Апплеты

Апплеты — это маленькие приложения, которые размещаются на серверах Internet, транспортируются клиенту по сети, автоматически устанавливаются и запускаются на месте, как часть документа HTML. Когда апплет прибывает к клиенту, его доступ к ресурсам ограничен.

Ниже приведен исходный код канонической программы HelloWorld, оформленной в виде апплета:

```
import Java.AWT.*;
import Java.Applet.*;
public class HelloWorldApplet extends Applet {
   public void paint(Graphics g) {
      g.drawString("Hello World!", 20, 20);
   }
}
```

Этот апплет начинается двумя строками, которые импортируют все пакеты иерархий **Java.Applet** и **Java.AWT**. Дальше в нашем примере присутствует метод **paint**, замещающий одноименный метод класса **Applet**. При вызове этого метода ему передается аргумент, содержащий ссылку на объект класса **Graphics**. Последний используется для прорисовки нашего апплета. С помощью метода **drawString**, вызываемого с этим объектом типа **Graphics**, в позиции экрана (20,20) выводится строка “Hello World”.

Для того, чтобы с помощью браузера запустить этот апплет, нам придется написать несколько строк html-текста.

```
<Applet CODE="HelloWorldApplet" WIDTH=200 HEIGHT=40>
</Applet>
```

Вы можете поместить эти строки в отдельный html-файл (HelloWorldApplet.html), либо вставить их в текст этой программы в виде комментария и запустить программу **appletViewer** с его исходным текстом в качестве аргумента.

Тег HTML <Applet>

Тег <**Applet**> используется для запуска апплета как из HTML-документа, так и из программы **appletViewer**. Программа **appletViewer** выполняет каждый найденный ей тег <**Applet**> в отдельном окне, в то время как браузеры позволяют разместить на одной странице несколько апплетов. Синтаксис тэга <**APPLET**> в настоящее время таков :

```
<APPLET

CODE = appletFile
OBJECT = appletSerialFile
WIDTH = pixels
HEIGHT = pixels
[ARCHIVE = JARFiles]
[CODEBASE = codebaseURL]
[ALT = ALTernateText]
[NAME = appletInstanceName]
```

```
[ALIGN = alignment]
[VSPACE = pixels]
[HSPACE = pixels]
>
[< PARAM NAME = AttributeName1 VALUE = AttributeValue1 >]
[< PARAM NAME = AttributeName2 VALUE = AttributeValue2 >]
[HTML-текст, отображаемый при отсутствии поддержки Java]
</APPLET>
```

CODE = appletClassFile

CODE — *обязательный* атрибут, задающий имя файла, в котором содержится оттранслированный код апплета. Имя файла задается относительно **CODEBASE**, то есть либо от текущего каталога, либо от каталога, указанного в атрибуте **CODEBASE**. В **Java** 1.1 вместо этого атрибута может использоваться атрибут **OBJECT.**

OBJECT = appletClassSerialFile

Указывает имя файла, содержащего сериализованный апплет, из которого последний будет восстановлен. При запуске определяемого таким образом апплета должен вызываться не метод **init**(), а метод **start**(). Для апплета необходимо задать либо атрибут **CODE**, либо атрибут **OBJECT**, но задавать эти атрибуты одновременно нельзя.

WIDTH = pixels

HEIGHT = pixels

WIDTH и **HEIGHT** — *обязательные* атрибуты, задающие начальный размер видимой области апплета.

ARCHIVE = JARFiles

Задает список **JAR**-файлов (разделяется запятыми), которые предварительно загружаются в Web-броузер. В этих архивных файлах могут содержаться файлы классов, изображения, звуки и любые другие ресурсы, необходимые апплету. Для создания архивов используется утилита **JAR.**

c:\> **JAR** cf soundmap.jar *.class image.gif sound.wav

Очевидно, что передача сжатых **JAR**-файлов повышает эффективность работы. Поэтому многие средства разработки (Lotus JavaBeans, Borland JBuilder) уже имеют средства для публикации апплетов в виде **JAR**-файлов.

CODEBASE = codebaseURL

CODEBASE — *необязательный* атрибут, задающий базовый URL кода апплета, являющийся каталогом, в котором будет выполняться поиск исполняемого файла апплета (задаваемого в признаке **CODE**). Если этот атрибут не задан, по умолчанию используется каталог данного HTML-документа. **CODEBASE** не обязательно должен указывать на тот же узел, с которого был загружен HTML-документ.

ALT = ALTernateAppletText

Признак **ALT** — *необязательный* атрибут, задающий короткое текстовое сообщение, которое должно быть выведено в том случае, если используемый браузер распознает синтаксис тега <**Applet**>, но выполнять апплеты не умеет. Это не то же самое, что HTML-текст, который можно вставлять между <**Applet**> и </**Applet**> для браузеров, вообще не поддерживающих апплетов.

NAME = appletInstanceName

NAME — *необязательный* атрибут, используемый для задания имени для данного экземпляра апплета. Присвоение апплетам имен необходимо для того, чтобы другие апплеты на этой же странице могли находить их и общаться с ними. Для того, чтобы получить доступ к подклассу MyApplet класса **Applet** с именем “Duke”, нужно написать:

```
MyApplet a = getAppletContext().getApplet("Duke");
```

После того, как вы получили таким образом дескриптор именованного экземпляра апплета, вы можете вызывать его методы точно так же, как это делается с любым другим объектом.

ALIGN = alignment

ALIGN — *необязательный* атрибут, задающий стиль выравнивания апплета. Этот атрибут трактуется так же, как в теге IMG, возможные его значения — **LEFT**, **RIGHT**, TOP, TEXTTOP, MIDDLE, ABSMIDDLE, BASELINE, BOTTOM, ABSBOTTOM.

VSPACE = pixels

HSPACE = PIXELS

Эти *необязательные* атрибуты задают ширину свободного пространства в пикселях сверху и снизу апплета (VSPACE), и слева и справа от него (HSPACE). Они трактуются точно так же, как одноименные атрибуты тега IMG.

PARAM NAME = appletAttribute1 VALUE = value1

Этот тег дает возможность передавать из HTML-страницы апплету необходимые ему аргументы. Апплеты получают эти атрибуты, вызывая метод **getParameter**(), описываемый ниже.

Передача параметров

getParameter(String)

Метод **getParameter** возвращает значение типа **String**, соответствующее указанному имени параметра. Если вам в качестве параметра требуется значение какого-либо другого типа, вы должны преобразовать строку-параметр самостоятельно. Вы сейчас увидите некоторые примеры использования метода **getParameter** для извлечения параметров из приведенного ниже примера:

```
<Applet CODE=Testing WIDTH=40 HEIGHT=40>
<param name=fontName value=Univers>
<param name=fontSize value=14>
```

```
<param name=leading value=2>
<param name=accountEnabled value=true>
```

Ниже показано, как извлекается каждый из этих параметров:

```
String FontName = getParameter("fontName");
String FontSize = Integer.parseInt(getParameter("fontSize"));
String Leading = Float.valueOf(getParameter("leading"));
String PaidUp = Boolean.valueOf(getParameter("accountEnabled"));
```

Контекст апплета

getDocumentBase и getCodeBase

Возможно, Вы будете писать апплеты, которым понадобится явно загружать данные и текст. **Java** позволяет апплету загружать данные из каталога, в котором располагается HTML-документ, запустивший апплет (база документа - **getDocumentBase**), и из каталога, из которого был загружен class-файл с кодом апплета (база кода - **getCodeBase**).

AppletContext и showDocument

AppletContext представляет собой средства, позволяющие получать информацию об окружении работающего апплета. Метод **showDocument** приводит к тому, что заданный его параметром документ отображается в главном окне браузера или фрейме.

Отладочная печать

Отладочную печать можно выводить в два места: на консоль и в статусную строку программы просмотра апплетов. Для того, чтобы вывести сообщение на консоль, надо написать:

```
System.out.println("Hello there, welcome to Java");
```

Сообщения на консоли очень удобны, поскольку консоль обычно не видна пользователям апплета, и в ней достаточно места для нескольких сообщений. Обычно в браузерах консоль **Java** доступна из меню Options или «инструменты».

Метод **showStatus** выводит текст в статусной области программы **appletviewer** или браузера с поддержкой **Java**. В статусной области можно вывести только одну строку сообщения.

Порядок инициализации апплета

Ниже приведен порядок, в котором вызываются методы класса **Applet**, с пояснениями, нужно или нет переопределять данный метод.

init

Метод **init** вызывается первым. В нем вы должны инициализировать свои переменные.

start

Метод **start** вызывается сразу же после метода **init**. Он также используется в качестве стартовой точки для возобновления работы после того, как апплет был остановлен. В то время, как метод **init** вызывается только однажды — при загрузке апплета, **start** вызывается каждый раз при выводе HTML-документа, содержащего апплет, на экран. Так, например, если пользователь перейдет к новой WWW-странице, а затем вернется назад к странице с апплетом, апплет продолжит работу с метода **start.**

paint

Метод **paint** вызывается каждый раз при повреждении апплета. **AWT** следит за состоянием окон в системе и замечает такие случаи, как, например, перекрытие окна апплета другим окном. В таких случаях, после того, как апплет снова оказывается видимым, для восстановления его изображения вызывается метод **paint.**

update

Используемый по умолчанию метод **update** класса **Applet** сначала закрашивает апплет цветом фона по умолчанию, после чего вызывает метод **paint**. Если вы в методе **paint** заполняете фон другим цветом, пользователь будет видеть вспышку цвета по умолчанию при каждом вызове метода **update** — то есть, всякий раз, когда вы перерисовываете апплет. Чтобы избежать этого, нужно заместить метод **update**. В общем случае нужно выполнять операции рисования в методе **update**, а в методе **paint**, к которому будет обращаться **AWT**, просто вызвать **update.**

stop

Метод **stop** вызывается в тот момент, когда браузер покидает HTML-документ, содержащий апплет. При вызове метода **stop** апплет еще работает. Вы должны использовать этот метод для приостановки тех подпроцессов, работа которых необязательна при невидимом апплете. После того, как пользователь снова обратится к этой странице, вы должны будете возобновить их работу в методе **start.**

destroy

Метод **destroy** вызывается тогда, когда среда (например, браузер) решает, что апплет нужно полностью удалить из памяти. В этом методе нужно освободить все ресурсы, которые использовал апплет.

Перерисовка

Возвратимся к апплету HelloWorldApplet. В нем мы заместили метод **paint**, что позволило апплету выполнить отрисовку. В классе **Applet** предусмотрены дополнительные методы рисования, позволяющие эффективно закрашивать части экрана. При разработке первых апплетов порой непросто понять, почему метод **update** никогда не вызывается. Для инициации **update** предусмотрены три варианта метода **repaint.**

repaint

Метод **repaint** используется для принудительного перерисовывания апплета. Этот метод, в свою очередь, вызывает метод **update**. Однако, если ваша система медленная или

сильно загружена, метод **update** может и не вызваться. Близкие по времени запросы на перерисовку могут объединяться **AWT**, так что метод **update** может вызываться спорадически. Если вы хотите добиться ритмичной смены кадров изображения, воспользуйтесь методом **repaint**(time) — это позволит уменьшить количество кадров, нарисованных не вовремя.

repaint(time)

Вы можете вызывать метод **repaint**, устанавливая крайний срок для перерисовки (этот период задается в миллисекундах относительно времени вызова **repaint**).

repaint(x, y, w, h)

Эта версия ограничивает обновление экрана заданным прямоугольником, изменены будут только те части экрана, которые в нем находятся.

repaint(time, x, y, w, h)

Этот метод — комбинация двух предыдущих.

Задание размеров графических изображений.

Графические изображения вычерчиваются в стандартной для компьютерной графики системе координат, в которой координаты могут принимать только целые значения, а оси направлены слева направо и сверху вниз. У апплетов и изображений есть метод **size**, который возвращает объект **Dimension**. Получив объект **Dimension**, вы можете получить и значения его переменных **WIDTH** и **HEIGHT**:

```
Dimension d = size();
System.out.println(d. WIDTH + "," + d.HEIGHT);
```

Простые методы класса Graphics

У объектов класса **Graphics** есть несколько простых функций рисования. Каждую из фигур можно нарисовать заполненной, либо прорисовать только ее границы. Каждый из методов **drawRect**, **drawOval**, **fillRect** и **fillOval** вызывается с четырьмя параметрами: **int** x, **int** y, **int WIDTH** и **int HEIGHT**. Координаты x и y задают положение верхнего левого угла фигуры, параметры **WIDTH** и **HEIGHT** определяют ее границы.

drawLine

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

Этот метод вычерчивает отрезок прямой между точками с координатами (x1,y1) и (x2,y2). Эти линии представляют собой простые прямые толщиной в 1 пиксель. Поддержка разных перьев и разных толщин линий не предусмотрена.

drawArc и fillArc

Форма методов **drawArc** и **fillArc** следующая:

```
drawArc(int x, int y, int WIDTH, int HEIGHT, int startAngle,
        int sweepAngle)
```

Эти методы вычерчивают (**fillArc** заполняет) дугу, ограниченную прямоугольником (x,y,**WIDTH**, **HEIGHT**), начинающуюся с угла startAngle и имеющую угловой размер sweepAngle. Ноль градусов соответствует положению часовой стрелки на 3 часа, угол отсчитывается против часовой стрелки (например, 90 градусов соответствуют 12 часам, 180 — 9 часам, и так далее).

drawPolyson и fillPolyson

Прототипы для этих методов:

```
drawPolygon(int[], int[], int)
```

```
fillPolygon(int[], int[], int)
```

Метод **drawPolygon** рисует контур многоугольника (ломаную линию), задаваемого двумя массивами, содержащими x и y координаты вершин, третий параметр метода — число пар координат. Метод **drawPolygon** не замыкает автоматически вычерчиваемый контур. Для того, чтобы прямоугольник получился замкнутым, координаты первой и последней точек должны совпадать.

Цвет

Цветовая система **AWT** разрабатывалась так, чтобы была возможность работы со всеми цветами. После того, как цвет задан, **Java** отыскивает в диапазоне цветов дисплея тот, который ему больше всего соответствует. Вы можете запрашивать цвета в той семантике, к которой привыкли — как смесь красного, зеленого и голубого, либо как комбинацию оттенка, насыщенности и яркости. Вы можете использовать статические переменные класса **Color.black** для задания какого-либо из общеупотребительных цветов **- black, white, red, green, blue, cyan, yellow, magenta, orange, pink, gray, darkGray** и **lightGray**.

Для создания нового цвета используется один из трех описанных ниже конструкторов.

Color(int, int, int)

Параметрами для этого конструктора являются три целых числа в диапазоне от 0 до 255 для красного, зеленого и голубого компонентов цвета.

Color(int)

У этого конструктора — один целочисленный аргумент, в котором в упакованном виде заданы красный, зеленый и голубой компоненты цвета. Красный занимает биты 16-23, зеленый — 8-15, голубой — 0-7.

Color(float, float, float)

Последний из конструкторов цвета, **Color**(**float**, **float**, **float**), принимает в качестве параметров три значения типа **float** (в диапазоне от 0.0 до 1.0) для красного, зеленого и голубого базовых цветов.

Методы класса Color

HSBtoRGB(float, float, float)
RGBtoHSB(int, int, int, float[1)

HSBtoRGB преобразует цвет, заданный оттенком, насыщенностью и яркостью (**HSB**), в целое число в формате RGB, готовое для использования в качестве параметра конструктора **Color**(**int**). **RGBtoHSB** преобразует цвет, заданный тремя базовыми компонентами, в массив типа **float** со значениями **HSB**, соответствующими данному цвету.

Цветовая модель **HSB** (Hue-Saturation-Brightness, оттенок-насыщенность-яркость) является альтернативой модели Red-Green-Blue для задания цветов. В этой модели оттенки можно представить как круг с различными цветами (оттенок может принимать значения от 0.0 до 1.0, цвета на этом круге идут в том же порядке, что и в радуге — красный, оранжевый, желтый, зеленый, голубой, синий, фиолетовый). Насыщенность (значение в диапазоне от 0.0 до 1.0) - это шкала глубины цвета, от легкой пастели до сочных цветов. Яркость - это также число в диапазоне от 0.0 до 1.0, причем меньшие значения соответствуют более темным цветам, а большие - более ярким.

getRedQ, getGreenO, setBlue()

Каждый из этих методов возвращает в младших восьми битах результата значение соответствующего базового компонента цвета.

getRGB()

Этот метод возвращает целое число, в котором упакованы значения базовых компонентов цвета, причем

```
red = Oxff & (getRGB() >> 16);
green = Oxff & (getRGB() >> 8);
blue = Oxff & getRGB();
```

setPaintMode() и setXORMode(Color)

Режим отрисовки **paint** — используемый по умолчанию метод заполнения графических изображений, при котором цвет пикселей изменяется на заданный. **XOR** устанавливает режим рисования, когда результирующий цвет получается выполнением операции **XOR** (исключающее или) для текущего и указанного цветов (особенно полезно для анимации).

Шрифты

Библиотека **AWT** обеспечивает большую гибкость при работе со шрифтами благодаря предоставлению соответствующих абстракций и возможности динамического выбора шрифтов. Вот очень короткая программа, которая печатает на консоли **Java** имена всех имеющихся в системе шрифтов.

```
/*
* <Applet CODE="WhatFontsAreHere" WIDTH=100 HEIGHT=40>
* </Applet>
*
*/
import Java.Applet.*;
import Java.AWT.*;
public class WhatFontsAreHere extends Applet {
public void init() {
String FontList[];
FontList = getToolkit().getFontList();
for (int i=0; i < FontList.length; i++) {
System.out.println(i + ": " + FontList[i]);
}
} }
```

drawString

В предыдущих примерах использовался метод **drawString**(**String**, x, y). Этот метод выводит строку с использованием текущих шрифта и цвета. Точка с координатами (x,y) соответствует левой границе базовой линии символов, а не левому верхнему углу, как это принято в других методах рисования. Для того, чтобы понять, как при этом располагается описывающий строку прямоугольник, прочтите раздел о метрике шрифта в конце этой главы.

Использование шрифтов

Конструктор класса **Font** создает новый шрифт с указанным именем, стилем и размером в пунктах:

```
Font StrongFont = new Font("Helvetica", Font.BOLD|Font.ITALIC,
                                        24);
```

В настоящее время доступны следующие имена шрифтов: **Dialog**, **Helvetica**, **TimesRoman**, **Courier** и **Symbol**. Для указания стиля шрифта внутри данного семейства предусмотрены три статические переменные. — **Font**.**PLAIN**, **Font**.**BOLD** и **Font**.**ITALIC**, что соответствует обычному стилю, курсиву и полужирному.

Теперь давайте посмотрим на несколько дополнительных методов.

getFamily и **getName**
Метод **getFamily** возвращает строку с именем семейства шрифтов. С помощью метода **getName** можно получить логическое имя шрифта.

getSize
Этот метод возвращает целое число, представляющее собой размер шрифта в пунктах.

getStyle
Этот метод возвращает целое число, соответствующее стилю шрифта. Полученный результат можно побитово сравнить со статическими переменными класса **Font**: — **PLAIN**, **BOLD** и **ITALIC**.

isBold, **isItalic**, **isPlain**

Эти методы возвращают **true** в том случае, если стиль шрифта — полужирный (**BOLD**), курсив (**ITALIC**) или обычный (**PLAIN**), соответственно.

Позиционирование и шрифты: FontMetrics

В **Java** используются различные шрифты, а класс **FontMetrics** позволяет программисту точно задавать положение выводимого в апплете текста. Прежде всего нам нужно понять кое-что из обычной терминологии, употребляемой при работе со шрифтами:

- *Высота* (HEIGHT) — размер от верхней до нижней точки самого высокого символа в шрифте.
- *Базовая линия* (baseline) — линия, по которой выравниваются нижние границы символов (не считая снижения (descent)).
- *Подъем* (ascent) — расстояние от базовой линии до верхней точки символа.
- *Снижение* (descent) — расстояние от базовой линии до нижней точки символа.

Использование FontMetrics

Ниже приведены некоторые методы класса **FontMetrics**:

stringWidth
Этот метод возвращает длину заданной строки для данного шрифта.

bytesWidth, **charsWidth**

Эти методы возвращают ширину указанного массива байтов для текущего шрифта.

getAscent, **getDescent**, **getHeight**

Эти методы возвращают подъем, снижение и ширину шрифта. Сумма подъема и снижения дают полную высоту шрифта. Высота шрифта — это не просто расстояние от самой нижней точки букв g и y до самой верхней точки заглавной буквы T и символов вроде скобок. Высота включает подчеркивания и т.п.

getMaxAscent и **getMaxDescent**

Эти методы служат для получения максимальных подъема и снижения всех символов в шрифте.

Центрирование текста

Давайте теперь воспользуемся методами объекта **FontMetrics** для получения подъема, снижения и длины строки, которую требуется нарисовать, и с помощью полученных значений отцентрируем ее в нашем апплете.

```
/*
* <Applet CODE="HelloWorld" WIDTH=200 HEIGHT=100>
* </Applet>
*
```

```
*/
import Java.Applet.*;
import Java.AWT.*;
public class HelloWorld extends Applet {
final Font f = new Font("Helvetica", Font.BOLD, 18);
public void paint(Graphics g) {
Dimension d = this.size();
g.setColor(Color.white);
g.fillRect(0,0,d.WIDTH,d.HEIGHT);
g.setColor(Color.black);
g.setFont(f);
drawCenteredString("Hello World!", d.WIDTH, d.HEIGHT, g);
g.drawRect(0,0,d.WIDTH-1,d.HEIGHT-1);
}
public void drawCenteredString(String s, int w, int h, Graphics
g) {
FontMetrics fm = g.getFontMetrics();
int x = (w - fm.stringWidth(s)) / 2;
int y = (fm.getAscent() + (h - (fm.getAscent() +
fm.getDescent()))/2);
g.drawString(s, x, y);
} }
```

Набор абстракций для работы с окнами

Трудность при создании независимой от платформы библиотеки заключается в том, что ее разработчикам либо приходится требовать, чтобы все приложения на всех платформах вели себя и выглядели одинаково, либо для поддержки, скажем, трех различных разновидностей интерфейса приходится писать в три раза больше кода. Существуют два взгляда на эту проблему. Один подход заключается в том, что упор делается на графику низкого уровня — рисование пикселей, при этом разработчики библиотеки сами заботятся о внешнем виде каждого компонента. При другом подходе создаются абстракции, подходящие для библиотек каждой из операционных систем, и именно “родные” пакеты данной операционной системы служат подъемной силой для архитектурно-нейтральной библиотеки на каждой из платформ. В Java при создании библиотеки Abstraction **Window** Toolkit (**AWT**) выбран второй подход.

Классы **Graphics** и **Fonts** мы уже обсуждали выше. Далее будут обсуждаться различные возможности работы с изображениями. Здесь мы пройдемся по базовой архитектуре **AWT**, касающейся интерфейсных объектов.

Компоненты

Component — это абстрактный класс, который инкапсулирует все атрибуты визуального интерфейса - обработка ввода с клавиатуры, управление фокусом, взаимодействие с мышью, уведомление о входе/выходе из окна, изменения размеров и положения окон, прорисовка своего собственного графического представления, сохранение текущего текстового шрифта, цветов фона и переднего плана (более 10 методов). Перейдем к некоторым конкретным подклассам класса **Component**.

Container

Container — это абстрактный подкласс класса **Component**, определяющий дополнительные методы, которые дают возможность помещать в него другие компоненты, что дает возможность построения иерархической системы визуальных объектов. **Container** отвечает за расположение содержащихся в нем компонентов с помощью интерфейса **LayoutManager**.

Panel

Класс **Panel** — это очень простая специализация класса **Container**. В отличие от последнего, он не является абстрактным классом. Поэтому о **Panel** можно думать, как о допускающем рекурсивную вложенность экранном компоненте. С помощью метода **add** в объекты **Panel** можно добавлять другие компоненты. После того, как в него добавлены какие-либо компоненты, можно вручную задавать их положение и изменять размер с помощью методов **move, resize** и **reshape** класса **Component.**

Ранее мы уже использовали один из подклассов **Panel** — **Applet**. Каждый раз, когда мы создавали **Applet**, методы **paint** и **update** рисовали его изображение на *поверхности* объекта **Panel**. Прежде, чем мы углубимся в методы **Panel**, давайте познакомимся с компонентом **Canvas**, который можно вставлять в пустую **Panel** при работе с объектом **Applet.**

Canvas

Основная идея использования объектов **Canvas** в том, что они являются семантически свободными компонентами. Вы можете придать объекту **Canvas** любое поведение и любой желаемый внешний вид. Его имя подразумевает, что этот класс является пустым холстом, на котором вы можете “нарисовать” любой компонент — такой, каким вы его себе представляете.

Произведем от **Canvas** подкласс GrayCanvas, который будет просто закрашивать себя серым цветом определенной насыщенности. Наш апплет будет создавать несколько таких объектов, каждый со своей интенсивностью серого цвета.

```
/* <Applet code = “PanelDemo”

width=300

height=300>

</Applet>

*/

import java.AWT.*;

import java.Applet.*;

class GrayCanvas extends Canvas {

Color gray;
```

```
public GrayCanvas(float g) {

gray = new Color(g, g, g);

}

public void paint(Graphics g) {

Dimension size = size();

g.setColor(gray);

g.fillRect(0, 0, size.width, size.height);

g.setColor(Color.black);

g.drawRect(0, 0, size.width-1, size.height-1);

} }

public class PanelDemo extends Applet {

static final int n = 4;

public void init() {

setLayout(null);

int width = Integer.parseInt(getParameter("width"));

int height = Integer.parseInt(getParameter("height"));

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

float g = (i * n + j) / (float) (n * n);

Canvas c = new GrayCanvas(g);

add(c);

c.resize(width / n, height / n);

c.move(i * width / n, j * height / n);

}

}

} }
```

Мы устанавливаем размер каждого из объектов **Canvas** на основе значения, полученного с помощью метода size, который возвращает объект класса **Dimension**. Обратите внимание на то, что для размещения объектов **Canvas** в нужные места используются методы **resize** и **move**. Такой способ станет очень утомительным, когда мы перейдем к более сложным компонентам и более интересным вариантам расположения. А

пока в нашем апплете для выключения упомянутого механизма использован вызов метода **setLayout**(**null**).

Label

Функциональность класса **Label** сводится к тому, что он знает, как нарисовать объект **String** — текстовую строку, выровняв ее нужным образом. Шрифт и цвет, которыми отрисовывается строка метки, являются частью базового определения класса **Component**. Для работы с этими атрибутами предусмотрены пары методов **getFont**/**setFont** и **getForeground**/**setForeground**. Задать или изменить текст строки после создания объекта с помощью метода **setText**. Для задания режимов выравнивания в классе **Label** определены три константы — **LEFT**, **RIGHT** и **CENTER**. Ниже приведен пример, в котором создаются три метки, каждая — со своим режимом выравнивания.

```
/* <Applet code = "LabelDemo" width=100 height=100>

</Applet>

*/

import java.AWT.*;

import java.Applet.*;

public class LabelDemo extends Applet {

public void init() {

setLayout(null);

int width = Integer.parseInt(getParameter("width"));

int height = Integer.parseInt(getParameter("height"));

Label LEFT = new Label("Left", Label.LEFT);

Label RIGHT = new Label("RIGHT", Label.RIGHT);

Label CENTER = new Label("CENTER", Label.CENTER);

add(left);

add(RIGHT);

add(CENTER);

left.reshape(0, 0, width, height / 3);

RIGHT.reshape(0, height / 3, width, height / 3);

CENTER.reshape(0, 2 * height / 3, width, height / 3);

} }
```

На этот раз, чтобы одновременно переместить и изменить размер объектов **Label**, мы использовали метод **reshape**. Ширина каждой из меток равна полной ширине апплета, высота — 1/3 высоты апплета.

Button

Объекты-кнопки помечаются строками, причем эти строки нельзя выравнивать подобно строкам объектов **Label** (они всегда центрируются внутри кнопки). Позднее в данной главе речь пойдет о том, как нужно обрабатывать события, возникающие при нажатии и отпускании пользователем кнопки. Ниже приведен пример, в котором создаются три расположенные по вертикали кнопки.

```
/* <Applet code = "ButtonDemo" width=100 height=100>

</Applet>

*/

import java.AWT.*;

import java.Applet.*;

public class ButtonDemo extends Applet {

public void init() {

setLayout(null);

int width = Integer.parseInt(getParameter("width"));

int height = Integer.parseInt(getParameter("height"));

Button yes = new Button("Yes");

Button no = new Button("No");

Button maybe = new Button("Undecided");

add(yes);

add(no);

add(maybe);

yes.reshape(0, 0, width, height / 3);

no.reshape(0, height / 3, width, height / 3);

maybe.reshape(0, 2 * height / 3, width, height / 3);

} }
```

Checkbox

Класс **Checkbox** часто используется для выбора одной из двух возможностей. При создании объекта **Checkbox** ему передается текст метки и логическое значение, чтобы задать исходное состояние окошка с отметкой. Программно можно получать и устанавливать состояние окошка с отметкой с помощью методов **getState** и **setState**. Ниже приведен пример с тремя объектами **Checkbox**, задаваемое в этом примере исходное состояние соответствует отметке в первом объекте.

```
/* <Applet code = "CheckBoxDemo" width=120 height=100>

</Applet>

*/

import java.AWT.*;

import java.Applet.*;

public class CheckboxDemo extends Applet {

public void init() {

setLayout(null);

int width = Integer.parseInt(getParameter("width"));

int height = Integer.parseInt(getParameter("height"));

Checkbox win95 = new Checkbox("Windows 95/98", null, true);

Checkbox Solaris = new Checkbox("Solaris 2.5");

Checkbox mac = new Checkbox("MacOS 7.5");

add(win95);

add(solaris);

add(mac);

win95.reshape(0, 0, width, height / 3);

Solaris.reshape(0, height / 3, width, height / 3);

mac.reshape(0, 2 * height / 3, width, height / 3);

} }
```

CheckboxGroup

Второй параметр конструктора **Checkbox** (в предыдущем примере мы ставили там **null**) используется для группирования нескольких объектов **Checkbox**. Для этого сначала создается объект **CheckboxGroup**, затем он передается в качестве параметра любому количеству конструкторов **Checkbox**, при этом предоставляемые этой группой варианты выбора становятся взаимоисключающими (только один может быть задействован). Предусмотрены и методы, которые позволяют получить и установить группу, к которой принадлежит конкретный объект **Checkbox** — **getCheckboxGroup** и **setCheckboxGroup**. Вы можете пользоваться методами **getCurrent** и **setCurrent** для получения и установки состояния выбранного в данный момент объекта **Checkbox**. Ниже приведен пример, отличающийся от предыдущего тем, что теперь различные варианты выбора в нем взаимно исключают друг друга.

```
/* <Applet code = "CheckboxGroupDemo" width=120 height=100>

</Applet>

*/

import java.AWT.*;

import java.Applet.*;

public class CheckboxGroupDemo extends Applet {

public void init() {

setLayout(null);

int width = Integer.parseInt(getParameter("width"));

int height = Integer.parseInt(getParameter("height"));

CheckboxGroup g = new CheckboxGroup();

Checkbox win95 = new Checkbox("Windows 95/98", g, true);

Checkbox solaris = new Checkbox("Solaris 2.5", g, false);

Checkbox mac = new Checkbox("MacOS 7.5", g, false);

add(win95);

add(solaris);

add(mac);

win95.reshape(0, 0, width, height / 3);

solaris. reshape(0, height / 3, width, height / 3);
```

```
mac.reshape(0, 2 * height / 3, width, height / 3);

} }
```

Обратите внимание — окошки изменили свою форму, теперь они не квадратные, а круглые.

Choice

Класс **Choice** (выбор) используется при создании раскрывающихся списочных меню (выпадающих списков типа **ComboBox** в **Windows**). Компонент **Choice** занимает ровно столько места, сколько требуется для отображения выбранного в данный момент элемента, когда пользователь щелкает мышью на нем, раскрывается меню со всеми элементами, в котором можно сделать выбор. Каждый элемент меню — это строка, которая выводится, выровненная по левой границе. Элементы меню выводятся в том порядке, в котором они были добавлены в объект **Choice**. Метод **countItems** возвращает количество пунктов в меню выбора. Вы можете задать пункт, который выбран в данный момент, с помощью метода **select**, передав ему либо целый индекс (пункты меню перечисляются с нуля), либо строку, которая совпадает с меткой нужного пункта меню. Аналогично, с помощью методов **getSelectedItem** и **getSelectedIndex** можно получить, соответственно, строку-метку и индекс выбранного в данный момент пункта меню. Вот очередной простой пример, в котором создается два объекта **Choice**.

```
/* <Applet code = "ChoiceDemo" width=200 height=100>

</Applet>

*/

import java.AWT.*;

import java.Applet.*;

public class ChoiceDemo extends Applet {

public void init() {

setLayout(null);

int width = Integer.parseInt(getParameter("width"));

int height = Integer.parseInt(getParameter("height"));

Choice os = new Choice();

Choice browser = new Choice();

os.addItem("Windows 95/98");

os.addItem("Solaris 2.5");

os.addItem("MacOS 7.5");
```

```
browser.addItem("Netscape Navigator 3.0");

browser.addItem("Netscape Communicator 4.5");

browser.addItem("Internet Explorer 3.0");

browser.addItem("Mosaic 3.0");

browser.addItem("Lynx 2.4");

browser.select("Netscape Communicator 4.5");

add(os);

add(browser);

os.reshape(0, 0, width, height / 2);

browser.reshape(0, height / 2, width, height / 2);

} }
```

List

Класс **List** представляет собой компактный список с возможностью выбора нескольких вариантов и с прокруткой (аналог **ListBox** в **Windows**). Ниже приведен пример с двумя списками выбора, один из которых допускает выбор нескольких элементов, а второй — выбор единственного элемента.

```
/* <Applet code = "ListDemo" width=200 height=100>

</Applet>

*/

import java.AWT.*;

import java.Applet.*;

public class ListDemo extends Applet {

public void init() { setLayout(null);

int width = Integer.parseInt(getParameter("width"));

int height = Integer.parseInt(getParameter("height"));

List os = new List(0, true);

List browser = new List(0, false);
```

```
os.addItem("Windows 95/98");

os.addItem("Solaris 2.5");

os.addItem("MacOS 7.5");

browser.addItem("Netscape Navigator 3.0");

browser.addItem("Netscape Communicator 4.5");

browser.addItem("lnternet Explorer 4.0");

browser.addItem("Mosaic 3.0");

browser.addItem("Lynx 2.4");

browser.select(1);

add(os);

add(browser);

os.reshape(0, 0, width, height / 2);

browser.reshape(0, height / 2, width, height / 2);

} }
```

Заметьте, что у нижнего списка имеется линейка прокрутки, поскольку все его элементы не уместились в заданный нами размер.

Scrollbar

Объекты **Scrollbar** (линейки прокрутки) используются для выбора подмножества значений между заданными минимумом и максимумом. Визуально у линейки прокрутки есть несколько органов управления, ориентированных либо вертикально, либо горизонтально. Стрелки на каждом из ее концов показывают, что, нажав на них, вы можете продвинуться на один шаг в соответствующем направлении. Текущее положение отображается с помощью движка линейки прокрутки, которым пользователь также может управлять, устанавливая требуемое положение линейки.

Конструктор класса **Scrollbar** позволяет задавать ориентацию линейки прокрутки — для этого предусмотрены константы **VERTICAL** и **HORIZONTAL**. Кроме того с помощью конструктора можно задать начальное положение и размер движка, а так же минимальное и максимальное значения, в пределах которых линейка прокрутки может изменять параметр. Для получения и установки текущего состояния линейки прокрутки используются методы **getValue** и **setValue**. Кроме того воспользовавшись методами **getMinimum** и **getMaximum**, вы можете получить рабочий диапазон объекта. Ниже приведен пример, в котором создается и вертикальная, и горизонтальная линейки прокрутки.

```
/* <Applet code = "ScrollbarDemo" width=200 height=100>
```

```
</Applet>

*/

import java.AWT.*;

import java.Applet.*;

public class ScrollbarDemo extends Applet {

public void init() {

setLayout(null);

int width = Integer.parseInt(getParameter("width"));

int height = Integer. parseInt(getParameter("height"));

Scrollbar hs = new Scrollbar(Scrollbar.HORIZONTAL, 50, width /
10, 0, 100);

Scrollbar vs = new Scrollbar(Scrollbar.VERTICAL, 50, height / 2,
0, 100);

add(hs);

add(vs);

int thickness = 16;

hs.reshape(0, height - thickness, width - thickness, thickness);

vs.reshape(width - thickness, 0, thickness, height - thickness);

} }
```

В этом примере скроллируется, конечно, пустая область.

TextField

Класс **TextField** представляет собой реализацию однострочной области для ввода текста. Такие области часто используются в формах для пользовательского ввода. Вы можете “заморозить” содержимое объекта **TextField** с помощью метода **setEditable**, а метод **isEditable** сообщит вам, можно ли редактировать текст в данном объекте. Текущее значение объекта можно получить методом **getText** и установить методом **setText**. С помощью метода **select** можно выбрать фрагмент строки, задавая его начало и конец, отсчитываемые с нуля. Для выбора всей строки используется метод **selectAll**.

Метод **setEchoChar** задает символ, который будет выводиться вместо любых вводимых символов. Вы можете проверить, находится ли объект **TextField** в этом режиме, с помощью метода **echoCharIsSet**, и узнать, какой именно символ задан для эхо-печати, с

помощью метода **getEchoChar**. Вот пример, в котором создаются классические поля для имени пользователя и пароля.

```
/* <Applet code = "TextFieldDemo" width=200 height=100>

</Applet>

*/

import java.AWT.*;

import java.Applet.*;

public class TextFieldDemo extends Applet {

public void init() {

setLayout(null);

int width = Integer.parseInt(getParameter("width"));

int height = Integer.parseInt(getParameter("height"));

Label namep = new Label("Name : ", Label.RIGHT);

Label passp = new Label("Password : ", Label.RIGHT);

TextField name = new Text Field(8);

TextField pass = new TextField(8);

pass.setEchoChar('*');

add(namep);

add(name);

add(passp);

add(pass);

int space = 25;

int w1 = width / 3;

namep.setBounds(0, (height - space) / 2, w1, space);

name.setBounds(w1, (height - space) / 2, w1, space);

passp.setBounds(0, (height + space) / 2, w1, space);

pass.setBounds(w1, (height + space) / 2, w1, space);
```

```
} }
```

Обратите внимание, что в этом примере мы заменили устаревший в JDK 1.1 **reshape** на **setBounds**.

TextArea

Порой одной строки текста оказывается недостаточно для конкретной задачи. **AWT** включает в себя очень простой многострочный редактор обычного текста, называемый **TextArea**. Конструктор класса **TextArea** воспринимает значение типа **String** в качестве начального текста объекта. Кроме того, в конструкторе указывается число колонок и строк текста, которые нужно выводить. Есть три метода, которые позволяют программе модифицировать содержимое объекта **TextArea**: appendText добавляет параметр типа **String** в конец буфера; **insertText** вставляет строку в заданное отсчитываемым от нуля индексом место в буфере; **replaceText** копирует строку-параметр в буфер, замещая ею текст, хранящийся в буфере между первым и вторым параметрами-смещениями. Ниже приведена программа, создающая объект **TextArea** и вставляющая в него строку.

```
/* <Applet code = "TextAreaDemo" width=200 height=100>

</Applet>

*/

import java.AWT.*;

import java.Applet.*;

public class TextAreaDemo extends Applet {

public void init() {

setLayout(null);

int width = Integer.parseInt(getParameter("width"));

int height = Integer.parseInt(getParameter("height"));

String val = "There are two ways of constructing " +

"a software design.\n" +

"One way is to make it so simple\n" +

"that there are obviously no deficiencies.\n" +

"And the other way is to make it so complicated\n" +

"that there are no obvious deficiencies.\n\n" +

"C.A.R. Hoare\n\n" +
```

```
"There's an old story about the person who wished\n" +

"his computer were as easy to use as his telephone. \n" +

"That wish has come true,\n" +

"since I no longer know how to use my telephone. \n\n" +

"Bjarne Stroustrup, AT&T (inventor of C++)";

TextArea text = new TextArea(val, 80, 40);

add(text);

text.setBounds(0, 0, width, height);

}}
```

Прочитайте (можете перевести на русский язык) этот текст с юмором.

Layout

Все компоненты, с которыми мы работали до сих пор в этой главе, размещались “вручную”. И в каждом примере мы вызывали загадочный метод **setLayout**(**null**). Этот вызов запрещал использование предусмотренного по умолчанию механизма управления размещением компонентов. Для решения подобных задач в **AWT** предусмотрены диспетчеры размещения (**LayoutManagers**).

LayoutManager.

Каждый класс, реализующий интерфейс **LayoutManager**, следит за списком компонентов, которые хранятся с именами типа **String**. Всякий раз, когда вы добавляете компонент в **Panel**, диспетчер размещения уведомляется об этом. Если требуется изменить размер объекта **Panel**, то идет обращение к диспетчеру посредством методов **minimumLayoutSize** и **preferredLayoutSize**. В каждом компоненте, который приходится обрабатывать диспетчеру, должны присутствовать реализации методов **preferredSize** и **minimumSize**. Эти методы должны возвращать предпочтительный и минимальный размеры для прорисовки компонента, соответственно. Диспетчер размещения по возможности будет пытаться удовлетворить эти запросы, в то же время заботясь о целостности всей картины взаимного расположения компонентов.

В Java есть несколько предопределенных классов — диспетчеров размещения, описываемых ниже.

Диспетчеры размещения

FlowLayout

Класс **FlowLayout** реализует простой стиль размещения, при котором компоненты располагаются, начиная с левого верхнего угла, слева направо и сверху вниз. Если в данную строку не помещается очередной компонент, он располагается в левой позиции новой строки. Справа, слева, сверху и снизу компоненты отделяются друг от друга

небольшими промежутками. Ширину этого промежутка можно задать в конструкторе **FlowLayout**. Каждая строка с компонентами выравнивается по левому или правому краю, либо центрируется в зависимости от того, какая из констант **LEFT**, **RIGHT** или **CENTER** была передана конструктору. Режим выравнивания по умолчанию — **CENTER**, используемая по умолчанию ширина промежутка — 5 пикселей.

Ниже приведен пример, в котором в **Panel** включается несколько компонентов **Label**. Объект **Panel** использует **FlowLayout** с выравниванием **RIGHT**.

```
/* <Applet code = "FlowLayoutDemo" width=200 height=100>

</Applet>

*/

import java.AWT.*;

import java.Applet.*;

import java.util.*;

public class FlowLayoutDemo extends Applet {

public void init() {

setLayout(new FlowLayout(FlowLayout.RIGHT, 10, 3));

int width = Integer.parseInt(getParameter("width"));

int height = Integer.parseInt(getParameter("height"));

String val = "Data is not information " +

"is not knowledge is not wisdom.";

StringTokenizer st = new StringTokenizer(val);

while (st.hasMoreTokens()) {

add(new Button(st.nextToken()));

}

} }
```

Необходимо вызвать пример для двух различных размеров – FlowLayoutDemo1.html, FlowLayoutDemo2.html для того, чтобы проиллюстрировать, как объекты **Label** перетекают из строки в строку, и при этом строки выравниваются по правому краю (или Вы можете изменять размеры окна **AppletViewer**).

BorderLayout

Класс **BorderLayout** реализует обычный стиль размещения для окон верхнего уровня, в котором предусмотрено четыре узких компонента фиксированной ширины по краям, и одна большая область в центре, которая может расширяться и сужаться в двух направлениях, занимая все свободное пространство окна. У каждой из этих областей есть строки-имена: **String.North**, **String.South**, **String.East** и **String.West** соответствуют четырем краям, а **CENTER** — центральной области. Ниже приведен пример **BorderLayout** с компонентом в каждой из названных областей.

```
/* <Applet code = "BorderLayoutDemo" width=300 height=200>

</Applet>

*/

import java.AWT.*;

import java.Applet.*;

import java.util.*;

public class BorderLayoutDemo extends Applet {

public void init() {

setLayout(new BorderLayout());

int width = Integer.parseInt(getParameter("width"));

int height = Integer.parseInt(getParameter("height"));

add("North", new Button("This is across the top"));

add("South", new Label("The footer message might go here"));

add("East", new Button("Left"));

add("West", new Button("RIGHT"));

String msg = "The reasonable man adapts " +

"himself to the world;\n" +

"the unreasonable one persists in " +

"trying to adapt the world to himself.\n" +

"Therefore all progress depends " +

"on the unreasonable rnan.\n\n" +

"George Bernard Shaw\n\n";
```

```
add("CENTER", new TextArea(msg));

} }
```

Опять читаем фразу со смыслом (спасибо Бернарду Шоу) – BorderLayoutDemo.html.

GridLayout

Класс **GridLayout** размещает компоненты в простой равномерной сетке. Конструктор этого класса позволяет задавать количество строк и столбцов. Ниже приведен пример, в котором **GridLayout** используется для создания сетки 4x4, 15 квадратов из 16 заполняются кнопками, помеченными соответствующими индексами. Как вы уже, наверное, поняли, это — панель для игры в “пятнашки”.

```
/* <Applet code = “GridLayoutDemo” width=200 height=200>

</Applet>

*/

import java.AWT.*;

import java.Applet.*;

public class GridLayoutDemo extends Applet {

static final int n = 4;

public void init() {

setLayout(new GridLayout(n, n));

setFont(new Font("Helvetica", Font.BOLD, 24));

int width = Integer.parseInt(getParameter("width"));

int height = Integer.parseInt(getParameter("height”));

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

int k = i * n + j;

if (k > 0)

add(new Button("" + k));

}

}
```

```
} }
```

Если доработать этот пример – получится неплохая игра – GridLayoutDemo.html.

Insets

Класс **Insets** используется для того, чтобы вставлять в объект **Panel** границы, напоминающие горизонтальные и вертикальные промежутки между объектами, которые делает диспетчер размещения. Для того, чтобы добиться вставки границ в объект **Panel**, нужно заместить метод **Insets** реализацией, возвращающей новый объект **Insets** с четырьмя целыми значениями, соответствующими ширине верхнего, нижнего, левого и правого краев.

```
public Insets Insets() {

return new Insets(10, 10, 10, 10);

}
```

CardLayout

Класс **CardLayout** по своему уникален. Он отличается от других программ управления размещением компонентов тем, что представляет несколько различных вариантов размещения, которые можно сравнить с колодой карт. Колоду можно тасовать так, чтобы в данный момент времени наверху была только одна из карт. Это может быть полезно при создании интерфейсов пользователя, в которых есть необязательные компоненты, включаемые и выключаемые динамически в зависимости от реакции пользователя.

Window

Класс **Window** во многом напоминает **Panel** за тем исключением, что он создает свое собственное окно верхнего уровня. Большая часть программистов скорее всего будет использовать не непосредственно класс **Window**, а его подкласс **Frame**.

Frame

Frame — это как раз то, что обычно и считают окном на рабочей поверхности экрана. У объекта **Frame** есть строка с заголовком, управляющие элементы для изменения размера и линейка меню. Для того, чтобы вывести/спрятать изображение объекта **Frame**, нужно использовать методы **show** и **hide**. Ниже приведен пример апплета, который показывает объект **Frame** с содержащимся в нем компонентом **TextArea**.

```
/* <Applet code = "FrameDemo" width=200 height=200>

</Applet>

*/

import java.AWT.*;

import java.Applet.*;
```

```
public class FrameDemo extends Applet {

public void init() {

int width = Integer.parseInt(getParameter("width"));

int height = Integer.parseInt(getParameter("height"));

String val = "There are two ways of constructing " +

"a software design.\n" +

"One way is to make it so simple\n" +

"that there are obviously no deficiencies.\n" +

"And the other way is to make it so complicated" +

"that there are no obvious deficiencies.\n\n" +

“C.A.R. Hoare\n\n";

TextArea text = new TextArea(val, 80, 40);

Frame f = new Frame("Demo Frame");

f.setSize(width, height);

f.add("CENTER", text);

f.show();

} }
```

Уже знакомая фраза во фрейме – FrameDemo.html.

Меню

С каждым окном верхнего уровня может быть связана линейка меню. Объект MenuBar может включать в себя несколько объектов **Menu**. Последние, в свою очередь, содержат в себе список вариантов выбора — объектов **MenuItem**. **Menu** — подкласс **MenuItem**, так что объекты **Menu** также могут включаться в этот список, что позволяет создавать иерархически вложенные подменю. Вот пример, в котором к окну добавлены несколько вложенных меню.

```
/* <Applet code = “MenuDemo” width=200 height=200>

</Applet>

*/

import java.AWT.*;
```

```
import java.Applet. *;
public class MenuDemo extends Applet {
public void init() {
int width = Integer.parseInt(getParameter("width"));
int height = Integer.parseInt(getParameter("height"));
Frame f = new Frame("Demo Frame");
f.setSize(width, height);
MenuBar mbar = new MenuBar();
f.setMenuBar(mbar);
Menu file = new Menu("File");
file.add(new MenuItem("New... "));
file.add(new MenuItem("0pen..."));
file.add(new MenuItem("Close"));
file.add(new MenuItem("-"));
file.add(new MenuItem("Quit..."));
mbar.add(file);
Menu edit = new Menu("Edit");
edit.add(new MenuItem("Cut"));
edit.add(new MenuItem("Copy"));
edit.add(new MenuItem("Paste"));
edit.add(new MenuItem("-"));
Menu sub = new Menu("Special");
sub.add(new MenuItem("First"));
sub.add(new MenuItem("Second"));
sub.add(new MenuItem("Third"));
edit.add(sub);
```

```
edit.add(new CheckBoxMenuItem("Debug"));

edit.add(new CheckBoxMenuItem("Testing"));

mbar.add(edit);

f.show();

} }
```

Посмотрим на практически классическое меню – MenuDemo.html.

AWT в своем нынешнем виде делает прекрасную работу, являясь общим знаменателем - библиотекой, единой для всех платформ. Некоторый недостаток **AWT** в том, что, поскольку каждый из **AWT**-компонентов реализован на основе соответствующего компонента базовой операционной системы, их поведение и внешний вид может меняться при смене платформы. Хорошо известны расширения и аналоги **AWT** – **Swing**, Java Foundation Classes (Netscape), Application Foundation Classes (Microsoft).