину приращения установите равной 0,1.

Одно из возможных решений данной задачи приведено ниже.

// Перебор значений двойной точности.

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

class SpinDoubles {

    JSpinner jspin;
    JLabel jlab;

    SpinDoubles() {
        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Spin Doubles");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма.
        jfrm.setSize(160, 120);

        // Завершение программы при закрытии окна пользователем.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание модели инкрементного регулятора
        // для перебора значений типа double.
        SpinnerNumberModel spm =
            new SpinnerNumberModel(0.0, 0.0, 9.9, 0.1);
```

```

// Создание компонента JSpinner
// с использованием модели.
jspin = new JSpinner(spm);

// Установка предпочтительных размеров
// инкрементного регулятора.
jspin.setPreferredSize(new Dimension(40, 20));

// Создание метки для отображения значения,
// выбранного пользователем.
jlab = new JLabel(" Current value is: 0 ");

// Связывание с инкрементным регулятором
// обработчика событий изменения состояния.
jspin.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent ce) {
        // Получение текущего значения.
        Double val = (Double) jspin.getValue();

        // Вывод текущего значения.
        jlab.setText(" Current value is: " + val + " ");
    }
});

// Включение инкрементного регулятора и метки
// в состав панели содержимого.
jfrm.getContentPane().add(jspin);
jfrm.getContentPane().add(jlab);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new SpinDoubles();
        }
    });
}
}

```

14. Включите в программу `DynamicComboBox` компонент `JList`. Каждый раз, когда пользователь удаляет из раскрывающегося списка название сорта яблок, соответствующий пункт должен быть включен в состав `JList`. Таким образом, `JList` будет содержать удаленные записи.

Одно из возможных решений данной задачи имеет следующий вид:

```
// Включение элемента, удаленного из раскрывающегося списка,  
// в состав компонента JList.
```

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

```
class DynamicComboBox {
```

```
    JComboBox jcbb;  
    JLabel jlab;  
    JButton jbtnRemove;  
    JList jlst;
```

```
    // Создание массива сортов яблок.
```

```
    String apples[] = { "Winesap", "Cortland", "Red Delicious",  
                        "Golden Delicious", "Gala", "Fuji",  
                        "Granny Smith", "Jonathan" };
```

```
    DynamicComboBox() {
```

```
        // Создание нового контейнера JFrame.
```

```
        JFrame jfrm = new JFrame("Dynamic JComboBox");
```

```
        // Установка диспетчера компоновки FlowLayout.
```

```
        jfrm.getContentPane().setLayout(new FlowLayout());
```

```
        // Установка начальных размеров фрейма.
```

```
        jfrm.setSize(220, 240);
```

```
        // Завершение программы при закрытии окна пользователем.
```

```
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        // Создание компонента JComboBox.
```

```
        jcbb = new JComboBox(apples);
```

```
        // Разрешение редактирования содержимого JComboBox.
```

```
        jcbb.setEditable(true);
```



```
// Создание метки, предназначенной для отображения
// выбора пользователя.
jlab = new JLabel();

// Создание компонента JList и включение его
// в состав панели с прокруткой.
jlst = new JList(new DefaultListModel());
JScrollPane jscrlp = new JScrollPane(jlst);

// Установка предпочтительных размеров
// панели с прокруткой.
jscrlp.setPreferredSize(new Dimension(120, 90));

// Связывание обработчика событий с раскрывающимся
// списком. Новый пункт, введенный пользователем,
// добавляется к списку.
jcbb.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        // Получение ссылки на выбранный пункт.
        String item = (String) jcbb.getSelectedItem();

        // Если ни один пункт не выбран, никакие
        // действия не выполняются.
        if(item==null) return;

        // Отображение выбранного пункта.
        jlab.setText("Current selection: " + item);

        // Если пункт отсутствует в списке,
        // он включается в список.
        int i;

        // Проверка наличия пункта в списке.
        for(i=0; i < jcbb.getItemCount(); i++)
            if(item.equals(jcbb.getItemAt(i)))
                break; // Пункт в списке.

        // Если пункт отсутствует,
        // он включается в список.
        if(i==jcbb.getItemCount())
            jcbb.addItem(item);
    }
}
```

```
});

// Первоначально выбирается первый пункт списка.
jcbb.setSelectedIndex(0);

// Создание кнопки Remove Selection.
jbtnRemove = new JButton("Remove Selection");

// Связывание обработчика события действия с кнопкой.
jbtnRemove.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        // Получение ссылки на выбранный пункт.
        String item = (String) jcbb.getSelectedItem();

        // Если ни один пункт не выбран, никакие
        // действия не выполняются.
        if(item==null) return;

        // Удаление пункта.
        jcbb.removeItem(item);

        // Отображение выбранного пункта.
        jlab.setText("Removed " + item);

        // Добавление удаленных пунктов
        // к компоненту JList.
        DefaultListModel dlm =
            (DefaultListModel) jlst.getModel();
        dlm.addElement(item);
    }
});

// Включение компонентов в панель содержимого.
jfrm.getContentPane().add(jcbb);
jfrm.getContentPane().add(jlab);
jfrm.getContentPane().add(jbtnRemove);
jfrm.getContentPane().add(jscrlp);

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
```

```
// Фрейм создается в потоке обработки событий.  
SwingUtilities.invokeLater(new Runnable() {  
    public void run() {  
        new DynamicComboBox();  
    }  
});  
}  
}
```

## Модуль 6. Текстовые компоненты

1. Что такое `JTextComponent`? Можно ли создать экземпляр класса `JTextComponent`?

`JTextComponent` — это класс, потомками которого являются все текстовые компоненты `Swing`. Экземпляр класса `JTextComponent` создать невозможно, так как этот класс является абстрактным.

2. Назовите шесть текстовых компонентов, определенных в составе `Swing`.

`JTextField`, `JPasswordField`, `JFormattedTextField`, `JTextArea`, `JEditorPane` и `JTextPane`.

3. Какое событие генерируется при перемещении текстового курсора компонента `JTextField`? Какое событие будет сгенерировано, если, работая с компонентом `JTextField`, пользователь нажмет клавишу `<Enter>`?

При перемещении текстового курсора компонент `JTextField` генерирует событие `CaretEvent`. Если пользователь нажмет клавишу `<Enter>`, будет сгенерировано событие `ActionEvent`.

4. Какой компонент следует использовать для ввода пароля?

`JPasswordField`.

5. В чем состоит главное преимущество компонента `JFormattedTextField` по сравнению с `JTextField`?

Основное преимущество компонента `JFormattedTextField` по сравнению с `JTextField` состоит в том, что `JFormattedTextField` позволяет редактировать и выводить значения в требуемом формате.

6. Какую маску надо использовать для определения формата ввода телефонных номеров, которые имеют вид, подобный следующему:  
1 (555) 555-5555?

# (###) ###-####.

7. Назовите четыре политики потери фокуса ввода компонента `JFormattedTextField` и объясните их назначение.

Четыре политики потери фокуса ввода компонента `JFormattedTextField` перечислены ниже.

`COMMIT`

`REVERT`

`COMMIT_OR_REVERT`

`PERSIST`

По умолчанию принимается политика `COMMIT_OR_REVERT`. Согласно ей, корректное значение принимается в качестве текущего, а в случае некорректного значения происходит возврат к предыдущему значению. Политика `COMMIT` указывает на то, что допустимое значение должно быть принято, а недопустимое — оставаться в поле редактирования, не влияя на текущее значение. Согласно политике `REVERT`, должно быть извлечено и выведено предыдущее значение. Политика `PERSIST` задает использование текущих отредактированных данных, но значение не изменяется.

8. События какого типа генерируются при изменении значения поля редактирования с поддержкой формата?

`PropertyChangeEvent`.

9. Компонент `JTextArea` обычно включается в состав `JScrollPane`. Да или нет?

Да.

10. Какой метод надо вызвать, чтобы перевести текстовую область в режим автоматического перехода на новую строку?

`setLineWrap()`.

11. Какой метод надо вызвать, чтобы изменить в текстовой области размер табуляторов?

`setTabSize()`.

12. Можно ли вырезать и вставлять текст в текстовых компонентах? Можно ли делать это из программы? Если да, то какие методы следует использовать?

Да, в текстовых компонентах поддерживается вырезание и вставка текста, причем эти действия можно выполнять из программы. Для поддержки вырезания и вставки предусмотрены методы `cut()` и `paste()`. Также доступен метод `copy()`.

13. Добавьте в программе SimpleTextEditor, созданной в рамках проекта 6.1, возможность замены текста. Замене подлежит текст, заданный в поле редактирования Search For. Добавьте еще одно поле редактирования, в котором будет вводиться новая строка. Создайте также кнопку Replace, после каждого щелчка на которой будет заменяться одно вхождение исходной строки.

Одно из решений данной задачи приведено ниже.

```
// Модификация программы, созданной в рамках
// проекта 6.1. Модифицированная программа
// поддерживает поиск и замену.

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

class SimpleTextEditor {

    JLabel jlabMsg;

    JTextArea jta;

    JTextField jtffName;
    JTextField jtffFind;
    JTextField jtffReplace;

    JButton jbtnSave;
    JButton jbtnLoad;
    JButton jbtnFind;
    JButton jbtnFindNext;
    JButton jbtnReplace;

    int findIdx;

    public SimpleTextEditor() {

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("A Simple Text Editor");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());
```

```
// Установка начальных размеров фрейма.
jfrm.setSize(270, 480);

// Завершение программы при закрытии окна пользователем.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Создание метки для отображения сообщений.
jlabMsg = new JLabel();
jlabMsg.setPreferredSize(new Dimension(200, 30));
jlabMsg.setHorizontalAlignment(SwingConstants.CENTER);

// Создание метки без текста
// для выделения пустого пространства.
JLabel jlabSeparator = new JLabel();
jlabSeparator.setPreferredSize(new Dimension(200, 30));

// Создание меток Search For и Filename.
JLabel jlabFind = new JLabel("Search For:");
jlabFind.setPreferredSize(new Dimension(70, 20));
jlabFind.setHorizontalAlignment(SwingConstants.RIGHT);

JLabel jlabFilename = new JLabel("Filename:");
jlabFilename.setPreferredSize(new Dimension(70, 20));
jlabFilename.setHorizontalAlignment(
    SwingConstants.RIGHT);

// Создание метки Replace.
JLabel jlabReplace = new JLabel("Replace:");
jlabReplace.setPreferredSize(new Dimension(70, 20));
jlabReplace.setHorizontalAlignment(SwingConstants.RIGHT);

// Создание текстовой области.
jta = new JTextArea();

// Включение текстовой области
// в состав панели с прокруткой.
JScrollPane jscrlp = new JScrollPane(jta);
jscrlp.setPreferredSize(new Dimension(250, 200));

// Поле редактирования для ввода имени файла.
jtfFName = new JTextField(15);
```

```
// Связывание обработчика событий текстового курсора
// с текстовой областью. Обработчик отображает
// число символов в файле и обновляет эти данные при
// каждом изменении положения текстового курсора.
// В переменную findIdx записывается текущая позиция
// текстового курсора.
jta.addCaretListener(new CaretListener() {
    public void caretUpdate(CaretEvent ce) {
        String str = jta.getText();
        jlabMsg.setText("Current size: " + str.length());
        findIdx = jta.getCaretPosition();
    }
});

// Создание кнопок Save File и Load File.
jbtnSave = new JButton("Save File");
jbtnLoad = new JButton("Load File");

// Связывание обработчика событий с кнопкой Save File.
jbtnSave.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        save();
    }
});

// Связывание обработчика событий с кнопкой Load File.
jbtnLoad.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
        load();
    }
});

// Создание поля редактирования Search For.
jtfFind = new JTextField(15);

// Создание поля редактирования Replace.
jtfReplace = new JTextField(15);

// Создание кнопок Find From Top и Find Next.
jbtnFind = new JButton("Find From Top");
jbtnFindNext = new JButton("Find Next");

// Связывание обработчика событий
```

## А Ответы на вопросы для самоконтроля



```
jfrm.setVisible(true);
}

// Метод, предназначенный для сохранения файла.
void save() {
    FileWriter fw;

    // Извлечение имени файла из поля редактирования.
    String fname = jtffName.getText();

    // Проверка наличия имени.
    if(fname.length() == 0) {
        jlabMsg.setText("No filename present.");
        return;
    }

    // Сохранение файла.
    try {
        fw = new FileWriter(fname);
        jta.write(fw);
        fw.close();
    } catch(IOException exc) {
        jlabMsg.setText("Error opening or writing file.");
        return;
    }

    jlabMsg.setText("File written successfully.");
}

// Метод, предназначенный для загрузки файла.
void load() {
    FileReader fw;

    // Извлечение имени файла из поля редактирования.
    String fname = jtffName.getText();

    // Проверка наличия имени файла.
    if(fname.length() == 0) {
        jlabMsg.setText("No filename present.");
        return;
    }

    // Загрузка файла.
```

```
try {
    fw = new FileReader(fname);
    jta.read(fw, null);
    fw.close();
} catch(IOException exc) {
    jlabMsg.setText("Error opening or reading file.");
    return;
}

// При загрузке нового файла текущая позиция обнуляется.
findIdx = 0;
jlabMsg.setText("File loaded successfully.");
}

// Поиск текста.
void find(int start) {
    // Получение текущего текста в виде строки.
    String str = jta.getText();

    // Извлечение строки поиска.
    String findStr = jtfFind.getText();

    // Поиск первого вхождения указанной строки
    int idx = str.indexOf(findStr, start);

    // Проверка, найдено ли соответствие.
    if(idx > -1) {
        // Если поиск завершен успешно, текстовый
        // курсор помещается в позицию, соответствующую
        // найденному тексту.
        jta.setCaretPosition(idx);
        findIdx = idx;
        jlabMsg.setText("String found.");
    }
    else
        jlabMsg.setText("String not found.");

    // Передача фокуса ввода окну редактора.
    jta.requestFocusInWindow();
}

// Замена строки.
void replace(int start) {
```

```
// Получение текущего текста в виде строки.
String str = jta.getText();

// Получение заменяемой строки.
String findStr = jtffind.getText();

// Получение заменяющей строки.
String repString = jtffReplace.getText();

// Поиск первого вхождения указанной строки.
int idx = str.indexOf(findStr, start);

// Проверка, найдена ли строка.
if(idx > -1) {
    // Если поиск завершен успешно, текстовый
    // курсор помещается в позицию, соответствующую
    // найденному тексту.
    jta.setCaretPosition(idx);
    findIdx = idx;

    // Замена текста.
    jta.replaceRange(repString, findIdx,
        findIdx+findStr.length());

    jlabMsg.setText("String replaced.");
}
else
    jlabMsg.setText("String not found.");

// Передача фокуса ввода окну редактора.
jta.requestFocusInWindow();
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new SimpleTextEditor();
        }
    });
}
```

14. Измените код программы SimpleTextEditor так, чтобы вместо счетчика символов отображался счетчик строк.

Чтобы в программе SimpleTextEditor вместо счетчика символов отображался счетчик строк, надо изменить обработчик событий текстового курсора так, как показано ниже.

```
jta.addCaretListener(new CaretListener() {  
    public void caretUpdate(CaretEvent ce) {  
        jlabMsg.setText("Line count: " + jta.getLineCount());  
        findIdx = jta.getCaretPosition();  
    }  
});
```

15. В программе SimpleTextEditor задайте комбинации клавиш для активизации кнопок Find From Top и Find Next. Для Find From Top это должна быть комбинация <Alt+F>, а для Find Next — <Alt+N>.

Для того чтобы решить данную задачу, надо добавить к коду программы следующие выражения:

```
jbtnFind.setMnemonic('F');  
jbtnFindNext.setMnemonic('N');  
jbtnFindNext.setDisplayedMnemonicIndex(5);
```

## Модуль 7. Меню

1. Назовите основные классы Swing, используемые для создания меню.

JMenu, JMenuItem и JMenuBar.

2. С помощью какого класса создается меню? Какой класс позволяет создать контекстное меню? Какой класс следует использовать для создания строки меню?

Класс JMenu создает меню, JPopupMenu создает контекстное меню, JMenuBar создает строку меню.

3. Какое событие генерируется при выборе пользователем пункта меню?

При выборе пользователем пункта меню генерируется событие действия.

4. В состав пунктов меню нельзя включать изображения. Да или нет?

Утверждение неверно, изображения допустимы в составе меню.

5. Какой метод включает строку меню в окно?

setJMenuBar().

6. С помощью какого метода можно определить мнемоническое обозначение

ние для пункта меню?

`setMnemonic()`.

7. Можно ли использовать пиктограмму в качестве пункта меню? Если да, запрещает ли она использовать имя пункта?

Да, в качестве пункта меню может быть применена пиктограмма. Она не запрещает использовать имя пункта.

8. Какой класс создает пункт меню в виде кнопки переключателя опций?

`JRadioButtonMenuItem`.

9. В составе меню могут присутствовать флажки опций, однако они ухудшают интерфейс, так как при этом меню выглядит непривычно для пользователя. Да или нет?

Нет. Флажки опций — стандартная разновидность пунктов меню. Они часто встречаются в составе меню.

10. Контекстное меню обычно вызывается \_\_\_\_\_ мыши.

Щелчком правой кнопкой.

11. Какой метод, объявленный в интерфейсе `MouseListener`, следует переопределить для вызова контекстного меню? Какой метод класса `MouseEvent` следует вызвать, чтобы определить, следует ли отображать контекстное меню?

Для поддержки контекстного меню в обработчике событий надо переопределить методы `mousePressed()` и `mouseReleased()`. Чтобы определить, следует ли отображать контекстное меню, надо вызвать метод `isPopupTrigger()`.

12. Какой метод надо вызвать, чтобы получить ссылку на компонент, вызывающий контекстное меню?

`getComponent()`.

13. Экземпляр какого класса реализует панель инструментов?

`JToolBar`.

14. Можно ли связать панель инструментов со строкой меню?

Нет.

15. Какой интерфейс используется для определения действия?

`Action`.

**16. Перечислите свойства, определенные посредством действия?**

Посредством действия определены следующие свойства.

- Клавиша быстрого доступа.
- Клавиша, соответствующая мнемоническому обозначению.
- Имя.
- Пиктограмма.
- Строка подсказки.
- Строка команды действия.
- Состояние (разрешение/запрет доступа).

**17. Какие действия выполняет метод `System.exit()`?**

Метод `System.exit()` завершает программу.

## Модуль 8. Таблицы и деревья

1. Для обеспечения работы `JTable` используются несколько классов и интерфейсов. В каком пакете они находятся?

`javax.swing.table.`

2. Какой метод позволяет задать размеры прокручиваемой области просмотра для таблицы?

`setPreferredScrollableViewportSize()`.

3. Как называется модель, определяющая таблицу?

`TableModel.`

4. Если компонент `JTable` не помещен в панель с прокруткой, для отображения заголовка таблицы необходимо предпринять специальные действия. Да или нет?

Да.

5. Предположим, вы хотите организовать обработку событий выбора для таблицы. С каким объектом надо связать обработчик `ListSelectionListener`?

С моделью столбца таблицы.

6. Для обработки событий модели таблицы надо зарегистрировать обработчик \_\_\_\_\_.

`TableModelListener.`

7. Какие действия выполняет метод `setValueAt()` для модели `TableModel`?

Метод `setValueAt()` устанавливает значение ячейки, заданной посредством номера строки и номера столбца. Координаты ячейки указываются относительно модели, а не относительно представления.

8. Какой метод класса `JTable` изменяет режим автоматического изменения размеров?

`setAutoResizeMode()`.

9. Какой класс следует расширить для того, чтобы реализовать модель таблицы, определяемую разработчиком?

`AbstractTableModel`.

10. Какой интерфейс определяет объект воспроизведения для таблицы?

`TableCellRenderer`.

11. Компонент `JTree` применим только для отображения бинарных деревьев. Да или нет?

Нет. С помощью компонента `JTree` можно представлять любые данные, имеющие иерархическую структуру.

12. Что такое лист?

Лист — это узел дерева, не имеющий дочерних узлов.

13. Какие действия надо выполнить для создания дерева из объектов `DefaultMutableTreeNode`?

В первую очередь надо создать корневой узел. Затем следует добавить к нему дочерние узлы, используя метод `add()`. Дочерние узлы могут выступать в качестве корневых узлов для поддеревьев. Таким образом можно создать сколь угодно сложное дерево.

14. Какое событие возникает при изменении данных в составе дерева?

`TreeModelEvent`.

15. Какой метод надо вызвать для формирования объекта `Enumeration`, содержащего узлы дерева, посредством обхода в глубину в обратном порядке?

`postorderEnumeration()` (можно также использовать метод `depthFirstEnumeration()`).

16. Создайте программу, которая выводит таблицу, помогающую рассчитывать стоимость товара. В первом столбце должна отображаться цена изделий с шагом 1. В последующих трех столбцах отображаются вели-

чины 10, 20 и 30% от этой суммы. В последнем столбце пользователь должен иметь возможность ввести свое предложение. Используйте объект воспроизведения, отображающий значения в формате денежных единиц. Предусмотрите в программе возможность указания числа строк при создании таблицы. Готовая таблица должна выглядеть так, как показано ниже.

Одно из возможных решений имеет следующий вид:

| Price   | 10% Tip | 15% Tip | 20% Tip | Suggested |
|---------|---------|---------|---------|-----------|
| \$47.00 | \$4.70  | \$7.05  | \$9.40  |           |
| \$48.00 | \$4.80  | \$7.20  | \$9.60  |           |
| \$49.00 | \$4.90  | \$7.35  | \$9.80  | \$8.00    |
| \$50.00 | \$5.00  | \$7.50  | \$10.00 |           |
| \$51.00 | \$5.10  | \$7.65  | \$10.20 |           |

New suggested tip: \$8.00

// Программа, отображающая заданную часть стоимости  
 // товара в виде таблицы. В программе применяется  
 // объект воспроизведения, определяемый разработчиком.  
 // С его помощью данные выводятся в формате денежных  
 // единиц. Кроме того, пользователь может включать  
 // в состав таблицы произвольные значения.

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

// Простой объект воспроизведения для вывода данных
// в формате денежных единиц.
class MoneyRenderer extends DefaultTableCellRenderer {
    public Component getTableCellRendererComponent(
       .JTable jtab, Object v,
        boolean selected, boolean focus,
        int r, int c) {

        // Получение компонента по умолчанию.
        JLabel rendComp = (JLabel)
            super.getTableCellRendererComponent(
                jtab, v, selected, focus, r, c);
```



```
// Получение формата для вывода денежных единиц.
NumberFormat nf = NumberFormat.getCurrencyInstance();

// Если значение не равно 0, оно форматируется
// для представления денежных единиц. В противном
// случае данные не отображаются.
if(((Double) v).doubleValue() != 0.0)
    rendComp.setText(nf.format(v));
else
    rendComp.setText("");

return rendComp;
}
}

// Модель таблицы, определяемая разработчиком.
class TipModel extends AbstractTableModel {
    int numRows;

    String colNames[] = { "Price", "10% Tip",
                          "15% Tip", "20% Tip",
                          "Suggested" };

    // Массив для хранения данных, введенных пользователем.
    double[] other;

    // Число строк для отображения передается с помощью
    // параметра len.
    TipModel(int len) {
        super();
        numRows = len;
        other = new double[numRows];
    }

    public int getRowCount() { return numRows; }
    public int getColumnCount() { return 5; }

    public String getColumnName(int c) {
        return colNames[c];
    }
}

// Данный метод возвращает цену для первого столбца
```

```
// и определенный процент цены
// для остальных трех столбцов.
public Object getValueAt(int r, int c) {
    if(c==0) return new Double(r+1);
    else if(c>0 & c<4)
        return new Double((r+1) * ((c+1)*0.05));
    else
        return new Double(other[r]);
}

// Данный метод устанавливает значение в столбце
// Suggested (в случае, если это значение было
// введено пользователем). Он также генерирует
// событие модели таблицы, оповещающее о том, что
// данные изменились.
public void setValueAt(Object v, int r, int c) {
    if(c==4) {
        other[r] = ((Double) v).doubleValue();
        fireTableCellUpdated(r, c);
    }
}

// Данный метод возвращает значение false
// для всех столбцов, за исключением столбца 4,
// который редактируется пользователем.
public boolean isCellEditable(int r, int c) {
    if(c==4) return true;
    return false;
}

// Данный метод возвращает класс Double
// для всех столбцов.
public Class getColumnClass(int c) {
    return Double.class;
}

// Отображение таблицы, содержащей цену и
// указанный процент от нее.
class TipTable {

    JTable jtabTip;
    JLabel jlab;
```

```
TipTable(int size) {
    // Создание нового контейнера JFrame.
    JFrame jfrm = new JFrame("A Tip Calculator");

    // Установка диспетчера компоновки FlowLayout.
    jfrm.getContentPane().setLayout(new FlowLayout());

    // Установка начальных размеров фрейма.
    jfrm.setSize(500, 200);

    // Завершение программы при закрытии окна пользователем.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Создание метки для отображения предлагаемого варианта.
    jlab = new JLabel();

    // Создание таблицы, отображающей данные.
    jtabTip = new JTable(new TipModel(size));

    // Включение таблицы в состав панели с прокруткой.
    JScrollPane jscrlp = new JScrollPane(jtabTip);

    // Установка размеров области просмотра.
    jtabTip.setPreferredScrollableViewportSize(
        new Dimension(450, 80));

    // Установка объекта воспроизведения, определенного
    // разработчиком.
    jtabTip.setDefaultRenderer(Double.class,
        new MoneyRenderer());

    // Получение модели выбора.
    ListSelectionModel lsm = jtabTip.getSelectionModel();
    lsm.setSelectionMode(
        ListSelectionModel.SINGLE_SELECTION);

    // Получение модели таблицы.
    TableModel tm = jtabTip.getModel();

    // Связывание обработчика событий с моделью таблицы.
    tm.addTableModelListener(new TableModelListener() {
        // Отображение предлагаемых значений.
```

```

        public void tableChanged(TableModelEvent tme) {
            if(tme.getType() == TableModelEvent.UPDATE) {
                jlab.setText("New suggested tip: " +
                    NumberFormat.getCurrencyInstance().format(
                        jtabTip.getModel().getValueAt(
                            tme.getFirstRow(),
                            tme.getColumn()))
                );
            }
        }
    });

    // Включение таблицы в состав панели содержимого.
    jfrm.getContentPane().add(jscrlp);
    jfrm.getContentPane().add(jlab);

    // Отображение фрейма.
    jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new TipTable(100);
        }
    });
}
}

```

17. Какое выражение надо включить в программу TreeDemo, чтобы корневой узел не отображался на экране?

```
jtree.setRootVisible(false);
```

18. Измените проект 8.2 так, чтобы программа при посещении узла сообщала, является ли он листом или имеет дочерние узлы.

Решение данной задачи выглядит следующим образом:

```
TreeNode tn;
```

```
Enumeration preorder = root.preorderEnumeration();
```

```
while(preorder.hasMoreElements()) {
```

```
tn = (TreeNode) preorder.nextElement();
if(tn.isLeaf()) System.out.println(tn + " is leaf");
else System.out.println(tn + " has children");
}
```

## Модуль 9. Диалоговые окна

1. Диалоговое окно содержит два или более компонентов. Оно отображает сообщение для пользователя и ожидает ответа. Да или нет?

Да.

2. Какие классы, предназначенные для создания диалоговых окон, содержатся в составе Swing? Назовите два класса общего назначения.

JDialog, JOptionPane, JFileChooser и JColorChooser. Первые два из них являются классами общего назначения.

3. Какой из методов класса JOptionPane создает диалоговое окно для ввода данных? Какой из них создает диалоговое окно с сообщением?

Метод `showInputDialog()` создает диалоговое окно для ввода данных. Для создания окна с сообщением используется метод `showMessageDialog()`.

4. Какой метод класса JOptionPane обычно используется для создания диалогового окна, посредством которого запрашивается согласие пользователя на сохранение изменений, внесенных в документ? Как должен выглядеть вызов этого метода?

Для этой цели обычно используется метод `showConfirmDialog()`, определенный в классе JOptionPane. Пример вызова этого метода приведен ниже.

```
showConfirmDialog(null, "Save Changes?");
```

5. Предположим, вы используете диалоговое окно для получения подтверждения от пользователя. Какое возвращаемое значение указывает на то, что пользователь щелкнул на кнопке Yes?

YES\_OPTION.

6. Какое значение указывает на то, что в диалоговом окне, предназначенном для получения подтверждения от пользователя, должны отображаться только кнопки Yes и No?

YES\_NO\_OPTION.

7. Предположим, вы хотите получить от пользователя ответ в виде строки. Какой метод класса JOptionPane следует для этого использовать?

`showInputDialog()`.

8. Обязательно ли параметр метода `show...()` класса `JOptionPane`, определяющий сообщение, должен представлять собой строку символов? Обоснуйте свой ответ.

Нет. Параметр метода `show...()` класса `JOptionPane`, определяющий сообщение, может ссылаться как на строку, так и на объект любого другого типа.

9. `JDialog` является контейнером верхнего уровня. Да или нет?

Да.

10. Какие четыре действия надо выполнить для создания и отображения диалогового окна на основе класса `JDialog`?

- Создать объект `JDialog`.
- Установить диспетчер компоновки, размер и, возможно, политику закрытия диалогового окна.
- Включить компоненты в состав панели содержимого диалогового окна.
- Отобразить диалоговое окно, вызвав метод `setVisible(true)`.

11. Можно ли с помощью класса `JDialog` создать немодальное диалоговое окно?

Да.

12. Чем отличаются вызовы методов `setVisible(false)` и `dispose()` для диалогового окна?

В результате вызова `setVisible(false)` лишь запрещается отображение диалогового окна на экране. Метод `dispose()` удаляет окно, а также освобождает все используемые им ресурсы.

13. Какой метод класса `JFileChooser` создает диалоговое окно **Save** для выбора файла? Какой метод позволяет явно указывать заголовок создаваемого диалогового окна?

Создать диалоговое окно **Save** для выбора файла позволяет метод `showSaveDialog()`. Метод `showDialog()` дает возможность задать произвольный заголовок.

14. Какие два метода надо переопределить при реализации фильтра (`FileFilter`) для `JFileChooser`?

`accept()` и `getDescription()`.

15. Можно ли с помощью окна выбора файла выбирать каталоги? Если да, то как это сделать?

Да. Для того чтобы разрешить выбор каталога, надо вызвать метод `setFileSelectionMode()`.

16. Значения какого типа возвращает метод `showDialog()` класса `JColorChooser`?

Ссылку на объект `Color`.

17. Модуль 7 был посвящен меню. В нем рассматривался пример создания меню `File`, содержащего пункт `Exit`. Обработчик событий действия, соответствующий данному пункту, приведен ниже.

```
// Обработка событий действия для пункта меню.
public void actionPerformed(ActionEvent ae) {
    // Получение команды действия, соответствующей
    // выбранному пункту.
    String comStr = ae.getActionCommand();

    // Если пользователь выбрал пункт Exit,
    // программа завершает выполнение.
    if(comStr.equals("Exit")) System.exit(0);
    ...
}
```

Измените данный код так, чтобы при выборе пункта `Exit` отображалось диалоговое окно, запрашивающее у пользователя разрешение на завершение программы.

Для обработки пункта меню `Exit` можно использовать следующий фрагмент кода:

```
// Если пользователь выбрал пункт Exit,
// программа завершает выполнение.
if(comStr.equals("Exit")) {
    int response = JOptionPane.showConfirmDialog(
        null,
        "Exit Now?",
        "Terminate Program",
        JOptionPane.YES_NO_OPTION);

    if(response == JOptionPane.YES_OPTION) {
        System.exit(0);
    }
}
```

## Модуль 10. Поток, апплеты, рисование и компоновка

1. Код, взаимодействующий с компонентами Swing, должен выполняться в потоке \_\_\_\_\_.

Обработки событий.

2. Предположим, что в программе используется класс `javax.swing.Timer`. Какое событие генерируется таймером?

`ActionEvent`.

3. Какие методы используются для запуска и остановки таймера, реализуемого классом `javax.swing.Timer`?

`start()` и `stop()`.

4. Назовите четыре метода, определяющих жизненный цикл апплета. Когда вызываются эти методы?

Жизненный цикл апплета определяют методы `init()`, `start()`, `stop()` и `destroy()`. Метод `init()` вызывается первым. Он используется для инициализации апплета. Метод `start()` вызывается для того, чтобы начать (или возобновить) работу апплета. Обращение к данному методу осуществляется тогда, когда надо организовать отображение апплета в составе документа. Метод `stop()` вызывается для остановки апплета. Необходимость в остановке может возникнуть тогда, когда от страницы, содержащей апплет, пользователь переходит к другому документу. Метод `destroy()` вызывается тогда, когда выполнение апплета должно быть завершено и апплет удален из памяти.

5. Какой метод следует применять при создании пользовательского интерфейса апплета?

`invokeAndWait()`.

6. Можно ли в программе организовать рисование на поверхности компонента?

Да.

7. Какие три метода, связанные с рисованием, вызываются при отображении компонента Swing?

`paintComponent()`, `paintBorder()` и `paintChildren()`.

8. К какому методу следует обратиться, чтобы вызвать перерисовку содержимого компонента?

`repaint()`.



9. Какие методы позволяют определить текущую ширину и высоту компонента?

`getWidth()` и `getHeight()`.

10. Диспетчер компоновки реализует объединение небольших \_\_\_\_\_.  
Таблиц.

11. Каждому компоненту в контейнере, связанном с диспетчером компоновки `GridBagLayout`, ставится в соответствие набор ограничений. В каком классе инкапсулированы эти ограничения?

`GridBagConstraints`.

12. Почему в ряде случаев диспетчер компоновки `BoxLayout` оказывается более эффективным, чем `GridBagLayout`?

Программа, использующая `BoxLayout`, оказывается короче; кроме того, данный диспетчер компоновки проще в настройке, чем `GridBagLayout`.

13. Может ли в горизонтальном блоке, созданном с помощью класса `Box`, содержаться больше одной строки компонентов?

Нет. В нем может быть только одна строка компонентов.

14. Модифицируйте программу, отображающую гистограмму, так, чтобы цвет обрамления изменялся каждую секунду. В разные моменты времени должна отображаться красная, зеленая и синяя рамка. Кроме того, обеспечьте изменение размера рамки с одного на пять пикселей и наоборот через одну пятую секунды. Для решения этой задачи используйте класс `javax.swing.Timer`.

Одно из возможных решений данной задачи выглядит следующим образом:

// Ответ на вопрос 14.

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

```
// Данный класс является подклассом JPanel. В нем
// переопределен метод paintComponent(), с помощью
// которого в составе панели отображаются данные,
// сгенерированные случайным образом.
class PaintPanel extends JPanel {
```

```
Insets ins; // Размеры обрамления панели.

Random rand; // Используется для генерации
               // псевдослучайных значений.

PaintPanel(int w, int h) {

    // Панель должна быть непрозрачной.
    setOpaque(true);

    // Вначале рамка имеет вид линии красного цвета.
    setBorder(
        BorderFactory.createLineBorder(Color.RED, 1));

    // Установка предпочтительных размеров.
    setPreferredSize(new Dimension(w, h));

    rand = new Random();
}

// Переопределение метода paintComponent().
protected void paintComponent(Graphics g) {
    // Тело метода начинается с вызова метода суперкласса.
    super.paintComponent(g);

    // Определение высоты и ширины компонента.
    int height = getHeight();
    int width = getWidth();

    // Получение размеров обрамления.
    ins = getInsets();

    // В составе панели выводятся псевдослучайные значения,
    // представленные в виде гистограммы.
    for(int i=ins.left+5; i <= width-ins.right-5; i += 4) {
        // Получение псевдослучайного значения в интервале от 0
        // до максимальной высоты области отображения.
        int h = rand.nextInt(height-ins.bottom);

        // Коррекция значения, располагающегося слишком
        // близко к рамке.
        if(h <= ins.top) h = ins.top+1;
    }
}
```

```
// Вывод линии, представляющей значение.
g.drawLine(i, height-ins.bottom, i, h);
}
}

// Изменение размера и цвета рамки.
public void changeBorder(Color color, int size) {
    setBorder(
        BorderFactory.createLineBorder(color, size));
}
}

// Демонстрация рисования на поверхности панели.
class PaintDemo {

    JButton jbtnMore;
    JButton jbtnSize;
    JLabel jlab;
    JPanel pp;

    int bsize = 1; // Текущий размер рамки.
    int cidx = 0; // Текущий цвет панели.

    // Массив значений цвета для перебора,
    Color colors[] = {Color.RED, Color.BLUE, Color.GREEN };

    boolean big; // Используется для изменения размеров панели.

    Timer borderSizeT; // Таймер, управляющий изменением
                      // размеров обрамления.
    Timer borderColorT; // Таймер, управляющий изменением
                      // цвета обрамления.

    PaintDemo() {

        // Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame("Painting Demo");

        // Установка диспетчера компоновки FlowLayout.
        jfrm.getContentPane().setLayout(new FlowLayout());

        // Установка начальных размеров фрейма,
        jfrm.setSize(240, 260);
```

```
// Завершение программы при закрытии окна пользователем.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Создание панели для вывода данных.
pp = new JPanel(100, 100);

// Создание кнопок.
jbtnMore = new JButton("Show More Data");
jbtnSize = new JButton("Change Border Size");

// Описание графика.
jlab = new JLabel("Bar Graph of Random Data");

// Изменение размеров обрамления.
ActionListener borderSize = new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        pp.changeBorder(colors[cidx], bsize);
        bsize++;
        if(bsize > 5) bsize = 1;
    }
};

// Изменение цвета обрамления.
ActionListener borderColor = new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        pp.changeBorder(colors[cidx], bsize);
        cidx++;
        if(cidx == 3) cidx = 0;
    }
};

// Создание и запуск таймеров.
borderSizeT = new Timer(200, borderSize);
borderColorT = new Timer(1000, borderColor);

borderSizeT.start();
borderColorT.start();

// Перерисовка панели по щелчку на кнопке
// Show More Data.
jbtnMore.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
```

```

        pp.repaint();
    }
});

// Установка размеров обрамления по щелчку
// на кнопке Change Border Size.
// Изменение размеров обрамления автоматически
// приводит к перерисовке компонента.
jbtnSize.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        if(!big) bsize = 5;
        else bsize = 1;
        pp.changeBorder(colors[cidx], bsize);
        big = !big;
    }
});

// Включение кнопок, метки и панели
// в состав панели содержимого.
jfrm.getContentPane().add(jlab);
jfrm.getContentPane().add(pp);
jfrm.getContentPane().add(jbtnMore);
jfrm.getContentPane().add(jbtnSize);

big = false;

// Отображение фрейма.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Фрейм создается в потоке обработки событий.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new PaintDemo();
        }
    });
}
}

```

15. Модифицируйте код апплета в проекте 1.10 так, чтобы направление перемещения текста периодически изменялось. Например, пусть текст некоторое время перемещается влево, затем вправо и т.д. Время, через которое

направление перемещения будет изменяться, выберите сами. Программа, приведенная в приложении к данной книге, изменяет направление каждые 20 секунд.

Одно из возможных решений данной задачи приведено ниже.

```
// Ответ на вопрос 15.
```

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

```
/*
```

HTML-код, используемый для загрузки данного апплета,  
имеет следующий вид:

```
<object code="ScrollText" width=240 height=100>
</object>
```

```
*/
```

```
public class ScrollText extends JApplet {
```

```
    JLabel jlab;
```

```
    String msg = " Swing makes the GUI move! ";
```

```
    ActionListener scroller;
```

```
    Timer stTimer; // Таймер, определяющий
                   // скорость перемещения.
```

```
    int counter;   // Таймер, используемый для изменения
                   // направления движения.
```

```
    // Значение, определяющее, в какой момент направление
    // движения должно измениться.
```

```
    int scrollLimit;
```

```
    // Инициализация апплета.
```

```
    public void init() {
```

```
        counter = 0;
```

```
        scrollLimit = 100;
```

```
try {
    SwingUtilities.invokeLaterAndWait(new Runnable () {
        public void run() {
            guiInit();
        }
    });
} catch (Exception exc) {
    System.out.println("Can't create because of "+ exc);
}

}

// Активизация таймера при запуске апплета.
public void start() {
    stTimer.start();
}

// Остановка таймера при остановке апплета.
public void stop() {
    stTimer.stop();
}

// Остановка таймера при удалении апплета.
public void destroy() {
    stTimer.stop();
}

// Инициализация интерфейса.
private void guiInit() {

    // Создание метки для перемещения сообщения.
    jlab = new JLabel(msg);
    jlab.setHorizontalAlignment(SwingConstants.CENTER);

    // Создание обработчика событий для таймера.
    // Данный вариант обработчика изменяет направление
    // перемещения каждые 20 секунд.
    scroller = new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            if(counter < scrollLimit) {
                // Перемещение сообщения влево на один символ.
                char ch = msg.charAt(0);
                msg = msg.substring(1, msg.length());
                msg += ch;
            }
        }
    };
}
```

```
    } else {  
        // Перемещение сообщения вправо на один символ.  
        char ch = msg.charAt(msg.length()-1);  
        msg = msg.substring(0, msg.length()-1);  
        msg = ch + msg;  
        if(counter == scrollLimit*2) counter = 0;  
    }  
    counter++;  
    jlab.setText(msg);  
}  
};  
  
// Создание таймера.  
stTimer = new Timer(200, scroller);  
  
// Включение метки в состав панели содержимого апплета.  
getContentPane().add(jlab);  
}  
}
```





# Предметный указатель

## A

Abstract Window Toolkit, 23  
Appletviewer, 586  
AWT, 23

## G

Graphical User Interface, 22  
GUI, 22

## H

HTML-код в составе метки, 95

## J

Java Foundation Classes, 23  
JFC, 23

## M

Model-View-Controller, 26  
MVC, 26

## A

автоматическое изменение размеров, 462  
апплет, 585; 586  
архитектура модель-представитель, 26  
архитектура модель-представлениеконтроллер, 26  
архитектура с выделенной моделью, 26

## B

ведущий пункт, 276  
ветвь, 482  
визуальный компонент, 25  
вкладка, 239  
восточная область, 69  
вспомогательный маркер, 173  
выбор столбцов, 444  
выбор строк, 443  
выбор цвета, 564  
выбор ячеек, 444  
выделенный текст, 321  
выравнивание содержимого метки по

вертикали, 79

выравнивание содержимого метки по  
горизонтали, 79  
графический контекст, 600

## D

двойная буферизация, 212  
действие, 415  
дерево, 434; 481  
диалоговое окно, 504  
диспетчер компоновки, 69  
BorderLayout, 69  
BoxLayout, 69; 617  
FlowLayout, 69  
GridBagLayout, 69; 609  
GridLayout, 69; 84  
SpringLayout, 69

## Ж

жесткая область, 618

## З

заголовок столбца, 225; 228  
заголовок строки, 225; 228  
западная область, 69

## И

иерархия контейнеров, 27  
изображение в составе метки, 85  
изображение на кнопке, 104  
изображение по умолчанию, 126  
инвертирование диапазона, 179  
индикатор хода процесса, 202  
индикатор хода процесса, 152  
инкрементный регулятор, 301  
интерфейс  
Action, 415  
ActionListener, 39; 40; 54  
Adjustable, 155  
AdjustmentListener, 40; 159  
AncestorListener, 40  
Border, 74

BoundedRangeModel, 152; 155  
 ButtonModel, 100  
 Caret, 321  
 CaretListener, 40; 323  
 ChangeListener, 40; 102  
 ComboBoxModel, 289  
 FocusListener, 40  
 HyperlinkListener, 40  
 ItemListener, 40; 101  
 ItemSelectable, 102; 375; 382  
 KeyListener, 40  
 LayoutManager, 69  
 ListCellRenderer, 277  
 ListDataListener, 40  
 ListModel, 264  
 ListSelectionListener, 41; 446; 450  
 ListSelectionModel, 264; 435  
 MenuDragMouseListener, 376  
 MenuElement, 376  
 MenuKeyListener, 376  
 MenuListener, 41  
 MouseListener, 40; 405  
 MouseMotionListener, 40  
 MouseWheelListener, 40  
 MutableComboBoxModel, 289  
 MutableTreeNode, 484  
 PopupMenuListener, 377  
 PropertyChangeListener, 341  
 Runnable, 574  
 Scrollable, 234  
 ScrollPaneConstants, 233  
 SingleSelectionModel, 376  
 SpinnerModel, 301  
 TableCellEditor, 473  
 TableColumnModel, 435  
 TableModel, 435  
 TableModelListener, 41; 456  
 TreeExpansionListener, 41; 490  
 TreeModel, 482  
 TreeModelListener, 41  
 TreeNode, 484  
 TreeSelectionListener, 41  
 TreeSelectionModel, 482  
 WindowListener, 40  
 ButtonModel, 114

исключение HeadlessException, 508

источник события, 37; 38

## **К**

клавиша быстрого доступа, 388

клавиша быстрого доступа, 374

класс

AbstractButton, 98; 104; 376  
 AbstractSpinnerModel, 302  
 AbstractTableModel, 467  
 ActionEvent, 40; 43  
 AdjustmentEvent, 40; 159  
 AncestorEvent, 40  
 Applet, 586  
 AWTEvent, 39  
 BorderFactory, 74  
 ButtonGroup, 135  
 CaretEvent, 40  
 ChangeEvent, 40; 102  
 Component, 28  
 Container, 28  
 DefaultBoundedRangeModel, 154  
 DefaultButtonModel, 100  
 DefaultCaret, 321  
 DefaultComboBoxModel, 289  
 DefaultListModel, 264  
 DefaultListSelectionModel, 264  
 DefaultMutableTreeNode, 497  
 DefaultMutableTreeNode, 484  
 DefaultSingleSelectionModel, 376  
 DefaultTreeModel, 482  
 EventObject, 39  
 FileFilter, 546  
 FocusAdapter, 42  
 FocusEvent, 40  
 GridBagConstraints, 609  
 HyperlinkEvent, 40  
 Icon, 85  
 ImageIcon, 86  
 InputEvent, 389  
 ItemEvent, 40; 101; 102  
 JApplet, 28; 586  
 JButton, 43  
 JButton, 38; 97; 103  
 JCheckBox, 97; 120; 127  
 JCheckBoxMenuItem, 375; 401  
 JColorChooser, 504; 565  
 JComboBox, 264; 288  
 JComponent, 27  
 JDialog, 531

JDialog, 28; 504  
JEditorPane, 320  
JFileChooser, 504; 540  
JFormattedTextField, 336  
JFrame, 28; 33  
JInternalFrame, 507  
JLabel, 30; 33; 77  
JLayeredPane, 29  
JList, 264  
JMenu, 375; 377; 379  
JMenuBar, 375; 377; 378; 382  
JMenuItem, 375; 377; 379; 381  
JOptionPane, 504; 505; 511  
JPanel, 29; 212  
JPasswordField, 320; 332  
JPopupMenu, 375; 405  
JProgressBar, 202  
JRadioButton, 97; 120; 134  
JRadioButtonMenuItem, 402  
JRadioButtonMenuItem, 375  
JRootPane, 29; 109  
JScrollBar, 155  
JScrollPane, 224; 265  
JSeparator, 375  
JSlider, 170  
JSpinner, 264; 301  
JSplitPane, 252  
JTabbedPane, 239  
JTable, 434  
JTextArea, 320; 348  
JTextComponent, 320  
JTextField, 54; 320; 324  
JTextPane, 320  
JToggleButton, 97; 120  
JTree, 434; 481  
JWindow, 28  
KeyAdapter, 42  
KeyEvent, 40; 389  
ListDataEvent, 40  
ListSelectionEvent, 41; 446  
MenuEvent, 41  
MouseAdapter, 42  
MouseEvent, 40  
MouseInputAdapter, 42  
MouseMotionAdapter, 41; 42  
MouseWheelEvent, 40  
PlainDocument, 324

PropertyChangeEvent, 341  
SpinnerDateModel, 302; 311  
SpinnerListModel, 302; 303  
SpinnerNumberModel, 302; 307  
SwingUtilities, 35  
TableModelEvent, 41; 446  
Timer, 580  
TreeExpansionEvent, 41  
TreeModelEvent, 41  
TreeSelectionEvent, 41  
WindowAdapter, 42  
WindowEvent, 40  
адаптера, 41  
кнопка, 42; 97  
переключатель, 119  
по умолчанию, 109  
команда действия, 43; 57; 324  
компиляция, 31  
компонент, 25; 27  
контейнер, 27; 28  
контейнер верхнего уровня, 27; 28  
контекстное меню, 375; 405  
контроллер, 26  
корневая панель, 29  
корневой узел, 482

## Л

легковесный компонент, 24  
линейный регулятор, 152; 170  
лист, 482

## М

маркер, 171; 497  
меню, 374  
метка, 30; 33; 77  
actionPerformed, 100  
getPath, 490  
itemStateChanged, 101  
tableChanged, 456  
valueChanged, 446  
accept, 546  
actionPerformed, 43; 54; 100; 135; 415  
add, 378; 379; 411; 484  
add(), 34  
addActionListener, 43; 54; 98  
addChangeListener, 98; 154  
addItem, 294; 299

- addKeyListener, 38
- addMouseListener, 405
- addMouseMotionListener, 38
- addPropertyChangeListener, 416
- addSeparator, 380; 413
- addTab, 243
- addTableModelListener, 456; 466
- addTypeListener, 38
- adjustmentValueChanged, 159
- append, 354
- breadthFirstEnumeration, 497
- changeBorderSize, 607
- copy, 321; 326; 349
- createEmptyBorder, 75
- createEtchedBorder, 75
- createGlue, 623
- createHorizontalGlue, 623
- createHorizontalStrut, 623
- createLineBorder, 74
- createRigidArea, 618
- createStandardLabels, 174
- createVerticalGlue, 623
- createVerticalStrut, 623
- cut, 321; 326; 349
- depthFirstEnumeration, 497
- destroy, 586
- dispose, 533
- doClick, 98
- drawLine, 600
- drawRect, 600
- drawString, 600
- exists, 542
- getActionCommand, 43
- getAdjustable, 159
- getAnchorSelectionIndex, 277
- getBlockIncrement, 159
- getCalendarField, 312
- getColumn, 454; 456; 457
- getColumnClass, 466
- getColumnCount, 466
- getColumnModel, 450
- getColumnName, 466
- getComponent, 406
- getContentPane(), 34
- getCurrentDirectory, 551
- getDescription, 546
- getDot, 323
- getEnd, 312
- getExtent, 153; 172
- getFirstRow, 456; 457
- getHorizontalScrollBar, 232
- getIcon, 104
- getInsets, 601
- getInverted, 180
- getItem, 102; 121; 123
- getItemAt, 299
- getItemCount, 299
- getItemSelectable, 102
- getKeyStroke, 389
- getLabelTable, 174
- getLastPathComponent, 490
- getLastRow, 456; 457
- getLeadSelectionIndex, 277
- getLineCount, 354
- getLineWrap, 350
- getList, 303
- getListCellRendererComponent, 277
- getMajorTickSpacing, 174
- getMark, 323
- getMaximum, 153; 157; 172; 203; 307
- getMaxWidth, 465
- getMenuComponentCount, 380
- getMenuComponents, 380
- getMenuCount, 378
- getMinimum, 153; 157; 172; 203; 307
- getMinorTickSpacing, 174
- getMinWidth, 465
- getModel, 98; 115; 300; 456; 491
- getModelIndex, 454
- getModifiers, 101
- getName, 542
- getNumber, 308
- getPaintLabels, 174
- getPaintTicks, 174
- getPassword, 333
- getPath, 489; 551
- getPreferredWidth, 465
- getRowCount, 466
- getSelectedColumn, 451
- getSelectedColumns, 451
- getSelectedFile, 542
- getSelectedFiles, 551
- getSelectedIndex, 266
- getSelectedIndex, 290

- getSelectedIndices, 270
- getSelectedItem, 289; 290
- getSelectedRow, 447
- getSelectedRows, 447; 450
- getSelectedText, 326
- getSelectedValue, 274
- getSelectedValues, 274
- getSelectionModel, 447
- getSnapToTicks, 179
- getSource, 101
- getStart, 312
- getStateChange, 129
- getStateChange, 102; 121; 129
- getStepSize, 307
- getString, 208
- getSubElements, 378
- getText, 54; 95; 98; 325
- getTreePath, 491
- getType, 457
- getUnitIncrement, 158
- getValue, 153; 157; 159; 172; 203; 415; 416
- getValueAt, 466
- getValueIsAdjusting, 447
- getValueIsAdjusting, 153; 158; 160
- getVerticalScrollBar, 232
- getVisibleAmount, 158
- getWhen, 101
- getX, 406
- getY, 406
- hidePopup, 300
- init, 586
- insert, 354
- insertSeparator, 380
- invokeAndWait, 35; 573
- invokeLater, 35; 573
- isArmed, 115
- isBorderPainted, 208
- isCellEditable, 466
- isDirectory, 543
- isEnabled, 105; 115; 416; 417
- isFile, 543
- isIndeterminate, 207
- isLeaf, 497
- isOpaque, 214
- isPopupTrigger, 406; 408
- isPressed, 115
- isRollover, 115
- isRolloverEnabled, 104
- isSelected, 98; 115; 121; 378
- isSelectionEmpty, 276
- isStringPainted, 204
- itemStateChanged, 121; 128
- mouseClicked, 405
- mouseDragged, 41
- mouseEntered, 406
- mouseExited, 406
- mouseMoved, 41
- mousePressed, 406; 408
- mouseReleased, 406; 408
- moveCaretPosition, 321; 326; 349
- moveColumn, 472
- paint, 599
- paintBorder, 599
- paintChildren, 599
- paintComponent, 599
- paste, 321; 326; 349
- postorderEnumeration, 497
- preorderEnumeration, 497
- print, 472
- propertyChange, 341
- putValue, 415; 416
- remove, 378; 380; 497
- removeActionListener, 43; 54
- removeAllItems, 294
- removeChangeListener, 154
- removeColumn, 472
- removeItem, 294
- removeItemAt, 294
- removeKeyListener, 38
- removePropertyChangeListener, 416
- removeTableModelListener, 467
- repaint, 601
- replaceRange, 354
- revalidate, 607
- select, 321
- setActionCommand, 57; 324
- setBackgroundAt, 244
- setBlockIncrement, 159; 170
- setBorder, 75
- setBorderPainted, 208
- setBottomComponent, 257
- setCalendarField, 312
- setCaretPosition, 321; 326; 349
- setCellRenderer, 277

- setCellSelectionEnabled, 445
- setColumnHeaderView, 228
- setColumnSelectionAllowed, 445
- setContentPane, 219
- setContinuousLayout, 257
- setCorner, 233
- setDefaultButton, 109
- setDefaultCloseOperation(), 33
- setDefaultEditor, 473
- setDefaultRenderer, 473
- setDisabledIcon, 89; 104; 126; 393
- setDisabledIconAt, 245
- setDisplayedMnemonic, 92
- setDisplayedMnemonicIndex, 389
- setDividerLocation, 258
- setDividerSize, 258
- setEditable, 293
- setEnabled, 89; 98; 105; 127; 300; 381; 416; 417
- setEnabledAt, 245
- setEnd, 312
- setExtent, 153; 172
- setFileHidingEnabled, 551
- setFileSelectionMode, 550
- setFloatable, 414
- setForegroundAt, 244
- setGridColor, 472
- setHorizontalAlignment, 79; 98; 133
- setHorizontalScrollBarPolicy, 232
- setHorizontalTextPosition, 85
- setHorizontalTextPosition, 99; 126; 133; 392
- setIcon, 104; 392
- setIndeterminate, 207
- setInverted, 179
- setLabelFor, 93
- setLabelTable, 173
- setLeftComponent, 257
- setLineWrap, 349
- setList, 303
- setListData, 277
- setMajorTickSpacing, 173
- setMaximum, 154; 157; 203; 307
- setMinimum, 154; 157; 172; 203; 307
- setMinorTickSpacing, 173
- setMnemonic, 99; 388
- setMnemonicAt, 245
- setModel, 300
- setOneTouchExpandable, 257
- setOpaque, 214
- setOrientation, 256
- setPaintTicks, 173
- setPopupVisible, 300
- метод
  - setPreferredScrollableViewportSize, 435
  - setPreferredSize, 166
  - setPressedIcon, 99; 104; 126
  - setRangeProperties, 154
  - setRepeats, 581
  - setResizeWeight, 258
  - setRightComponent, 257
  - setRollover, 414
  - setRolloverEnabled, 104
  - setRolloverIcon, 99; 104; 126
  - setRolloverSelectedIcon, 99
  - setRootVisible, 496
  - setRowHeaderView, 228
  - setRowSelectionAllowed, 444; 445
  - setSelected, 99; 128; 136
  - setSelectedIcon, 99; 126
  - setSelectedIndex, 245; 275
  - setSelectedIndices, 276
  - setSelectedItem, 289
  - setSelectionMode, 265; 443
  - setShowGrid, 472
  - setShowHorizontalLines, 472
  - setShowsRootHandles, 497
  - setShowVerticalLines, 472
  - setSnapToTicks, 179
  - setStart, 312
  - setStepSize, 307
  - setString, 208
  - setStringPainted, 203
  - setTabSize, 354
  - setText, 54; 95; 99; 325
  - setToolTipText, 259; 393
  - setTopComponent, 257
  - setUnitIncrement, 158
  - setUserObject, 497
  - setValue, 154; 157; 172; 203
  - setValueAt, 457; 467
  - setValueIsAdjusting, 153; 154; 158
  - setValues, 158
  - setVerticalAlignment, 79; 99; 133

setVerticalPosition, 80  
setVerticalScrollBarPolicy, 232  
setVerticalTextPosition, 85; 99; 126; 133  
setViewportBorder, 229  
setVisible, 531  
setVisible, 34; 533  
setVisibleAmount, 158  
setWrapStyleWord, 350  
show, 406; 408  
showConfirmDialog, 506; 513  
showDialog, 542; 565  
showInputDialog, 506; 519  
showInternalMessageDialog, 507  
showMessageDialog, 507  
showMessageDialog, 506  
showOpenDialog, 541  
showOptionDialog, 506; 525  
showSaveDialog, 541  
start, 581; 586  
stateChanged, 102  
stop, 581; 586  
toString, 473  
treeCollapsed, 490  
treeExpanded, 490  
treeNodesChanged, 490  
treeNodesInserted, 491  
treeNodesRemoved, 491  
treeStructureChanged, 491  
valueChanged, 265; 450; 489  
getSelectedColumns, 544  
getValueAt, 457  
setJMenuBar, 378  
caretUpdate, 323  
мнемоническое обозначение, 92; 110; 388  
многопоточность, 572  
модель, 26; 155  
    SingleSelectionModel, 240  
    выбора пунктов, 482  
    делегирования событий, 37  
    дерева, 482  
    столбца, 435  
    таблицы, 435

## Н

настраиваемые стили, 24  
неименованный внутренний класс, 68; 111  
неопределенное состояние индикатора

хода процесса, 207  
непрозрачный компонент, 213

## О

область  
    отображения, 601  
    постраничного просмотра, 155  
    просмотра, 224  
обработчик  
    события, 39  
    события, 37  
обрамление, 74  
обход дерева  
    в глубину в обратном направлении, 498  
    в глубину в прямом направлении, 498  
    в ширину, 498  
объект воспроизведения, 434; 482  
окно  
    выбора файлов, 540  
    для ввода данных, 505  
    для вывода сообщений, 505  
    для получения подтверждения от пользователя, 505  
    для установки опций, 505  
основное меню, 382  
основной маркер, 173

## П

пакет  
    java.awt.event, 39  
    java.beans, 39  
    java.util, 39  
    javax.swing, 28; 33; 70  
    javax.swing.border, 70  
    javax.swing.colorchooser, 70  
    javax.swing.event, 39; 70  
    javax.swing.filechooser, 70; 546  
    javax.swing.plaf, 70  
    javax.swing.plaf.basic, 70  
    javax.swing.plaf.metal, 70  
    javax.swing.plaf.multi, 70  
    javax.swing.plaf.synth, 70  
    javax.swing.table, 70; 434  
    javax.swing.text, 70; 320  
    javax.swing.text.html, 70  
    javax.swing.text.html.parser, 70



javax.swing.text.rtf, 70  
 javax.swing.tree, 70; 482  
 javax.swing.undo, 70  
 панель, 29; 212  
   с вкладками, 239  
   слоя, 29  
   содержимого, 29; 219  
   с прокруткой, 224  
 переключатель опций, 134; 400  
 плавающая панель, 411  
 поддерево, 482  
 поддержка событий, 37  
 поле редактирования, 324  
 ползунок, 155; 171  
 политика полос прокрутки, 231  
 полоса прокрутки, 152; 155  
 пользовательский интерфейс, 22  
 поток, 572  
   обработки событий, 35  
 представитель пользовательского  
   интерфейса, 26  
 представление, 26  
 привязка к маркерам, 179  
 приращение блока, 155  
 пришвартованная панель, 411  
 прозрачный компонент, 213  
 прокручиваемая область просмотра, 435  
 пустая рамка, 74  
 путь, 482

## Р

разделитель, 252; 380  
 разделяемая панель, 252  
 размер таблицы, 462  
 рамка, состоящая из линий, 74  
 раскрывающийся список, 288  
 раскрывающийся список с  
   редактированием, 293  
 распорка, 623  
 расширение, 152  
 регистрация обработчика, 38  
 редактор ячейки, 434  
 режим выбора, 443; 482  
   пунктов списка, 265  
 рельефная рамка, 74  
 рисование, 598

## С

северная область, 69  
 сервлет, 585  
 склейка, 623  
 событие, 37; 38  
   CaretEvent, 323  
   ListDataEvent, 264  
   ListSelectionEvent, 446  
   MenuDragMouseEvent, 376  
   MenuEvent, 376  
   MenuKeyEvent, 376  
   PopupMenuEvent, 377  
   TreeExpansionEvent, 483; 489; 490  
   TreeModelEvent, 483; 489; 490  
   TreeSelectionEvent, 483; 489  
   действия, 100  
   действия, 100  
   изменения свойств, 340  
   изменения состояния, 102  
   изменения состояния, 100; 240  
   регулировки, 154; 159  
   элемента, 101  
   элемента, 100  
 список, 264  
 стеклянная панель, 29  
 стиль, GTK+, 25  
   Java, 25  
   Mac, 25  
   metal, 25  
   Motif, 25  
   Windows, 25  
 строка подсказки, 244; 258; 393

## Т

таблица, 434  
 табулятор, 354  
 тяжеловесный компонент, 23  
 текстовый компонент, 320  
 текстовый курсор, 321  
 терминальный узел, 482

## У

угловая область, 225; 232

**Ф**

фильтр файлов, 546  
флажок опции, 127; 400

**Ц**

центральная область, 69

**Э**

элементарное приращение, 155

**Ю**

южная область, 69

**Я**

якорь, 276  
ярлык, 240  
ячейка, 434

*Научно-популярное издание*

**Герберт Шилдт**

# **SWING: руководство для начинающих**

Литературный редактор *И.А. Попова*

Верстка *А.В. Плаксюк*

Художественный редактор *С.А. Чернокозинский*

Корректор *Л.А. Гордиенко*

Издательский дом "Вильямс"

127055, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 22.06.2007. Формат 70х100/16.

Гарнитура Petersburg. Печать офсетная.

Усл. печ. л. 56,76. Уч.-изд. л. 30,00.

Тираж 2000 экз. Заказ №1947

Отпечатано по технологии СтР

в ОАО "Печатный двор" им. А. М. Горького  
197110, Санкт-Петербург, Чкаловский пр., 15.