Принципы построения графического интерфейса

Графические примитивы

Методы класса Graphics
Как задать цвет
Как нарисовать чертеж
Класс Polygon
Как вывести текст. Как установить шрифт.
Как задать шрифт
Класс FontMetrics
Возможности Java 2D
Преобразование координат. Класс AffineTransform.
Рисование фигур средствами Java2D. Класс BasicStroke.
Класс GeneralPath
Классы GradientPaint и TexturePaint
Вывод текста средствами Java 2D
Методы улучшения визуализации

Основные компоненты

Класс Component
Класс Cursor
Как создать свой курсор
События. Класс Container.
События. Компонент Label.
События. Компонент Button.
События. Компонент Checkbox.
События. Класс CheckboxGroup.
Как создать группу радиокнопок
Компонент Choice. События.
Компонент List
События
Компоненты для ввода текста. Класс TextComponent. События.
Компонент TextField. События.
Компонент TextArea
События
Компонент Scrollbar
События
Контейнер Panel
Контейнер ScrollPane
Контейнер Window. События.
Контейнер Frame
События
Контейнер Dialog
События
Контейнер FileDialog. События.
Создание собственных компонентов. Компонент Canvas.
Создание "легкого" компонента

Размещение компонентов

Размещение компонентов
Менеджер FlowLayout
Менеджер BorderLayout
Менеджер GridLayout
Менеджер CardLayout
Менеджер GridBagLayout

Обработка событий

Обработка событий
Событие ActionEvent
Обработка действий мыши
Классы-адаптеры
Обработка действий клавиатуры. Событие TextEvent.
Обработка действий с окном
Событие ComponentEvent. Событие ContainerEvent.
Событие FocusEvent. Событие ItemEvent.
Событие AdjustmentEvent
Несколько слушателей одного источника
Диспетчеризация событий
Создание собственного события

Создание меню

Принципы построения графического интерфейса

В предыдущих главах мы писали программы, связанные с текстовым терминалом и запускающиеся из командной строки. Такие программы называются **консольными приложениями**. Они разрабатываются для выполнения на серверах, там, где не требуется интерактивная связь с пользователем.

Программы, тесно взаимодействующие с пользователем, воспринимающие сигналы от клавиатуры и мыши, работают в графической среде. Каждое приложение, предназначенное для работы в графической среде, должно создать хотя бы одно окно, в котором будет происходить его работа, и зарегистрировать его в графической оболочке операционной системы, чтобы окно могло взаимодействовать с операционной системой и другими окнами: перекрываться, перемещаться, менять размеры, сворачиваться в ярлык.

Есть много различных графических систем: MS Windows, X Window System, Macintosh. В каждой из них свои правила построения окон и их компонентов: меню, полей ввода, кнопок, списков, полос прокрутки. Эти правила сложны и запутанны. Графические API содержат сотни функций.

Для облегчения создания окон и их компонентов написаны библиотеки классов: MFC, Motif, OpenLook, Qt, Tk, Xview, OpenWindows и множество других. Каждый класс такой библиотеки описывает сразу целый графический компонент, управляемый методами этого и других классов.

В технологии Java дело осложняется тем, что приложения Java должны работать в любой или хотя бы во многих графических средах. Нужна библиотека классов, независимая от конкретной графической системы. В первой версии JDK задачу рещили следующим образом: были разработаны интерфейсы, содержащие методы работы с графическими объектами. Классы библиотеки AWT реализуют эти интерфейсы для создания приложений. Приложения Java используют данные методы для размещения и перемещения графических объектов, изменения их размеров, взаимодействия объектов.

С другой стороны, для работы с экраном в конкретной графической среде эти интерфейсы реализуются в каждой такой среде отдельно. В каждой графической оболочке это делается по-своему, средствами этой оболочки с помощью, графических библиотек данной операционной системы. Такие интерфейш были названы **peer-интерфейсами**.

Библиотека классов Java, основанных на peer-интерфейсах, получила название **AWT**(Abstract Window Toolkit). При выводе объекта, созданного в приложении Java и основанного на peer-интерфейсе, на экран создается парный ему (**peer-to-peer**) объект графической подсистемы операционной системы, который и отображается на экране. Эти объекты тесно взаимодействуют во время работы приложения. Поэтому графические объекты AWT в каждой графической среде имеют вид, характерный для этой среды: в MS Windows, Motif, OpenLook, OpenWindows, везде окна, созданные в AWT, выглядят как "родные" окна.

Именно из-за такой реализации peer-интерфейсов и других "родных" методов, написанных, главным образом, на языке C++, приходится для каждой платформы выпускать свой вариант JDK.

В версии JDK 1.1 библиотека AWT была переработана. В нее добавлена возможность создания компонентов, полностью написанных на Java и не зависящих от peer-интерфейсов. Такие компоненты стали называть "**легкими**" (lightweight) в отличие от компонентов, реализованных через peer-интерфейсы, названных "**тяжелыми**" (heavy).

"Легкие" компоненты везде выглядят одинаково, сохраняют заданный при создании вид (**look and feel**). Более того, приложение можно разработать таким образом, чтобы после его запуска можно было выбрать какой-то определенный вид: Motif, Metal, Windows 95 или какой-нибудь другой, и сменить этот вид в любой момент работы.

Эта интересная особенность "легких" компонентов получила название **PL&F** (Pluggable Look and Feel) или "plaf.

Была создана обширная библиотека "легких" компонентов Java, названная Swing. В ней были переписаны все компоненты библиотеки AWT, так что библиотека Swing может использоваться самостоятельно, несмотря на то, что все классы из нее расширяют классы библиотеки AWT.

Библиотека классов Swing поставлялась как дополнение к JDK 1.1. В состав Java 2 SDK она включена как основная графическая библиотека классов, реализующая идею "100% Pure Java", наряду с AWT.

В Java 2 библиотека AWT значительно расширена добавлением новых средств рисования, вывода текстов и изображений, получивших название Java 2D, и средств, реализующих перемещение текста методом **DnD** (Drag and Drop).

Кроме того, в Java 2 включены новые методы ввода/вывода Input Method Framework и средства связи с дополнительными устройствами ввода/вывода, такими как световое перо или клавиатура Бройля, названные Accessibility.

Все эти средства Java 2: AWT, Swing, Java 2D, DnD, Input Method Framework и Accessibility составили библиотеку графических средств Java, названную **JFC** (Java Foundation Classes).

Описание каждого из этих средств составит целую книгу, поэтому мы вынуждены ограничиться представлением только основных средств библиотеки AWT.

Компонент и контейнер

Основное понятие графического интерфейса пользователя (ГИП) – **компонент**(component) графической системы. В русском языке это слово подразумевает просто составную часть, элемент чего-нибудь, но в графическом интерфейсе это понятие гораздо конкретнее. Оно означает отдельный, полностью определенный элемент, который можно использовать в графическом интерфейсе независимо от других элементов. Например, это поле ввода, кнопка, строка меню, полоса прокрутки, радиокнопка. Само окно приложения – тоже его компонент. Компоненты могут быть и невидимыми, например, панель, объединяющая компоненты, тоже является компонентом.

Вы не удивитесь, узнав, что в AWT компонентом считается объект класса Component или объект всякого класса, расширяющего класс component. В классе component собраны общие методы работы с любым компонентом графического интерфейса пользователя. Этот класс – центр библиотеки AWT.

Каждый компонент перед выводом на экран помещается в **контейнер** (container). Контейнер "знает", как разместить компоненты на экране. Разумеется, в языке Java контейнер – это объект класса Container или всякого его расширения. Прямой наследник этого класса – класс **jcomponent** – вершина иерархии многих классов библиотеки Swing.

Создав компонент – объект класса Component или его расширения, следует добавить его к предварительно созданному объекту класса container или его расширения одним из методов **add ()**.

Класс **Container** сам является невидимым компонентом, он расширяет класс **Component**. Таким образом, в контейнер наряду с компонентами можно помещать контейнеры, в которых находятся какие-то другие компоненты, достигая тем самым большой гибкости расположения компонентов.

Основное окно приложения, активно взаимодействующее с операционной системой, необходимо построить по правилам графической системы. Оно должно перемещаться по экрану, изменять размеры, реагировать на действия мыши и клавиатуры. В окне должны быть, как минимум, следующие стандартные компоненты.

- **Строка заголовка** (title bar), с левой стороны которой необходимо разместить кнопку контекстного меню, а с правой – кнопки сворачивания и разворачивания окна и кнопку закрытия приложения.
- Необязательная **строка меню** (menu bar) с выпадающими пунктами меню.
- Горизонтальная и вертикальная **полосы прокрутки** (scrollbars).
- Окно должно быть окружено **рамкой** (border), реагирующей на действия мыши.

Окно с этими компонентами в готовом виде описано в классе Frame. Чтобы создать окно, достаточно сделать свой класс расширением класса Frame, как показано в листинге 8.1. Всего восемь строк текста и окно готово.

Листинг 8.1. Слишком простое окно приложения.

```
import j ava.awt.*;
class TooSimpleFrame extends Frame{
public static void main(String[] args){
Frame fr = new TooSimpleFrame();
fr.setSize(400, 150);
fr.setVisible(true);
}
}
```

Класс **TooSimpleFrame** обладает всеми свойствами класса Frame, являясь его расширением. В нем создается экземпляр окна fr, и устанавливаются размеры окна на экране– 400x150 пикселов – методом **setsize()**. Если не задать размер окна, то на экране появится окно минимального размера – только строка заголовка. Конечно, потом его можно растянуть с помощью мыши до любого размера.

Затем окно выводится на экран методом **setvisibie(true)**. Дело в том, что, с точки зрения библиотеки AWT, создать окно значит выделить область оперативной памяти, заполненную нужными пикселами, а вывести содержимое этой области на экран – уже другая задача, которую и решает метод **setVisible(true).**

Конечно, такое окно непригодно для работы. Не говоря уже о том, что у него нет заголовка и поэтому окно нельзя закрыть. Хотя его можно перемещать по экрану, менять размеры, сворачивать на панель задач и раскрывать, но команду завершения приложения мы не запрограммировали. Окно нельзя закрыть ни щелчком кнопки мыши на кнопке с крестиком в правом верхнем углу окна, ни комбинацией клавиш **ALT + F4**. Приходится завершать работу приложения средствами операционной системы, например, комбинацией клавиш **CTRL + C**.

В листинге 8.2 к программе листинга 8.1 добавлены заголовок окна и обращение к методу, позволяющему завершить приложение.

Листинг 8.2. Простое окно приложения.

```
import java.awt.*;
import j ava.awt.event.*;
class SimpleFrame extends Frame{
SimpleFrame(String s){
super (s);
setSize(400, 150);
setvisible(true);
addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit (0);
}
});
}
public static void main(String[] args){
new SimpleFrame(" Моя программа");
}
}
```

В программу добавлен конструктор класса SimpleFrame, обращающийся к конструктору своего суперкласса Frame, который записывает свой аргумент s в строку заголовка окна.

В конструктор перенесена установка размеров окна, вывод его на экран и добавлено обращение к методу **addWindowListener ()**, реагирующему на действия с окном. В качестве аргумента этому методу передается экземпляр безымянного внутреннего класса, расширяющего класс WindowAdapter. Этот безымянный класс реализует метод**windowclosing ()**, обрабатывающий попытку закрытия окна. Данная реализация очень проста – приложение завершается статическим методом **exit ()** класса system. Окно при этом закрывается автоматически.

Все это мы подробно разберем в главе 12, а пока просто добавляйте эти строчки во все ваши программы для закрытия окна и завершения работы приложения.

Итак, окно готово. Но оно пока пусто. Выведем в него, по традиции, приветствие "Hello, World!", правда, слегка измененное.

В листинге 3.3 приведена полная программа этого вывода, а рис. 8.1 демонстрирует окно.

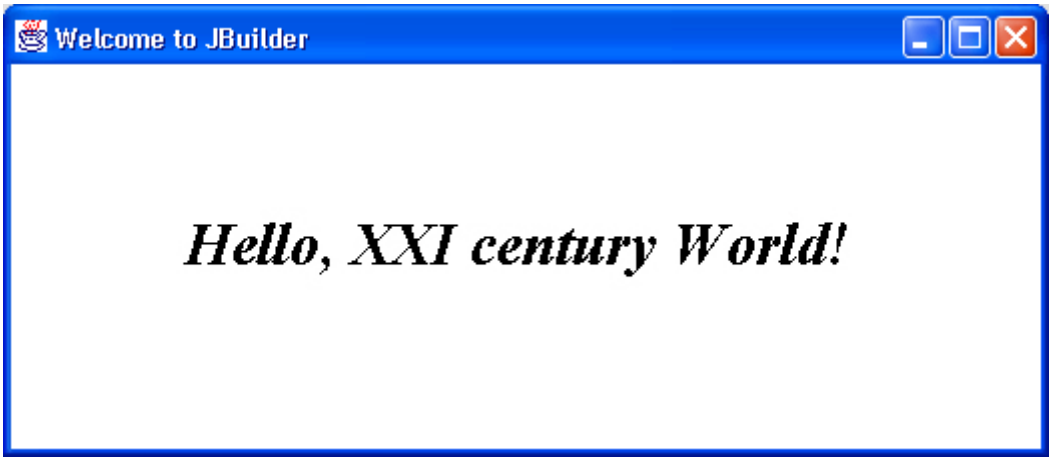

Рис. 8.1. Окно программы-приветствия

Листинг 8.3. Графическая программа с приветствием.

```
import java.awt.*;
import java.awt.event.*;
class HelloWorldFrame extends Frame{
HelloWorldFrame(String s){
super(s);
}
public void paint(Graphics g){
g.setFont(new Font("Serif", Font.ITALIC | Font.BOLD, 30));
g.drawstring("Hello, XXI century World!", 20, 100);
}
public static void main(String[] args){
Frame f = new HelloWorldFrame("Здравствуй, мир XXI века!");
f.setSize(400, 150);
f.setvisible(true);
f.addWindowListener(new WindowAdapter(){
public void windowciosing(WindowEvent ev)(
System.exit(0);
}
});
}
}
```

Для вывода текста мы переопределяем метод **paint ()** класса component. Класс Frame наследует этот метод с пустой реализацией.

Метод **paint ()** получает в качестве аргумента экземпляр g класса Graphics, умеющего, в частности, выводить текст методом **drawstring ()**. В этом методе кроме текста мы указываем положение начала строки в окне – 20 пикселов от левого края и 100 пикселов сверху. Эта точка – левая нижняя точка первой буквы текста н.

Кроме того, мы установили новый шрифт "Serif" большего размера – 30 пунктов, полужирный, курсив. Всякий шрифт – объект класса Font, а задается он методом **setFont ()**класса Graphics.

Работу со шрифтами мы рассмотрим в следующей главе.

В листинге 8.3, для разнообразия, мы вынесли вызовы методов установки размеров окна, вывода его на экран и завершения программы в метод **main ()**.

Как вы видите из этого простого примера, библиотека AWT большая и разветвленная, в ней множество классов, взаимодействующих друг с другом. Рассмотрим иерархию некоторых наиболее часто используемых классов AWT.

Иерархия классов AWT

На рис. 8.2 показана иерархия основных классов AWT. Основу ее составляют готовые компоненты: **Button**, **Canvas**, **Checkbox**, **Choice**, **Container**, **Label**, **List**, **Scrollbar**, **TextArea**, **TextField**, **Menubar**, **Menu**, **PopupMenu**, **MenuItem**, **CheckboxMenuItem**. Если этого набора не хватает, то от класса Canvas можно породить собственные "тяжелые" компоненты, а от класса Component – "легкие" компоненты.

Основные контейнеры – это классы **Panel**, **ScrollPane**, **Window**, **Frame**, **Dialog**,**FileDialog**. Свои "тяжелые" контейнеры можно породить от класса Panel, а "легкие" – от класса container.

Целый набор классов помогает размещать компоненты, задавать цвет, шрифт, рисунки и изображения, реагировать на сигналы от мыши и клавиатуры.

На рис. 8.2 показаны и начальные классы иерархии библиотеки Swing – классы**JComponent**, **JWindow**, **JFrame**, **JDialog**, **JApplet**.

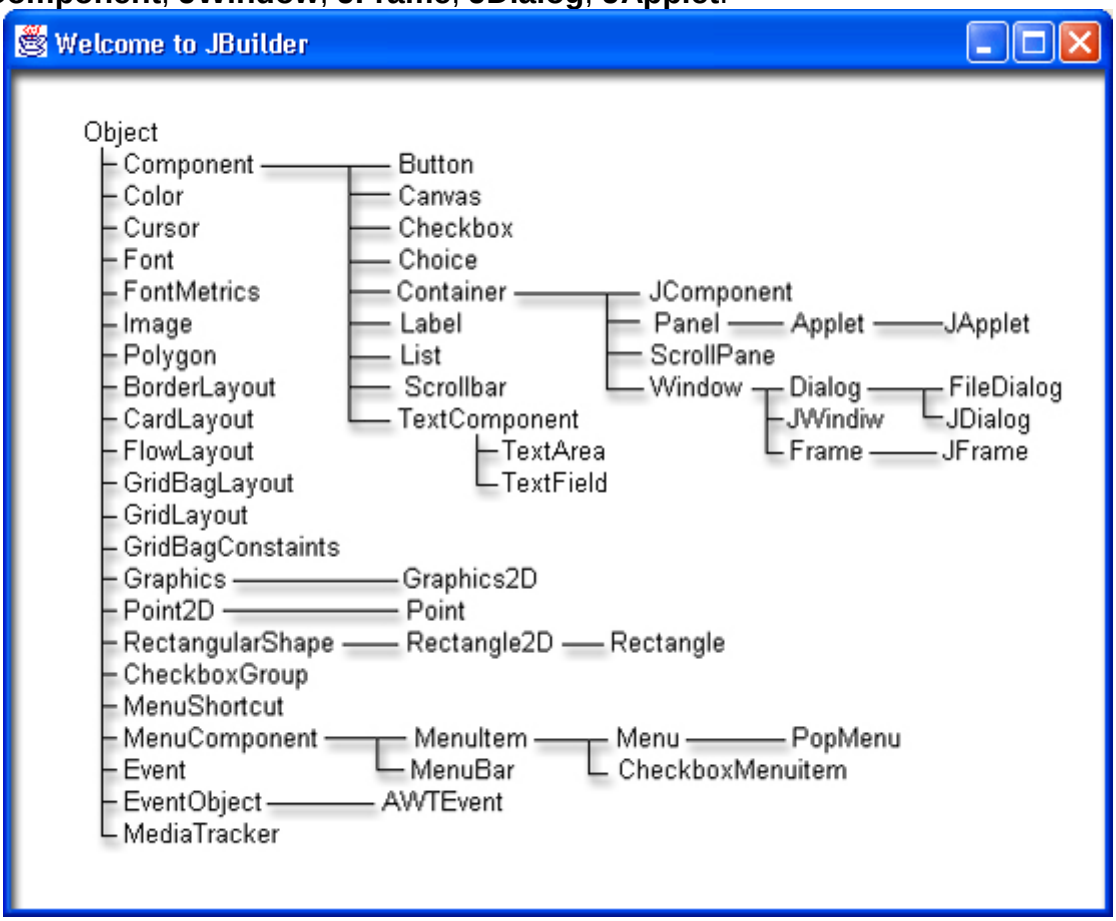

Рис. 8.2. Иерархия основных классов AWT

Заключение

Как видите, библиотека графических классов AWT очень велика и детально проработана. Это многообразие классов только отражает многообразие задач построения графического интерфейса. Стремление улучшить интерфейс безгранично. Оно приводит к созданию все новых библиотек классов и расширению существующих. Независимыми производителями создано уже много графических библиотек Java: KL Group, JBCL, и появляются все новые и новые библиотеки. Сведения о них можно получить на сайтах, указанных во **введении**.

В следующих главах мы подробно рассмотрим, как можно использовать библиотеку AWT для создания собственных приложений с графическим интерфейсом пользователя, изображениями, анимацией и звуком.

Графические примитивы

- ### Методы класса Graphics
  При создании компонента, т. е. объекта класса Component, автоматически формируется его графический контекст (graphics context). В контексте размещается область рисования и вывода текста и изображений.
- ### Как задать цвет
  Цвет, как и все в Java, – объект определенного класса, а именно, класса color. Основу класса составляют семь конструкторов цвета. Самый простой конструктор: | Color(int red, int green, int blue) | …создает цвет, получающийся как смесь красной red, зеленой green и синей blue составляющих.
- ### Как нарисовать чертеж
  Основной метод рисования: | drawLine(int x1, int y1, int x2, int y2) | …вычерчивает текущим цветом отрезок прямой между точками с координатами (x1, y1) и (x2, y2). | Одного этого метода достаточно, чтобы, нарисовать любую картину по точкам, вычерчивая каждую точку с координатами (x, y) методом drawLine (x, y, x, y) и меняя цвета от точки к точке. Но никто, разумеется, не станет этого делать.
- ### Класс Polygon
  Этот класс предназначен для работы с многоугольником, в частности, с треугольниками и произвольными четырехугольниками. | Объекты этого класса можно создать двумя конструкторами: | Polygon () – создает пустой объект;
- ### Как вывести текст. Как установить шрифт.
  Для вывода текста в область рисования текущим цветом и шрифтом, начиная с точки (x, y), в, классе Graphics есть несколько методов: | drawstring (String s, int x, int y) – выводит строку s; | drawBytes(byte[] b, int offset, int length, int x, int y) – выводит length элементов массива байтов ь, начиная с индекса offset;
- ### Как задать шрифт
  Объекты класса Font хранят начертания (glyphs) символов, образующие шрифт. Их можно создать двумя Конструкторами: | Font (Map attributes) – задает шрифт с заданными аргументом attributes атрибутами. Ключи атрибутов и некоторые их значения задаются константами класса TextAttnbute из пакета java.awt.font. Этот конструктор характерен для Java 2D и будет рассмотрен далее в настоящей главе.
- ### Класс FontMetrics
  Класс FontMetrics является абстрактным, поэтому нельзя воспользоваться его конструктором. Для получения объекта класса FontMetrics, содержащего набор метрических характеристик шрифта f, надо обратиться к методу getFontMetrics (f) класса Graphics или класса Component.
- ### Возможности Java 2D
  В систему пакетов и классов Java 2D, основа которой– класс Graphics2D пакета java.awt, внесено несколько принципиально новых положений. | Кроме координатной системы, принятой в классе Graphics и названной координатным пространством пользователя (User Space), введена еще система координат устройства вывода (Device Space): экрана монитора, принтера.
- ### Преобразование координат. Класс AffineTransform.
  Правило преобразования координат пользователя в координаты графического устройства (transform) задается автоматически при создании графического контекста так же, как цвет и шрифт. В дальнейшем его можно изменить методом setTransform() так же, как меняется цвет или шрифт.
- ### Рисование фигур средствами Java2D. Класс BasicStroke.
  Характеристики пера для рисования фигур описаны в интерфейсе stroke. В Java 2D есть пока только один класс, реализующий этот интерфейс – класс BasicStroke. | Конструкторы класса BasicStroke определяют характеристики пера.
- ### Класс GeneralPath
  Вначале создается пустой объект класса GeneralPath конструктором по умолчанию GeneralPath () или объект, содержащий одну фигуру, конструктором GeneralPath (Shape sh). | Затем к этому объекту добавляются фигуры методом append(Shape sh, boolean connect).
- ### Классы GradientPaint и TexturePaint
  Класс GradientPaint предлагает сделать заливку следующим образом. | В двух точках м и N устанавливаются разные цвета. В точке M(xi, yi) задается цвет cl, в точке b1(x2, y2) – цвет c2. Цвет заливки гладко меняется от el к c2 вдоль прямой, соединяющей точки м и м, оставаясь постоянным вдоль каждой прямой, перпендикулярной прямой мы.
- ### Вывод текста средствами Java 2D
  Шрифт – объект класса Font – кроме имени, стиля и размера имеет еще полтора десятка атрибутов: подчеркивание, перечеркивание, наклон, цвет шрифта и цвет фона, ширину и толщину символов, аффинное преобразование, расположение слева направо или справа налево.
- ### Методы улучшения визуализации
  Визуализацию (rendering) созданной графики можно усовершенствовать, установив один из методов (hint) улучшения одним из методов класса Graphics2D: | setRenderingHints(RenderingHints.Key key, Object value) | setRenderingHints(Map hints) | Ключи – методы улучшения – и их значения задаются константами класса RenderingHints, перечисленными в табл. 9.2. | Таблица 9.2. Методы визуализации и их значения.

Методы класса Graphics

При создании компонента, т. е. объекта класса Component, автоматически формируется его **графический контекст** (graphics context). В контексте размещается область рисования и вывода текста и изображений. Контекст содержит текущий и альтернативный цвет рисования и цвет фона – объекты класса color, текущий шрифт для вывода текста – объект класса Font.

В контексте определена система координат, начало которой с координатами (0, 0) расположено в верхнем левом углу области рисования, ось Ox направлена вправо, ось Oy – вниз. Точки координат находятся между пикселами.

Управляет контекстом класс **Graphics** или новый класс **Graphics2D**, введенный в Java 2. Поскольку графический контекст сильно зависит от конкретной графической платформы, эти классы сделаны абстрактными. Поэтому нельзя непосредственно создать экземпляры класса Graphics или Graphics2D.

Однако каждая виртуальная машина Java реализует методы этих классов, создает их экземпляры для компонента и предоставляет объект класса Graphics методом **getGraphics ()** класса Component или как аргумент методов **paint()** и **update()**.

Посмотрим сначала, какие методы работы с графикой и текстом предоставляет нам класс Graphics.

При создании контекста в нем задается текущий цвет для рисования, обычно черный, и цвет фона области рисования – белый или серый. Изменить текущий цвет можно методом**setcolor (Color newColor)**, аргумент **newcolor** которого – объект класса Color.

Узнать текущий цвет можно методом **getcolor ()**, возвращающим объект класса color.

Как задать цвет

Цвет, как и все в Java, – объект определенного класса, а именно, класса **color**. Основу класса составляют семь конструкторов цвета. Самый простой конструктор:

```
Color(int red, int green, int blue)
```

…создает цвет, получающийся как смесь красной red, зеленой green и синей blue составляющих. Эта цветовая модель называется RGB. Каждая составляющая меняется от 0 (отсутствие составляющей) до 255 (полная интенсивность этой составляющей). Например:

```
Color pureRed = new Color(255, 0, 0);
Color pureGreen = new Color(0, 255, 0);
```

…определяют чистый ярко-красный pureRed и чистый ярко-зеленый pureGreen цвета.

Во втором конструкторе интенсивность составляющих можно изменять более гладко вещественными числами от 0.0 (отсутствие составляющей) до 1.0 (полная интенсивность составляющей):

```
Color(float red, float green, float blue)
```

Например:

```
Color someColor = new Color(O.OSf, 0.4f, 0.95f);
```

Третий конструктор:

```
Color(int rgb)
```

…задает все три составляющие в одном целом числе. В битах 16-23 записывается красная составляющая, в битах 8-15 – зеленая, а в битах 0-7 – синяя составляющая цвета. Например:

```
Color c = new Color(OXFF8F48FF);
```

Здесь красная составляющая задана с интенсивностью 0x8F, зеленая – 0x48, синяя – 0xFF.

Следующие три конструктора:

```
Color(int red, int green, int blue, int alpha)
Color(float red, float green, float blue, float alpha)
Color(int rgb, boolean hasAlpha)
```

…вводят четвертую составляющую цвета, так называемую "альфу", определяющую прозрачность цвета. Эта составляющая проявляет себя при наложении одного цвета на другой. Если альфа равна 255 или 1.0, то цвет совершенно непрозрачен, предыдущий цвет не просвечивает сквозь него. Если альфа равна 0 или 0.0, то цвет абсолютно прозрачен, для каждого пиксела виден только предыдущий цвет.

Последний из этих конструкторов учитывает составляющую альфа, находящуюся в битах 24-31, если параметр hasAipha равен true. Если же hasAipha равно false, то составляющая альфа считается равной 255, независимо от того, что записано в старших битах параметра**rgb**.

Первые три конструктора создают непрозрачный цвет с альфой, равной 255 или 1.0.

Седьмой конструктор:

```
Color(ColorSpace cspace, float[] components, float alpha)
```

…позволяет создавать цвет не только в цветовой модели (**color model**) RGB, но и в других моделях: CMYK, HSB, CIEXYZ, определенных объектом класса **ColorSpace**.

Для создания цвета в модели HSB можно воспользоваться статическим методом:

```
getHSBColor(float hue, float saturation, float brightness).
```

Если нет необходимости тщательно подбирать цвета, то можно просто воспользоваться одной из тринадцати статических констант типа color, имеющихся в классе color. Вопреки соглашению "Code Conventions" они записываются строчными буквами: black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, white, yellow.

Методы класса Color позволяют получить составляющие текущего цвета: **getRed()**,**getGreen()**, **getBlue()**, **getAlpha()**, **getRGB()**, **getColorSpace ()**, **getComponents ()**.

Два метода создают более яркий **brighter()** и более темный **darker ()** цвета по сравнению с текущим цветом. Они полезны, если надо выделить активный компонент или, наоборот, показать неактивный компонент бледнее остальных компонентов.

Два статических метода возвращают цвет, преобразованный из цветовой модели RGB в HSB и обратно:

```
float[] RGBtoHSB(int red, int green, int blue, float[] hsb)
int HSBtoRGB(int hue, int saturation, int brightness)
```

Создав цвет, можно рисовать им в графическом контексте.

Как нарисовать чертеж

Основной метод рисования:

```
drawLine(int x1, int y1, int x2, int y2)
```

…вычерчивает текущим цветом отрезок прямой между точками с координатами (x1, y1) и (x2, y2).

Одного этого метода достаточно, чтобы, нарисовать любую картину по точкам, вычерчивая каждую точку с координатами (x, y) методом **drawLine (x, y, x, y)** и меняя цвета от точки к точке. Но никто, разумеется, не станет этого делать.

Другие графические примитивы:

- **drawRect(int x, int y, int width, int height)** – чертит прямоугольник со сторонами, параллельными краям экрана, задаваемый координатами верхнего левого угла (x, y), шириной width пикселов и высотой height пикселов;
- **draw3DRect(int x, int y, int width, int height, boolean raised)** – чертит прямоугольник, как будто выделяющийся из плоскости рисования, если аргумент raised равен true, или как будто вдавленный в плоскость, если аргумент raised равен false;
- **drawOval(int x, int y, int width, int height)** – чертит овал, вписанный в прямоугольник, заданный аргументами метода. Если width == height, то получится окружность;
- **drawArc(int x, int y, int width, int height, int startAngle, int arc)** – чертит дугу овала, вписанного в прямоугольник, заданный первыми четырьмя аргументами. Дуга имеет величину arc градусов и отсчитывается от угла startAngle. Угол отсчитывается в градусах от оси Ox. Положительный угол отсчитывается против часовой стрелки, отрицательный – по часовой стрелке;
- **drawRoundRect (int x, int y, int width, int height, int arcWidth, int arcHeight)** – чертит прямоугольник с закругленными краями. Закругления вычерчиваются четвертинками овалов, вписанных в прямоугольники шириной arcwidth и высотой arcHeight, построенные в углах основного прямоугольника;
- **drawPolyline(int[] xPoints, int[] yPoints, int nPoints)** – чертит ломаную с вершинами в точках xPoints[i], ypoints[i]) и числом вершин nPoints;
- **drawPolygon(int[] xPoints, int[] yPoints, int nPoints)** – чертит ЗЗМК-нутую ломаную, проводя замыкающий отрезок прямой между первой и последней точкой;
- **drawPolygon(Polygon p)** – чертит замкнутую ломаную, вершины которой заданы объектом p класса Polygon.

Класс **Polygon** рассмотрим подробнее.

Класс Polygon

Этот класс предназначен для работы с многоугольником, в частности, с треугольниками и произвольными четырехугольниками.

Объекты этого класса можно создать двумя конструкторами:

- **Polygon ()** – создает пустой объект;
- **Polygon(int[] xPoints, int[] yPoints, int nPoints)** – задаются вершины многоугольника (xPoints[i], yPoints[i]) и их число nPoints

После создания объекта в него можно добавлять вершины методом **addPoint(int x, int y)**.

Логические методы **contains ()** позволяют проверить, не лежит ли в многоугольнике заданная аргументами метода точка, отрезок прямой или целый прямоугольник со сторонами, параллельными сторонам экрана.

Логические методы **intersects ()** позволяют проверить, не пересекается ли с данным многоугольником отрезок прямой, заданный аргументами метода, или прямоугольник со сторонами, параллельными сторонам экрана.

Методы **getBounds()** и **getBounds2D()** возвращают прямоугольник, целиком содержащий в себе данный многоугольник.

Вернемся к методам класса Graphics. Несколько методов вычерчивают фигуры, залитые текущим цветом: **fillRect()**, **fill3DRect()**, **filiArc()**, **filioval()**, **filiPoiygon()**, **filiRoundRect()**. У них такие же аргументы, как и у соответствующих методов, вычерчивающих незаполненные фигуры.

Например, если вы хотите изменить цвет фона области рисования, то установите новый текущий цвет и начертите им заполненный прямоугольник величиной во всю область:

```
public void paint(Graphics g)(
Color initColor = g.getColor(); // Сохраняем исходный цвет
g.setColor(new Color(0, 0, 255)); // Устанавливаем цвет фона
// Заливаем область рисования
g.fillRect(0, 0, getSizeO.width – 1, getSize().height – 1);'
g.setColor(initColor); // Восстанавливаем исходный цвет
```

```
// Дальнейшие действия
}
```

Как видите, в классе Graphics собраны только самые необходимые средства рисования. Нет даже метода, задающего цвет фона (хотя можно задать цвет фона компонента методом**setBackground()** класса component). Средства рисования, вывода текста в область рисования и вывода изображений значительно дополнены и расширены в подклассе Graphics2D, входящем в систему Java 2D. Например, в нем есть метод задания цвета фона**setBackground(Color c).**

Перед тем как обратиться к классу Graphics2D, рассмотрим средства класса Graphics для вы вод а текста.

Как вывести текст. Как установить шрифт.

Для вывода текста в область рисования текущим цветом и шрифтом, начиная с точки (x, y), в, классе **Graphics** есть несколько методов:

- **drawstring (String s, int x, int y)** – выводит строку s;
- **drawBytes(byte[] b, int offset, int length, int x, int y)** – выводит length элементов массива байтов ь, начиная с индекса offset;
- **drawChars(chart] ch, int offset, int length, int x, int y)** – выводит length элементов массива символов ch, начиная с индекса offset.

Четвертый метод выводит текст, занесенный в объект класса, реализующего интерфейс**AttributedCharacteriterator**. Это позволяет задавать свой шрифт для каждого выводимого символа:

```
drawstring(AttributedCharacteriterator iter, int x, int y)
```

Точка (x, y) – это левая нижняя точка первой буквы текста на базовой линии (**baseline**) вывода шрифта.

Как установить шрифт

Метод **setFont(Font newFont)** класса Graphics устанавливает текущий шрифт для вывода текста.

Метод **getFont ()** возвращает текущий шрифт.

Как и все в языке Java, шрифт – это объект класса Font. Посмотрим, какие возможности предоставляет этот класс.

Как задать шрифт

Объекты класса Font хранят начертания (**glyphs**) символов, образующие шрифт. Их можно создать двумя Конструкторами:

- **Font (Map attributes)** – задает шрифт с заданными аргументом attributes атрибутами. Ключи атрибутов и некоторые их значения задаются константами класса TextAttnbute из пакета java.awt.font. Этот конструктор характерен для Java 2D и будет рассмотрен далее в настоящей главе.
- **Font (String name, int style, int size)** – задает Шрифт по имени name, со стилем style и размером size типографских пунктов. Этот конструктор характерен для JDK 1.1, но широко используется и в Java 2D в силу своей простоты.

Типографский пункт в России и некоторых европейских странах равен 0.376 мм, Точнее, 1/72 части французского дюйма. В англо-американской системе мер пункт равен 1/72 части английского дюйма, 0.351 мм. Этот-то пункт и применяется в компьютерной графике.

Имя шрифта name может быть строкой с физическим именем шрифта, например, "Courier New", или одна из строк "Dialog", "DialogInput",' "Monospaced", "Serif", "SansSerif", "Symbol". Это так называемые **логические имена шрифтов** (logical font names). Если name == null, то задается шрифт по умолчанию.

Стиль шрифта style – это одна из констант класса Font:

- **BOLD** – полужирный;
- **ITALIC** – курсив;
- **PLAIN** – обычный.

Полужирный курсив (**bolditalic**) можно задать операцией побитового сложения, Font. BOLD | Font. ITALIC, как это сделано в листинге 8.3.

При выводе текста логическим именам шрифтов и стилям сопоставляются **физические имена шрифтов** (font face name) или **имена семейств шрифтов** (font name). Это имена реальных шрифтов, имеющихся в графической подсистеме операционной системы.

Например, логическому имени "Serif" может быть сопоставлено имя семейства (**family**) шрифтов Times New Roman, а в сочетании со стилями – конкретные физические имена Times New Roman Bold, Times New Roman Italic. Эти шрифты должны находиться в составе шрифтов графической системы той машины, на которой выполняется приложение.

Список имен доступных шрифтов можно просмотреть следующими операторами:

```
Font[] fnt = Toolkit.getGraphicsEnvironment.getAHFonts();
for (int i = 0; i< fnt.length; i++)
System.out.println(fnt[i].getFontName());
```

В состав SUN J2SDK входит семейство шрифтов Lucida. Установив SDK, вы можете быть уверены, что эти шрифты есть в вашей системе.

Таблицы сопоставления логических и физических имен шрифтов находятся в файлах с именами:

- font.properties;
- font.properties.ar;
- font.properties.ja;
- font.properties.ru.

… и т. д. Эти файлы должны быть расположены в JDK в каталоге jdkl.3\jre\lib или каком-либо Другом подкаталоге lib корневого каталога JDK той машины, на которой выполняется приложение.

Нужный файл выбирается виртуальной машиной Java по окончании имени файла. Это окончание совпадает с международным кодом языка, установленного в локали или в системном свойстве **user.language** (см. рис. 6.2). Если у вас установлена русская локаль с международным кодом языка "ru", то для сопоставления будет выбран файл font.properties.ru. Если такой файл не найден, то применяется файл font.properties, не соответствующий никакой конкретной локали.

Поэтому можно оставить в системе только один файл font.properties, переписав в него содержимое нужного файла или создав файл заново. Для любой локали будет использоваться этот файл.

В листинге 9.1 показано сокращенное содержимое файла font.propeities.ru из JDK 1.3 для платформы MS Windows.

Листинг 9.1. Примерный файл font.properties.ru.

```
# %W% %E%
# Это просто комментарии
# AWT Font default Properties for Russian Windows
#
# Три сопоставления логическому имени "Dialog":
dialog.0=Arial,RUSSIAN_CHARSET
dialog.l=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
dialog.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED
# По три сопоставления стилям ITALIC, BOLD, ITALIC+BOLD:
dialog.italic.0=Arial Italic,RUSSIAN_CHARSET
dialog.italic.l=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
dialog.italic.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED
```

```
dialog.bold.0=Arial Bold,RUSSIAN_CHARSET
dialog.bold.l=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
dialog.bold.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED
dialog.bolditalic.0=Arial Bold Italic,RUSSIAN_CHARSET
dialog.bolditalic.l=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
dialog.bolditalic.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED
# По три сопоставления имени "Dialoglnput" и стилям:
dialoginput.0=Courier New,RUSSIAN_CHARSET
dialoginput.l=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
dialoginput.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED
dialoginput.italic.0=Courier New Italic,RUSSIAN_CHARSET
# И так далее
#
# По три сопоставления имени "Serif" и стилям:
serif.0=Times New Roman,RUSSIAN_CHARSET
serif.l=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
serif.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED
serif.italic.0=Times New Roman Italic,RUSSIAN_CHARSET
# И так далее
# Прочие логические имена
sansserif. CMArial,RUSSIAN_CHARSET
sansserif.l=WingDings,SVMBOL_CHARSET,NEED_CONVERTED
sansserif.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED
sansserif.italic. 0=Arial Italic,ROSSIAN_CHARSET
# И так далее
#
monospaced.0=Courier New,RUSSIAN_CHARSET
monospaced.l=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
monospaced.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED
monospaced.italic.0=Courier New Italic,RUSSIAN_CHARSET
# И так далее
# Default font definition
#
default.char=2751
# for backword compatibility
# Старые логические имена версии JDK 1.0
timesroman.0=Times New Roman,RUSSIAN_CHARSET
helvetica.0=Arial,RUSSIAN_CHARSET
courier.0=Courier New,RUSSIAN_CHARSET
zapfdingbats.0=WingDings,SYMBOL_CHARSET
# font filenames for reduced initialization time
# Файлы со шрифтами
filename.Arial=ARIAL.TTF
filename.Arial_Bold=ARIALBD.TTF
filename.Arial_Italic=ARIALI.TTF
filename.Arial_Bold_Italic=ARIALBI.TTF
filename.Courier_New=COUR.TTF
filename.Courier_New_Bold=COURBD.TTF
filename.Courier_New_Italic=COURI.TTF
filename.Courier_New_Bold_Italic=COURBI.TTF
filename.Times_New_Roman=TIMES.TTF
filename.Times_New_Roman_Bold=TlMESBD.TTF
filename.Times_New_Roman_Italic=TIMES3.TTF
filename.Times_New_Roman_Bold Italic=TIMESBI.TTF
filename.WingDings=WINGDING.TTF
filename.Symbol=SYMBOl.TTF
# name aliases
# Псевдонимы логических имен закомментированы
# alias.timesroman=serif
# alias.helvetica=sansserif
# alias.courier=monospaced
# Static FontCharset info
#
# Классы преобразования символов в байты
fontcharset.dialog.0=sun.io.CharToByteCP1251
fontcharset.dialog.l=sun.awt.windows.CharToByteWingDings
fontcharset.dialog.2=sun.awt.CharToByteSymbol
```

```
fontcharset.dialoginput.0=sun.io.CharToByteCP1251
fontcharset.dialoginput.1=sun.awt.windows.CharToByteWingDings
fontcharset.dialoginput.2=sun.awt.CharToByteSymbol
fontcharset.serif.0=sun.io.CharToByteCP1251
fontcharset.serif.1=sun.awt.windows.CharToByteWingDings
fontcharset.serif.2=sun.awt.CharToByteSymbol
fontcharset.sansserif.0=sun.io.CharToByteCP1251
fontcharset.sansserif.1=sun.awt.windows.CharToByteWingDings
fontcharset.sansserif.2=sun.awt.CharToByteSymbol
fontcharset.monospaced.0=sun.io.CharToByteCP1251
fontcharset.monospaced.1=sun.awt.windows.CharToByteWingDings
fontcharset.monospaced.2=sun.awt.CharToByteSymbol
# Exclusion Range info
#
# Не просматривать в этом шрифте указанные диапазоны
exclusion.dialog.0=0100-0400.0460-ffff
exclusion.dialoginput.0=0100-0400, 0460-ffff
exclusion.serif.0=0100-0400.04 60-ffff
exclusion.sansserif.0=0100-0400, 0460-ffff
exclusion.monospaced.0=0100-0400.0460-ffff
# charset for text input
#
# Вводимые байтовые символы кодируются в кириллический диапазон
# кодировки Unicode
inputtextcharset=RUSSIAN_CHARSET
```

Большая часть этого файла занята сопоставлениями логических и физических имен. Вы видите, что под номером 0:

- логическому имени "Dialog" сопоставлено имя семейства Arial;
- логическому имени "Dialoginput" сопоставлено имя семейства Courier New;
- логическому имени "Serif" сопоставлено имя семейства Times New Roman;
- логическому имени "Sansserif" сопоставлено имя семейства Arial;
- логическому имени "Monospaced" сопоставлено имя семейства Courier New.

Там, где указан стиль: dialog.italic, dialog.bold и т.д., подставлен соответствующий физический шрифт.

В строках листинга 9.1, начинающихся со слова filename, указаны файлы с соответствующими физическими шрифтами, например:

```
filename.Arial=ARIAL.TTF
```

Эти строки необязательны, но они ускоряют поиск файлов со шрифтами. Теперь посмотрите на последние строки листинга 9.1. Строка:

```
exclusion.dialog.0=0100-0400, 0460-ffff
```

…означает, что в шрифте, сопоставленном логическому имени "Dialog" под номером 0, а именно, Arial, не станут отыскиваться начертания (**glyphs**) символов с кодами в диапазонах '\u0100' – '\u0400' и '\u0460' – '\uFFFF'. Они будут взяты из шрифта, сопоставленного этому имени под номером 1, а именно, WingDings.

То же самое будет происходить, если нужные начертания не найдены в шрифте, сопоставленному логическому имени под номером 0. Не все файлы со шрифтами Unicode содержат начертания (**glyphs**) всех символов.

Если нужные начертания не найдены и в сопоставлении 1 (в данном примере в шрифте WingDings), они будут отыскиваться в сопоставлении 2 (т. е. в шрифте Symbol) и т. д. Подобных сопоставлений можно написать сколько угодно.

Таким образом, каждому логическому имени шрифта можно сопоставить разные диапазоны различных реальных шрифтов, а также застраховаться от отсутствия начертаний некоторых символов в шрифтах Unicode.

Все сопоставления под номерами 0, 1, 2, 3, 4 следует повторить для всех стилей: bold, italic, bolditalic.

Если в графической системе используются шрифты Unicode, как, например, в MS Windows NT/2000, то больше ни о чем беспокоиться не надо.

Если же графическая система использует байтовые ASCII-шрифты как, например, MS Windows 95/98/ME, то следует позаботиться об их правильной перекодировке в Unicode и обратно.

Для этого на платформе MS Windows используются константы Win32 API RUSSIAN_CHARSET, SYMBOL_CHARSET, ANSI_CHARSET, OEM_CHARSET И др., показывающие, какую кодовую таблицу использовать при перекодировке, так же, как это отмечалось в **главе 5** при создании строки из массива байтов.

Если логическим именам сопоставлены байтовые ASCII-шрифты (в примере это шрифты WingDings и Symbol), то необходимость перекодировки отмечается константой NEED_CONVERTED.

Перекодировкой занимаются методы специальных классов **charToByteCP1251**,**TiarToByteWingDings**, **CharToByteSyrnbol**. Они указываются для каждого сопоставления имен в строках, начинающихся со слова fontcharset. Эти строки обязательны для всех шрифтов, помеченных константой NEED_CONVERTED.

В последней строке файла указана кодовая страница для перекодировки в Unicode символов, вводимых в поля ввода:

```
inputtextcharset = RUSSIAN_CHARSET
```

Эта запись задает кодовую таблицу CP1251.

Итак, собираясь выводить строку str в графический контекст методом **drawstring ()**, мы создаем текущий шрифт конструктором класса Font, указывая в нем логическое имя шрифта, например, "Serif". Исполняющая система Java отыскивает в файле font.properties, соответствующем локальному языку, сопоставленный этому логическому имени физический шрифт операционной системы, например, Times New Roman. Если это Unicode-шрифт, то из него извлекаются начертания символов строки str по их кодировке Unicode и отображаются в графический контекст. Если это байтовый ASCII-шрифт, то строка str предварительно перекодируется в массив байтов методами класса, указанного в одной из строк fontcharset, например, CharToByteCP1251.

Хорошие примеры файлов font.properties.ru собраны на странице Сергея Астахова, указанной во введении.

Обсуждение этих вопросов и примеры файлов font.properties для X Window System даны в документации SUN J2SDK в файле docs/guide/intl /fontprop.html.

Завершая обсуждение логических и физических имен шрифтов, следует сказать, что в JDK 1.0 использовались логические имена "Helvetica", "TimesRoman", "Courier", замененные В JDK 1.1 Н3 "SansSerif", "Serif", "Monospaced", соответственно, из лицензионных соображений. Старые имена остались в файлах font.properties для совместимости.

При выводе строки в окно приложения очень часто возникает необходимость расположить ее определенным образом относительно других элементов изображения: центрировать, вывести над или под другим графическим объектом. Для этого надо знать метрику строки: ее высоту и ширину. Для измерения размеров отдельных символов и строки в целом разработан класс **FontMetrics**.

В Java 2D класс FontMetrics заменен классом TextLayout. Его мы рассмотрим в конце этой главы, а сейчас выясним, какую пользу можно извлечь из методов класса FontMetrics.

Класс FontMetrics

Класс **FontMetrics** является абстрактным, поэтому нельзя воспользоваться его конструктором. Для получения объекта класса FontMetrics, содержащего набор метрических характеристик шрифта f, надо обратиться к методу **getFontMetrics (f)** класса Graphics или класса Component.

Подробно с характеристиками компьютерных шрифтов можно познакомиться по книге.

Класс FontMetri.cs позволяет узнать ширину отдельного символа ch в пикселах методом charwidth(ch), общую ширину всех символов массива или под-массива символов или байтов методами **getchars()** и **getBytes()**, ширину целой строки str в пикселах методом**stringwidth(str)**.

Несколько методов возвращают в пикселах вертикальные размеры шрифта.

Интерлиньяж (leading) – расстояние между нижней точкой свисающих элементов таких букв, как р, у и верхней точкой выступающих элементов таких букв, как б, и, **в** следующей строке – возвращает метод **getLeading ()**.

Среднее расстояние от базовой линии шрифта до верхней точки прописных букв и выступающих элементов той же строки (**ascent**) возвращает метод **getAscent ()**, а максимальное – метод **getMaxAscent ()**.

Среднее расстояние свисающих элементов от базовой линии той же строки (**descent**) возвращает метод **getDescent ()**, а максимальное – метод **getMaxDescent()**.

Наконец, высоту шрифта (**height**) – сумму ascent + descent + leading – возвращает метод**getHeight ()**. Высота шрифта равна расстоянию между базовыми линиями соседних строк.

Эти элементы показаны на рис. 9.1.

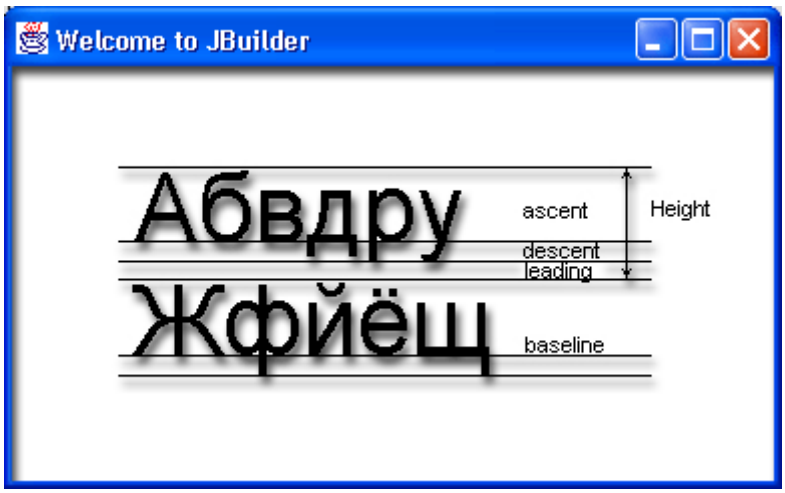

Рис. 9.1. Элементы шрифта

Дополнительные характеристики шрифта можно определить методами класса**LineMetrics** из пакета java.awt.font. Объект этого класса можно получить несколькими методами **getLineMetrics ()** класса FontMetrics.

Листинг 9.2 показывает применение графических примитивов и шрифтов, а рис. 9.2 – результат выполнения программы из этого листинга.

Листинг 9.2. Использование графических примитивов и шрифтов.

```
import java.awt.*;
import j ava.awt.event.*;
class GraphTest extends Frame{
GraphTest(String s) {
super(s);
setBounds(0, 0, 500, 300);
setVisible(true);
}
public void paint(Graphics g){
Dimension d = getSize();
```

```
int dx = d.width / 20, dy – d.height / 20;
g.drawRect(dx, dy + 20,
d.width -2 * dx, d.height -2 * dy -20);
g.drawRoundRect(2 * dx, 2 * dy + 20,
d.width -4 * dx, d.height -4 * dy -20, dx, dy);
g.fillArctd.width / 2 -dx, d.height -2 * dy + 1,
2 * dx, dy -1, 0, 360);
g.drawArctd.width / 2-3 * dx, d.height -3 * dy / 2-5,
dx, dy / 2, 0, 360);
g.drawArctd.width / 2 + 2 * dx, d.height -3 * dy / 2-5,
dx, dy / 2, 0, 360);
Font fl = new Font("Serif", Font.BOLD(Font.ITALIC, 2 * dy);
Font f2 = new Font ("Serif", Font.BOLD, 5 * dy / 2);
FontMetrics fml = getFontMetrics(fl);
FontMetrics fm2 = getFontMetrics(f2);
String s1 = "Всякая последняя ошибка";
String s2 = "является предпоследней.";
String s3 = "Закон отладки";
int firstLine = d.height / 3;
int nextLine = fml.getHeight();
int secondLine = firstLine + nextLine / 2;
g.setFont(f2);
g.drawstring(s3, (d.width-fm2.stringWidth(s3)) / 2, firstLine);
g.drawLine(d.width / 4, secondLine -2,
3 * d.width / 4, secondLine -2);
g.drawLine(d.width / 4, secondLine -1,
3 * d.width / 4, secondLine -1);
g.drawLine(d.width / 4, secondLine,
3 * d.width / 4, secondLine);
g.setFont(fl);
g.drawstring(s1, (d.width – fml.stringWidth(si)) 12,
firstLine + 2 * nextLine);
g.drawString(s2, (d.width – fml.stringWidth(s2)) / 2,
firstLine + 3 * nextLine);
}
public static void main(String[] args){
GraphTest f = new GraphTest(" Пример рисования");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
```

В листинге 9.2 использован простой класс **Dimension**, главная задача которого – хранить ширину и высоту прямоугольного объекта в своих полях width и height. Метод **getsize ()**класса component возвращает размеры компонента в виде объекта класса Dimension. В листинге 9.2 размеры компонента f типа GrapMest установлены в конструкторе методом**setBounds()** равными 500x300 пикселов.

Еще одна особенность листинга 9.2 – для вычерчивания толстой линии, отделяющей заголовок от текста, пришлось провести три параллельные прямые на расстоянии один пиксел друг от друга.

Как вы увидели из обзора класса Graphics и сопутствующих ему классов, средства рисования и вывода текста в этом классе весьма ограничены. Линии можно проводить только сплошные и только толщиной в один пиксел, текст выводится только горизонтально и слева направо, не учитываются особенности устройства вывода, например, разрешение экрана.

Эти ограничения можно обойти разными хитростями: чертить несколько параллельных линий, прижатых друг к другу, как в листинге 9.2, или узкий заполненный прямоугольник, выводить текст по одной букве, получить разрешение экрана методом **getScreenSize()**класса Java.awt.Toolkit и использовать его в дальнейшем. Но все

это затрудняет программирование, лишает его стройности и естественности, нарушает принцип KISS.

В Java 2 класс Graphics, в рамках системы Java 2D, значительно расширен классом Graphics2D.

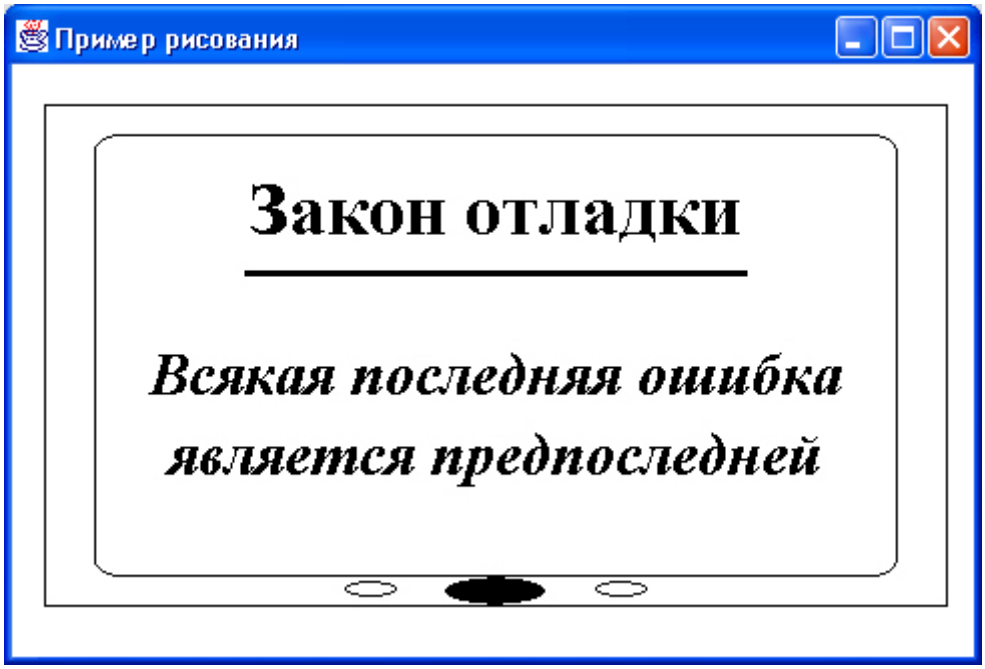

Рис. 9.2. Пример использования класса Graphics

Возможности Java 2D

В систему пакетов и классов Java 2D, основа которой– класс Graphics2D пакета java.awt, внесено несколько принципиально новых положений.

- Кроме координатной системы, принятой в классе Graphics и названной координатным пространством пользователя (**User Space**), введена еще система координат устройства вывода (**Device Space**): экрана монитора, принтера. Методы класса Graphics2D автоматически переводят (**transform**) систему координат пользователя в систему координат устройства при выводе графики.
- Преобразование координат пользователя в координаты устройства можно задать "вручную", причем преобразованием способно служить любое аффинное преобразование плоскости, в частности, поворот на любой угол и/или сжатие/растяжение. Оно определяется как объект класса **AffineTransform**. Его можно установить как преобразование по умолчанию методом **setTransform()**. Возможно выполнять преобразование "на лету" методами **transform ()** и **translate ()** и делать композицию преобразований методом **concatenate()**.
- Поскольку аффинное преобразование вещественно, координаты задаются вещественными, а не целыми числами.
- Графические примитивы: прямоугольник, овал, дуга и др., реализуют теперь новый интерфейс **shape** пакета java.awt. Для их вычерчивания можно использовать новый единый для всех фигур метод **draw()**, аргументом которого способен служить любой объект, реализовавший интерфейс shape. Введен метод **fill ()**, заполняющий фигуры–объекты класса, реализовавшего интерфейс shape.
- Для вычерчивания (**stroke**) линий введено понятие пера (**pen**). Свойства пера описывает интерфейс stroke. Класс **Basicstroke** реализует этот интерфейс. Перо обладает четырьмя характеристиками:
  - оно имеет толщину (**width**) в один (по умолчанию) или несколько пикселов;

- оно может закончить линию (**end cap**) закруглением – статическая константа CAP_ROUND, прямым обрезом – CAP_SQUARE (по умолчанию), или не фиксировать определенный способ окончания – CAP_BUTT;
- оно может сопрягать линии (**line joins**) закруглением – статическая константа**JOIN_ROOND**, отрезком прямой – JOIN_BEVEL, или просто состыковывать – JOIN_MITER (по умолчанию);
- оно может чертить линию различными пунктирами (**dash**) и штрих-пунктирами, длины штрихов и промежутков задаются в массиве, элементы массива с четными индексами задают длину штриха, с нечетными индексами – длину промежутка между штрихами.

- Методы заполнения фигур описаны в интерфейсе **Paint**. Три класса реализуют этот интерфейс. Класс **color** реализует его сплошной (**solid**) заливкой, класс **GradientPaint**– градиентным (**gradient**) заполнением, при котором цвет плавно меняется от одной заданной точки к другой заданной точке, класс **Texturepaint** – заполнением по предварительно заданному образцу (**pattern fill**).
- Буквы текста понимаются как фигуры, т. е. объекты, реализующие интерфейс shape, и могут вычерчиваться методом **draw ()** с использованием всех возможностей этого метода. При их вычерчивании применяется перо, все методы заполнения и преобразования.
- Кроме имени, стиля и размера, шрифт получил много дополнительных атрибутов, например, преобразование координат, подчеркивание или перечеркивание текста, вывод текста справа налево. Цвет текста и его фона являются теперь атрибутами самого текста, а не графического контекста. Можно задать разную ширину символов шрифта, надстрочные и подстрочные индексы. Атрибуты устанавливаются константами класса **TextAttribute**.
- Процесс визуализации (**rendering**) регулируется правилами (**hints**), определенными константами класса **RenderingHints**.

С такими возможностями Java 2D стала полноценной системой рисования, вывода текста и изображений. Посмотрим, как реализованы эти возможности, и как ими можно воспользоваться.

Преобразование координат. Класс AffineTransform.

Правило преобразования координат пользователя в координаты графического устройства (**transform**) задается автоматически при создании графического контекста так же, как цвет и шрифт. В дальнейшем его можно изменить методом **setTransform()** так же, как меняется цвет или шрифт. Аргументом этого метода служит объект класса AffineTransform из пакета java.awt.geom, подобно объектам класса color или Font при задании цвета или шрифта.

Рассмотрим подробнее класс AffineTransform.

Аффинное преобразование координат задается двумя основными конструкторами класса**AffineTransform**:

- **AffineTransform(double a, double b, double c, double d, double e, double f)**
- **AffineTransform (float a, float b, float c, float d, float e, float f)**

При этом точка с координатами (x, y) в пространстве пользователя перейдет в точку с координатами (a*x+c*y+e, b*x+d*y+f) в пространстве графического устройства.

Такое преобразование не искривляет плоскость – прямые линии переходят в прямые, углы между линиями сохраняются. Примерами аффинных преобразований служат повороты вокруг любой точки на любой угол, параллельные сдвиги, отражения от осей, сжатия и растяжения по осям.

Следующие два конструктора используют в качестве аргумента массив (a, b, c, d, e, f) или (a, b, c, d}, если e = f = o, составленный из таких же коэффициентов в том же порядке:

```
AffineTransform(double[] arr)
AffineTransform(float[] arr)
```

Пятый конструктор создает новый объект по другому, уже имеющемуся, объекту:

```
AffineTransform(AffineTransform at)
```

Шестой конструктор – конструктор по умолчанию – создает тождественное преобразование:

```
AffineTransform ()
```

Эти конструкторы математически точны, но неудобны при задании конкретных преобразований. Попробуйте рассчитать коэффициенты поворота на 57° вокруг точки с координатами (20, 40) или сообразить, как будет преобразовано пространство пользователя после выполнения методов:

```
AffineTransform at = new AffineTransform(-1.5, 4.45, – 0.56, 34.7, 2.68, 0.01);
g.setTransform(at);
```

Во многих случаях удобнее создать преобразование статическими методами, возвращающими объект класса AffineTransform.

- **getRotateInstance (double angle)** – возвращает поворот на угол angle, заданный в радианах, вокруг начала координат. Положительное направление поворота таково, что точки оси Ox поворачиваются в направлении к оси Oy. Если оси координат пользователя не менялись преобразованием отражения, то положительное значение angle задает поворот по часовой стрелке.
- **getRotateInstance(double angle, double x, double y)** – такой же поворот вокруг точки с координатами (x, y).
- **getScaleInstance (double sx, double sy)** – изменяет масштаб по оси Ox в sx раз, по оси Oy – в sy раз.
- **getShareInstance (double shx, double shy)** – преобразует каждую точку (x, y) в точку (x+shx*y, shy*x+y).
- **getTranslateInstance (double tx, double ty)** – сдвигает каждую точку (x, y) в точку (x+tx, y+ty).

Метод **createinverse ()** возвращает преобразование, обратное текущему преобразованию.

После создания преобразования его можно изменить методами:

- **setTransform(AffineTransform at)**
- **setTransform(double a, double b, double c, double d, double e, double f)**
- **setToIdentity()**
- **setToRotation(double angle)**
- **setToRotation(double angle, double x, double y)**
- **setToScale(double sx, double sy)**
- **setToShare(double shx, double shy)**
- **setToTranslate(double tx, double ty)**

… сделав текущим преобразование, заданное одним из этих методов.

Преобразования, заданные методами:

- **concatenate(AffineTransform at)**
- **rotate(double angle)**
- **rotate(double angle, double x, double y)**
- **scale(double sx, double sy)**
- **shear(double shx, double shy)**
- **translate(double tx, double ty)**

… выполняются перед текущим преобразованием, образуя композицию преобразований.

Преобразование, заданное методом **preconcatenate(AffineTransform at)**, напротив, осуществляется после текущего преобразования.

Прочие методы класса **AffineTransform** производят преобразования различных фигур в пространстве пользователя.

Пора привести пример. Добавим в начало метода **paint ()** в листинге 9.2 четыре оператора, как записано в листинге 9.3.

Листинг 9.3. Преобразование пространства пользователя.

```
// Начало листинга 9.2…
public void paint(Graphics gr){
Graphics2D g = (Graphics2D)gr;
AffineTransform at =
AffineTransform.getRotatelnstance(-Math.PI/4.0, 250.0.150.0);
at.concatenate(
new AffineTransform(0.5, 0.0, 0.0, 0.5, 100.0, 60.0));
g.setTransform(at);
Dimension d = getSize();
// Продолжение листинга 9.2
```

Метод **paint ()** начинается с получения экземпляра **g** класса Graphics2D простым приведением аргумента gr к типу Graphics2D. Затем, методом **getRotatelnstance ()**определяется поворот на 45° против часовой стрелки вокруг точки (250.0, 150.0). Это преобразование– экземпляр at класса AffineTransform. Метод **concatenate ()**, выполняемый объектом at, добавляет к этому преобразованию сжатие в два раза по обеим осям координат и перенос начала координат в точку (100.0, 60.0). Наконец, композиция этих преобразований устанавливается как текущее преобразование объекта g методом **setTransform().**

Преобразование выполняется в следующем порядке. Сначала пространство пользователя сжимается в два раза вдоль обеих осей, затем начало координат пользователя – левый верхний угол – переносится в точку (100.0, 60.0) пространства графического устройства. Потом картинка поворачивается на угол 45° против часовой стрелки вокруг точки (250.0, 150.0).

Результат этих преобразований показан на рис. 9.3.

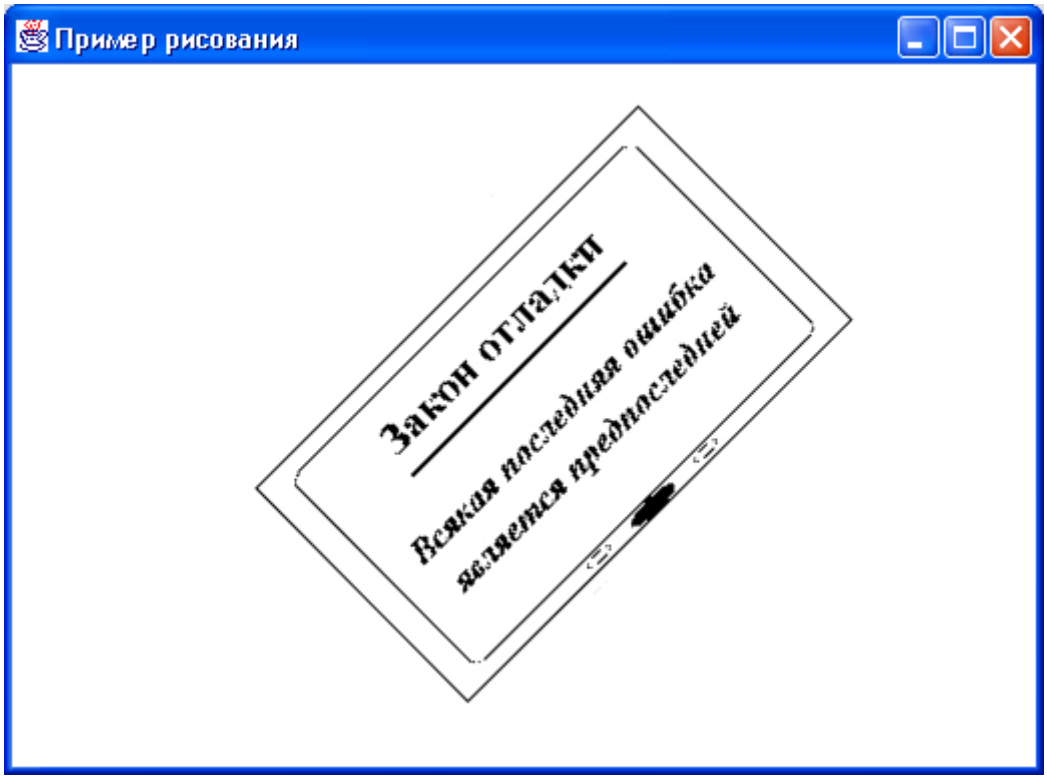

Рис. 9.3. Преобразование координат

Рисование фигур средствами Java2D. Класс BasicStroke.

Характеристики пера для рисования фигур описаны в интерфейсе stroke. В Java 2D есть пока только один класс, реализующий этот интерфейс – класс **BasicStroke**.

Конструкторы класса **BasicStroke** определяют характеристики пера. Основной конструктор:

```
BasicStroke(float width, int cap, int join, float miter, float[] dash, float dashBegin)
```

…задает:

- толщину пера width в пикселах;
- оформление конца линии cap; это одна из констант:
  - CAP_ROUND – закругленный конец линии;
  - CAP_SQUARE – квадратный конец линии;
  - CAP_BUTT – оформление отсутствует;
- способ сопряжения линий join; это одна из констант:
  - JOIN_ROUND – линии сопрягаются дугой окружности;
  - JOIN_BEVEL – линии сопрягаются отрезком прямой, перпендикуляр-ным биссектрисе угла между линиями;
  - JOIN_MITER – линии просто стыкуются;
- расстояние между линиями miter, начиная с которого применяется сопряжение JOIN_MITER;
- длину штрихов и промежутков между штрихами – массив dash; элементы массива с четными индексами задают длину штриха в пикселах, элементы с нечетными индексами – длину промежутка; массив перебирается циклически;
- индекс dashBegin, начиная с которого перебираются элементы массива
- dash.

Остальные конструкторы задают некоторые характеристики по умолчанию:

- BasicStroke (float width, int cap, int join, float miter) – сплошная линия;
- BasicStroke (float width, int cap, int join) – сплошная линия с сопряжением JOIN_ROUND или JOIN_BEVEL; для сопряжения JOIN_MITER задается значение miter = 10.0f;
- BasicStroke (float width) – прямой обрез CAP_SQUARE и сопряжение JOIN_MITER со значением miter = 10.0f;
- BasicStroke () – ширина1. 0f.

Лучше один раз увидеть," чем сто раз прочитать. В листинге 9.4 определено пять перьев с разными характеристиками, рис, 9.4 показывает, как они рисуют.

Листинг 9.4. Определение перьев.

```
import j ava.awt.*;
import j ava.awt.geom. *;
import j ava.awt.event.*;
class StrokeTest extends Frame{
StrokeTest(String s) {
super (s);
setSize(500, 400);
setvisible(true);
addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev)(
System.exit(0);
}
});
}
public void paint(Graphics gr){
Graphics2D g = (Graphics2D)gr;
g.setFont(new Font("Serif", Font.PLAIN, 15));
BasicStroke penl = new BasicStroke(20, BasicStroke.CAP_BUTT,
BasicStroke.JOIN_MITER,30);
BasicStroke pen2 = new BasicStroke(20, BasicStroke.CAP_ROUND,
BasicStroke.JOIN_ROUND);
BasicStroke pen3 = new BasicStroke(20, BasicStroke.CAP_SQUARE,
BasicStroke.JOIN_BEVEL);
```

```
floatf] dashl = {5, 20};
BasicStroke pen4 = new BasicStroke(10, BasicStroke.CAP_ROUND,
BasicStroke.JOIN_BEVEL, 10, dashl, 0);
float[] dash2 = (10, 5, 5, 5};
BasicStroke pen5 = new BasicStroke(10, BasicStroke.CAP_BUTT, BasicStroke.JOIN_BEVEL,
10, dash2, 0);
g.setStroke(penl);
g.draw(new Rectangle2D.Double(50, 50, 50, 50));
g.draw(new Line2D.Double(50, 180, 150, 180));
g.setStroke(pen2);
g.draw(new Rectangle2D.Double(200, 50, 50, 50));
g.draw(new Line2D.Double(50, 230, 150, 230));
g.setStroke(pen3);
g.draw(new Rectangle2D.Double(350, 50, 50, 50));
g.draw(new Line2D.Double(50, 280, 150, 280));
g.drawstring("JOIN_MITER", 40, 130);
g.drawstring("JOIN_ROUND", 180, 130);
g.drawstring("JOINJBEVEL", 330, 130);
g.drawstring("CAP_BUTT", 170, 190);
g.drawstring("CAP_ROUND", 170, 240);
g.drawstring("CAP_SQUARE", 170, 290);
g.setStroke(pen5);
g.drawfnew Line2D.Double(50, 330, 250, 330));
g.setStroke(pen4);
g.draw(new Line2D.Double(50, 360, 250, 360));
g.drawString("{10, 5, 5, 5,…}", 260, 335);
g.drawstring(<strong>(5</strong>, 10,…}", 260, 365);
}
public static void main(String[] args){
new StrokeTest("Моя программа");
}
}
```

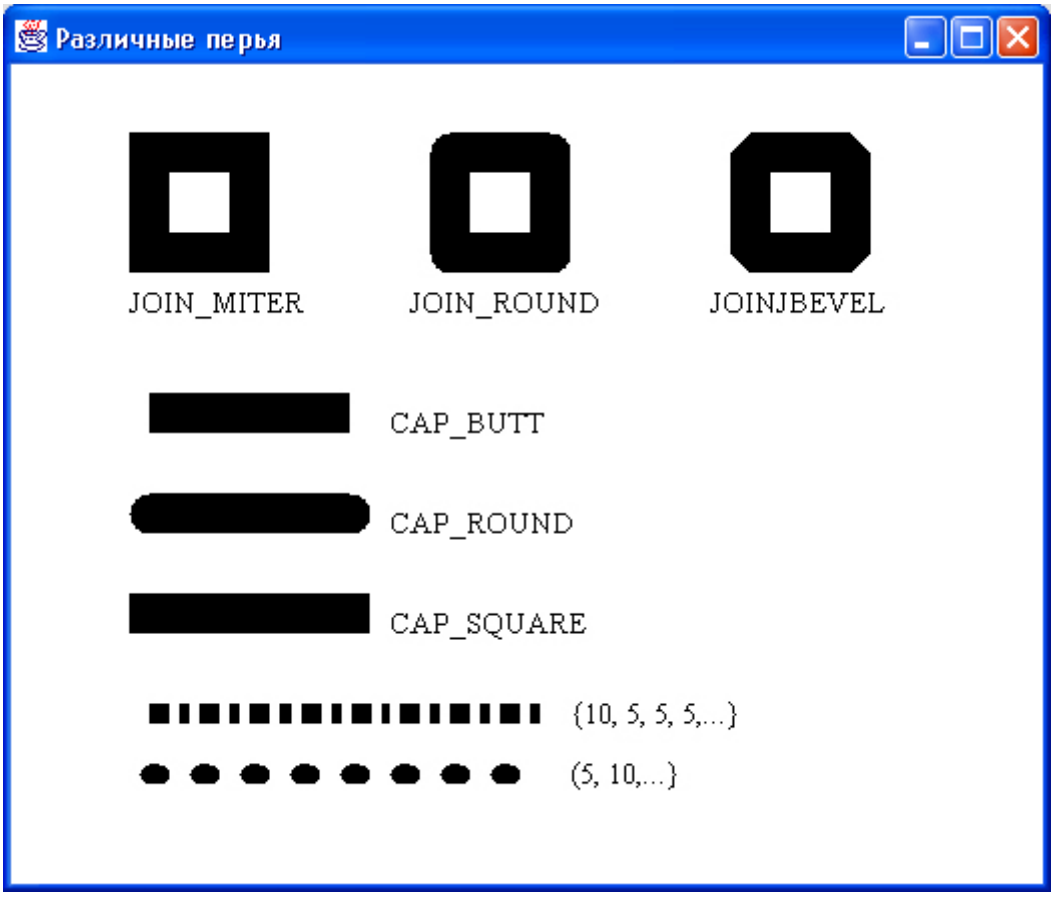

Рис. 9.4. Перья с различными характеристиками

После создания пера одним из конструкторов и установки пера методом **setStroke ()**можно рисовать различные фигуры методами **draw()** и **fill()**.

Общие свойства фигур, которые можно нарисовать методом **draw ()** класса Graphics2D, описаны в интерфейсе shape. Этот интерфейс реализован для создания обычного набора фигур – прямоугольников, прямых, эллипсов, дуг, точек – классами **Rectangle2D**,**RoundRectangle2D**, **Line2D**, **Ellipse2D**, **Arc2D**, **Point2D** пакета java.awt.geom. В этом пакете есть еще классы Cubiccurve2D и QuadCurve2D для создания кривых третьего и второго порядка.

Все эти классы абстрактные, но существуют их реализации – вложенные классы **Double**и **Float** для задания координат числами соответствующего типа. В Листинге 9.4 использованы классы **Rectangle2D.Double** и **Line2d.Double** для вычерчивания прямоугольников и отрезков.

В пакете java.awt.geom есть еще один интересный класс – **GeneralPath**. Объекты этого класса могут содержать сложные конструкции, составленные из отрезков прямых или кривых линий и прочих фигур, соединенных или не соединенных между собой. Более того, поскольку этот класс реализует интерфейс shape, его экземпляры сами являются фигурами и могут быть элементами других объектов класса GeneralPath.

Класс GeneralPath

Вначале создается пустой объект класса GeneralPath конструктором по умолчанию**GeneralPath ()** или объект, содержащий одну фигуру, конструктором **GeneralPath (Shape sh)**.

Затем к этому объекту добавляются фигуры методом **append(Shape sh, boolean connect)**.

Если параметр connect равен true, то новая фигура соединяется с предыдущими фигурами с помощью текущего пера.

В объекте есть текущая точка. Вначале ее координаты (0, 0), затем ее можно переместить в точку (x, y) методом **moveTo (float x, float y)**.

От текущей точки к точке (x, y) можно провести:

- отрезок прямой методом **lineTo(float x, float y);**
- отрезок квадратичной кривой методом **quadTo(float x1, float y1, float x, float y)**,
- кривую Безье методом **curveTo(float xl, float yl, float x2, float y2, float x, float y)**.

Текущей точкой после этого становится точка (x, y). Начальную и конечную точки можно соединить методом **closePath()**. Вот как можно создать треугольник с заданными вершинами:

```
GeneralPath p = new GeneralPath();
p.moveTo(xl, yl); // Переносим текущую точку в первую вершину,
p.lineTo(x2, y2); // проводим сторону треугольника до второй вершины,
p.lineTo(x3, y3); // проводим вторую сторону,
p.closePathf); // проводим третью сторону до первой вершины
```

Способы заполнения фигур определены в интерфейсе Paint. В настоящее время Java 2D содержит три реализации этого интерфейса – классы color, **GradientPaint** и **TexturePamt**. Класс Color нам известен, посмотрим, какие способы заливки предлагают классы**GradientPaint** и **TexturePaint**.

Классы GradientPaint и TexturePaint

Класс GradientPaint предлагает сделать заливку следующим образом.

В двух точках м и N устанавливаются разные цвета. В точке M(xi, yi) задается цвет cl, в точке b1(x2, y2) – цвет c2. Цвет заливки гладко меняется от el к c2 вдоль прямой, соединяющей точки м и м, оставаясь постоянным вдоль каждой прямой, перпендикулярной прямой мы. Такую заливку создает конструктор:

```
GradientPaint(float xl, float yl, Color cl, float x2, float y2, Color c2)
```

При этом вне отрезка мы цвет остается постоянным: за точкой м – цвет c1, за точкой b1 – цвет c2.

Второй конструктор:

```
GradientPaint(float xl, float yl, Color cl, float x2, float y2, Color c2, boolean cyclic)
```

…при задании параметра cyclic == true повторяет заливку полосы мы во всей заливаемой фигуре.

Еще два конструктора задают точки как объекты класса Point2D.

Класс TexturePaint поступает сложнее. Сначала создается буфер – объект класса**BufferedImage** из пакета java.awt .image. Это большой сложный класс.

Мы с ним еще встретимся в главе 15, а пока нам понадобится только его графический контекст, управляемый экземпляром класса Graphics2D. Этот экземпляр можно получить методом **createGraphics ()** класса Bufferedlmage. Графический контекст буфера заполняется фигурой, которая будет служить образцом заполнения.

Затем по буферу создается объект класса TexturePaint. При этом еще задается прямоугольник, размеры которого будут размерами образца заполнения. Конструктор выглядит так:

```
TexturePaint(Bufferedlmage buffer, Rectangle2D anchor)
```

После создания заливки – объекта класса color, GradientPaint или TexturePaint – она устанавливается в графическом контексте методом **setPaint (Paint p)** и используется в дальнейшем методом **fill (Shape sh)**. Все это демонстрирует листинг 9.5 и рис. 9.5.

Листинг 9.5. Способы заливки.

```
import java.awt.*;
import Java.awt.geom.*;
import java.awt.image.*;
import j ava.awt.event.*;
class PaintTest extends Frame{ PaintTest(String s){ super(s);
setSize(300, 300);
setVisible(true);
addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0); }
});
}
public void paint(Graphics gr){
Graphics2D g = (Graphics2D)gr;
Bufferedlmage bi =
new Bufferedlmage(20, 20, Bufferedlmage.TYPE_IMT_RGB);
Graphics2D big = bi.createGraphics();
big.draw(new Line2D.Double(0.0, 0.0, 10.0, 10.0));
big.draw(new Line2D.Double(0.0, 10.0, 10.0, 0.0));
TexturePaint tp = new TexturePaint(bi,
new Rectangle2D.Double(0.0, 0.0, 10.0, 10.0));
g.setPaint(tp);
g.fil(new Rectangle2D. Double (50, 50, 200, 200));
GradientPaint gp =
new GradientPaint(100, 100, Color.white,
150, 150, Color.black, true); g.setPaint(gp);
g.filKnew Ellipse2D.Double (100, 100, 200, 200));
}
public static void main(String[] args){
new PaintTest(" Способы заливки");
}
}
```

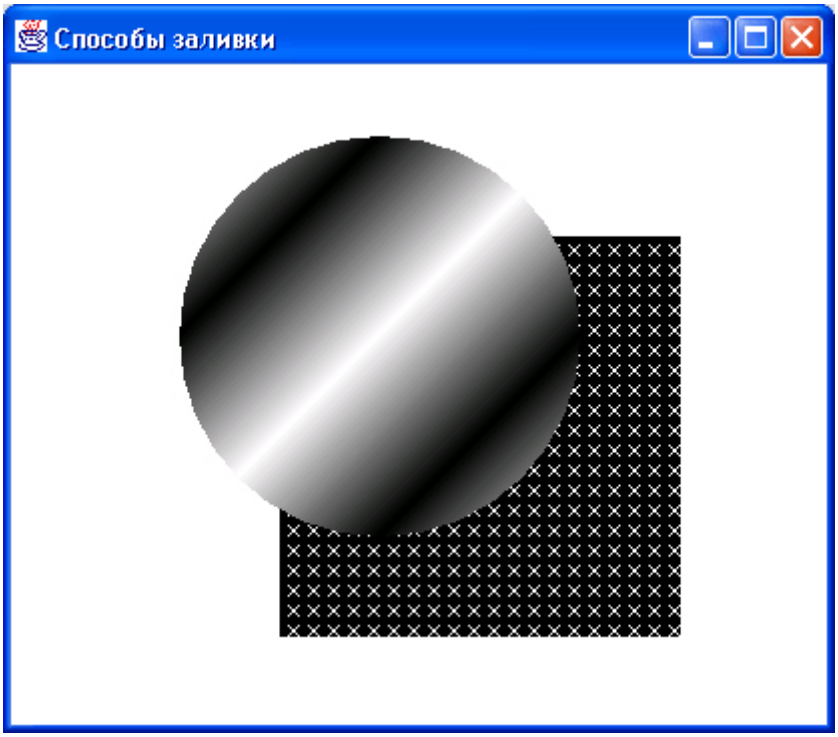

Рис. 9.5. Способы заливки

Вывод текста средствами Java 2D

Шрифт – объект класса Font – кроме имени, стиля и размера имеет еще полтора десятка атрибутов: подчеркивание, перечеркивание, наклон, цвет шрифта и цвет фона, ширину и толщину символов, аффинное преобразование, расположение слева направо или справа налево.

Атрибуты шрифта задаются как статические константы класса TextAttribute. Наиболее используемые атрибуты перечислены в табл. 9.1.

Таблица 9.1. Атрибуты шрифта.

Атрибут	Значение
BACKGROUND	Цвет фона. Объект, реализующий интерфейс Paint
FOREGROUND	Цвет текста. Объект, реализующий интерфейс Paint
BIDI EMBEDDED	Уровень вложенности просмотра текста. Целое от 1 до 1 5
CHAR_REPLACEMENT	Фигура, заменяющая символ. Объект GraphicAttribute
FAMILY	Семейство шрифта. Строка типа string
FONT	Шрифт. Объект класса Font
JUSTIFICATION	Допуск при выравнивании абзаца. Объект класса Float со значениями от 0.0 до 1.0. Есть две константы: JUSTIFICATION FULL И JUSTIFICATION NONE
POSTURE	Наклон шрифта. Объект класса Float. Есть две константы: POSTURE_OBLIQUE И POSTURE_REGULAR
RUNJHRECTION	Просмотр текста: RUN DIRECTION LTR – слева направо, RUN DIRECTION RTL – справа налево
SIZE	Размер шрифта в пунктах. Объект класса Float
STRIKETHROUGH	Перечеркивание шрифта. Задается константой STRIKETHROUGH ON, по умолчанию перечеркивания нет
SUPERSCRIPT	Подстрочные или надстрочные индексы. Константы: SUPERSCRIPT_NONE, SUPERSCRIPT_SUB, SUPERSCRIPT_SUPER
SWAP COLORS	Замена местами цвета текста и цвета фона. Константа SWAP COLORS ON, по умолчанию замены нет
TRANSFORM	Преобразование шрифта. Объект класса AffineTransform

Атрибут	Значение
UNDERLINE	Подчеркивание шрифта. Константы: UNDERLINE_ON, UNDERLINE_LOW_DASHED, UNDERLINE_LOW_DOTTED, UNDERLINE LOW GRAY, UNDERLINE LOW ONE PIXEL, UNDERLINE_LOW_TWO_PIXEL
WEIGHT	Толщина шрифта. Константы: WEIGHT ULTRA LIGHT, WEIGHT _ EXTRA_LIGHT, WEIGHT _ LIGHT, WEIGHT _ DEMILIGHT, WEIGHT _ REGULAR, WEIGHT _ SEMIBOLD, WEIGHT MEDIUM, WEIGHT DEMIBOLD, WEIGHT _ BOLD, WEIGHT HEAVY, WEIGHT _ EXTRABOLD, WEIGHT _ ULTRABOLD
WIDTH	Ширина шрифта. Константы: WIDTH CONDENSED,WIDTH SEMI CONDENSED, WIDTH REGULAR, WIDTH_SEMI_EXTENDED, WIDTH_EXTENDED

К сожалению, не все шрифты позволяют задать все атрибуты. Посмотреть список допустимых атрибутов для данного шрифта можно методом **getAvailableAttributes()** класса Font.

В классе Font есть конструктор **Font(Map attributes)**, которым можно сразу задать нужные атрибуты создаваемому шрифту. Это требует предварительной записи атрибутов в специально созданный для этой цели объект класса, реализующего Интерфейс Map: Класса HashMap, WeakHashMap или Hashtable (см. главу 7). Например:

```
HashMap hm = new HashMap ();
hm.put(TextAttribute.SIZE, new Float(60.Of));
hm.put(TextAttribute.POSTURE, TextAttribute.POSTUREJDBLIQUE);
Font f = new Font(hm);
```

Можно создать шрифт и вторым конструктором, которым мы пользовались в листинге 9.2, а потом добавлять и изменять атрибуты методами **deriveFont ()** Класса Font.

Текст в Java 2D обладает собственным контекстом – объектом класса FontRenderContext, хранящим всю информацию, необходимую для вывода текста. Получить его можно методом**getFontRendercontext ()** класса Graphics2D.

Вся информация о тексте, в том числе и об его контексте, собирается в объекте класса TextLayout. Этот класс в Java 2D заменяет класс FontMetrics.

В конструкторе класса TextLayout задается текст, шрифт и контекст. Начало метода **paint ()** со всеми этими определениями может выглядеть так:

```
public void paint(Graphics gr){
Graphics2D g = (Graphics2D)gr;
FontRenderContext frc = g.getFontRenderContex-t();
Font f = new Font("Serif", Font.BOLD, 15);
String s = "Какой-то текст";
TextLayout tl = new TextLayout(s, f, frc); // Продолжение метода }
```

В классе TextLayout есть не только более двадцати методов **getxxxo**, позволяющих узнать различные сведения о тексте, его шрифте и контексте, но и метод:

```
draw(Graphics2D g, float x, float y)
```

…вычерчивающий содержимое объекта класса TextLayout в графический области g, начиная с точки (x, y).

Еще один интересный метод:

```
getOutline(AffineTransform at)
```

…возвращает контур шрифта в виде объекта shape. Этот контур можно затем заполнить по какому-нибудь образцу или вывести только контур, как показано в листинге 9.6.

Листинг 9.6. Вывод текста средствами Java 20.

```
import java.awt.*;
import j ava.awt.font.*;
import j ava.awt.geom.*;
import java.awt.event.*
class StillText extends Frame{
StillText(String s) {
super(s);
setSize(400, 200);
```

```
setvisible(true);
addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
public void paint(Graphics gr){
Graphics2D g = (Graphics2D)gr;
int w = getSize().width, h = getSize().height;
FontRenderContext frc = g.getFontRenderContext();
String s = "Тень";
Font f = new Font("Serif", Font.BOLD, h/3);
TextLayout tl = new TextLayout(s, f, frc);
AffineTransform at = new AffineTransform();
at.setToTranslation(w/2-tl.getBounds().getwidth()/2, h/2);
Shape sh = tl.getOutline(at);
g.draw(sh);
AffineTransform atsh =
new AffineTransform(1, 0.0, 1.5, - 1, 0.0, 0.0);
g.transform(at);
g.transform(atsh);
Font df = f.deriveFont(atsh);
TextLayout dtl = new TextLayout(s, df, frc);
Shape sh2 = dtl.getOutline(atsh);
g.fill(sh2); } public static void main(Stnng[] args) {
new StillText(" Эффект тени");
}
}
```

На рис. 9.6 показан вывод этой программы.

Рис. 9.6. Вывод текста средствами Java 2D

Еще одна возможность создать текст с атрибутами – определить объект класса Attributedstring из пакета j ava.text. Конструктор этого класса:

```
AttributedString(String text, Map attributes)
```

…задает сразу и текст, и его атрибуты. Затем можно добавить или изменить характеристики текста одним их трех методов **addAttibute ()**.

Если текст занимает несколько строк, то встает вопрос его форматирования. Для этого вместо класса TextLayout используется класс LineBreakMeasurer, методы которого позволяют отформатировать абзац. Для каждого сегмента текста можно получить экземпляр класса TextLayout и вывести текст, используя его атрибуты.

Для редактирования текста необходимо отслеживать курсором (**caret**) текущую позицию в тексте. Это осуществляется методами класса TextHitinfo, а методы класса TextLayout позволяют получить позицию курсора, выделить блок текста" и подсветить его.

Наконец, можно задать отдельные правила для вывода каждого символа текста. Для этого надо получить экземпляр класса Glyphvector методом **createGiyphvector ()** класса Font, изменить позицию символа методом **setciyphPosition()**, задать преобразование символа, если это допустимо для данного шрифта, методом **setciyphTransform()**, и вывести измененный текст методом **drawGiyphVector ()** класса Graphics2D. Все это показано в листинге 9.7 и на рис. 9.7 – выводе программы листинга 9.7.

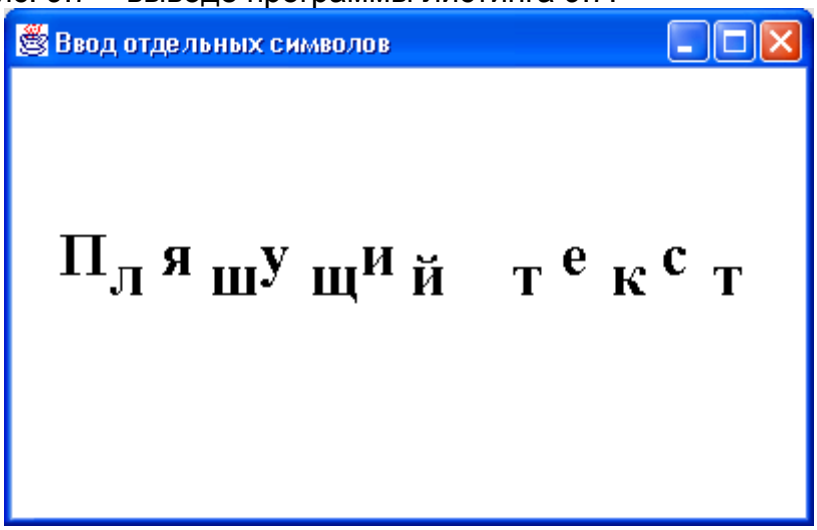

Рис. 9.7. Вывод отдельных символов

Листинг 9.7. Вывод отдельных символов.

```
import j ava.awt.*;
import Java.awt.font.*;
import java.awt.geom.*;
import j ava.awt.event.*;
class GlyphTest extends Frame{ GlyphTest(String s){ super(s);
setSize(400, 150);
setVisible(true);
addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
public void paint(Graphics gr){
int h = 5;
Graphics2D g = (Graphics2D)gr;
FontRenderContext frc = g.getFontRenderContext();
Font f = new Font("Serif", Font.BOLD, 30);
GlyphVector gv = f.createGiyphvector(frc, "Пляшущий текст");
int len = gv.getNumGlyphs();
for (int i = 0; i < len; i++){
Point2D.Double p = new Point2D.Double(25 * i, h = - h);
gv.setGlyphPosition(i, p);
}
g.drawGiyphVector(gv, 10, 100); }
public static void main(String[] args)(
new GlyphTest(" Вывод отдельных символов");
}
}
```

Методы улучшения визуализации

Визуализацию (**rendering**) созданной графики можно усовершенствовать, установив один из методов (**hint**) улучшения одним из методов класса Graphics2D:

```
setRenderingHints(RenderingHints.Key key, Object value)
setRenderingHints(Map hints)
```

Ключи – методы улучшения – и их значения задаются константами класса RenderingHints, перечисленными в табл. 9.2.

Таблица 9.2. Методы визуализации и их значения.

Метод	Значение
KEY_ANTIALIASING	Размывание крайних пикселов линий для гладкости изображения; три значения, задаваемые константами: VALUE ANTIALIAS DEFAULT, VALUE ANTIALIAS ON, VALUE~ANTIALIAS OFF
KEY_TEXT_ANTTALIAS ING	То же для текста. Константы: VALUE TEXT ANTIALIASING_DEFAULT, VALUE TEXT ANTIALIASING ON, VALUE_TEXT_ANTIALIASING_OFF
KEY RENDERING	Три типа визуализации. Константы: VALUE RENDER SPEED, VALUE RENDER QUALITY, VALUE RENDER DEFAULT
KEY COLOR RENDERING	То же для цвета. Константы: VALUE COLOR RENDER_SPEED, VALUE COLOR RENDER QUALITY, VALUE COLOR RENDER DEFAULT
KEY ALPHA INTERPOLATION	Плавное сопряжение линий. Константы: VALUE ALPHA INTERPOLATION_SPEED, VALUE ALPHA INTERPOLATION QUALITY, VALUE_ALPHA_INTERPOLATION_DEFAULT
KEY_INTERPOLATION	Способы сопряжения. Константы: VALUE_INTERPOLATION_BILINEAR, VALUE INTERPOLATION BICUBIC, VALUE INTERPOLATION_NEAREST_NEIGHBOR
KEY_DITHERING	Замена близких цветов. Константы: VALUE DITHER ENABLE, VALUE DITHER DISABLE, VALUE DITHER DEFAULT

Не все графические системы обеспечивают выполнение этих методов, поэтому задание указанных атрибутов не означает, что определяемые ими методы будут применяться на самом деле.

Вот как может выглядеть начало метода **paint ()** с указанием методов улучшения визуализации:

```
public void paint(Graphics gr){
Graphics2D g = (Graphics2D)gr;
g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
g.setRenderingHint(RenderingHints.KEY_RENDERING,
RenderingHints.VALUE_RENDER_QUALITY);
// Продолжение метода
}
```

Заключение

В этой главе мы, разумеется, не смогли подробно разобрать все возможности Java 2D. Мы не коснулись моделей задания цвета и смешивания цветов, печати графики и текста, динамической загрузки шрифтов, изменения области рисования. В главе 15 мы рассмотрим средства Java 2D для работы с изображениями, в главе 18 – средства печати.

В документации SUN J2SDK, в каталоге docs\guide\2d\spec, есть руководство Java 2 API Guide с обзором всех возможностей Java 2D. Там помещены ссылки на руководства и пособия по Java 2D. В каталоге demo\jfc\Java2D\src приведены исходные тексты программ, использующих Java 2D.

Основные компоненты

- ## Класс Component
  Графическая библиотека AWT предлагает более двадцати готовых компонентов. Они показаны на рис. 8.2. Наиболее часто используются подклассы класса Component: классы Button, Canvas, Checkbox, Choice, Container, Label, List, Scrollbar, TextArea, TextField, Panel, ScrollPane, Window, Dialog, FileDialog, Frame.
- ## Класс Cursor
  Основа класса – статические константы, определяющие форму курсора: | CROSSHAIR_CURSOR – курсор в виде креста, появляется обычно при поиске позиции для размещения какого-то элемента; | DEFAULT_CURSOR – обычная форма курсора – стрелка влево вверх;
- ## Как создать свой курсор
  Кроме этих предопределенных курсоров можно задать свою собственную форму курсора. Ее тип носит название CUSTOM_CURSOR. Сформировать свой курсор можно методом: | createCustomCursor(Image cursor, Point hotspot, String name) | …создающим объект класса cursor и возвращающим ссылку на него.
- ## События. Класс Container.
  Событие ComponentEvent происходит при перемещении компонента, изменении его размера, удалении с экрана и появлении на экране. | Событие FocusEvent возникает при получении или потере фокуса. | Событие KeyEvent проявляется при каждом нажатии и отпускании клавиши, если компонент имеет фокус ввода.
- ## События. Компонент Label.
  Кроме событий Класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent при добавлении и удалении компонентов в контейнере происходит событие ContainerEvent. | Перейдем к рассмотрению конкретных компонентов. Самый простой компонент описывает класс Label.
- ## События. Компонент Button.
  В классе Label происходят события классов Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent. | Немногим сложнее класс Button. | Компонент Button | Компонент Button – это кнопка стандартного для данной графической системы вида с надписью, умеющая реагировать на щелчок кнопки мыши – при нажатии она "вдавливается" в плоскость контейнера, при отпускании – становится "выпуклой".
- ## События. Компонент Checkbox.
  Кроме событий класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent. При воздействии на кнопку происходит Событие ActionEvent. | Немного сложнее класса Label класс checkbox, создающий кнопки выбора.
- ## События. Класс CheckboxGroup.
  В классе Checkbox происходят события класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent, а при изменении состояния кнопки возникает событие ItemEvent. | В библиотеке AWT радиокнопки не образуют отдельный компонент.
- ## Как создать группу радиокнопок
  Чтобы организовать группу радиокнопок, надо сначала сформировать объект класса CheckboxGroup, а затем создавать кнопки конструкторами: | Checkbox(String label, CheckboxGroup group, boolean state) | Checkbox(String label, boolean state, CheckboxGroup group) | Эти конструкторы идентичны, просто при записи конструктора можно не думать о порядке следования его аргументов.
- ## Компонент Choice. События.
  Компонент choice – это раскрывающийся список, один, выбранный, пункт (item) которого виден в поле, а другие появляются при щелчке кнопкой мыши на небольшой кнопке справа от поля компонента. | Вначале конструктором Choice () создается пустой список.
- ## Компонент List
  Компонент List – это список с полосой прокрутки, в котором можно выделить один или несколько пунктов. Количество видимых на экране пунктов определяется конструктором списка и размером компонента. | В классе три конструктора: | List() – создает пустой список с четырьмя видимыми пунктами;
- ## События
  Кроме событий класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent, при двойном щелчке кнопкой мыши на выбранном пункте происходит событие ActionEvent. | В листинге 10.2 задаются компоненты, аналогичные компонентам листинга 10.1, с помощью классов choice и List, а рис.
- ## Компоненты для ввода текста. Класс TextComponent. События.
  В библиотеке AWT есть два компонента для ввода текста с клавиатуры: TextField, позволяющий ввести только одну строку, и TextArea, в который можно ввести множество строк. | Оба класса расширяют класс Textcomponent, в котором собраны их общие методы, такие как выделение текста, позиционирование курсора, получение текста.
- ## Компонент TextField. События.
  Компонент TextField – это поле для ввода одной строки текста. Ширина поля измеряется в колонках (column). Ширина колонки – это средняя ширина символа в шрифте, которым вводится текст. Нажатие клавиши Enter заканчивает ввод и служит сигналом к началу обработки введенного текста, т. е.
- ## Компонент TextArea
  Компонент TextArea – это область ввода с произвольным числом строк. Нажатие клавиши Enter просто переводит курсор в начало следующей строки. В области ввода могут быть установлены линейки прокрутки, одна или обе.
- ## События
  Кроме Событий класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent, при изменении текста пользователем происходит событие TextEvent. | В листинге 10.3 создаются три поля: tf1, tf2, tf3 для ввода имени

пользователя, его пароля и заказа, и не редактируемая область ввода, в которой накапливается заказ. В поле ввода пароля tf2 появляется эхо-символ *. Результат показан на рис. 10.3.

- ## Компонент Scrollbar

  Компонент Scrollbar – это полоса прокрутки, но в библиотеке AWT класс Scrollbar используется еще и для организации ползунка (slider). Объект может располагаться горизонтально или вертикально, обычно полосы прокрутки размещают внизу и справа.

- ## События

  Кроме событий класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent, при изменении значения пользователем происходит событие AdjustmentEvent. | В листинге 10.4 создаются три вертикальные полосы прокрутки – красная, зеленая и синяя, позволяющие выбрать какое-нибудь значение соответствующего цвета в диапазоне 0-255, с начальным значением 127.

- ## Контейнер Panel

  Контейнер Panel – это невидимый компонент графического интерфейса, служащий для объединения нескольких других компонентов в один объект типа Panel. | Класс Panel очень прост, но важен. В нем всего два конструктора: | Panel () – создает контейнер с менеджером размещения по умолчанию FlowLayoutJ

- ## Контейнер ScrollPane

  Контейнер ScrollPane может содержать только один компонент, но зато такой, который не помещается целиком в окне. Контейнер обеспечивает средства прокрутки для просмотра большого компонента.

- ## Контейнер Window. События.

  Контейнер window – это пустое окно, без внутренних элементов: рамки, строки заголовка, строки меню, полос прокрутки. Это просто прямоугольная область на экране. Окно типа window самостоятельно, не содержится ни в каком контейнере, его не надо заносить в контейнер методом add().

- ## Контейнер Frame

  Контейнер Frame – это полноценное готовое окно со строкой заголовка, в которую помещены кнопки контекстного меню, сворачивания окна в ярлык и разворачивания во весь экран и кнопка закрытия приложения. Заголовок окна записывается в конструкторе или методом setTitie(string title).

- ## События

  Кроме событий класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent, при изменении размеров окна, его перемещении или удалении с экрана, а также показа на экране происходит событие windowEvent.

- ## Контейнер Dialog

  Контейнер Dialog – это окно обычно фиксированного размера, предназначенное для ответа на сообщения приложения. Оно автоматически регистрируется в оконном менеджере графической оболочки, следовательно, его можно перемещать по экрану, менять его размеры.

- ## События

  Кроме Событий класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent, при изменении размеров окна, его перемещении или удалении с экрана, а также показа на экране происходит событие windowEvent. | В листинге 10.6. создается модальное окно доступа, в которое вводится имя и пароль.

- ## Контейнер FileDialog. События.

  Контейнер FileDialog – это модальное окно с владельцем типа Frame, содержащее стандартное окно выбора файла операционной системы для открытия (константа LOAD) или сохранения (константа SAVE). Окна операционной системы создаются и помещаются в объект класса FileDialog автоматически.

- ## Создание собственных компонентов. Компонент Canvas.

  Создать свой компонент, дополняющий свойства и методы уже существующих компонентов AWT, очень просто – надо лишь образовать свой класс как расширение существующего класса Button, TextField или другого класса-компонента.

- ## Создание "легкого" компонента

  "Легкий" компонент, не имеющий своего peer-объекта в графической системе, создается как прямое расширение класса component или Container. При этом необходимо задать те действия, которые в "тяжелых" компонентах выполняет peer-объект.

Класс Component

Графическая библиотека AWT предлагает более двадцати готовых компонентов. Они показаны на рис. 8.2. Наиболее часто используются подклассы класса Component: классы**Button**, **Canvas**, **Checkbox**, **Choice**, **Container**, **Label**, **List**, **Scrollbar**, **TextArea**, **TextField**,**Panel**, **ScrollPane**, **Window**, **Dialog**, **FileDialog**, **Frame**.

Еще одна группа компонентов – это компоненты меню – классы **Menuitem**, **MenuBar**,**Menu**, **PopupMenu**, **CheckboxMenuItem**. Мы рассмотрим их в главе 13.

Забегая вперед, для каждого компонента перечислим события, которые в нем происходят. Обработку событий мы разберем в главе 12.

Начнем изучать эти компоненты от простых компонентов к сложным и от наиболее часто используемых к применяемым реже. Но сначала посмотрим на то общее, что есть во всех этих компонентах, на сам класс component.

Класс **component** – центр библиотеки AWT – очень велик и обладает большими возможностями. В нем пять статических констант, определяющих размещение компонента внутри пространства, выделенного для компонента в содержащем его контейнере: BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT, и около сотни методов.

Большинство методов – это методы доступа **getxxx()**, **isxxx()**, **setxxx()**. Изучать их нет смысла, надо просто посмотреть, как они используются в подклассах.

Конструктор класса недоступен – он защищенный (**protected**), потому, что класс component абстрактный, он не может использоваться сам по себе, применяются только его подклассы.

Компонент всегда занимает прямоугольную область со сторонами, параллельными сторонам экрана и в каждый момент времени имеет определенные размеры, измеряемые в пикселах, которые можно узнать методом **getsize()**, возвращающим объект класса Dimension, или целочисленными методами **getHeight()** и **getwidth()**, возвращающими высоту и ширину прямоугольника. Новый размер компонента можно установить из программы методами **setSize(Dimension d)** или **setSize(int width, int height)**, если это допускает менеджер размещения контейнера, содержащего компонент.

У компонента есть предпочтительный размер, при котором компонент выглядит наиболее пропорционально. Его можно получить методом **getPreferredSize()** В виде объекта Dimension.

Компонент обладает минимальным и максимальным размерами. Их возвращают методы**getMinimumSize()** и **getMaximumSize ()** в виде объекта Dimension.

В компоненте есть система координат. Ее начало – точка с координатами (0, 0) – находится в левом верхнем углу компонента, ось Ox идет вправо, ось Oy – вниз, координатные точки расположены между пикселами.

В компоненте хранятся координаты его левого верхнего угла в системе координат объемлющего контейнера. Их можно узнать методами **getLocation ()**, а изменить – методами**setLocation()**, переместив компонент в контейнере, если это позволит менеджер размещения компонентов.

Можно выяснить сразу и положение, и размер прямоугольной области компонента методом **getBounds ()**, возвращающим объект класса Rectangle, и изменить разом и положение, и размер компонента методами **setBounds ()**, если это позволит сделать менеджер размещения.

Компонент может быть недоступен для действий пользователя, тогда он выделяется на экране обычно светло-серым цветом. Доступность компонента можно проверить логическим методом **isEnabied()**, а изменить– методом **setEnabled(boolean enable)**.

Для многих компонентов определяется графический контекст – объект класса Graphics, – который управляется методом paint (), описанным в предыдущей главе, и который можно получить методом **getGraphics ()**.

В контексте есть текущий цвет и цвет фона – объекты класса color. Цвет фона можно получить методом **getBackground()**, а изменить– методом **setBackground(Color color)**. Текущий цвет можно получить методом **getForeground()**, а изменить – методом**setForeground(Color color)**.

В контексте есть шрифт – объект класса Font, возвращаемый методом **getFont()** и изменяемый методом **setFont(Font font)**.

В компоненте определяется локаль – объект класса Locale. Его можно получить методом**getLocale()**, изменить – методом **setLocale(Locale locale)**.

В компоненте существует курсор, показывающий положение мыши, – объект класса Cursor. Его можно получить методом **getcursor ()**, изменяется форма курсора в "тяжелых" компонентах с помощью метода **setcursor(Cursor cursor)**. Остановимся на этом классе подробнее.

Класс Cursor

Основа класса – статические константы, определяющие форму курсора:

- CROSSHAIR_CURSOR – курсор в виде креста, появляется обычно при поиске позиции для размещения какого-то элемента;
- DEFAULT_CURSOR – обычная форма курсора – стрелка влево вверх;
- HAND_CURSOR – "указующий перст", появляется обычно при выборе какого-то элемента списка;
- MOVE_CURSOR – крест со стрелками, возникает обычно при перемещении элемента;
- TEXT_CURSOR – вертикальная черта, появляется в текстовых полях;
- WAIT_CURSOR – изображение часов, появляется при ожидании.

Следующие курсоры появляются обычно при приближении к краю или углу компонента:

- E_RESIZE_CURSOR – стрелка вправо с упором;
- N_RESIZE_CURSOR – стрелка вверх с упором;
- NE_RESIZE_CURSOR – стрелка вправо вверх, упирающаяся в угол;
- NW_RESIZE_CURSOR – стрелка влево вверх, упирающаяся в угол;
- S_RESIZE_CURSOR – стрелка вниз с упором;
- SE_RESIZE_CURSOR – стрелка впрзво вниз, упирающаяся в угол;
- SW_RESIZE_CURSOR – стрелка влево вниз, упирающаяся в угол;
- W_RESIZE_CURSOR – стрелка влево с упором.

Перечисленные константы являются аргументом type в конструкторе класса Cursor(int type).

Вместо конструктора можно обратиться к статическому методу **getPredefinedCursor(int type)**, создающему объект класса Cursor и возвращающему ссылку на него.

Получить курсор по умолчанию можно статическим методом **getDefauitcursor ()**. Затем созданный курсор надо установить в компонент. Например, после выполнения:

```
Cursor curs = new Cursor(Cursor.WAIT_CURSOR);
omeComp.setCursor(curs);
```

…при появлении указателя мыши в компоненте somecomp указатель примет вид часов.

Как создать свой курсор

Кроме этих предопределенных курсоров можно задать свою собственную форму курсора. Ее тип носит название CUSTOM_CURSOR. Сформировать свой курсор можно методом:

```
createCustomCursor(Image cursor, Point hotspot, String name)
```

…создающим объект класса cursor и возвращающим ссылку на него. Перед этим следует создать изображение курсора cursor – объект класса image. Как это сделать, рассказывается в главе 15. Аргумент name задает имя курсора, можно написать просто null. Аргумент hotspot задает точку фокуса курсора. Эта точка должна быть в пределах изображения курсора, точнее, в пределах, показываемых методом:

```
getBestCursorSize(int desiredWidth, int desiredHeight)
```

…возвращающим ссылку на объект класса Dimension. Аргументы метода означают желаемый размер курсора. Если графическая система не допускает создание курсоров, возвращается (0, 0). Этот метод показывает приблизительно размер того курсора, который создаст графическая система, например, (32, 32). Изображение cursor будет подогнано под этот размер, при этом возможны искажения.

Третий метод – **getMaximumCursorColors()** – возвращает наибольшее количество цветов, например, 256, которое можно использовать в изображении курсора.

Это методы класса java.awt.Toolkit, с которым мы еще не работали. Класс Toolkit содержит некоторые методы, связывающие приложение Java со средствами платформы, на которой выполняется приложение. Поэтому нельзя создать экземпляр класса Toolkit конструктором, для его получения следует выполнить статический метод **Toolkit.getDefaultTooikit()**.

Если приложение работает в окне window или его расширениях, например, Frame, то можно получить экземпляр Toolkit методом **getToolkit()** класса Window.

Соберем все это вместе:

```
Toolkit tk = Toolkit.getDefaultTooikit();
int colorMax = tk.getMaximumCursorColors(); // Наибольшее число цветов
Dimension d = tk.getBestCursorSize(50, 50); // d – размер изображения
int w = d.width, h = d.height, k = 0;
Point p = new Point(0, 0); // Фокус курсора будет
// в его верхнем левом углу
int[] pix = new int[w * h]; // Здесь будут пикселы изображения
for(int i = 0; i < w; i++)
for(int j = 0; j < h; j++)
if (j < i) pix[k++] = 0xFFFF0000; // Левый нижний угол – красный
else pix[k++] = 0; // Правый верхний угол – прозрачный
// Создается прямоугольное изображение размером (w, h),
// заполненное массивом пикселов pix, с длиной строки w
Image im = createImage(new MemoryImageSource(w, h, pix, 0, w));
Cursor curs = tk.createCustomCursor(im, p, null);
someComp.setCursor(curs);
```

В этом примере создается курсор в виде красного прямоугольного треугольника с катетами размером 32 пиксела и устанавливается в каком-то компоненте someComp.

События. Класс Container.

Событие **ComponentEvent** происходит при перемещении компонента, изменении его размера, удалении с экрана и появлении на экране.

Событие **FocusEvent** возникает при получении или потере фокуса.

Событие **KeyEvent** проявляется при каждом нажатии и отпускании клавиши, если компонент имеет фокус ввода.

Событие **MouseEvent** происходит при манипуляциях мыши на компоненте.

Каждый компонент перед выводом на экран помещается в контейнер – подкласс класса container. Познакомимся с этим классом.

Класс Container

Класс **container** – прямой подкласс класса component, и наследует все его методы. Кроме них основу класса составляют методы добавления компонентов в контейнер:

- **add (Component comp)** – компонент comp добавляется в конец контейнера;
- **add (Component comp, int index)** – компонент comp добавляется в позицию index в контейнере, если index == -1, то компонент добавляется в конец контейнера;
- **add (Component comp, object constraints)** – менеджеру размещения контейнера даются указания объектом constraints;
- **add (String name. Component comp)** – компонент получает имя name.

Два метода удаляют компоненты из контейнера:

- **remove (Component comp)** – удаляет компонент с именем comp;
- **remove (int index)** – удаляет компонент с индексом index в контейнере.

Один из компонентов в контейнере получает **фокус ввода** (input focus), на него надравляется ввод с клавиатуры. Фокус можно переносить с одного компонента на другой клавишами **Tab** и **SHIFT + Tab**. Компонент может запросить фокус методом requestFocus () и передать фокус следующему компоненту методом transferFocus(). Компонент может проверить, имеет ли он фокус, своим логическим методом hasFocus(). Это методы класса Component.

Для облегчения размещения компонентов в контейнере определяется **менеджер размещения** (layout manager) – объект, реализующий интерфейс LayoutManager или его подынтерфейс LayoutManager2. Каждый менеджер размещает компоненты в каком-то своем порядке: один менеджер расставляет компоненты в таблицу, другой норовит растащить компоненты по сторонам, третий просто располагает их один за другим, как слова в тексте. Менеджер определяет смысл слов "добавить в конец контейнера" и "добавить в позицию index".

В контейнере в любой момент времени может быть установлен только один менеджер размещения. В каждом контейнере есть свой менеджер по умолчанию, установка другого менеджера производится методом:

```
setLayout(LayoutManager manager)
```

Менеджеры размещения мы рассмотрим подробно в следующей главе. В данной главе мы будем размещать компоненты вручную, отключив менеджер по умолчанию методом**setLayout (null)**.

События. Компонент Label.

Кроме событий Класса Component: **ComponentEvent**, **FocusEvent**, **KeyEvent**,**MouseEvent** при добавлении и удалении компонентов в контейнере происходит событие**ContainerEvent**.

Перейдем к рассмотрению конкретных компонентов. Самый простой компонент описывает класс Label.

Компонент Label

Компонент Label – это просто строка текста, оформленная как графический компонент для размещения в контейнере. Текст можно поменять только методом доступа **setText(string text)**, но не вводом пользователя с клавиатуры или с помощью мыши.

Создается объект этого класса одним из трех конструкторов:

- **Label ()** – пустой объект без текста;
- **Label (string text)** – объект с текстом text, который прижимается клевому краю компонента;
- **Label (String text, int alignment)** – объект с текстом text и определенным размещением в компоненте текста, задаваемого одной из трех констант: CENTER, LEFT, RIGHT.

Размещение можно изменить методом доступа **setAlignment(int alignment)**.

Остальные методы, кроме методов, унаследованных от класса component, позволяют получить текст **getText ()** и размещение **getAlignment ()**.

События. Компонент Button.

В классе **Label** происходят события классов Component: **ComponentEvent**, **FocusEvent**,**KeyEvent**, **MouseEvent**.

Немногим сложнее класс **Button**.

Компонент Button

Компонент **Button** – это кнопка стандартного для данной графической системы вида с надписью, умеющая реагировать на щелчок кнопки мыши – при нажатии она "вдавливается" в плоскость контейнера, при отпускании – становится "выпуклой".

Два конструктора **Button ()** и **Button (string label)** создают кнопку без надписи и с надписью label соответственно.

Методы доступа **getLabel()** и **setLabel (String label)** позволяют получить и изменить надпись на кнопке.

Главная функция кнопки – реагировать на щелчки мыши, и прочие методы класса обрабатывают эти действия. Мы рассмотрим их в главе 12.

События. Компонент Checkbox.

Кроме событий класса Component: **ComponentEvent**, **FocusEvent**, **KeyEvent**,**MouseEvent**. При воздействии на кнопку происходит Событие **ActionEvent**.

Немного сложнее класса Label класс checkbox, создающий кнопки выбора.

Компонент Checkbox

Компонент **checkbox** – это надпись справа от небольшого квадратика, в котором в некоторых графических системах появляется галочка после щелчка кнопкой мыши – компонент переходит в состояние (state) on. После следующего щелчка галочка пропадает – это состояние off. В других графических системах состояние on отмечается "вдавливанием" квадратика. В компоненте checkbox состояния on/off отмечаются логическими значениями true/false соответственно.

Три конструктора **Checkbox ()**, **Checkbox (String label)**, **Checkbox (String label, boolean state)** создают компонент без надписи, с надписью label в состоянии off, и в заданном состоянии state.

Методы доступа **getLabel()**, **setLabel (String label)**, **getState()**, **setstate (boolean state)**возвращают и изменяют эти параметры компонента.

Компоненты checkbox удобны для быстрого и наглядного выбора из списка, целиком расположенного на экране, как показано на рис. 10.1. Там же продемонстрирована ситуация, в которой нужно выбрать только один пункт из нескольких. В таких ситуациях образуется группа так называемых **радиокнопок** (radio buttons). Они помечаются обычно кружком или ромбиком, а не квадратиком, выбор обозначается жирной точкой в кружке или "вдавливанием" ромбика.

События. Класс CheckboxGroup.

В классе Checkbox происходят события класса Component: **ComponentEvent**,**FocusEvent**, **KeyEvent**, **MouseEvent**, а при изменении состояния кнопки возникает событие ItemEvent.

В библиотеке AWT радиокнопки не образуют отдельный компонент. Вместо этого несколько компонентов checkbox объединяются в группу с помощью объекта класса checkboxGroup.

Класс CheckboxGroup

Класс **CheckboxGroup** очень мал, поскольку его задача – просто дать общее имя всем объектам checkbox, образующим одну группу. В него входит один конструктор по умолчанию**CheckboxGroup ()** и два метода доступа:

- **getSelectedCheckbox()**, возвращающий выбранный объект Checkbox;
- **setSelectedCheckbox (Checkbox box)**, задающий выбор.

Как создать группу радиокнопок

Чтобы организовать группу радиокнопок, надо сначала сформировать объект класса**CheckboxGroup**, а затем создавать кнопки конструкторами:

```
Checkbox(String label, CheckboxGroup group, boolean state)
Checkbox(String label, boolean state, CheckboxGroup group)
```

Эти конструкторы идентичны, просто при записи конструктора можно не думать о порядке следования его аргументов.

Только одна радиокнопка в группе может иметь состояние state = true.

Пора привести пример. В листинге 10.1 приведена программа, помещающая в контейнер Frame две метки Label сверху, под ними слева три объекта checkbox, справа – группу радиокнопок. Внизу – три кнопки **Button**. Результат выполнения программы показан на рис. 10.1.

Листинг 10.1. Размещение компонентов.

```
import java.awt.*;
import j ava.awt.event.*;
class SimpleComp extends Frame{
SimpleComp(String s){ super(s);
setLayout(null);
Font f = new Font("Serif", Font.BOLD, 15);
setFont(f);
Label 11 = new Label("Выберите товар:", Labe1.CENTER);
l1.setBoundsdO, 50, 120, 30); add(11);
Label 12 = new Label("Выберите способ оплаты:");
l2.setBounds(160, 50, 200, 30); add(12);
Checkbox chl = new Checkbox("Книги");
chl.setBounds(20, 90, 100, 30); add(chl);
Checkbox ch2 = new Checkbox("Диски");
ch2.setBounds(20, 120, 100, 30); add(ch2);
Checkbox ch3 = new Checkbox("Игрушки");
ch3.setBounds(20, 150, 100, 30); add(ch3);
CheckboxGroup grp = new CheckboxGroup();
Checkbox chgl = new Checkbox("Почтовым переводом", grp,-true);
chgl.setBounds{170, 90, 200, 30); add(chgl);
Checkbox chg2 = new Checkbox{"Кредитной картой", grp, false);
chg2.setBounds(170, 120, 200, 30); add(chg2);
Button b1 = new Button("Продолжить");
bl.setBounds(30, 220, 100, 30); add(bl));
Button b2 = new Button("Отменить");
b2.setBounds(140, 220, 100, 30); add(b2);
Button b3 = new Button("Выйти");
b3.setBounds(250, 220, 100, 30); add(b3);
setSize(400, 300);
setVisible(true);
}
public static void main(String[] args)(
Frame f = new SimpleComp (" Простые компоненты");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
```

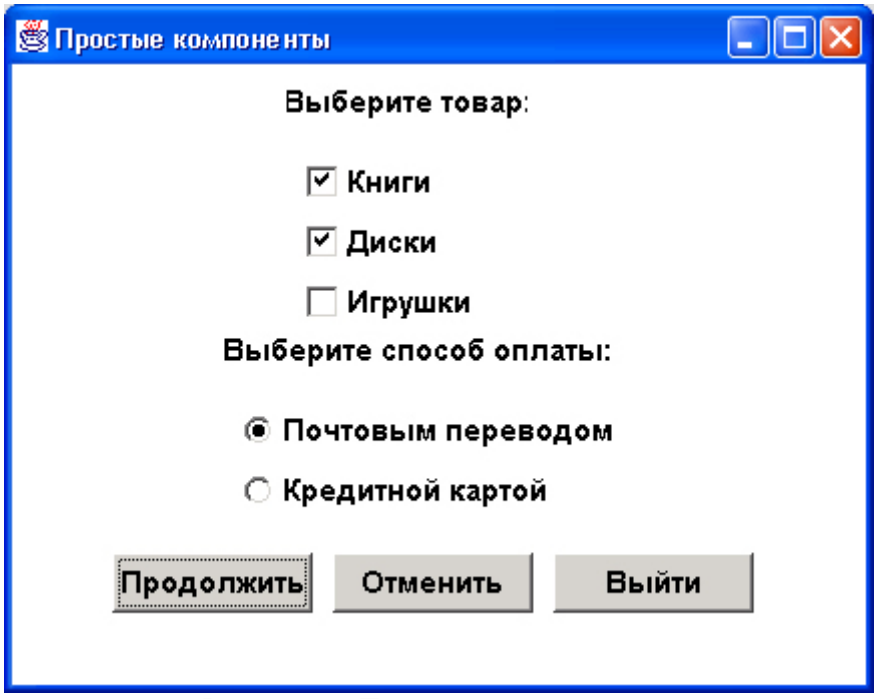

Рис. 10.1. Простые компоненты

Заметьте, что каждый создаваемый компонент следует заносить в контейнер, в данном случае Frame, методом **add()**. Левый верхний угол компонента помещается в точку контейнера с координатами, указанными первыми двумя аргументами метода **setBounds()**. Размер компонента задается последними двумя параметрами этого метода.

Если нет необходимости отображать весь список на экране, то вместо группы радиокнопок можно создать раскрывающийся список – объект класса Choice.

Компонент Choice. События.

Компонент **choice** – это раскрывающийся список, один, выбранный, пункт (**item**) которого виден в поле, а другие появляются при щелчке кнопкой мыши на небольшой кнопке справа от поля компонента.

Вначале конструктором **Choice ()** создается пустой список.

Затем, методом **add (string text)**, в список добавляются новые пункты с текстом text. Они располагаются в порядке написания методов add() и нумеруются от нуля.

Вставить новый пункт в нужное место можно методом **insert (string text, int position)**.

Выбор пункта можно произвести из программы методом **select (String text)** или **select(int position)**.

Удалить один пункт из списка можно методом **remove (String text)** или **remove (int position)**, а все пункты сразу – методом **removeAll()**.

Число пунктов в списке можно узнать методом **getitemCount ()**.

Выяснить, какой пункт находится в позиции pos можно методом **getitem(int pos)**, возвращающим строку.

Наконец, определение выбранного пункта производится методом **getselectedindex ()**, возвращающим позицию этого пункта, или методом **getseiecteditem()**, возвращающим выделенную строку.

События

В классе Choice происходят события класса Component: **ComponentEvent**, **FocusEvent**,**KeyEvent**, **MouseEvent**, а при выборе пункта возникает событие ItemEvent.

Если надо показать на экране несколько пунктов списка, то создайте объект класса List.

Компонент List

Компонент **List** – это список с полосой прокрутки, в котором можно выделить один или несколько пунктов. Количество видимых на экране пунктов определяется конструктором списка и размером компонента.

В классе три конструктора:

- **List()** – создает пустой список с четырьмя видимыми пунктами;
- **List (int rows)** – создает пустой список с rows видимыми пунктами;
- **List (int rows, boolean multiple)** – создает пустой список в котором можно отметить несколько пунктов, если multiple == true.

После создания объекта в список добавляются пункты с текстом item:

- метод **add (String item)** – добавляет новый пункт в конец списка;
- метод **add (String item, int position)** – добавляет новый пункт в позицию position.

Позиции нумеруются по порядку, начиная с нуля.

Удалить пункт можно методами **remove (String item)**, **remove (int position)**, **removeAll ()**.

Метод **replaceitem(string newitem, int pos)** позволяет заменить текст пункта в позиции pos.

Количество пунктов в списке возвращает метод **getitemcount ()**.

Выделенный пункт можно получить методом **getselecteditem()**, а его позицию – методом**getSelectedIndex ()**.

Если список позволяет осуществить множественный выбор, то выделенные пункты в виде массива типа string[] можно получить методом **getselecteditems ()**, позиции выделенных пунктов в виде массива типа int[] – методом **getSelectedIndexes ()**.

Кроме этих необходимых методов класс List содержит множество других, позволяющих манипулировать пунктами списка и получать его характеристики.

События

Кроме событий класса Component: **ComponentEvent**, **FocusEvent**, **KeyEvent**,**MouseEvent**, при двойном щелчке кнопкой мыши на выбранном пункте происходит событие**ActionEvent**.

В листинге 10.2 задаются компоненты, аналогичные компонентам листинга 10.1, с помощью классов **choice** и **List**, а рис. 10.2 показывает, как изменится при этом интерфейс.

Листинг 10.2. Использование списков.

```
import j ava.awt.*;
import j ava.awt.event.*;
class ListTest extends Frame{
ListTest(String s){ super(s);
setLayout(null);
setFont(new Font("Serif", Font.BOLD, 15));
Label 11 = new Label("Выберите товар:", Label.CENTER);
l1.setBoundsdO, 50, 120, 30); add (11);
Label 12 = new Label("Выберите способ оплаты:");
12.setBounds(170, 50, 200, 30); add(12);
List 1 = new List(2, true);
l.add("Книги");
l.add("Диски");
l.add("Игрушки");
l.setBounds(20, 90, 100, 40); add(l);
Choice ch = new Choice();
ch.add("Почтовым переводом");
ch.add("Кредитной картой");
ch.setBounds(17O, 90, 200.30); add(ch);
Button bl = new Button("Продолжить");
bl.setBounds(30, 150, 100, 30); add(bl);
```

```
Button b2 = new Button("Отменить");
b2.setBounds(140, 150, 100, 30); add(b2);
Button b3 = new Button("Выйти");
b3.setBounds(250, 150, 100, 30); add(b3);
setSize(400, 200); setVisible(true);
}
public static void main(StringH args){
Frame f = new ListTest(" Простые компоненты");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
```

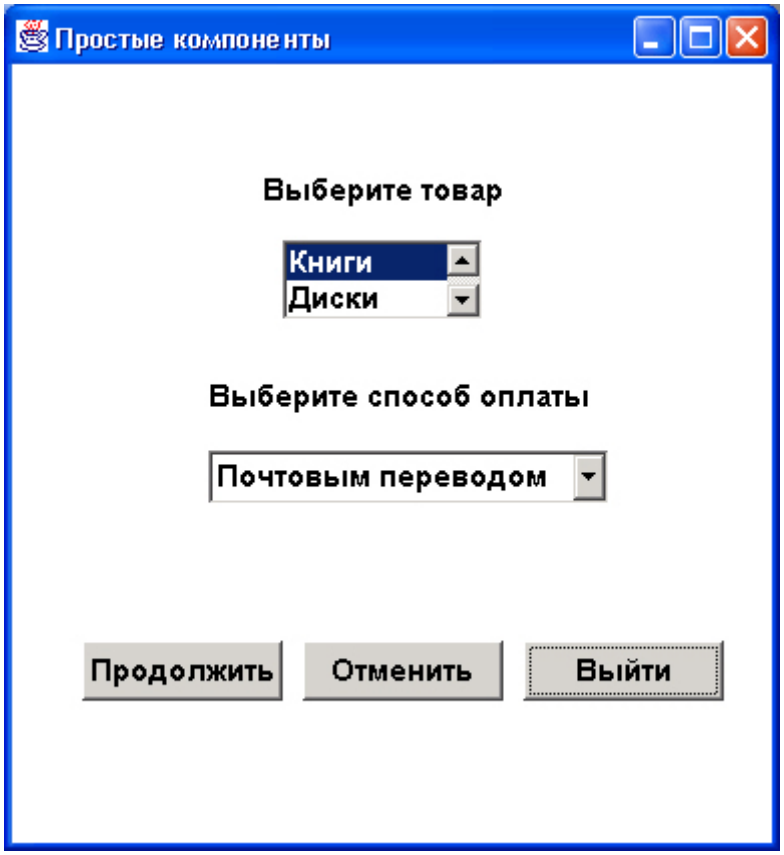

Рис. 10.2. Использование списков

Компоненты для ввода текста. Класс TextComponent. События.

В библиотеке AWT есть два компонента для ввода текста с клавиатуры: **TextField**, позволяющий ввести только одну строку, и **TextArea**, в который можно ввести множество строк.

Оба класса расширяют класс Textcomponent, в котором собраны их общие методы, такие как выделение текста, позиционирование курсора, получение текста.

Класс TextComponent

В классе **TextComponent** нет конструктора, этот класс не используется самостоятельно.

Основной метод класса – метод **getText ()** – возвращает текст, находящийся в поле ввода, в виде строки string.

Поле ввода может быть нередактируемым, в этом состоянии текст в поле нельзя изменить с клавиатуры или мышью. Узнать состояние поля можно логическим методом**isEditable()**, изменить значения в нем – методом **setEditable(boolean editable)**.

Текст, находящийся в поле, хранится как объект класса string, поэтому у каждого символа есть индекс (у первого – индекс 0). Индекс используется для определения позиции

курсора (**caret**) методом **getCaretPosition()**, для установки позиции курсора методом**setcaretposition(int ind)** и для выделения текста.

Текст выделяется, как обычно, мышью или клавишами со стрелками при нажатой клавише **SHIFT**, но можно выделить его из программы методом **select tin(begin, int end)**. При этом помечается текст от символа с индексом begin включительно, до символа с индексом end исключительно.

Весь текст выделяет метод **selectAll()**. Можно отметить начало выделения методом**setselectionstart (int ind)** и конец выделения методом **setSelectionEnd(int ind)**.

Важнее все-таки не задать, а получить выделенный текст. Его возвращает метод**getSeiectedText ()**, а начальный и конечный индекс выделения возвращают методы**getSelectionStart()** и **getSelectionEnd()**.

События

Кроме событий класса Component: **ComponentEvent**, **FocusEvent**, **KeyEvent**,**MouseEvent**, при изменении текста пользователем происходит событие **TextEvent**.

Компонент TextField. События.

Компонент **TextField** – это поле для ввода одной строки текста. Ширина поля измеряется в колонках (**column**). Ширина колонки – это средняя ширина символа в шрифте, которым вводится текст. Нажатие клавиши **Enter** заканчивает ввод и служит сигналом к началу обработки введенного текста, т. е. при этом происходит событие ActionEvent.

В классе четыре конструктора:

- **TextField ()** – создает пустое поле шириной в одну колонку;
- **TextField (int columns)** – создает пустое поле с числом колонок columns;
- **TextField (string text)** – создает поле с текстом text;
- **TextField(String text, int columns)** – создает поле с текстом text и числом колонок columns.

К методам, унаследованным от класса TextComponent, добавляются еще методы**getColumns()** и **setColumns(int col)**.

Интересная разновидность поля ввода – поле для ввода пароля. В таком поле вместо вводимых символов появляется какой-нибудь особый эхо-символ, чаще всего звездочка, чтобы пароль никто не подсмотрел через плечо.

Данное поле ввода получается выполнением метода **setEcnoCnar(char echo)**. Аргумент echo – это символ, который будет появляться в поле. Проверить, установлен ли эхо-символ, можно логическим методом **echoCharisSet()**, получить эхо-символ – методом **getEchoChar ()**.

Чтобы вернуть поле ввода в обычное состояние, достаточно выполнить метод**setEchoChar(0)**.

События

Кроме событий класса Component: **ComponentEvent**, **FocusEvent**, **KeyEvent**,**MouseEvent**, при изменении текста пользователем происходит событие **TextEvent**, а при нажатии на клавишу **Enter** – событие **ActionEvent**.

Компонент TextArea

Компонент **TextArea** – это область ввода с произвольным числом строк. Нажатие клавиши **Enter** просто переводит курсор в начало следующей строки. В области ввода могут быть установлены линейки прокрутки, одна или обе.

Основной конструктор класса:

```
TextArea(String text, int rows, int columns, int scrollbars)
```

…создает область ввода с текстом text, числом видимых строк rows, числом колонок columns, и заданием полос.прокрутки scrollbars одной из четырех констант: SCROLLBARS_NONE, SCROLLBARS_HORIZONTAL_ONLY, SCROLLBARS_VERTICAL_ONLY, SCROLLBARS_BOTH.

Остальные конструкторы задают некоторые параметры по умолчанию:

- **TextArea (String text, int rows, int columns)** – присутствуют обе полосы прокрутки;
- **TextArea (int rows, int columns)** – в поле пустая строка;
- **TextArea (string text)** – размеры устанавливает контейнер;
- **TextArea ()** – конструктор по умолчанию.

Среди методов класса TextArea наиболее важны методы:

- **append (string text)**, добавляющий текст text в конец уже введенного текста;
- **insert (string text, int pos)**, вставляющий текст в указанную позицию pos;
- **replaceRange (String text, int begin, int end)**, удаляющий текст начиная с индекса begin включительно по end исключительно, и помещающий вместо него текст text.

Другие методы позволяют изменить и получить количество видимых строк.

События

Кроме Событий класса Component: **ComponentEvent**, **FocusEvent**, **KeyEvent**,**MouseEvent**, при изменении текста пользователем происходит событие **TextEvent**.

В листинге 10.3 создаются три поля: tf1, tf2, tf3 для ввода имени пользователя, его пароля и заказа, и не редактируемая область ввода, в которой накапливается заказ. В поле ввода пароля tf2 появляется эхо-символ *****. Результат показан на рис. 10.3.

Листинг 10.3. Поля ввода.

```
import j ava.awt.*;
import j ava.awt.event.*;
class TextTest extends Frame{
TextTesttString s){
super(s);
setLayout(null);
setFont(new Font("Serif", Font.PLAIN, 14));
Label 11 = new Label("Ваше имя:", Label.RIGHT);
11.setBounds(20, 30, 70, 25); add(11);
Label 12 = new Label("Пароль:", Label.RIGHT);
12.setBounds(20, 60, 70, 25); add(12);
TextField tfl = new TextField(30);
tf1.setBounds(100, 30, 160, 25); add(tfl);
TextField tf2 = new TextField(30);
tf2.setBounds(100, 60, 160, 25);
add(tf2); tf2.setEchoChar('*');
TextField tf3 = new TextField("Введите сюда Ваш заказ", 30);
tf3.setBounds(10, 100, 250, 30); add(tf3);
TextArea ta = new TextArea("Ваш заказ:", 5, 50,
TextArea.SCROLLBARS_NONE);
ta.setEditable(false);
ta.setBounds(10, 150, 250, 140); add(ta);
Button bl = new Button("Применить");
Ы.setBounds(280, 180, 100, 30); add(bl);
```

```
Button b2 = new Button("Отменить");
b2.setBounds(280, 220, 100, 30); add(b2);
Button b3 = new Button("Выйти");
b3.setBounds(280, 260, 100, 30); add(b3);
setSize(400, 300); setVisible(true);
public static void main(String[] args){
Frame f = new TextTest(" Поля ввода");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
```

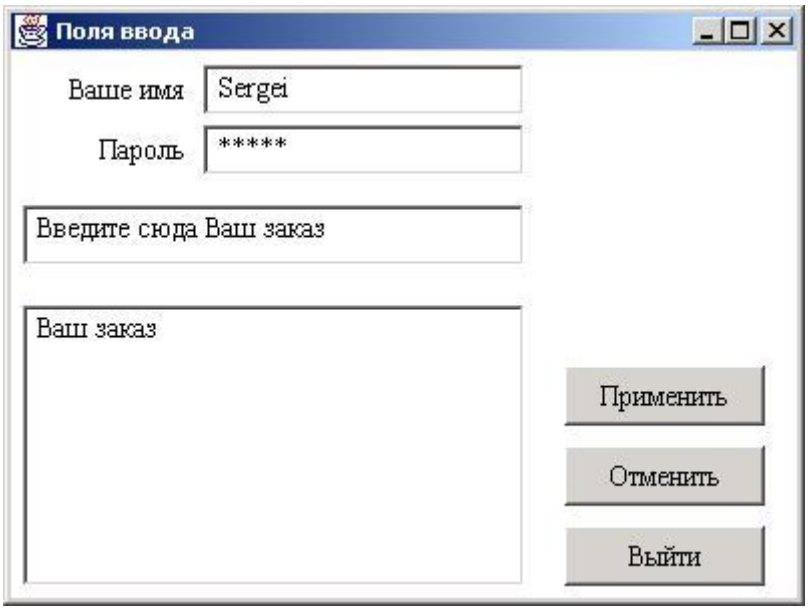

Рис. 10.3. Поля ввода

Компонент Scrollbar

Компонент **Scrollbar** – это полоса прокрутки, но в библиотеке AWT класс Scrollbar используется еще и для организации ползунка (**slider**). Объект может располагаться горизонтально или вертикально, обычно полосы прокрутки размещают внизу и справа.

Каждая полоса прокрутки охватывает некоторый диапазон значений и хранит текущее значение из этого диапазона. В линейке прокрутки есть пять элементов управления для перемещения по диапазону. Две стрелки на концах линейки вызывают перемещение на одну единицу (**unit**) в соответствующем направлении при щелчке на стрелке кнопкой мыши. Положение движка или бегунка (**bubble**, **thumb**) показывает текущее значение из диапазона и может его изменять при перемещении бегунка с помощью мыши. Два промежутка между движком и (Стрелками Позволяют переместиться на один блок (**block**) щелчком кнопки мыши.

Смысл понятий "единица" и "блок" зависит от объекта, с которым работает полоса прокрутки. Например, для вертикальной полосы прокрутки при просмотре текста это может быть строка и страница или строка и абзац.

Методы работы с данным компонентом описаны в интерфейсе Adjustable, который реализован классом scrollbar.

В классе **scrollbar** три конструктора:

- **Scrollbar ()** – создает вертикальную полосу прокрутки с диапазоном 0-100, текущим значением 0 и блоком 10 единиц;
- **Scrollbar (int orientation)** – ориентация orientation задается одной из двух констант HORIZONTAL или VERTICAL;

- **Scrollbar(int orientation, int value, int visible, int min, int max)** – задает, кроме ориентации, еще начальное значение value, размер блока visible, диапазон значений min-max.

Аргумент visible определяет еще и длину движка – она устанавливается пропорционально диапазону значений и длине полосы прокрутки. Например, конструктор по умолчанию задаст длину движка равной 0.1 длины полосы прокрутки.

Основной метод класса – **getvalue ()** – возвращает значение текущего положения движка на полосе прокрутки. Остальные методы доступа позволяют узнать и изменить характеристики объекта, примеры их использования показаны в листинге 12.6.

События

Кроме событий класса Component: **ComponentEvent**, **FocusEvent**, **KeyEvent**,**MouseEvent**, при изменении значения пользователем происходит событие**AdjustmentEvent**.

В листинге 10.4 создаются три вертикальные полосы прокрутки – красная, зеленая и синяя, позволяющие выбрать какое-нибудь значение соответствующего цвета в диапазоне 0-255, с начальным значением 127. Кроме них создается область, заполняемая получившимся цветом, и две кнопки. Линейки прокрутки, их заголовок и масштабные метки помещены в отдельный контейнер p типа Panel. Об этом чуть позже в данной главе.

Как все это выглядит, показано на рис. 10.4. В листинге 12.6 мы "оживим" эту программу.

Листинг 10.4. Линейки прокрутки для выбора цвета.

```
import java.awt.*;
import java.awt.event.*;
class ScrollTest extends Frame!
Scroiibar sbRed = new Scroiibar{Scroiibar.VERTICAL, 127, 10, 0, 255);
Scroiibar sbGreen = new Scroiibar(Scroiibar.VERTICAL, 127, 10, 0, 255);
Scroiibar sbBlue = new Scroiibar(Scroiibar.VERTICAL, 127, 10, 0, 255);
Color mixedColor =,new Color(127, 127, 127);
Label 1m = new Label();
Button Ы = new Button("Применить");
Button b2 = new Button("Отменить");
ScrollTest(String s){ super(s);
setLayout(null);
setFont(new Font("Serif", Font.BOLD, 15));
Panel p = new Panel();
p.setLayout(null);
p.setBounds(10.50, 150, 260); add(p);
Label Ic = new Label("Подберите цвет");
lc.setBounds(20, 0, 120, 30); p.add(lc);
Label Imin = new Label("0", Label.RIGHT);
lmin.setBounds(0, 30, 30, 30); p.add(lmin);
Label Imiddle = new Label("127", Label.RIGHT);
Imiddle.setBounds(0, 120, 30, 30); p.add(Imiddle);
Label Imax = new Label("255", Label.RIGHT);
Imax.setBounds(0, 200, 30, 30); p.add(Imax);
sbRed.setBackground(Color.red);
sbRed.setBounds(40, 30, 20, 200); p.add(sbRed);
sbGreen.setBackground(Color.green);
sbGreen.setBounds(70, 30, 20, 200); p.add(sbGreen);
sbBlue.setBackground(Color.blue);
sbBlue.setBounds(100, 30, 20, 200); p.add{sbBlue);
Label Ip = new Label("Образец:");
lp.setBounds(250, 50, 120, 30); add dp);
1m.setBackground(new Color(127, 127, 127));
lm.setBounds(220, 80, 120, 80); add(lm);
bl.setBounds(240, 200, 100, 30); add(bl);
b2.setBounds(240, 240, 100, 30); add(b2);
setSize(400, 300); setVisible(true);
```

```
}
public static void main(String[] args){
Frame f = new ScrollTestC' Выбор цвета");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
```

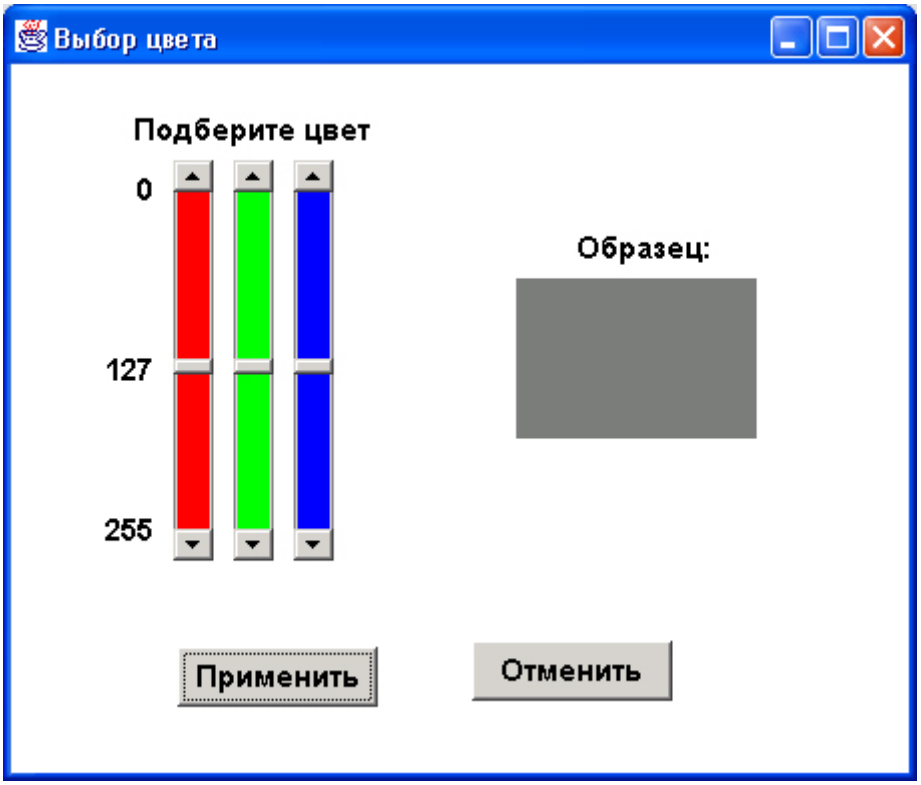

Рис. 10.4. Полосы прокрутки для выбора цвета

В листинге 10.4 использован контейнер **Panel**. Рассмотрим возможности этого класса.

Контейнер Panel

Контейнер **Panel** – это невидимый компонент графического интерфейса, служащий для объединения нескольких других компонентов в один объект типа Panel.

Класс Panel очень прост, но важен. В нем всего два конструктора:

- **Panel ()** – создает контейнер с менеджером размещения по умолчанию FlowLayoutJ
- **Panel (LayoutManager layout)** – создает контейнер с указанным менеджером размещения компонентов layout.

После создания контейнера в него добавляются компоненты унаследованным методом**add ()**:

```
Panel p = new Panel();
p.add(comp1);
p.add(comp2);
```

… и т. д. Размещает компоненты в контейнере его менеджер размещения, о чем мы поговорим в следующей главе.

Контейнер **Panel** используется очень часто. Он удобен для создания группы компонентов.

В листинге 10.4 три полосы прокрутки вместе с заголовком "Подберите цвет" и масштабными метками 0, 127 и 255 образуют естественную группу. Если мы захотим переместить ее в другое место окна, нам придется переносить каждый из семи компонентов, входящих в указанную группу. При этом придется следить за тем, чтобы их

взаимное положение не изменилось. Вместо этого мы создали панель p и разместили на ней все семь элементов. Метод **setBounds()** каждого из рассматриваемых компонентов указывает в данном случае положение и размер компонента в системе координат панели **p**, а не окна**Frame**. В окно мы поместили сразу целую панель, а не ее отдельные компоненты.

Теперь для перемещения всей группы компонентов достаточно переместить панель, и находящиеся на ней объекты автоматически переместятся вместе с ней, не изменив своего взаимного положения.

Контейнер ScrollPane

Контейнер **ScrollPane** может содержать только один компонент, но зато такой, который не помещается целиком в окне. Контейнер обеспечивает средства прокрутки для просмотра большого компонента. В контейнере можно установить полосы прокрутки либо постоянно, константой SCROLLBARS_ALWAYS, либо так, чтобы они появлялись только при необходимости (если компонент действительно не помещается в окно) константой SCROLLBARS_AS_NEEDED.

Если полосы прокрутки не установлены, это задает константа SCROLLBARS_NEVER, то перемещение компонента для просмотра нужно обеспечить из программы одним из методов**setScrollPosition ()**.

В классе два конструктора:

- **ScrollPane ()** – создает контейнер, в котором полосы прокрутки появляются по необходимости;
- **ScrollPane(int scroiibars)** – создает контейнер, в котором появление линеек прокрутки задается одной из трех указанных выше констант.

Конструкторы создают контейнер размером 100x100 пикселов, в дальнейшем можно изменить размер унаследованным методом – **setsize(int width, int height)**.

Ограничение, заключающееся в том, что scrollPane может содержать только один компонент, легко обходится. Всегда можно сделать этим единственным компонентом объект класса Panel, разместив на панели что угодно.

Среди методов класса интересны те, что позволяют прокручивать компонент в ScrollPane:

- методы **getHAdjustable()** и **getvAdjustabie ()** возвращают положение линеек прокрутки в виде интерфейса Adjustable;
- метод **getscrollPosition()** показывает координаты (x, y) точки компонента, находящейся в левом верхнем углу панели ScrollPane, в виде объекта класса Point;
- метод **setScrollPosition(Point p)** или **setScrollPosition(int x, int y)** прокручивает компонент в позицию (x, y).

Контейнер Window. События.

Контейнер **window** – это пустое окно, без внутренних элементов: рамки, строки заголовка, строки меню, полос прокрутки. Это просто прямоугольная область на экране. Окно типа window самостоятельно, не содержится ни в каком контейнере, его не надо заносить в контейнер методом **add()**. Однако оно не связано с оконным менеджером графической системы. Следовательно, нельзя изменить его размеры, переместить в другое место экрана. Поэтому оно может быть создано только каким-нибудь уже существующим окном,**владельцем** (owner) или **родителем** (parent) окна window. Когда окно-владелец убирается с экрана, вместе с ним убирается и порожденное окно. Владелец окна указывается В конструкторе:

- **window (Frame f)** – создает окно, владелец которого – фрейм f;
- **window (window owner)** – создает окно, владелец которого– уже имеющееся окно или подкласс класса window.

Созданное конструктором окно не выводится на экран автоматически. Его следует отобразить методом **show ()**. Убрать окно с экрана можно методом **hide ()**, а проверить, видно ли окно на экране – логическим методом **isShowing()**.

Окно типа window возможно использовать для создания всплывающих окон предупреждения, сообщения, подсказки. Для создания диалоговых окон есть подкласс Dialog, всплывающих меню – класс **popupMenu** (см. главу 13).

Видимое на экране окно выводится на передний план методом **toFront()** или, наоборот, помещается на задний план методом **toBack()**.

Уничтожить окно, освободив занимаемые им ресурсы, можно методом **dispose()**.

Менеджер размещения компонентов в окне по умолчанию – **BorderLayout**. Окно создает свой экземпляр класса Toolkit, который возможно получить методом **getToolkit()**.

События

Кроме событий класса Component: **ComponentEvent**, **FocusEvent**, **KeyEvent**,**MouseEvent**, при изменении размеров окна, его перемещении или удалении с экрана, а также показа на экране происходит событие windowEvent.

Контейнер Frame

Контейнер **Frame** – это полноценное готовое окно со строкой заголовка, в которую помещены кнопки контекстного меню, сворачивания окна в ярлык и разворачивания во весь экран и кнопка закрытия приложения. Заголовок окна записывается в конструкторе или методом **setTitie(string title)**. Окно окружено рамкой. В него можно установить строку меню методом **setMenuBar (MenuBar mb)**. Это мы обсудим В главе 13.

На кнопке контекстного меню в левой части строки заголовка изображена дымящаяся чашечка кофе – логотип Java. Вы можете установить там другое изображение методом**seticonimage(image icon)**, создав предварительно изображение icon в виде объекта класса image. Как это сделать, объясняется в главе 15.

Все элементы окна Frame вычерчиваются графической оболочкой операционной системы по правилам этой оболочки. Окно Frame автоматически регистрируется в оконном менеджере графической оболочки и может перемещаться, менять размеры, сворачиваться в панель задач (**task bar**) с помощью мыши или клавиатуры, как "родное" окно операционной системы.

Создать окно типа Frame можно следующими конструкторами:

- **Frame ()** – создает окно с пустой строкой заголовка;
- **Frame (string title)** – записывает аргумент title в строку заголовка.

Методы класса Frame осуществляют доступ к элементам окна, но не забывайте, что класс Frame наследует около двухсот методов классов Component, Container и window. В частности, наследуется менеджер размещения по умолчанию – **BorderLayout**.

События

Кроме событий класса Component: **ComponentEvent**, **FocusEvent**, **KeyEvent**,**MouseEvent**, при изменении размеров окна, его перемещении или удалении с экрана, а также показа на экране происходит событие **windowEvent**.

Программа листинга 10.5 создает два окна типа Frame, в которые помещаются строки – метки **Label**. При закрытии основного окна щелчком по соответствующей кнопке в строке заголовка или комбинацией клавиш **ALT + F4** выполнение программы завершается обращением к методу **system.exit (0)**, и закрываются оба окна. При закрытии второго окна происходит обращение к методу **dispose ()**, и закрывается только это окно.

Листинг 10.5. Создание двух окон.

```
import java.awt.*;
import java.awt.event.*;
class TwoFrames{
public static void main(String[] args){
Fr1 fl = new Frl(" Основное окно");
Fr2 f2 = new Fr2(" Второе окно");
}
}
class Frl extends Frame{
Fr1(String s){
super(s);
setLayout(null);
Font f = new Font("Serif", Font.BOLD, 15);
setFont(f);
Label 1 = new Label("Это главное окно", Labe1.CENTER);
l.setBounds(10, 30, 180, 30);
add(l);
setSize(200, 100);
setvisible(true);
addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit (0);
}
});
}
}
class Fr2 extends Frame{ Fr2(String s){
super(s);
setLayout(null);
Font f = new Font("Serif", Font.BOLD, 15);
setFont(f);
Label I <strong>=</strong> new Label("Это второе окно", Label.CENTER);
l.setBounds(10, 30, 180, 30);
add(l);
setBounds(50, 50, 200, 100);
setvisible(true);
addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev) {
dispose ();
}
});
}
}
```

На рис. 10.5 показан вывод этой программы. Взаимное положение окон определяется оконным менеджером операционной системы и может быть не таким, какое показано на рисунке.

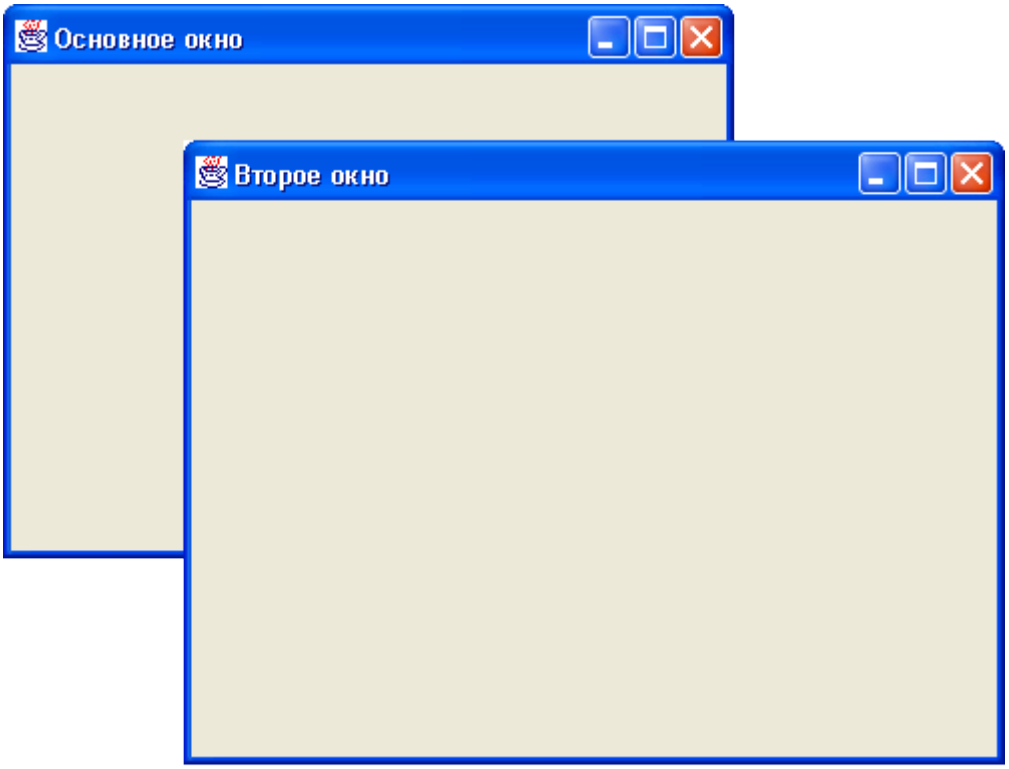

Рис. 10.5. Программа с двумя окнами

Контейнер Dialog

Контейнер **Dialog** – это окно обычно фиксированного размера, предназначенное для ответа на сообщения приложения. Оно автоматически регистрируется в оконном менеджере графической оболочки, следовательно, его можно перемещать по экрану, менять его размеры.

Но окно типа Dialog, как и его суперкласс – окно типа window, – обязательно имеет владельца owner, который указывается в конструкторе. Окно типа Dialog может быть**модальным** (modal), в котором надо обязательно выполнить все предписанные действия, иначе из окна нельзя будет выйти.

В классе семь конструкторов. Из них:

- **Dialog (Dialog owner)** – создает немодальное диалоговое окно с пустой строкой заголовка;
- **Dialog (Dialog owner, string title)** – создает немодальное диалоговое-окно со строкой заголовка title;
- **Dialog(Dialog owner, String title, boolean modal)** – создает диалоговое окно, которое будет модальным, если modal == true.

Четыре других конструктора аналогичны, но создают диалоговые окна, принадлежащие окну типа Frame:

- **Dialog(Frame owner)**
- **Dialog(Frame owner. String title)**
- **Dialog(Frame owner, boolean modal)**
- **Dialog(Frame owner, String title, Boolean modal)**

Среди методов класса интересны методы: **isModal ()**, проверяющий состояние модальности, и **setModal(boolean modal)**, меняющий это состояние.

События

Кроме Событий класса Component: **ComponentEvent**, **FocusEvent**, **KeyEvent**,**MouseEvent**, при изменении размеров окна, его перемещении или удалении с экрана, а также показа на экране происходит событие **windowEvent**.

В листинге 10.6. создается модальное окно доступа, в которое вводится имя и пароль. Пока не будет сделан правильный ввод, другие действия невозможны. На рис. 10.6 показан вид этого окна.

Листинг 10.6. Модальное окно доступа.

```
import java.awt.*;
import Java.awt.event.*;
class LoginWin extends Dialog{
LoginWin(Frame f, String s){
super(f, s, true);
setLayout(null);
setFont(new Font("Serif", Font.PLAIN, 14));
Label 11 = new Label("Ваше имя:", Label.RIGHT);
11.setBounds(20, 30, 70, 25); add(ll);
Label 12 = new Label("Пароль:", Label.RIGHT);
12.setBounds(20, 60, 70, 25); add(12);
TextField tfl = new TextField(30);
tfl.setBounds(100, 30, 160, 25); add(tfl);
TextField tf2 = new TextField(30);
tf2.setBounds(100, 60, 160, 25); add(tf2);
tf2.setEchoChar('*');
Button bl = new Button("Применить");
bl.setBounds(50, 100, 100, 30); add(bl);
Button b2 = new Button("Отменить");
b2.setBounds(160, 100, 100, 30); add(b2);
setBounds(50, 50, 300, 150); } }
class DialogTest extends Frame{ DialogTest(String s){ super(s);
setLayout(null); setSize(200, 100);
setvisible(true);
Dialog d = new LoginWin(this, " Окно входа"); d.setvisible(true);
}
public static void main(String[] args){
Frame f = new DialogTest(" Окно-владелец");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
```

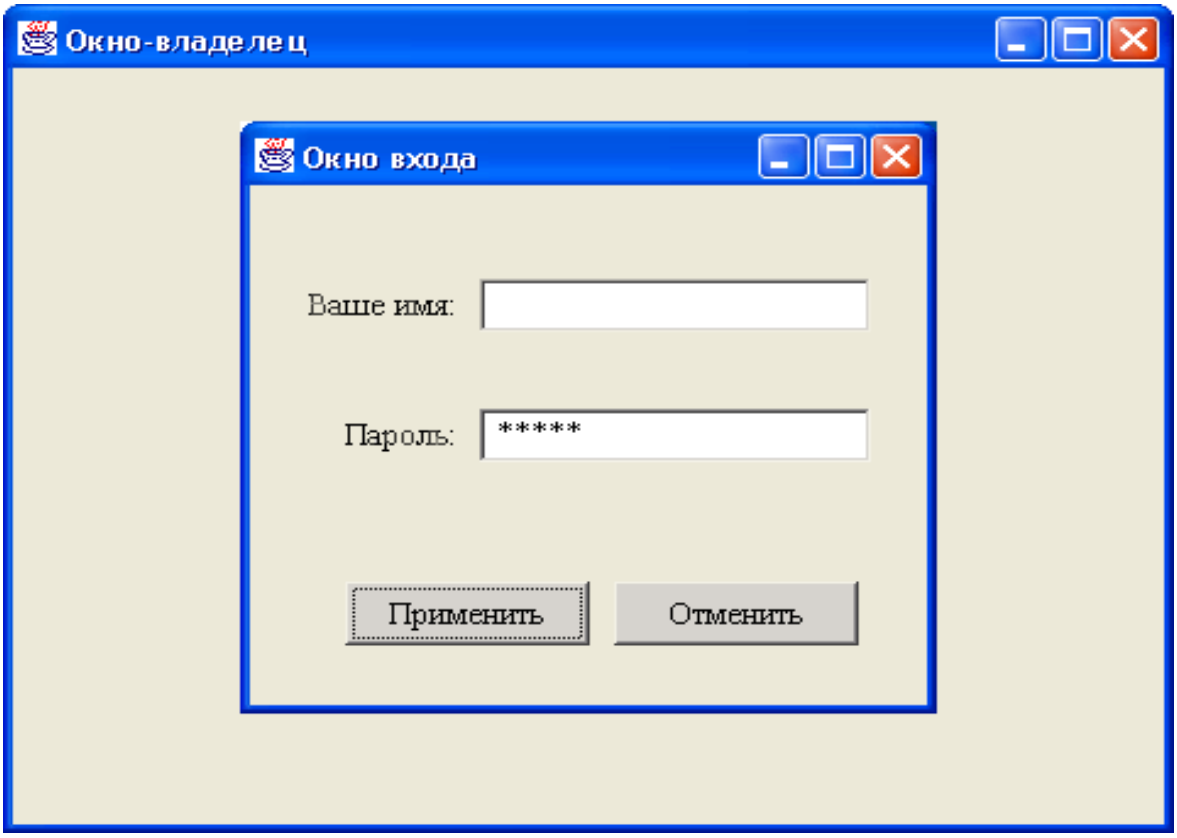

Рис. 10.6. Модальное окно доступа

Контейнер FileDialog. События.

Контейнер **FileDialog** – это модальное окно с владельцем типа Frame, содержащее стандартное окно выбора файла операционной системы для открытия (константа LOAD) или сохранения (константа SAVE). Окна операционной системы создаются и помещаются в объект класса FileDialog автоматически.

В классе три конструктора:

- **FileDialog (Frame owner)** – создает окно с пустым заголовком для открытия файлоа;
- **FileDialog (Frame owner, String title)** – создает окно открытия файла с заголовком title;
- **FileDialog(Frame owner, String title, int mode)** – создает окно открытия или сохранения документа; аргумент mode имеет два значения: **FileDialog.LOAD** и **FileDialog.SAVE**.

Методы класса **getoirectory ()** и **getFile()** возвращают только выбранный каталог и имя файла в виде строки string. Загрузку или сохранение файла затем нужно производить методами классов ввода/вывода, как рассказано в главе 18, там же приведены примеры использования класса FileDialog.

Можно установить начальный каталог для поиска файла и имя файла методами**setDirectory(String dir)** и **setFile(String fileName)**.

Вместо конкретного имени файла fileName можно написать шаблон, например, *.java (первые символы – звездочка и точка), тогда в окне будут видны только имена файлов, заканчивающиеся точкой и словом java.

Метод **setFilenameFilter(FilenameFilter filter)** устанавливает шаблон filter для имени выбираемого файла. В окне будут видны только имена файлов, подходящие под шаблон. Этот метод не реализован в SUN JDK на платформе MS Windows.

События

Кроме событий класса Component: **ComponentEvent**, **FocusEvent**, **KeyEvent**,**MouseEvent**, при изменении размеров окна, его перемещении или удалении с экрана, а также показа на экране происходит событие **windowEvent**.

Создание собственных компонентов. Компонент Canvas.

Создать свой компонент, дополняющий свойства и методы уже существующих компонентов AWT, очень просто – надо лишь образовать свой класс как расширение существующего класса Button, TextField или другого класса-компонента.

Если надо скомбинировать несколько компонентов в один, новый, компонент, то достаточно расширить класс Panel, расположив компоненты на панели.

Если же требуется создать совершенно новый компонент, то AWT предлагает две возможности: создать "тяжелый" или "легкий" компонент. Для создания собственных "тяжелых" компонентов в библиотеке AWT есть класс **canvas** – пустой компонент, для которого создается свой peer-объект графической системы.

Компонент Canvas

Компонент **canvas** – это пустой компонент. Класс canvas очень прост – в нем только конструктор по умолчанию **Canvas ()** и пустая реализация метода **paint(Graphics g)**.

Чтобы создать свой "тяжелый" компонент, необходимо расширить класс canvas, дополнив его нужными полями и методами, и при необходимости переопределить метод **paint ()**.

Например, как вы заметили, на стандартной кнопке Button можно написать только одну текстовую строку. Нельзя написать несколько строк или отобразить на кнопке рисунок. Создадим свой "тяжелый" компонент – кнопку с рисунком.

В листинге 10.7 кнопка с рисунком – класс **FlowerButton**. Рисунок задается методом**drawFiower ()**, а рисуется методом **paint ()**. Метод **paint ()**, кроме того, чертит по краям кнопки внизу и справа отрезки прямых, изображающих тень, отбрасываемую "выпуклой" кнопкой. При нажатии кнопки мыши на компоненте такие же отрезки чертятся вверху и слева – кнопка "вдавилась". При этом рисунок сдвигается на два пиксела вправо вниз – он "вдавливается" в плоскость окна.

Кроме этого, в классе **FlowerButton** задана реакция на нажатие и отпускание кнопки мыши. Это мы обсудим в главе 12, а пока скажем, что при каждом нажатии и отпускании кнопки меняется значение поля isDown и кнопка перечерчивается методом **repaint ()**. Это достигается выполнением методов **mousePressed()** и **mouseReleased()**.

Для сравнения рядом помещена стандартная кнопка типа Button того же размера. Рис. 10.7 демонстрирует вид этих кнопок.

Листинг 10.7. Кнопка с рисунком.

```
import java.awt.*;
import java.awt.event.*;
class FiowerButton extends Canvas implements MouseListener{
private boolean isDown=false;
public FiowerButton(){
super();
setBackground(Color.lightGray);
addMouseListener(this);
}
public void drawFlower(Graphics g, int x, int y, int w, int h){
g.drawOvalfx + 2*w/5-6, y, w/5, w/5);
g.drawLine(x + w/2-6, y + w/5, x + w/2-6, y + h -4);
g.drawOvalfx + 3*w/10-6, y + h/3-4, w/5, w/5);
g.drawOval(x + w/2-6, y + h/3-4, w/5, w/5); }
public void paint(Graphics g){
int w = getSizeO.width, h = getSize().height;
if (isDown){
g.drawLine(0, 0, w -1, 0);
g.drawLined, 1, w -1, 1);
g.drawLine(0, 0, 0, h -1);
g.drawUne (1, 1, 1, h -1);
drawFlower(g, 8, 10, w, h);
```

```
}
else
{
g.drawLine(0, h -2, w -2, h -2);
g.drawLined, h -1, w -1, h -1);
g.drawLinefw -2, h -2, w -2, 0);
g.drawLinefw -1, h -1, w -1, 1);
drawFlower (g, 6, 8, w, h); } }
public void mousePressed(MouseEvent e){
isDown=true; repaint(); }
public void mouseReleased(MouseEvent e){
isDown=false; repaint(); }
public void mouseEntered(MouseEvent e){}
public void mouseExited(MouseEvent e) {}
public void mouseClicked(MouseEvent e){()
}
class DrawButton extends Frame{
DrawButton(String s) {
super (s);
setLayout(null);
Button b = new Button("OK");
b.setBounds(200, 50, 100, 60); add(b);
FlowerButton d = new FlowerButton();
d.setBounds(50, 50, 100, 60); add(d);
setSize(400, 150);
setVisible(true);
}
public static void main(String[] args){
Frame f= new DrawButton(" Кнопка с рисунком");
f.addWindowListener(new WindowAdapter()(
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
```

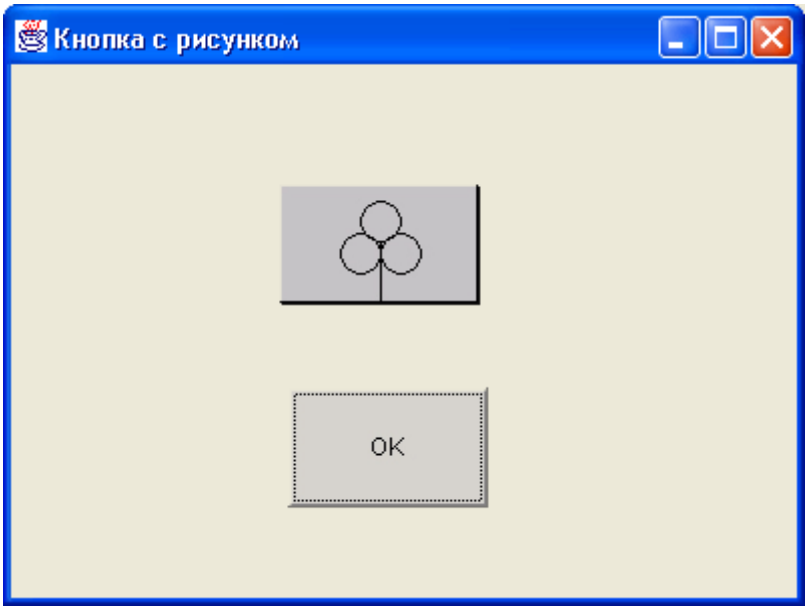

Рис. 10.7. Кнопка с рисунком

Создание "легкого" компонента

"Легкий" компонент, не имеющий своего peer-объекта в графической системе, создается как прямое расширение класса **component** или **Container**. При этом необходимо задать те действия, которые в "тяжелых" компонентах выполняет peer-объект.

Например, заменив в листинге 10.7 заголовок класса FlowerButton строкой:

```
class FlowerButton extends Component implements MouseListener{
```

…а затем перекомпилировав и выполнив программу, вы получите "легкую" кнопку, но увидите, что ее фон стал белым, потому что метод **setBackground(Color.lightGray)** не сработал.

Это объясняется тем, что теперь всю черную работу по изображению кнопки на экране выполняет не peer-двойник кнопки, а "тяжелый" контейнер, в котором расположена кнопка, в нашем случае класс Frame. Контейнер же ничего не знает о том, что надо обратиться к методу **setBackground ()**, он рисует только то, что записано в методе paint (). Придется убрать метод **setBackground ()** из конструктора и заливать фон серым цветом вручную в методе **paint ()**, как показано в листинге 10.8.

"Легкий" контейнер не умеет рисовать находящиеся в нем "легкие" компоненты, поэтому в конце метода **paint ()** "легкого" контейнера нужно обратиться к методу **paint ()** суперкласса:

```
super.paint(g);
```

Тогда рисованием займется "тяжелый" суперкласс-контейнер. Он нарисует и лежащий в нем "легкий" контейнер, и размещенные в контейнере "легкие" компоненты.

Совет

Завершайте метод paint () "легкого" контейнера обращением к методу paint () суперкласса.

Предпочтительный размер "тяжелого" компонента устанавливается peer-объектом, а для "легких" компонентов его надо задать явно, переопределив метод **getPreferredSize()**, иначе некоторые менеджеры размещения, например **FlowLayout ()**, установят нулевой размер, и компонент не будет виден на экране.

Совет

Переопределяйте метод getPref erredSize ().

Интересная особенность "легких" компонентов – они изначально рисуются прозрачными, не закрашенная часть прямоугольного объекта не будет видна. Это позволяет создать компонент любой видимой формы. Листинг 10.8 показывает, как можно изменить метод **paint ()** листинга 10.7 для создания круглой кнопки и задать дополнительные методы, а рис. 10.8 демонстрирует ее вид.

Листинг 10.8. Создание круглой кнопки.

```
public void paint(Graphics g){
int w = getSize().width, h = getSize().height;
int d = Math.min(w, h); // Диаметр круга
Color c = g.getColor(); // Сохраняем текущий цвет
g.setColor(Color.lightGray); // Устанавливаем серый цвет
g.fillArc(0, 0, d, d, 0, 360); // Заливаем круг серым цветом
g.setColor(c); // Восстанавливаем текущий цвет
if (isDown)(
g.drawArc(0, 0, d, d, 43, 180);
g.drawArcd, 1, d -2, d -2, 43, 180);
drawFlower(g, 8, 10, d, d);
}else{
g.drawArc(0, 0, d, -d, 229, 162);
g.drawArcd, 1, d -2, d -2, 225, 170);
drawFlower(g, 6, 8, d, d);
}
}
public Dimension getPreferredSize(){
return new Dimension(30.30);
}
public Dimension getMinimumSize()
{
return getPreferredSize(); }
```

```
public Dimension getMaximumSize(){
return getPreferredSize();
}
```

Рис. 10.8. Круглая кнопка

Сразу же надо дать еще одну рекомендацию. "Легкие" контейнеры не занимаются обработкой событий без специального указания. Поэтому в конструктор "легкого" компонента следует включить обращение к методу **enableEvents ()** для каждого типа событий. В нашем примере в конструктор класса **FlowerButton** полезно добавить строку:

```
enabieEvents(AWTEvent.MOUSE_EVENT_MASK);
```

…на случай, если кнопка окажется в "легком" контейнере. Подробнее об этом мы поговорим в главе 12.

В документации есть хорошие примеры создания "легких" компонентов, посмотрите страницу docs\guide\awt\demos\lightweight\index.html.

Размещение компонентов

- ## Размещение компонентов
  В предыдущей главе мы размешали компоненты "вручную", задавая их размеры и положение в контейнере абсолютными координатами в координатной системе контейнера. Для этого мы применяли метод setBounds(). | Такой способ размешает компоненты с точностью до пиксела, но не позволяет перемещать их.
- ## Менеджер FlowLayout
  Наиболее просто поступает менеджер размещения FlowLayout. Он укладывает в контейнер один компонент за другим слева направо как кирпичи, переходя от верхних рядов к нижним. При изменении размера контейнера "кирпичи" перестраиваются, как показано на рис. 11.1.
- ## Менеджер BorderLayout
  Менеджер размещения BorderLayout делит контейнер на пять неравных областей, полностью заполняя каждую область одним компонентом, как показано на рис. 11.2. Области получили географические названия NORTH, SOUTH, WEST, EAST И CENTER.
- ## Менеджер GridLayout
  Менеджер размещения GridLayout расставляет компоненты в таблицу с заданным в конструкторе числом строк rows и столбцов columns: | GridLayout(int rows, int columns) | Все компоненты получают одинаковый размер. Промежутков между компонентами нет.
- ## Менеджер CardLayout
  Менеджер размещения cardLayout своеобразен – он показывает в контейнере только один, первый (first), компонент. Остальные компоненты лежат под первым в определенном порядке как игральные карты в колоде. Их расположение определяется порядком, в котором написаны методы add ().
- ## Менеджер GridBagLayout
  Менеджер размещения GridBagLayout расставляет компоненты наиболее гибко, позволяя задавать размеры и положение каждого компонента. Но он оказался очень сложным и применяется редко. | В классе GridBagLayout есть только один конструктор по умолчанию, без аргументов.

Размещение компонентов

В предыдущей главе мы размешали компоненты "вручную", задавая их размеры и положение в контейнере абсолютными координатами в координатной системе контейнера. Для этого мы применяли метод **setBounds()**.

Такой способ размешает компоненты с точностью до пиксела, но не позволяет перемещать их. При изменении размеров окна с помощью мыши компоненты останутся на своих местах, привязанными к левому верхнему углу контейнера. Кроме того, нет гарантии, что все мониторы отобразят компоненты так, как вы задумали.

Чтобы учесть изменение размеров окна, надо задать размеры и положение компонента относительно размеров контейнера, например, так:

```
int w = getSize().width; // Получаем ширину
int h = getSize().height; //и высоту контейнера
Button b = new Button("OK"); // Создаем кнопку
b.setBounds(9*w/20, 4*h/5, w/10, h/10);
```

…и при всяком изменении размеров окна задавать расположение компонента заново.

Чтобы избавить программиста от этой кропотливой работы, в библиотеку AWT внесены два интерфейса: **LayoutManager** и порожденный от него интерфейс LayoutManager2, а также пять реализаций этих интерфейсов:
классы **BorlerLayout**, **CardLayout**, **FlowLayout**,**GridLayout** и **GridBagLayout**. Эти классы названы менеджерами размещения (**layout manager**) компонентов.

Каждый программист может создать свои менеджеры размещения, реализовав интерфейсы **LayoutManager** или LayoutManager2.

Посмотрим, как размещают компоненты эти классы.

Менеджер FlowLayout

Наиболее просто поступает менеджер размещения **FlowLayout**. Он укладывает в контейнер один компонент за другим слева направо как кирпичи, переходя от верхних рядов к нижним. При изменении размера контейнера "кирпичи" перестраиваются, как показано на рис. 11.1. Компоненты поступают в том порядке, в каком они заданы в методах **add ()**.

В каждом ряду компоненты могут прижиматься к левому краю, если в конструкторе аргумент align равен FlowLayout. LEFT, к правому краю, если этот аргумент FlowLayout. RIGHT, или собираться в середине ряда, если FlowLayout.CENTER.

Между компонентами можно оставить промежутки (**gap**) по горизонтали hgap и вертикали vgap. Это задается в конструкторе:

```
FlowLayout(int align, int hgap, int vgap)
```

Второй конструктор задает промежутки размером 5 пикселов:

```
FlowLayout(int align)
```

Третий конструктор определяет выравнивание по центру и промежутки 5 пикселов:

```
FlowLayout()
```

После формирования объекта эти параметры можно изменить методами:

```
setHgap(int hgap)
setVgap(int vgap)
setAlignment(int align)
```

В листинге 11.1 создаются кнопка Button, метка Label, кнопка выбора **checkbox**, раскрывающийся список choice, поле ввода **TextField** и все это размещается в контейнере Frame. Рис. 11.1 содержит вид этих компонентов при разных размерах контейнера.

Листинг 11.1. Менеджер размещения FlowLayout.

```
import j ava.awt.*;
import j ava.awt.event.*;
class FlowTest extends Frame{
FlowTest(String s) {
super(s);
setLayout (new FlowLayout (FlowLayout.LEFT, 10, 10));
add(new Button("Кнопка"));
add(new Label("Метка"));
add(new Checkbox("Выбор"));
add(new Choice());
add(new TextFieldt"Справка", 10));
setSize(300, 100); setVisible(true);
}
public static void main(String[] args){
Frame f= new FlowTest(" Менеджер FlowLayout");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
```

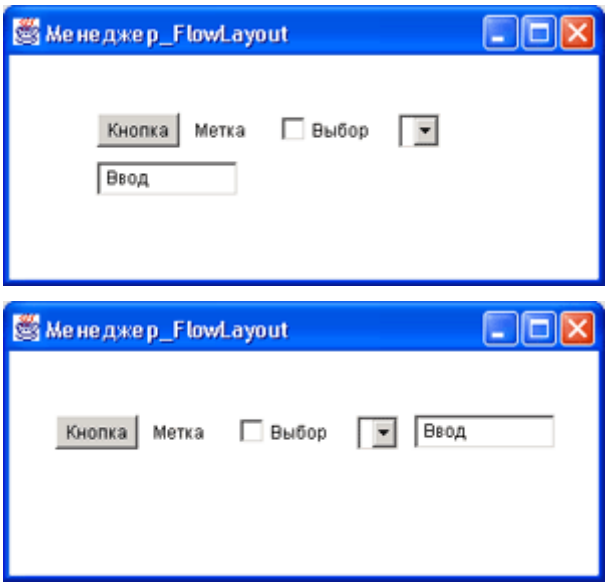

Рис. 11.1. Размещение компонентов с помощью FlowLayout

Менеджер BorderLayout

Менеджер размещения **BorderLayout** делит контейнер на пять неравных областей, полностью заполняя каждую область одним компонентом, как показано на рис. 11.2. Области получили географические названия NORTH, SOUTH, WEST, EAST И CENTER.

Метод **add ()** в случае применения BorderLayout имеет два аргумента: ссылку на компонент сотр и область region, в которую помещается компонент – одну из перечисленных выше констант:

```
add(Component comp, String region)
```

Обычный метод **add (Component comp)** с одним аргументом помещает компонент В область CENTER.

В классе два конструктора:

- **BorderLayout ()** – между областями нет промежутков;
- **BorderLayout(int hgap int vgap)** – между областями остаются горизонтальные hgap и вертикальные vgap промежутки, задаваемые в пикселах.

Если в контейнер помещается менее пяти компонентов, то некоторые области не используются и не занимают места в контейнере, как можно заметить на рис. 11.3. Если не занята область CENTER, то компоненты прижимаются к границам контейнера.

В листинге 11.2 создаются пять кнопок, размещаемых в контейнере. Заметьте отсутствие установки менеджера в контейнере **setLayout ()** – менеджер BorderLayout установлен в контейнере Frame по умолчанию. Результат размещения показан на рис. 11.2.

Листинг 11.2. Менеджер размещения BorderLayout.

```
import java.awt.*;
import java.awt.event.*;
class BorderTest extends Frame{
BorderTest(String s){ super(s);
add(new Button("North"), BorderLayout.NORTH);
add(new Button("South"), BorderLayout.SOUTH);
add(new Button("West"), BorderLayout.WEST);
add(new Button("East"), BorderLayout.EAST);
add(new Button("Center"));
setSize(300, 200);
setVisible(true);
}
public static void main(String[] args){
Frame f= new BorderTest(" Менеджер BorderLayout");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
```

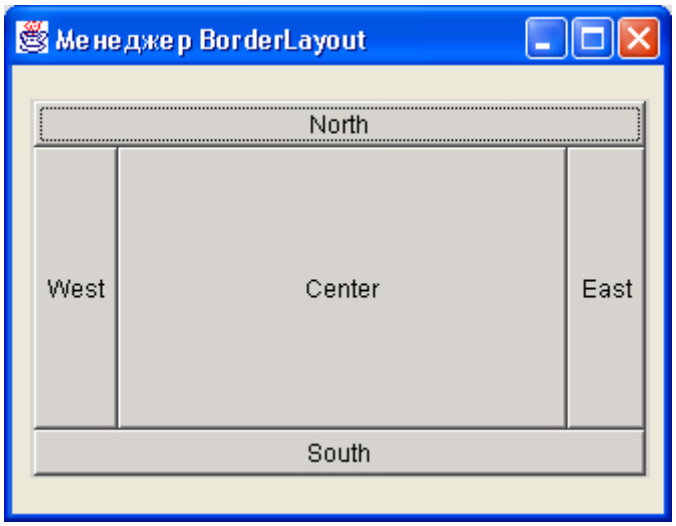

Рис. 11.2. Области размещения BorderLayout

Менеджер размещения **BorderLayout** кажется неудобным: он располагает не больше пяти компонентов, последние растекаются по всей области, области имеют странный вид. Но дело в том, что в каждую область можно поместить не компонент, а панель, и размещать компоненты на ней, как сделано в листинге 11.3 и показано на рис. 11.3. Напомним, что на панели Panel менеджер размещения по умолчанию **FlowLayout**.

Листинг 11.3. Сложная компоновка.

```
import java.awt.*;
import java.awt.event.*;
class BorderPanelTest extends Frame{
BorderPanelTest(String s){
super(s);
// Создаем панель p2 с тремя кнопками
Panel p2 = new Panel();
p2.add(new Button("Выполнить"));
p2.add(new Button("Отменить"));
p2.add(new Button("Выйти"));
```

```
Panel pi = new Panel ();
pi.setLayout(new BorderLayout());
// Помещаем панель p2 с кнопками на "юге" панели p1
p1.add(p2, BorderLayout.SOUTH);
// Поле ввода помещаем на "севере"
p1.add(new TextFieldt"Поле ввода", 20), BorderLayout.NORTH);
// Область ввода помещается в центре
p1.add(new TextArea("Область ввода", 5, 20, TextArea.SCROLLBARS_NONE),
BorderLayout.CENTER);
add(new Scrollbar(Scrollbar.HORIZONTAL), BorderLayout.SOUTH);
addfnew Scrollbar(Scrollbar.VERTICAL), BorderLayout.EAST);
// Панель p1 помещаем в "центре" контейнера add(p1, BorderLayout.CENTER);
setSize(300, 200);
setVisible(true);
}
public static void main(String[] args){
Frame f= new BorderPanelTest(" Сложная компоновка");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
```

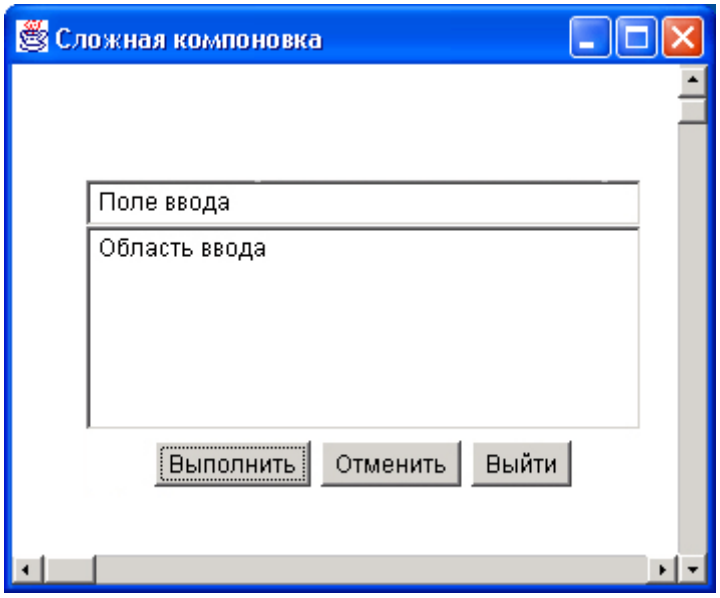

Рис. 11.3. Компоновка с помощью FiowLayout и BorderLayout

Менеджер GridLayout

Менеджер размещения **GridLayout** расставляет компоненты в таблицу с заданным в конструкторе числом строк **rows** и столбцов **columns**:

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

Все компоненты получают одинаковый размер. Промежутков между компонентами нет.

Второй конструктор позволяет задать промежутки между компонентами в пикселах по горизонтали **hgap** и вертикали **vgap**:

```
GridLayout(int rows, int columns, int hgap, int vgap)
```

Конструктор по умолчанию **GridLayout ()** задает таблицу размером 0x0 без промежутков между компонентами. Компоненты будут располагаться в одной строке.

Компоненты размещаются менеджером **GridLayout** слева направо по строкам созданной таблицы в том порядке, в котором они заданы в методах **add()**.

Нулевое количество строк или столбцов означает, что менеджер сам создаст нужное их число.

В листинге 11.4 выстраиваются кнопки для калькулятора, а рис. 11.4 показывает, как выглядит это размещение.

Листинг 11.4. Менеджер GridLayout.

```
import Java.awt.*;
import java.awt.event.*;
import java.util.*;
class GridTest extends Frame{
GridTest(String s){ super(s);
setLayout(new GridLayout(4, 4, 5, 5));
StringTokenizer st =
new StringTokenizer("7 89/456*123-0.=+");
while(st.hasMoreTokens())
add(new Button(st.nextToken()));
setSize(200, 200); setvisible(true);
}
public static void main(String[] args){
Frame f= new GridTestt" Менеджер GridLayout");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
```

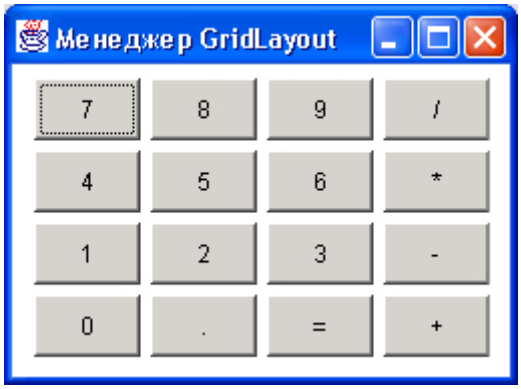

Рис. 11.4. Размещение кнопок менеджером GridLayout

Менеджер CardLayout

Менеджер размещения **cardLayout** своеобразен – он показывает в контейнере только один, первый (**first**), компонент. Остальные компоненты лежат под первым в определенном порядке как игральные карты в колоде. Их расположение определяется порядком, в котором написаны методы **add ()**. Следующий компонент можно показать методом **next (Container c)**, предыдущий – методом **previous (Container c)**, Последний– методом **last (Container c)**, первый – методом **first (Container c)**. Аргумент этих методов – ссылка на контейнер, в который помещены компоненты, обычно this.

В классе два конструктора:

- **CardLayout ()** – не отделяет компонент от границ контейнера;
- **CardLayout (int hgap, int vgap)** – задает горизонтальные hgap и вертикальные vgap поля.

Менеджер **CardLayout** позволяет организовать и произвольный доступ к компонентам. Метод add () для менеджера CardLayout имеет своеобразный вид:

```
add(Component comp, Object constraints)
```

Здесь аргумент **constraints** должен иметь тип string и содержать имя компонента. Нужный компонент с именем name можно показать методом:

```
show(Container parent, String name)
```

В листинге 11.5 менеджер размещения c1 работает с панелью p, помещенной в "центр" контейнера Frame. Панель p указывается как аргумент parent в методах **next ()** и **show ()**. На

"север" контейнера Frame отправлена панель p2 с меткой и раскрывающимся списком ch. Рис. 11.5 демонстрирует результат работы программы.

Листинг 11.5. Менеджер CardLayout.

```
import java.awt.*;
import java.awt.event.*;
class CardTest extends Frame{ CardTest(String s){
super(s);
Panel p = new Panel();
CardLayout cl = new CardLayout();
p.setLayout(cl);
p.add(new Button("Русская страница"),"page1");
p.add(new Button("English page"), "page2");
p.add(new Button("Deutsche Seite"), "page3");
add(p);
cl.next(p);
cl.show(p, "pagel");
Panel p2 = new Panel();
p2.add(new Label("Выберите язык:"));
Choice ch = new Choice();
ch.add("Русский");
ch.add("Английский");
ch.add("Немецкий");'
p2.add(ch);
add(p2, BorderLayout.NORTH);
setSize(400, 300);
setvisible(true); }
public static void main(String[] args){
Frame f= new CardTest{" Менеджер CardLayout");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
```

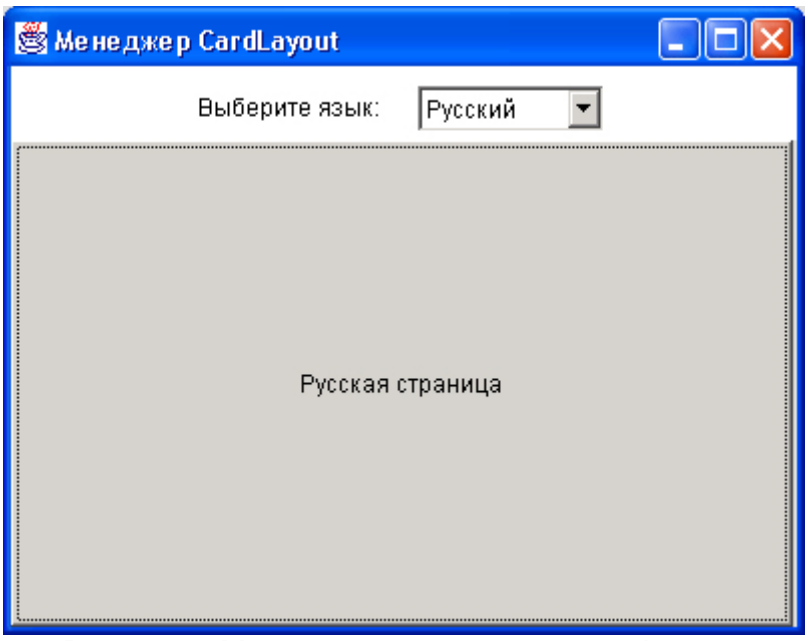

Рис. 11.5. Менеджер размещения CardLayout

Менеджер GridBagLayout

Менеджер размещения **GridBagLayout** расставляет компоненты наиболее гибко, позволяя задавать размеры и положение каждого компонента. Но он оказался очень сложным и применяется редко.

В классе **GridBagLayout** есть только один конструктор по умолчанию, без аргументов. Менеджер класса **GridBagLayout**, в отличие от других менеджеров размещения, не содержит правил размещения. Он играет только организующую роль. Ему передаются ссылка на компонент и правила расположения этого компонента, а сам он помещает данный компонент по указанным правилам в контейнер. Все правила размещения компонентов задаются в объекте другого класса, **GridBagConstraints**.

Менеджер размещает компоненты в таблице с неопределенным заранее числом строк и столбцов. Один компонент может занимать несколько ячеек этой таблицы, заполнять ячейку целиком, располагаться в ее центре, углу или прижиматься к краю ячейки.

Класс **GridBagConstraints** содержит одиннадцать полей, определяющих размеры компонентов, их положение в контейнере и взаимное положение, и несколько констант – значений некоторых полей. Они перечислены в табл. 11.1. Эти параметры определяются конструктором, имеющим одиннадцать аргументов. Второй конструктор – конструктор по умолчанию – присваивает параметрам значения, заданные по умолчанию.

Таблица 11.1. Поля класса GridBagConstraints.

Поле	Значение
anchor	Направление размещения компонента в контейнере. Константы: CENTER, NORTH, EAST, NORTHEAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST, и NORTHWEST; по умолчанию CENTER
fill	Растяжение компонента для заполнения ячейки. Константы: NONE, HORIZONTAL, VERTICAL, BOTH; ПО умолчанию NONE
gridheight	Количество ячеек в колонке, занимаемых компонентом. Целое типа int, по умолчанию 1. Константа REMAINDER означает, что компонент займет остаток колонки, RELATIVE – будет следующим по порядку в колонке
gridwidth	Количество ячеек в строке, занимаемых компонентом. Целое типа int, по умолчанию 1. Константа REMAINDER означает, что компонент займет остаток строки, RELATIVE – будет следующим в строке по порядку
gridx	Номер ячейки в строке. Самая левая ячейка имеет номер 0. По умолчанию константа RELATIVE, что означает: следующая по порядку
gridy	Номер ячейки в столбце. Самая верхняя ячейка имеет номер 0. По умолчанию константа RELATIVE, что означает: следующая по порядку
insets	Поля в контейнере. Объект класса insets; по умолчанию объект с нулями
ipadx,**ipady**	Горизонтальные и вертикальные поля вокруг компонентов; по умолчанию 0
weightx,**weighty**	Пропорциональное растяжение компонентов при изменении размера контейнера; по умолчанию 0.0

Как правило, объект класса **GridBagConstraints** создается конструктором по умолчанию, затем значения нужных полей меняются простым присваиванием новых значений, например:

```
GridBagConstraints gbc = new GridBagConstraints();
gbc.weightx = 1.0;
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.gridheight =2;
```

После создания объекта gbc класса **GridBagConstraints** менеджеру размещения указывается, что при помещении компонента comp в контейнер следует применять правила, занесенные в объект gbc. Для этого применяется метод:

```
add(Component comp, GridBagConstraints gbc)
```

Итак, схема применения менеджера GridBagLayout такова:

```
GridBagLayout gbl = new GridBagLayout(); // Создаем менеджер
setLayout(gbl); // Устанавливаем его в контейнер
// Задаем правила размещения по умолчанию
GridBagConstraints c = new GridBagConstraints();
Button b2 = new Button(); // Создаем компонент
c.gridwidth =2; // Меняем правила размещения
add(b1, c); // Помещаем компонент b2 в контейнер
// по указанным правилам размещения c
Button b2 = new Button(); // Создаем следующий компонент
c.gridwidth = 1; // Меняем правила для его размещения
add(b2, c); // Помещаем в контейнер
и т.д.
```

В документации к классу **GridBagLayout** приведен хороший пример использования этого менеджера размещения.

Заключение

Все менеджеры размещения написаны полностью на языке Java, в состав SUN J2SDK входят их исходные тексты. Если вы решили написать свой менеджер размещения, реализовав интерфейс **LayoutManager** или **LayoutManager2**, то посмотрите эти исходные тексты.

Обработка событий

- ## Обработка событий
  В двух предыдущих главах мы написали много программ, создающих интерфейсы, но, собственно, интерфейса, т. е. взаимодействия с пользователем, эти программы не обеспечивают. Можно щелкать по кнопке на экране, она будет "вдавливаться" в плоскость экрана, но больше ничего не будет происходить.
- ## Событие ActionEvent
  Это простое событие означает, что надо выполнить какое-то действие. При этом неважно, что вызвало событие: щелчок мыши, нажатие клавиши или что-то другое. | В классе ActionEvent есть два полезных метода:
- ## Обработка действий мыши
  Событие MouseEvent возникает в компоненте по любой из семи причин: | нажатие кнопки мыши – идентификатор MOUSE_PRESSED; | отпускание кнопки мыши – идентификатор MOUSE_RELEASED; | щелчок кнопкой мыши – идентификатор MOUSE_CLICKED (нажатие и отпускание не различаются);
- ## Классы-адаптеры
  Классы-адаптеры представляют собой пустую реализацию интерфейсов-слушателей, имеющих более одного метода. Их имена составляются из имени события и слова Adapter. Например, для действий с мышью есть два класса-адаптера.
- ## Обработка действий клавиатуры. Событие TextEvent.
  Событие KeyEvent происходит в компоненте по любой из трех причин: | нажата клавиша – идентификатор KEY_PRESSED; | отпущена клавиша – идентификатор KEY_RELEASED; | введен символ – идентификатор KEYJTYPED.
- ## Обработка действий с окном
  Событие windowEvent может произойти по семи причинам: | окно открылось – идентификатор WINDOW_OPENED; | окно закрылось – идентификатор WINDOW_CLOSED; | попытка закрытия окна – идентификатор WINDOW_CLOSING; | окно получило фокус – идентификатор WINDOW_ACTIVATED;
- ## Событие ComponentEvent. Событие ContainerEvent.
  Данное событие происходит в компоненте по четырем причинам: | компонент перемещается – идентификатор COMPONENT_MOVED; | компонент меняет размер – идентификатор COMPONENT_RESIZED; | компонент убран с экрана – идентификатор COMPONENT_HIDDEN;
- ## Событие FocusEvent. Событие ItemEvent.
  Событие возникает в компоненте, когда он получает фокус ввода – идентификатор FOCUS_GAINED, или теряет фокус – идентификатор FOCUS_LOST. | Соответствующий интерфейс: | public interface FocusListener extends EventListener{ | public void focusGainedtFocusEvent e); | public void focusLost(FocusEvent e);
- ## Событие AdjustmentEvent
  Это событие возникает для полосы прокрутки Scrollbar при всяком изменении ее бегунка и отмечается идентификатором ADJUSTMENT_VALUE_CHANGED. | Соответствующий интерфейс описывает один метод: | public interface AdjustmentListener extends EventListener{ | public void adjustmentValueChanged(AdjustmentEvent e);
- ## Несколько слушателей одного источника
  В начале этой главы, в листингах 12.1-12.3, мы привели пример класса TextMove, слушающего сразу два компонента: поле ввода tf типа TextFieid и кнопку b типа Button. | Чаще встречается обратная ситуация – несколько слушателей следят за одним компонентом.
- ## Диспетчеризация событий
  Если вам понадобится обработать просто действие мыши, не важно, нажатие это, перемещение или еще что-нибудь, то придется включать эту обработку во все семь методов двух классов-слушателей событий мыши. | Эту работу можно облегчить, выполнив обработку не в слушателе, а на более ранней стадии.
- ## Создание собственного события
  Вы можете создать собственное событие и определить источник и условия его возникновения. | В листинге 12.6 приведен пример создания события MyEvent, любезно предоставленный Вячеславом Педаком. | Событие MyEvent говорит о начале работы программы (START) и окончании ее работы (STOP).

Обработка событий

В двух предыдущих главах мы написали много программ, создающих интерфейсы, но, собственно, интерфейса, т. е. взаимодействия с пользователем, эти программы не обеспечивают. Можно щелкать по кнопке на экране, она будет "вдавливаться" в плоскость экрана, но больше ничего не будет происходить. Можно ввести текст в поле ввода, но он не станет восприниматься и обрабатываться программой. Все это происходит из-за того, что мы не задали обработку действий пользователя, обработку событий.

Событие (event) в библиотеке AWT возникает при воздействии на компонент какими-нибудь манипуляциями мышью, при вводе с клавиатуры, при перемещении окна, изменении его размеров.

Объект, в котором произошло событие, называется **источником** (source) события.

Все события в AWT классифицированы. При возникновении события исполняющая система Java автоматически создает объект соответствующего событию класса. Этот объект не производит никаких действий, он только хранит все сведения о событии.

Во главе иерархии классов-событий стоит класс **Eventobject** из пакета java.utii – непосредственное расширение класса object. Его расширяет абстрактный класс **AWTEvent**из пакета java.awt – глава классов, описывающих события библиотеки AWT. Дальнейшая иерархия классов-событий показана на рис. 12.1. Все классы, отображенные на рисунке, кроме класса AWTEvent, собраны в пакет java.awt.event.

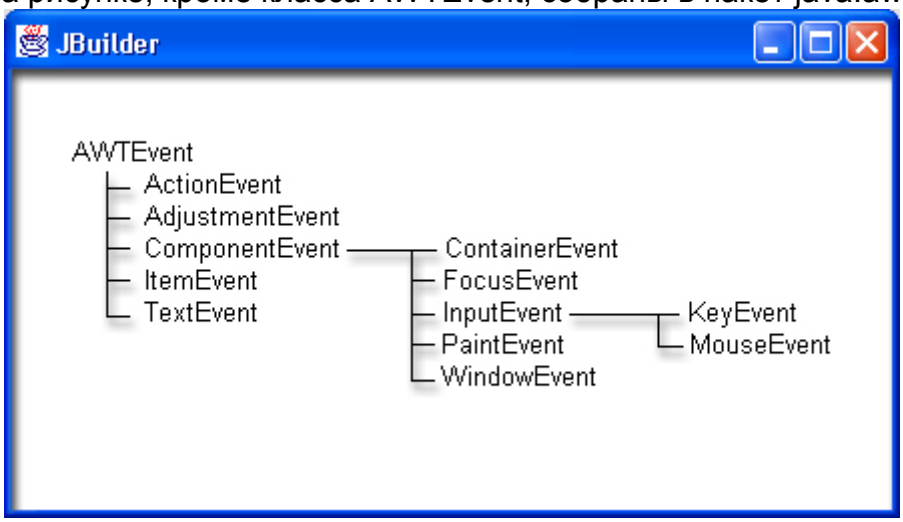

Рис. 12.1. Иерархия классов, описывающих события AWT

События типа **ComponentEvent**, **FbeusEvent**, **KeyEvent**, **MouseEvent** возникают во всех компонентах.

А события типа **ContainerEvent** – только в контейнерах: **Container**, **Dialog**, **FileDialog**,**Frame**, **Panel**, **ScrollPane**, **Window**.

События типа **WindowEvent** возникают только в окнах: **Frame**, **Dialog**, **FileDialog**,**Window**.

События типа **TextEvent** генерируются только в контейнерах **Textcomponent**, **TextArea**,**TextField**.

События типа **ActionEvent** проявляются только в контейнерах **Button**, **List**, **TextField**.

События типа **ItemEvent** возникают только в контейнерах **Checkbox**, **Choice**, **List**.

Наконец, события типа **AdjustmentEvent** возникают только в контейнере **Scrollbar**.

Узнать, в каком объекте произошло событие, можно методом **getsource()** класса Eventobject. Этот метод возвращает тип object.

В каждом из этих классов-событий определен метод paramstring (), возвращающий содержимое объекта данного класса в виде строки string. Кроме того, в каждом классе есть свои методы, предоставляющие те или иные сведения о событии. В частности, метод getioo возвращает **идентификатор** (identifier) события – целое число, обозначающее тип события. Идентификаторы события определены в каждом классе-событии как константы.

Методы обработки событий описаны в интерфейсах – **слушателях** (listener). Для каждого показанного на рис. 12.1 типа событий, кроме inputEvent (это событие редко используется самостоятельно), есть свой интерфейс. Имена интерфейсов составляются из имени события и слова Listener, например, ActionListener, MouseListener. Методы интерфейса "слушают", что происходит в потенциальном источнике события. При возникновении события эти методы автоматически выполняются, получая в качестве аргумента объект-событие и используя при обработке сведения о событии, содержащиеся в этом объекте.

Чтобы задать обработку события определенного типа, надо реализовать соответствующий интерфейс. Классы, реализующие такой интерфейс, классы-обработчики (handlers) события,, называются **слушателями** (listeners): они "слушают", что происходит в объекте, чтобы отследить возникновение события и обработать его.

Чтобы связаться с обработчиком события, классы-источники события должны получить ссылку на экземпляр eventHandier класса-обработчика события одним из методов**addXxxListener(XxxEvent eventHandier)**, где Xxx – имя события.

Такой способ регистрации, при котором слушатель оставляет "визитную карточку" источнику для своего вызова при наступлении события, называется **обратный вызов**(callback). Им часто пользуются студенты, которые, звоня родителям и не желая платить за телефонный разговор, говорят: "Перезвони мне по такому-то номеру". Обратное действие – отказ от обработчика, прекращение прослушивания – выполняется методом**removeXxxListener ().**

Таким образом, компонент-источник, в котором произошло событие, не занимается его обработкой. Он обращается к экземпляру класса-слушателя, умеющего обрабатывать события, **делегирует** (delegate) ему полномочия по обработке.

Такая схема получила название схемы **делегирования** (delegation). Она удобна тем, что мы можем легко сменить класс-обработчик и обработать событие по-другому или назначить несколько обработчиков одного и того же события. С другой стороны, мы можем один обработчик назначить на прослушивание нескольких объектов-источников событий.

Эта схема кажется слишком сложной, но мы ей часто пользуемся в жизни. Допустим, мы решили оборудовать квартиру. Мы помещаем в нее, как в контейнер, разные компоненты: мебель, сантехнику, электронику, антиквариат. Мы предполагаем, что может произойти неприятное событие – квартиру посетят воры, – и хотим его обработать. Мы знаем, что классы-обработчики этого события – охранные агентства, – и обращаемся к некоторому экземпляру такого класса. Компоненты-источники события, т. е. те, которые могут быть украдены, присоединяют к себе датчики методом **addXxxListener()**. Затем экземпляр-обработчик "слушает", что происходит в объектах, к которым он подключен. Он реагирует на наступление только одного события – похищения прослушиваемого объекта, – прочие события, например, короткое замыкание или обрыв водопроводной трубы, его не интересуют. При наступлении "своего" события он действует по контракту, записанному в методе обработки.

Замечание

*В JDK 1.0 была принята другая модель обработки событий. Не удивляйтесь, читая старые книги и просматривая исходные тексты старых программ, но и не пользуйтесь старой моделью***.**

Приведем пример. Пусть в контейнер типа Frame помещено поле ввода tf типа TextField, не редактируемая область ввода ta типа TextArea и кнопка ь типа Button. В поле tf вводится строка, после нажатия клавиши **Enter** или щелчка кнопкой мыши по кнопке ь строка переносится в область ta. После этого можно снова вводить строку в поле tf и т. д.

Здесь и при нажатии клавиши **Enter** и при щелчке кнопкой мыши возникает событие класса ActionEvent, причем оно может произойти в двух компонентах-источниках: поле tf или кнопке ь. Обработка события в обоих случаях заключается в получении строки текста из поля tf (например, методом tf.getText()) и помещений ее в область ta (скажем, методом ta .append ()). Значит, можно написать один обработчик события ActionEvent, реализовав соответствующий интерфейс, который называется **ActionListener**. В этом Интерфейсе всего один метод **actionPerformed().**

Итак, пишем:

```
class TextMove implements ActionListener{
private TextField tf;
private TextArea ta;
TextMove(TextField tf, TextArea ta){
```

```
this.tf = tf; this.ta = ta;
}
public void actionPerformed(ActionEvent ae){
ta.append(tf.getText()+"\n");
}
}
```

Обработчик событий готов. При наступлении события типа ActionEvent будет создан экземпляр класса-обработчика TextMove, конструктор получит ссылки на конкретные поля объекта-источника, метод **actionPerformed ()**, автоматически включившись в работу, перенесет текст из одного поля в другое.

Теперь напишем класс-контейнер, в котором находятся источники tf и события ActionEvent, и подключим к ним слушателя этого события TextMove, передав им ссылки на него методом **addActionListener()**, как показано в листинге 12.1.

Листинг 12.1. Обработка события ActionEvent.

```
import java.awt.*;
import java.awt.event.*;
class MyNotebook extends Frame{
MyNotebook(String title) {
super(title);
TextField tf = new TextField("Вводите текст", 50);
add(tf, BorderLayout.NORTH);
TextArea ta = new TextArea();
ta.setEditable(false);
add(ta);
Panel p = new Panel();
add(p, BorderLayout.SOUTH);
Button b = new Button("Перенести");
p.add(b);
tf.addActionListener(new TextMove(tf, ta));
b.addActionListener(new TextMove(tf, ta));
setSize(300, 200);
setVisible(true);
}
public static void main(String[] args){
Frame f = new MyNotebook(" Обработка ActionEvent");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
// Текст класса TextMove
//…
```

На рис. 12.2 показан результат работы с этой программой.

В листинге 12.1 в методах **addActionListener()** создаются два экземпляра класса TextMove – для прослушивания поля tf и для прослушивания кнопки. Можно создать один экземпляр класса TextMove, он будет прослушивать оба компонента:

```
TextMove tml = new TextMove(tf, ta);
tf.addActionListener(tml);
b.addActionListener(tml);
```

Но в первом случае экземпляры создаются после наступления события в соответствующем компоненте, а во втором – независимо от того, наступило событие или нет, что приводит к расходу памяти, даже если событие не произошло. Решайте сами, что лучше.

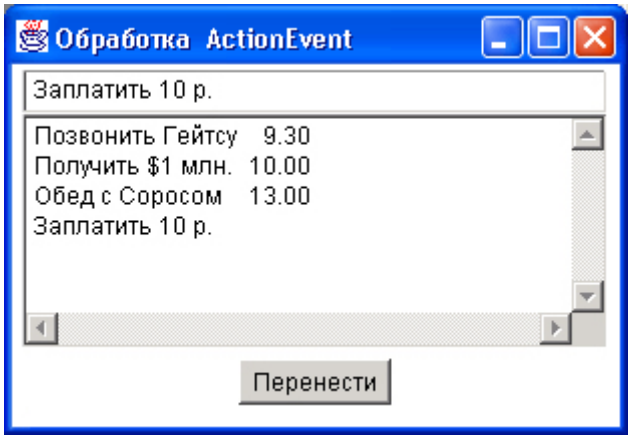

Рис. 12.2. Обработка события ActionEvent

Класс, содержащий источники события, может сам обрабатывать его. Вы можете самостоятельно прослушивать компоненты в своей квартире, установив пульт сигнализации у кровати.

Для этого достаточно реализовать соответствующий интерфейс прямо в классе-контейнере, как показано в листинге 12.2.

Листинг 12.2. Самообработка события ActionEvent.

```
import java.awt.*;
import java.awt.event.*;
class MyNotebook extends Frame implements ActionListener{
private TextField tf;
private TextArea ta;
MyNotebook(String title){
super(title);
tf = new TextField ("Вводите текст**", 50);
add(tf, BorderLayout.NORTH);
ta = new TextArea();
ta.setEditable(false);
add(ta);
Panel p = new Panel();
add(p, BorderLayout.SOUTH);
Button b = new Button("Перенести");
p.add(b);
tf.addActionListener(this);
b.addActionListener(this);
setSize(300, 200); setVisible(true); }
public void actionPerformed(ActionEvent ae){
ta.append(tf.getText()+"\n"); }
public static void main(String[] args){
Frame f = new MyNotebook(" Обработка ActionEvent");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
```

Здесь tf и ta уже не локальные переменные, а переменные экземпляра, поскольку они используются и в конструкторе, и в методе **actionPerformed ()**. Этот метод теперь – один из методов класса MyNotebook. Класс MyNotebook стал классом-обработчиком события ActionEvent – он реализует интерфейс **ActionListener**. В Методе **addActionListener ()**указывается аргумент this – класс сам слушает свои компоненты.

Рассмотренная схема, кажется, проще и удобнее, но она предоставляет меньше возможностей. Если вы захотите изменить обработку, например заносить записи в поле ta по алфавиту или по времени выполнения заданий, то придется переписать и перекомпилировать класс MyNotebook.

Еще один вариант – сделать обработчик вложенным классом. Это позволяет обойтись без переменных экземпляра и конструктора в классе-обработчике TextMove, как показано в листинге 12.3.

Листинг 12.3. Обработка вложенным классом.

```
import Java.awt.*;
import j ava.awt.event.*;
class MyNotebook extends Frame{ private TextField tf;
private TextArea ta;
MyNotebook(String title){
super(title);
tf = new TextField("Вводите текст", 50);
add(tf, BorderLayout.NORTH);
ta = new TextArea();
ta.setEditable(false);
add (tab-Panel p = new Panel();
add(p, BorderLayout.SOUTH);
Button b = new Button("Перенести");
p.add(b);
tf.addActionListener(new TextMove());
b.addActionListener(new TextMove());
setSizepOO, 200);
setVisible(true);
}
public static void main(String[] args){
Frame f = new MyNotebook(" Обработка ActionEvent");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit (0);
}
});
}
// Вложенный класс
class TextMove implements ActionListener{
public void actionPerformed(ActionEvent ae){
ta.appendftf.getText()+"\n");
}
}
}
```

Наконец, можно создать безымянный вложенный класс, что мы и делали в этой и предыдущих главах, обрабатывая нажатие комбинации клавиш **ALT + F4** или щелчок кнопкой мыши по кнопке закрытия окна. При этом возникает событие типа **windowEvent**, для его обработки мы обращались к методу **windowciosing**(), реализуя его обращением к методу завершения приложения **System.exit (0)**. Но для этого нужно иметь суперкласс определяемого безымянного класса, такой как **windowAdapter**. Такими суперклассами могут быть классы-адаптеры, о них речь пойдет чуть ниже.

Перейдем к детальному рассмотрению разных типов событий.

Событие ActionEvent

Это простое событие означает, что надо выполнить какое-то действие. При этом неважно, что вызвало событие: щелчок мыши, нажатие клавиши или что-то другое.

В классе **ActionEvent** есть два полезных метода:

- метод **getActionCommand ()** возвращает в виде строки string надпись на кнопке Button, точнее, то, что установлено методом **setActionCoramand (Strings)** класса Button, выбранный пункт списка List, или что-то другое, зависящее от компонента;
- метод **getModifiers()** возвращает код клавиш **ALT**, **CTRL**, **Meta** или **SHIFT**, если какая-нибудь одна или несколько из них были нажаты, в виде числа типа int; узнать, какие именно клавиши были нажаты, можно сравнением со статическими константами этого класса ALT_MASK, CTRL_MASK, META_MASK, SHIFT_MASK.

Примечание

*Клавиши **Meta** на PC-клавиатуре нет, ее действие часто назначается на клавишу **Esc** или левую клавишу **ALT**.*

Например:

```
public void actionPerformed(ActionEvent ae){
if (ae.getActionCommand() == "Open" &&
(ae.getModifiers() | ActionEvent.ALT_MASK)!= 0){
// Какие-то действия
}
}
```

Обработка действий мыши

Событие **MouseEvent** возникает в компоненте по любой из семи причин:

- нажатие кнопки мыши – идентификатор **MOUSE_PRESSED**;
- отпускание кнопки мыши – идентификатор **MOUSE_RELEASED**;
- щелчок кнопкой мыши – идентификатор **MOUSE_CLICKED** (нажатие и отпускание не различаются);
- перемещение мыши – идентификатор **MOUSE_MOVED**;
- перемещение мыши с нажатой кнопкой – идентификатор **MOUSE_DRAGGED**;
- появление курсора мыши в компоненте – идентификатор **MOUSE_ENTERED**;
- выход курсора мыши из компонента – идентификатор **MOUSE_EXITED**.

Для их обработки есть семь методов в двух интерфейсах:

```
public interface MouseListener extends EventListener{
public void mouseClicked(MouseEvent e);
public void mousePressed(MouseEvent e);
public void mouseReleased(MouseEvent e);
public void mouseEntered(MouseEvent e);
public void mouseExited(MouseEvent e);
}
public interface MouseMotionListener extends EventListener{
public void mouseDragged(MouseEvent e);
public void mouseMoved(MouseEvent e);
}
```

Эти методы могут получить от аргумента e координаты курсора мыши в системе координат компонента методами **e.getx()**, **e.getv()**, или одним методом **e.getPoint()**, возвращающим экземпляр класса Point.

Двойной щелчок кнопкой мыши можно отследить методом **e.getciickcount()**, возвращающим количество щелчков. При перемещении мыши возвращается 0.

Узнать, какая кнопка была нажата, можно с помощью метода **e.getModifiers()** класса inputEvent сравнением со следующими статическими константами класса inputEvent:

- **BUTTON1_MASK** – нажата первая кнопка, обычно левая;
- **BUTTON2_MASK** – нажата вторая кнопка, обычно средняя, или одновременно нажаты обе кнопки на двухкнопочной мыши;
- **BUTTON3_MASK** – нажата третья кнопка, обычно правая.

Приведем пример, уже ставший классическим. В листинге 12.4 представлен простейший вариант "рисовалки" – класс scribble. При нажатии первой кнопки мыши методом**mousePressed ()** запоминаются координаты курсора мыши. При протаскивании мыши вычерчиваются отрезки прямых между текущим и предыдущим положением курсора мыши методом **mouseDragged()**. На рис. 12.3 показан пример работы с этой программой.

Листинг 12.4. Простейшая программа рисования.

```
import java.awt.*;
import java.awt.event.*;
public class ScribbleTest extends Frame{
public ScribbleTest(String s){
super(s);
ScrollPane pane = new ScrollPane();
pane.setSize(300, 300);
add(pane, BorderLayout.CENTER);
Scribble scr = new Scribble(this, 500, 500);
pane.add(scr);
Panel p = new Panel 0;
add(p, BorderLayout.SOUTH);
Button bl = new Button("Красный");
p.add(bl);
bl.addActionListener(scr);
Button b2 = new Button("Зеленый");
p.add(b2);
b2.addActionListener(scr);
Button b3 = new Button("Синий");
p.add(b3);
b3.addActionListener(scr);
Button b4 = new Button("Черный");
p.add(b4);
b4.addActionListener(scr);
Button b5 = new Button("Очистить");
p.add(bS);
b5.addActionListener(scr);
addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e){
System.exit(0);
}
});
pack();
setvisible(true);
}
public static void main(String[] args){
new ScribbleTest(" \"Рисовалка\"");
}
}
class Scribble extends Component implements ActionListener, MouseListener,
MouseMotionListener{
protected int lastX, lastY, w, h;
protected Color currColor = Color.black;
protected Frame f;
public Scribble(Frame frame, int width, int height){
f = frame;
w = width;
h = height;
enableEvents(AWTEvent.MOUSE_EVENT_MASK | AWTEvent.MOUSE_MOTION_EVENT_MASK);
addMouseListener(this);
addMouseMotionListener(this); }
public Dimension getPreferredSize(){
return new Dimension(w, h); }
public void actionPerformed(ActionEvent event){
String s = event.getActionCommand();
if (s.equals ("Очистить")) repaint();
else if (s.equals ("Красный")) currColor = CofLor.red;
else if (s.equals("Зеленый")) currColor = Coior.green;
else if (s.equals("Синий")) currColor = Color.blue;
else if (s.equals("Черный")) currColor = Color.black; }
public void mousePressed(MouseEvent e){
if ((e.getModifiers() & MouseEvent.BUTTON 1__MASK) = 0) return;
lastX = e.getXO; lastY = e.getYO; }
public void mouseDragged(MouseEvent e){
if ((e.getModifiers() & MouseEvent.BUTTON1_MASK) = 0) return;
Graphics g = getGraphics();
```

```
g.setColor(currColor);
g.drawLinedastX, lastY, e.getX(), e.getY());
lastX = e.getX(); lastY = e.getY(); }
public void mouseReleased(MouseEvent e){}
public void mouseClicked(MouseEvent e){}
public void mouseEntered(MouseEvent e){}
public void mouseExited(MouseEvent e){}
public void mouseMoved(MouseEvent e){}
}
```

Рис. 12.3. Пример работы с программой рисования

При создании класса-слушателя **scribble** и реализации интерфейсов **MouseListener** и**MouseMotionListener** пришлось реализовать все их семь методов, хотя мы отслеживали только нажатие и перемещение мыши, и нам нужны были только методы **mousePressed ()** и**mouseDragged ()**. Для остальных методов мы задали пустые реализации.

Чтобы облегчить задачу реализации интерфейсов, имеющих более одного метода, созданы классы-адаптеры.

Классы-адаптеры

Классы-адаптеры представляют собой пустую реализацию интерфейсов-слушателей, имеющих более одного метода. Их имена составляются из имени события и слова **Adapter**. Например, для действий с мышью есть два класса-адаптера. Выглядят они очень просто:

```
public abstract class MouseAdapter implements MouseListener{
public void mouseClicked(MouseEvent e){}
public void mousePressed(MouseEvent e){}
public void mouseReleased(MouseEvent e){}
public void mouseEntered(MouseEvent e){}
public void mouseExited(MouseEvent e){}
}
public abstract class MouseMotionAdapter implements MouseMotionListener{
public void mouseDragged(MouseEvent e){}
public void mouseMoved(MouseEvent e){}
}
```

Вместо того чтобы реализовать интерфейс, можно расширять эти классы. Не бог весть что, но полезно для создания безымянного вложенного класса, как у нас и делалось для закрытия окна. Там мы использовали класс-адаптер **WindowAdapter**.

Классов-адаптеров всего семь. Кроме уже упомянутых трех классов, это классы**Component Adapter**, **ContainerAdapter**, **FocusAdapter** и **KeyAdapter**.

Обработка действий клавиатуры. Событие TextEvent.

Событие KeyEvent происходит в компоненте по любой из трех причин:

- нажата клавиша – идентификатор **KEY_PRESSED**;
- отпущена клавиша – идентификатор **KEY_RELEASED**;
- введен символ – идентификатор **KEYJTYPED**.

Последнее событие возникает из-за того, что некоторые символы вводятся нажатием нескольких клавиш, например, заглавные буквы вводятся комбинацией клавиш **SHIFT + буква**. Вспомните еще **Alt-ввод** в MS Windows. Нажатие функциональных клавиш, например**F1**, не вызывает событие **KEY_TYPED**.

Обрабатываются эти события тремя методами, описанными в интерфейсе:

```
public interface KeyListener extends EventListener{
public void keyTyped(KeyEvent e);
public void keyPressed(KeyEvent e);
public void keyReleased(KeyEvent e);
}
```

Аргумент e этих методов может дать следующие сведения.

Метод **e.getKeyChar()** возвращает символ Unicode типа char, связанный с клавишей. Если с клавишей не связан никакой символ, то возвращается константа CHAR_UNDEFINED.

Метод **e.getKeyCode ()** возвращает код клавиши в виде целого числа типа int. В классе KeyEvent определены коды всех клавиш в виде констант, называемых **виртуальными кодами** клавиш (virtual key codes), например, VK_FI, VK_SHIFT, VK_A, VK_B, VK_PLUS. Они перечислены в документации к классу KeyEvent. Фактическое значение виртуального кода зависит от языка и раскладки клавиатуры. Чтобы узнать, какая клавиша была нажата, надо сравнить результат выполнения метода **getKeyCode ()** с этими константами. Если кода клавиши нет, как происходит при наступлении события KEY_TYPED, то возвращается значение VK_UNDEFINED.

Чтобы узнать, не нажата ли одна или несколько клавиш-модификаторов **ALT**, **CTRL**,**Meta**, **SHIFT**, надо воспользоваться унаследованным от класса inputEvent методом**getModifiers()** и сравнить его результат с константами **ALT_MASK**, **CTRL_MASK**,**META_MASK**, **SHIFTJMASK**.

Другой способ – применить логические методы **isALTDown()**, **isControlDown()**,**isMetaDown()**, **isSHIFTDown()**.

Добавим в листинг 12.3 возможность очистки поля ввода tf после нажатия клавиши **Esc**. Для этого перепишем вложенный класс-слушатель **TextMove**:

```
class TextMove implements ActionListener, KeyListener{
public void actionPerformed(ActionEvent ae){
ta.append{tf.getText 0+"\n");
}
public void keyPressed(KeyEvent ke) {
if (ke.getKeyCodeO == KeyEvent.VK_ESCAPE) tf.setText("");
}
public void keyReleased(KeyEvent ke){()}
public void keyTyped(KeyEvent ke){}
}
```

Событие TextEvent

Событие **TextEvent** происходит только по одной причине – изменению текста – и отмечается идентификатором **TEXT_VALUE_CHANGED**.

Соответствующий интерфейс имеет только один метод:

```
public interface TextListener extends EventListener{
public void textValueChanged(TextEvent e);
}
```

От аргумента e этого метода можно получить ссылку на объект-источник события методом **getsource()**, унаследованным от класса Eventobject, например, так:

```
TextComponent tc = (TextComponent)e.getSpurce();
String s = tc.getText();
// Дальнейшая обработка
```

Обработка действий с окном

Событие **windowEvent** может произойти по семи причинам:

- окно открылось – идентификатор **WINDOW_OPENED**;
- окно закрылось – идентификатор **WINDOW_CLOSED**;
- попытка закрытия окна – идентификатор **WINDOW_CLOSING**;
- окно получило фокус – идентификатор **WINDOW_ACTIVATED**;
- окно потеряло фокус – идентификатор **WINDOW_DEACTIVATED**;
- окно свернулось в ярлык – идентификатор **WINDOW_ICONIFIED**;
- окно развернулось – идентификатор **WINDOW_DEICONIFIED**.

Соответствующий интерфейс содержит семь методов:

```
public interface WindowListener extends EventListener {
public void windowOpened(WindowEvent e);
public void windowClosing(WindowEvent e);
public void windowClosed(WindowEvent e);
public void windowlconified(WindowEvent e);
public void windowDeiconified(WindowEvent e);
public void windowActivated(WindowEvent e);
public void windowDeactivated(WindowEvent e); }
```

Аргумент **e** этих методов дает ссылку типа window на окно-источник методом**e.getwindow()**.

Чаще всего эти события используются для перерисовки окна методом **repaint()** при изменении его размеров и для остановки приложения при закрытии окна.

Событие ComponentEvent. Событие ContainerEvent.

Данное событие происходит в компоненте по четырем причинам:

- компонент перемещается – идентификатор **COMPONENT_MOVED**;
- компонент меняет размер – идентификатор **COMPONENT_RESIZED**;
- компонент убран с экрана – идентификатор **COMPONENT_HIDDEN**;
- компонент появился на экране – идентификатор **COMPONENT_SHOWN**.

Соответствующий интерфейс содержит описания четырех методов:

```
public interface ComponentListener extends EventListener{
public void componentResized(ComponentEvent e);
public void componentMoved(Comp©nentEvent e);
public void componentShown(ComponentEvent e);
public void componentHidden(ComponentEvent e);
}
```

Аргумент **e** методов этого интерфейса предоставляет ссылку на компонент-источник события методом **e.getComponent()**.

Событие ContainerEvent

Это событие происходит по двум причинам:

- в контейнер добавлен компонент – идентификатор **COMPONENT_ADDED**;
- из контейнера удален компонент – идентификатор **COMPONENT_REMOVED**.

Этим причинам соответствуют методы интерфейса:

```
public interface ContainerListener extends EventListener{
public void componentAdded(ContainerEvent e);
public void componentRemoved(ContainerEvent e);
}
```

Аргумент **e** предоставляет ссылку на компонент, чье добавление или удаление из контейнера вызвало событие, методом **e.getchild()**, и ссылку на контейнер – источник события методом **e.getcontainer ()**.

Обычно при наступлении данного события контейнер перемещает свои компоненты.

Событие FocusEvent. Событие ItemEvent.

Событие возникает в компоненте, когда он получает фокус ввода – идентификатор**FOCUS_GAINED**, или теряет фокус – идентификатор **FOCUS_LOST**.

Соответствующий интерфейс:

```
public interface FocusListener extends EventListener{
public void focusGainedtFocusEvent e);
public void focusLost(FocusEvent e);
}
```

Обычно при потере фокуса компонент перечеркивается бледным цветом, для этого применяется метод **brighter ()** класса **Color**, при получении фокуса становится ярче, что достигается применением метода **darker ()**.

Это приходится делать самостоятельно при создании своего компонента.

Событие ItemEvent

Это событие возникает при выборе или отказе от выбора элемента в списке List, **choice**или флажка **checkbox** и отмечается идентификатором **ITEM_STATE_CHANGED**.

Соответствующий интерфейс очень прост:

```
public interface ItemListener extends EventListener{
void itemStateChanged(ItemEvent e);
}
```

Аргумент e предоставляет ссылку на источник методом **e.getitemselectable()**, ссылку на выбранный пункт методом **e.getitem()** в виде object.

Метод **e.getstatechange()** позволяет уточнить, что произошло: значение **SELECTED**указывает на то, что элемент был выбран, значение **DESELECTED** – произошел отказ от выбора.

В следующей главе мы рассмотрим примеры использования этого события.

Событие AdjustmentEvent

Это событие возникает для полосы прокрутки **Scrollbar** при всяком изменении ее бегунка и отмечается идентификатором **ADJUSTMENT_VALUE_CHANGED**.

Соответствующий интерфейс описывает один метод:

```
public interface AdjustmentListener extends EventListener{
public void adjustmentValueChanged(AdjustmentEvent e);
}
```

Аргумент e этого метода предоставляет ссылку на источник события методом**e.getAdjustable()**, текущее значение положения движка полосы прокрутки методом**e.getvalue ()**, и способ изменения его значения методом **e.getAdjustmentType()**, возвращающим следующие значения:

- **UNIT__INCREMENT** – увеличение на одну единицу;
- **UNIT_DECREMENT** – уменьшение на одну единицу;
- **BLOCK_INCREMENT** – увеличение на один блок;
- **BLOCK_DECREMENT** – уменьшение на один блок;
- **TRACK** – процес передвижения бегунка полосы прокрутки.

"Оживим" программу создания цвета, приведенную в листинге 10.4, добавив необходимые действия. Результат этого приведен в листинге 12.5.

Листинг 12.5. Программа создания цвета.

```
import java.awt.*;
import java.awt.event.*;
class ScrollTestl extends Frame{
private Scroiibar
sbRed = new Scroiibar(Scroiibar.VERTICAL, 127, 16, 0, 271),
sbGreen = new Scroiibar(Scroiibar.VERTICAL, 127, 16, 0, 271),
sbBlue = new Scroiibar(Scroiibar.VERTICAL, 127, 16, 0, 271);
private Color c = new Color(127, 127, 127);
private Label 1m = new Label();
private Button
b1= new Button("Применить"),
b2 = new Button("Отменить");
ScrollTestl(String s){
super(s);
setLayout(null);
setFont(new Font("Serif", Font.BOLD, 15));
Panel p = new Panel();
p.setLayout(null);
p.setBounds(10.50, 150, 260); add(p);
Label Ic = new Label("Подберите цвет");
lc.setBounds(20, 0, 120, 30); p.add(lc);
Label Imin = new Label("0", Label.RIGHT);
lmin.setBounds(0, 30, 30, 30); p.add(lmin);
Label Middle = new Label("127", Label.RIGHT);
lmiddle.setBounds(0, 120, 30, 30); p.add(Imiddle);
Label Iroax = new Label("255", Label.RIGHT);
Imax.setBoundsfO, 200, 30, 30); p.add(lraax);
sbRed.setBackground(Color.red);
sbRed.setBounds(40, 30, 20, 200); p.add(sbRed);
sbRed.addAdjustmentListener(new ChColorO);
```

```
sbGreen.setBackground(Color.green);
sbGreen.setBounds(70, 30, 20, 200); p.add(sbGreen);
sbGreen.addAdjustmentListener(new ChColor());
sbBlue.setBackground(Color.blue);
sbBlue.setBoundsds (100, 30, 20, 200); p.add(sbBlue);
sbBlue.addAdjustmentListener(new ChColor());
Label Ip = new Label("Образец:");
lp.setBoundS(250, 50, 120, 30); add(lp);
1m.setBackground(new Color(127, 127, 127));
Im.setBounds(220, 80, 120, 80); add(lm);
b1.setBounds(240, 200, 100, 30); add(bl);
b1.addActionListener(new ApplyColor());
b2.setBounds(240, 240, 100, 30); add(b2);
b2.addActionListener(new CancelColor());
setSize(400, 300); setVisible(true);)
class ChColor implements AdjustmentListener{
public void adjustmentValueChanged(AdjustmentEvent e){
int red = c.getRed(), green = c.getGreen(), blue = c.getBlue();
if (e.getAdjustable() == sbRed) red = e.getValue();
else if (e.getAdjustablet) == sbGreen) green = e.getValue();
else if (e.getAdjustable() == sbBlue) blue = e.getValue();
c = new Color(red, green, blue);
lm.setBackground(c);
}
}
class ApplyColor implements ActionListener {
public void actionPerformed(ActionEvent ae){
setBackground(c);
}
}
class CancelColor implements ActionListener {
public void actionPerformed(ActionEvent ae){
c = new Color(127, 127, 127);
sbRed.setValue(127);
sbGreen.setValue(127);
sbBlue.setValue(127);
lm.setBackground(c);
setBackground(Color.white);
}
}
public static void main(String[] args){
Frame f = new ScrollTestl(" Выбор цвета");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
```

Несколько слушателей одного источника

В начале этой главы, в листингах 12.1-12.3, мы привели пример класса TextMove, слушающего сразу два компонента: поле ввода tf типа TextFieid и кнопку b типа Button.

Чаще встречается обратная ситуация – несколько слушателей следят за одним компонентом. В том же примере кнопка ь в ответ на щелчок по ней кнопки мыши совершала еще и собственные действия – она "вдавливалась", а при отпускании кнопки мыши становилась "выпуклой". В классе Button эти действия выполняет peer-объект.

В классе **FiowerButton** листинга 10.6 такие же действия выполняет метод **paint()** этого класса.

В данной модели реализован **design pattern** под названием **observer.**

К каждому компоненту можно присоединить сколько угодно слушателей одного и того же события или разных типов событий. Однако при этом не гарантируется какой-либо определенный

порядок их вызова, хотя чаще всего слушатели вызываются в порядке написания методов **addXxxListener ()**.

Если нужно задать определенный порядок вызовов слушателей для обработки события, то придется обращаться к ним друг из друга или создавать объект, вызывающий слушателей в нужном порядке.

Ссылки на присоединенные методами **addxxxbistener ()** слушатели можно было бы хранить в любом классе-коллекции, например, vector, но в пакет j ava .awt специально для этого введен класс **AWTEventMuiticaster**. Он реализует все одиннадцать интерфейсов xxxListener, значит, сам является слушателем любого события. Основу класса составляют своеобразные статические методы addo, написанные для каждого типа событий, например:

```
add(ActionListener a, ActionListener b)
```

Своеобразие этих методов двоякое: они возвращают ссылку на тот же интерфейс, в данном случае, **ActionListener**, и присоединяют объект а к объекту ь, создавая совокупность слушателей одного и того же типа. Это позволяет использовать их наподобие операций а += b. Заглянув в исходный текст класса Button, вы увидите, что метод **addActionListener ()**очень прост:

```
public synchronized void addActionListener(ActionListener 1){
if (1 = null){ return; }
actionListener = AWTEventMuiticaster.add(actionListener, 1);
newEventsOnly = true;
}
```

Он добавляет к совокупности слушателей **actionListener** нового слушателя 1.

Для событий типа **inputEvent**, а именно, **KeyEvent** и **MouseEvent**, есть возможность прекратить дальнейшую обработку события методом **consume ()**. Если записать вызов этого метода в класс-слушатель, то ни peer-объекты, ни следующие слушатели не будут обрабатывать событие. Этим способом обычно пользуются, чтобы отменить стандартные действия компонента, например, "вдавливание" кнопки.

Диспетчеризация событий

Если вам понадобится обработать просто действие мыши, не важно, нажатие это, перемещение или еще что-нибудь, то придется включать эту обработку во все семь методов двух классов-слушателей событий мыши.

Эту работу можно облегчить, выполнив обработку не в слушателе, а на более ранней стадии. Дело в том, что прежде чем событие дойдет до слушателя, оно обрабатывается несколькими методами.

Чтобы в компоненте произошло событие AWT, должно быть выполнено хотя бы одно из двух условий: к компоненту присоединен слушатель или в конструкторе компонента определена возможность появления события методом **enableEvents ().** В аргументе этого метода через операцию побитового сложения перечисляются константы класса AWTEvent, задающие события, которые могут произойти в компоненте, например:

```
enableEvents(AWTEvent.MOUSE_MOTION_EVENT_MASK |
AWTEvent.MOUSE_EVENT_MASK I AWTEvent.KEY_EVENT_MASK)
```

При появлении события создается объект соответствующего класса xxxEvent. Метод**dispatchEvent ()** определяет, где появилось событие – в компоненте или одном из его подкомпонентов, – и передает объект-событие методу **processEvent ()** компонента-источника.

Метод **processEvent ()** определяет тип события и передает его специализированному методу **processxxxEvent ()**. Вот начало этого метода:

```
protected void processEvent(AWTEvent e){
if (e instanceof FocusEvent){
processFocusEvent((FocusEvent)e);
}else if (e instanceof MouseEvent){
switch (e.getlDO) {
case MouseEvent.MOUSE_PRESSED:
case MouseEvent.MOUSE_RELEASED:
case MouseEvent.MOUSE_CLICKED:
case MouseEvent.MOUSE_ENTERED:
```

```
case MouseEvent.MOUSE_EXITED:
processMouseEvent((MouseEvent)e);
break/case MouseEvent.MOUSE_MOVED:
case MouseEvent.MOUSE_DRAGGED:
processMouseMotionEvent((MouseEvent)e);
break; } }else if (e instanceof KeyEvent){
processKeyEvent((KeyEvent)e); }
//…
```

Затем в дело вступает специализированный метод, например, **processKeyEvent ()**. Он-то и передает объект-событие слушателю. Вот исходный текст этого метода:

```
protected void processKeyEvent(KeyEvent e){
KeyListener listener = keyListener;
if (listener!= null){ int id = e.getlDf);
switch(id){
case KeyEvent.KEYJTYPED: listener.keyTyped(e);
break;
case KeyEvent.KEY_PRESSED: listener.keyPressed(e);
break;
case KeyEvent.KEY_RELEASED: listener.keyReleased(e);
break;
}
}
}
```

Из этого описания видно, что если вы хотите обработать любое событие типа AWTEvent, то вам надо переопределить метод **processEvent ()**, а если более конкретное событие, например, событие клавиатуры, – переопределить более конкретный метод**processKeyEvent ()**. Если вы не переопределяете весь метод целиком, то не забудьте в конце обратиться к методу суперкласса, например:

```
super.processKeyEvent(e);
```

Замечание

Не забывайте обращаться к методу processXxxEvent() суперкласса.

В следующей главе мы применим такое переопределение в листинге 13.2 для вызова всплывающего меню.

Создание собственного события

Вы можете создать собственное событие и определить источник и условия его возникновения.

В листинге 12.6 приведен пример создания события **MyEvent**, любезно предоставленный Вячеславом Педаком.

Событие **MyEvent** говорит о начале работы программы (**START**) и окончании ее работы (**STOP**).

Листинг 12.6, Создание собственного события.

```
// 1. Создаем свой класс события:
public class MyEvent extends java.util.EventObjectf protected int id;
public static final int START = 0, STOP = 1;
public MyEvent(Object source, int id){
super(source);
this.id = id;
}
public int getID(){ return id; }
}
// 2. Описываем Listener:
public interface MyListener extends java.util.EventListener{
public void start{MyEvent e);
public void stop(MyEvent e); }
// 3. В теле нужного класса создаем метод fireEvent():
protected Vector listeners = new Vector();
public void fireEvent(MyEvent e){
Vector list = (Vector) listeners.clone();
for (int i <strong>=</strong> 0; i < list.sizeO; i++) {
```

```
MyListener listener = (MyListener)list.elementAt(i);
switch(e.getlDO) {
case MyEvent.START: listener.start(e); break;
case MyEvent.STOP: listener.stop(e); break;
}
}
}
```

Все, теперь при запуске программы делаем:

```
fi reEvent(thi s, MyEvent.START);
```

…а при окончании:

```
fireEvent(this, MyEvent.STOP);
```

При этом все зарегистрированные слушатели получат экземпляры событий.

Создание меню

В контейнер типа **Frame** заложена возможность установки стандартной **строки меню**(menu bar), располагаемой ниже строки заголовка, как показано на рис. 13.1. Эта строка – объект класса MenuBar.

Все, что нужно сделать для установки строки меню в контейнере Frame – это создать объект класса MenuBar и обратиться к методу **setMenuBar ()**:

```
Frame f = new Frame("Пример меню");
MenuBar mb = new MenuBar();
f.setMenuBar(mb);
```

Если имя mb не понадобится, можно совместить два последних обращения к методам:

```
f.setMenuBar(new MenuBar());
```

Разумеется, строка меню еще пуста и пункты меню не созданы.

Каждый элемент строки меню – **выпадающее меню** (drop-down menu) – это объект класса Menu. Создать эти объекты и занести их в строку меню ничуть не сложнее, чем создать строку меню:

```
Menu mFile = new Menu("Файл");
mb.add(mFile);
Menu mEdit = new Menu("Правка");
mb.add(mEdit);
Menu mView = new Menu("Вид");
mb.add(mView);
Menu mHelp = new Menu("Справка");
mb.setHelpMenu(mHelp);
```

…и т. д. Элементы располагаются слева направо в порядке обращений к методам **add()**, как показано на рис. 13.1. Во многих графических системах принято меню Справка (**Help**) прижимать к правому краю строки меню. Это достигается обращением к методу**setHelpMenu ()**, но фактическое положение меню **Справка** определяется графической оболочкой.

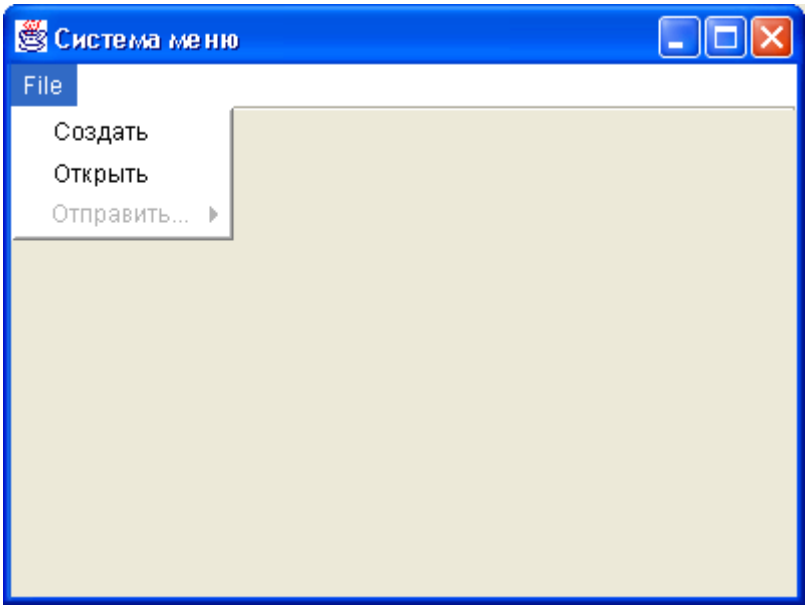

Рис. 13.1. Система меню

Затем определяем каждое выпадающее меню, создавая его пункты. Каждый пункт меню – это объект класса **Menuitem**. Схема его создания и добавления к меню точно такая же, как и самого меню:

```
Menuitem create = new Menuitem("Создать");
mFile.add(create);
Menuitem open = new Menuitem("Открыть…");
mFile.add(open);
```

…и т. д. Пункты меню будут расположены сверху вниз в порядке обращения к методам**add()**.

Часто пункты меню объединяются в группы. Одна группа от другой отделяется горизонтальной чертой. На рис. 13.1 черта проведена между командами **Открыть** и**Отправить**. Эта черта создается методом **addseparator ()** класса Menu или определяется как пункт меню с надписью специального вида – дефисом:

```
mFile.addfnew Menuitem("-"));
```

Интересно, что класс Menu расширяет класс Menuitem, а не наоборот. Это означает, что меню само является пунктом меню, и позволяет задавать меню в качестве пункта другого меню, тем самым организуя вложенные подменю:

```
Menu send = new Menu("Отправить");
mFile.add(send);
```

Здесь меню send добавляется в меню mFile как один из его пунктов. Подменю send заполняется пунктами меню как обычное меню.

Часто команды меню создаются для выбора из них каких-то возможностей, подобно компонентам checkbox. Такие пункты можно выделить щелчком кнопки мыши или отменить выделение повторным щелчком. Эти команды – объекты класса **CheckboxMenuItem**:

```
CheckboxMenuItem disk = new CheckboxMenuItem("Диск A:", true);
send.add(disk);
send.add(new CheckboxMenuItem("Архив"));
и т.д.
```

Все, что получилось в результате перечисленных действий, показано на рис. 13.1.

Многие графические оболочки, но не MS Windows, позволяют создавать **отсоединяемые**(tear-off) меню, которые можно перемещать по экрану. Это указывается в конструкторе:

```
Menu(String label, boolean tearOff)
```

Если tearoff == true и графическая оболочка умеет создавать отсоединяемое меню, то оно будет создано. В противном случае этот аргумент просто игнорируется.

Наконец, надо назначить действия командам меню. Команды меню типа Menuitem порождают события типа ActionEvent, поэтому нужно присоединить к ним объект класса-слушателя как к обычным компонентам, записав что-то вроде:

```
create.addActionListener(new SomeActionEventHandler())
open.addActionListener(new AnotherActionEventHandler())
```

Пункты типа **CheckboxMenuItem** порождают события типа ItemEvent, поэтому надо обращаться к объекту-слушателю этого события:

```
disk.addltemListener(new SomeltemEventHandler())
```

Очень часто действия, записанные в командах меню, вызываются не только щелчком кнопки мыши, но и "горячими" клавишами-акселераторами (**shortcut**), действующими чаще всего при нажатой клавише **CTRL**. На экране в пунктах меню, которым назначены "горячие" клавиши, появляются подсказки вида **CTRL + N**, **CTRL + O**, как на рис. 13.1. "Горячая" клавиша определяется объектом класса MenuShortcut и указывается в его конструкторе константой класса KeyEvent, например:

```
MenuShortcut keyCreate = new MenuShortcut(KeyEvent.VK_N);
```

После этого "горячей" будет комбинация клавиш **CTRL + N**. Затем полученный объект указывается в конструкторе класса Menuitem:

```
Menuitem create = new Menuitem("Создать", keyCreate);
```

Нажатие **CTRL + N** будет вызывать окно создания. Эти действия, разумеется, можно совместить, например:

```
Menuitem open = new Menultern("Открыть…",
new – MenuShortcut(KeyEvent.VK_O));
```

Можно добавить еще нажатие клавиши **SHIFT**. Действие пункта меню будет вызываться нажатием комбинации клавиш **SHIFT + CTRL + X**, если воспользоваться вторым конструктором:

```
MenuShortcut(int key, boolean useSHIFT)
```

…с аргументом **useSHIFT == true**.

Программа рисования, созданная в листинге 12.4 и показанная на рис. 12.3, явно перегружена кнопками. Перенесем их действия в пункты меню. Добавим возможность манипуляции файлами и команду завершения работы. Это сделано в листинге 13.1. Класс**scribble** не изменялся и в листинге не приведен. Результат показан на рис. 13.2.

Листинг 13.1. Программа рисования с меню.

```
import java.awt.*;
import java.awt.event.*;
public class MenuScribble extends Frame{
public MenuScribble(String s) { super(s);
ScrollPane pane = new ScrollPane();
pane.setSize(300, 300);
add(pane, BorderLayout.CENTER);
Scribble scr = new Scribble(this, 500, 500);
pane.add(scr);
MenuBar mb = new MenuBar();
setMenuBar(mb);
Menu f = new Menu("Файл");
Menu v = new Menu("Вий");
mb.add(f); mb.add(v);
Menuitem open = new Menuitem("Открыть…",
new MenuShortcut(KeyEvent.VK_0));
Menuitem save = new Menuitem("Сохранить",
new MenuShortcut(KeyEvent.VK_S));
Menuitem saveAs = new Menultera("Сохранить как…");
Menuitem exit = new Menuitem("Выход",
new MenuShortcut(KeyEvent.VK_Q));
f.add(open); f.add(save); f.add(saveAs);
f.addSeparator(); f.add(exit);
open.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
```

```
FileDialog fd = new FileDialog(new Frame(),
" Загрузить", FileDialog.LOAD);
fd.setVisible(true);
}
});
saveAs.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
FileDialog fd = new FileDialog(new Frame(),
" Сохранить", FileDialog.SAVE);
fd.setVisible(true);
}
exit.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
System.exit(0);
}
});
Menu c = new Menu("Цвет");
MenuItem clear = new MenuItem("Очистить",
new MenuShortcut(KeyEvent.VK_D));
v.add(c); v.add(clear);
MenuItem red = new MenuItem("Красный");
MenuItem green = new MenuItem("Зеленый");
MenuItem blue = new MenuItem("Синий");
MenuItem black = new MenuItem("Черный");
c.add(red); c.add(green); c.add(blue); c.add(black);
red.addActionListener(scr);
green.addActionListener(scr);
blue.addActionListener(scr);
black.addActionListener(scr);
clear.addActionListener(scr);
addWindowListener(new WinClose()); pack();
setVisible(true);
}
class WinClose extends WindowAdapter{
public void windowClosing(WindowEvent e){
System.exit(0);
}
}
public static void main(String[] args){
new MenuScribble(" \"Рисовалка\" с меню");
}
}
```

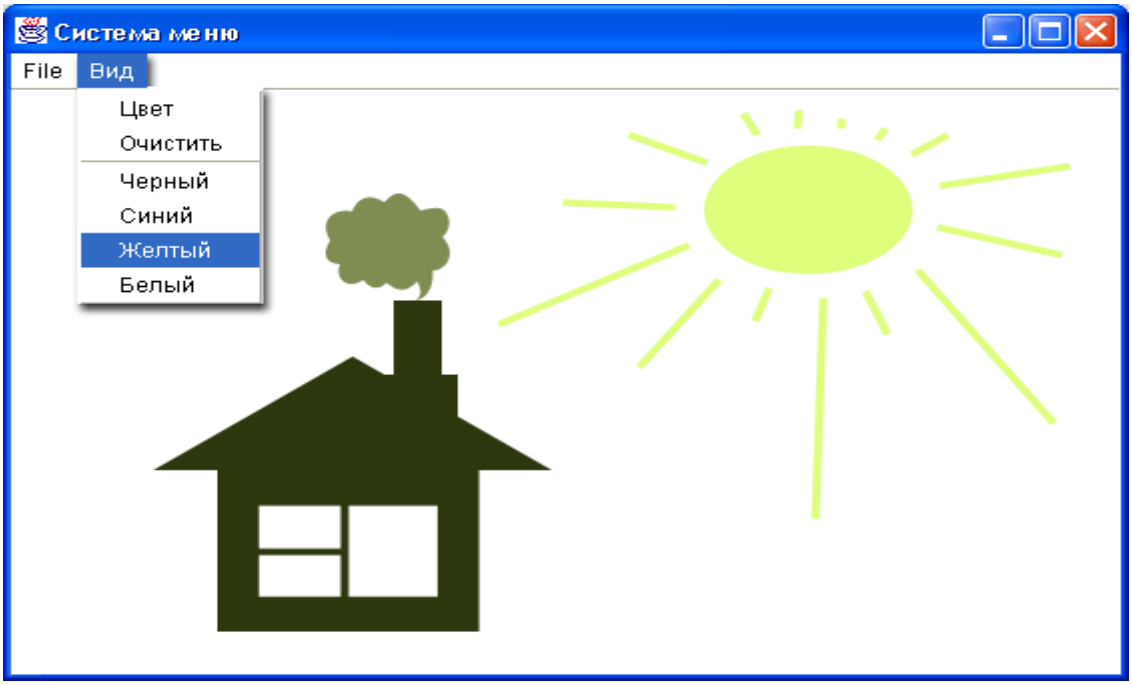

Рис. 13.2. Программа рисования с меню

Всплывающее меню

Всплывающее меню (popup menu) появляется обычно при нажатии или отпускании правой или средней кнопки мыши и является **контекстным** (context) меню. Его команды зависят от компонента, на котором была нажата кнопка мыши. В языке Java всплывающее меню – объект класса Pop и Pmenu. Этот класс расширяет класс Menu, следовательно, наследует все свойства меню и пункта меню MenuItem. Всплывающее меню присоединяется не к строке меню типа MenuBar или к меню типа Menu в качестве подменю, а к определенному компоненту. Для этого в классе component есть метод **add(PopupMenu menu)**.

У некоторых компонентов, например **TextField** и **TextArea**, уже существует всплывающее меню. Подобные меню нельзя переопределить.

Присоединить всплывающее меню можно только к одному компоненту. Если надо использовать всплывающее меню с несколькими компонентами в контейнере, то его присоединяют к контейнеру, а нужный компонент определяют с помощью метода**getcomponent ()** класса MouseEvent, как показано в листинге 13.2.

Кроме унаследованных свойств и методов, в классе **PopupMenu** есть метод **show (Component comp, int x, int y)**, показывающий всплывающее меню на экране так, что его левый верхний угол располагается в точке (x, y) в системе координат компонента comp. Чаще всего это компонент, на котором нажата кнопка мыши, возвращаемый методом**getcomponent ()**. Компонент comp должен быть внутри контейнера, к которому присоединено меню, иначе возникнет исключительная ситуация.

Всплывающее меню появляется в MS Windows при отпускании правой кнопки мыши, в Motif– при нажатии средней кнопки, а в других графических системах могут быть иные правила. Чтобы учесть эту разницу, в класс MouseEvent введен логический метод**isPopupTrigger ()**, показывающий, что возникшее событие мыши вызывает появление всплывающего меню. Его нужно вызывать при возникновении всякого события мыши, чтобы проверять, не является ли оно сигналом к появлению всплывающего меню, т. е. обращению к методу **show()**.

Было бы слишком неудобно включать такую проверку во все семь методов классов-слушателей событий мыши. Поэтому метод **IsPopupTrigger ()** лучше вызывать в методе**processMouseEvent()**.

Переделаем еще раз программу рисования из листинга 12.4, введя в класс scribble всплывающее меню для выбора цвета рисования и очистки окна и изменив обработку событий мыши. Для простоты уберем строку меню, хотя ее можно было оставить. Результат показан в листинге 13.2, а на рис. 13.3 – вид всплывающего меню в MS Windows.

Листинг 13.2. Программа рисования с всплывающим меню.

```
import java.awt.*;
import java.awt.event.*;
public class PopupMenuScribble extends Frame{
public PopupMenuScribble(String s){ super (s);
ScrollPane pane = new ScrollPane();
pane.setSize(300, 300);
add(pane, BorderLayout.CENTER);
Scribble scr = new Scribble(this, 500, 500);
pane.add(scr);
addWindowListener(new WinClose());
pack ();
setVisible(true);
}
class WinClose extends WindowAdapter{
public void windowClosing(WindowEvent e){
System.exit(0);
}
}
public static void main(String[] args){
new PopupMenuScribble(" \"Рисовалка\" с всплывающим меню");
}
}
```

```
class ScriBble extends Component implements ActionListener{
protected int lastX, lastY, w, h;
protected Color currColor = Color.black;
protected Frame f;
protected PopupMenu c;
public Scribble(Frame frame, int width, int height)!{
f = frame; w = width; h = height;
enableEvents(AWTEvent.MOUSE_EVENT_MASK |
AWTEvent.MOUSEJtoTIONJEVENT_MASK);
c = new PopupMenu ("Цвет");
add(c);
Menultera clear = new Menultem("Очистить",
new MenuShortcut(KeyEvent.VK_D));
Menultem red = new Menultem("Красный");
Menultem green = new Menultem("Зеленый");
Menultem blue = new Menultern("Синий");
Menultem black = new Menultem("Черный");
c.add(red); c.add(green); c.add(blue);
c.add(black); c.addSeparator(); c.add(clear);
red.addActionListener(this);
green.addActionListener(this);
blue.addActionListener(this);
black.addActionListener(this);
clear.addActionListener(this);
}
public Dimension getPreferredSize()
{
return new Dimension(w, h);
}
public void actionPerformed(ActionEvent event){
String s = event.getActionCommand();
if (s.equals("Очистить")) repaint();
else if (s.equals("Красный")) currColor = Color.red;
else if (s.equals("Зеленый")) currColor = Color.green;
else if (s.equals("Синий")) currColor = Color.blue;
else if (s.equals("Черный")) currColor = Color.black;
}
public void processMouseEvent(MouseEvent e){
if (e.isPopupTrigger())
c.show(e.getComponent (), e.getXO, e.getY());
else if (e.getlDO == MouseEvent.MOUSE_PRESSED){
lastX = e.getX(); lastY = e.getY(); }
else super.processMouseEvent(e); }
public void processMouseMotionEvent(MouseEvent e){
if (e.getlDO = MouseEvent.MOUSE_DRAGGED){
Graphics g = getGraphics();
g.setColor(currColor);
g.drawLinedastX, lastY, e.getX(), e.getY());
lastX = e.getX(); lastY = e.getY();
}
else super.processMouseMotionEvent(e);
}
}
```

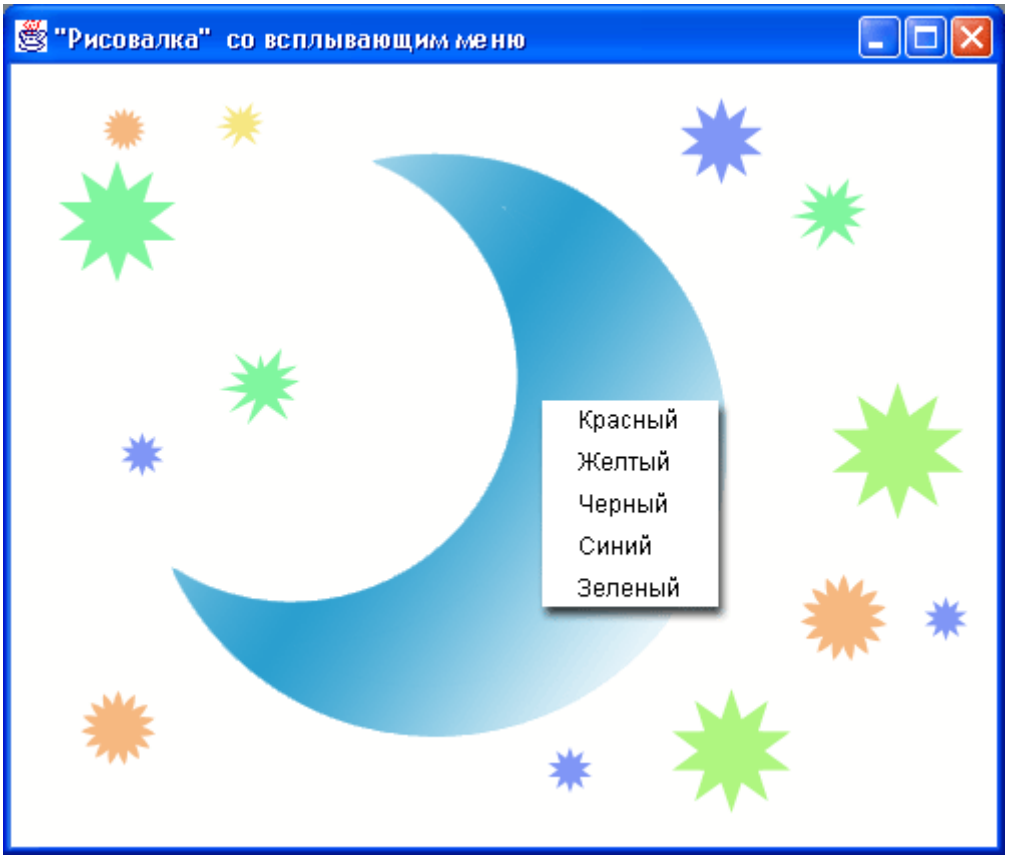

Рис. 13.3. Программа рисования с всплывающим меню