The second color constructor takes a single integer that contains the mix of red, green, and blue packed into an integer. The integer is organized with red in bits 16 to 23, green in bits 8 to 15, and blue in bits 0 to 7. Here is an example of this constructor:

```
int newRed = (0xff000000 | (0xc0 << 16) | (0x00 << 8) | 0x00);
Color darkRed = new Color(newRed);
```

The final constructor, **Color(float, float, float)**, takes three **float** values (between 0.0 and 1.0) that specify the relative mix of red, green, and blue.

Once you have created a color, you can use it to set the foreground and/or background color by using the **setForeground( )** and **setBackground( )** methods described in Chapter 23. You can also select it as the current drawing color.

Color Methods

The **Color** class defines several methods that help manipulate colors. Several are examined here.

Using Hue, Saturation, and Brightness

The *hue-saturation-brightness (HSB)* color model is an alternative to red-green-blue (RGB) for specifying particular colors. Figuratively, *hue* is a wheel of color. The hue can be specified with a number between 0.0 and 1.0, which is used to obtain an angle into the color wheel. (The principal colors are approximately red, orange, yellow, green, blue, indigo, and violet.) *Saturation* is another scale ranging from 0.0 to 1.0, representing light pastels to intense hues. *Brightness* values also range from 0.0 to 1.0, where 1 is bright white and 0 is black. **Color** supplies two methods that let you convert between RGB and HSB. They are shown here:

static int HSBtoRGB(float *hue*, float *saturation*, float *brightness*)
static float[ ] RGBtoHSB(int *red*, int *green*, int *blue*, float *values*[ ])

HSBtoRGB( ) returns a packed RGB value compatible with the **Color(int)** constructor. **RGBtoHSB( )** returns a **float** array of HSB values corresponding to RGB integers. If *values* is not **null**, then this array is given the HSB values and returned. Otherwise, a new array is created and the HSB values are returned in it. In either case, the array contains the hue at index 0, saturation at index 1, and brightness at index 2.

getRed( ), getGreen( ), getBlue( )

You can obtain the red, green, and blue components of a color independently using **getRed( )**, **getGreen( )**, and **getBlue( )**, shown here:

int getRed( )
int getGreen( )
int getBlue( )

Each of these methods returns the RGB color component found in the invoking **Color** object in the lower 8 bits of an integer.

getRGB( )

To obtain a packed, RGB representation of a color, use **getRGB( )**, shown here:

int getRGB( )

The return value is organized as described earlier.

Setting the Current Graphics Color

By default, graphics objects are drawn in the current foreground color. You can change this color by calling the **Graphics** method **setColor( )** :

void setColor(Color *newColor*)

Here, *newColor* specifies the new drawing color.

You can obtain the current color by calling **getColor( )**, shown here:

Color getColor( )

A Color Demonstration Applet

The following applet constructs several colors and draws various objects using these colors:

```
// Demonstrate color.
import java.awt.*;
import java.applet.*;
/*
<applet code="ColorDemo" width=300 height=200>
</applet>
*/

public class ColorDemo extends Applet {
  // draw lines
  public void paint(Graphics g) {
    Color c1 = new Color(255, 100, 100);
    Color c2 = new Color(100, 255, 100);
    Color c3 = new Color(100, 100, 255);

    g.setColor(c1);
    g.drawLine(0, 0, 100, 100);
    g.drawLine(0, 100, 100, 0);

    g.setColor(c2);
    g.drawLine(40, 25, 250, 180);
    g.drawLine(75, 90, 400, 400);

    g.setColor(c3);
    g.drawLine(20, 150, 400, 40);
    g.drawLine(5, 290, 80, 19);

    g.setColor(Color.red);
    g.drawOval(10, 10, 50, 50);
    g.fillOval(70, 90, 140, 100);
```

```
    g.setColor(Color.blue);
    g.drawOval(190, 10, 90, 30);
    g.drawRect(10, 10, 60, 50);

    g.setColor(Color.cyan);
    g.fillRect(100, 10, 60, 50);
    g.drawRoundRect(190, 10, 60, 50, 15, 15);
  }
}
```

Setting the Paint Mode

The *paint mode* determines how objects are drawn in a window. By default, new output to a window overwrites any preexisting contents. However, it is possible to have new objects XORed onto the window by using **setXORMode( )**, as follows:

void setXORMode(Color *xorColor*)

Here, *xorColor* specifies the color that will be XORed to the window when an object is drawn. The advantage of XOR mode is that the new object is always guaranteed to be visible no matter what color the object is drawn over.

To return to overwrite mode, call **setPaintMode( )**, shown here:

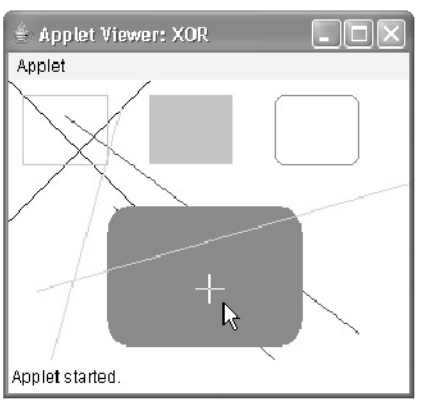
void setPaintMode( )

In general, you will want to use overwrite mode for normal output, and XOR mode for special purposes. For example, the following program displays cross hairs that track the mouse pointer. The cross hairs are XORed onto the window and are always visible, no matter what the underlying color is.

```
// Demonstrate XOR mode.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="XOR" width=400 height=200>
  </applet>
*/

public class XOR extends Applet {
  int chsX=100, chsY=100;

  public XOR() {
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseMoved(MouseEvent me) {
        int x = me.getX();
        int y = me.getY();
        chsX = x-10;
        chsY = y-10;
        repaint();
      }
    });
  }
```

```
  public void paint(Graphics g) {
    g.drawLine(0, 0, 100, 100);
    g.drawLine(0, 100, 100, 0);
    g.setColor(Color.blue);
    g.drawLine(40, 25, 250, 180);
    g.drawLine(75, 90, 400, 400);
    g.setColor(Color.green);
    g.drawRect(10, 10, 60, 50);
    g.fillRect(100, 10, 60, 50);
    g.setColor(Color.red);
    g.drawRoundRect(190, 10, 60, 50, 15, 15);
    g.fillRoundRect(70, 90, 140, 100, 30, 40);
    g.setColor(Color.cyan);
    g.drawLine(20, 150, 400, 40);
    g.drawLine(5, 290, 80, 19);

    // xor cross hairs
    g.setXORMode(Color.black);
    g.drawLine(chsX-10, chsY, chsX+10, chsY);
    g.drawLine(chsX, chsY-10, chsX, chsY+10);
    g.setPaintMode();
  }
}
```

Sample output from this program is shown here:



Working with Fonts

The AWT supports multiple type fonts. Years ago, fonts emerged from the domain of traditional typesetting to become an important part of computer-generated documents and displays. The AWT provides flexibility by abstracting font-manipulation operations and allowing for dynamic selection of fonts.

Fonts have a family name, a logical font name, and a face name. The *family name* is the general name of the font, such as Courier. The *logical name* specifies a name, such as Monospaced, that is linked to an actual font at runtime. The *face name* specifies a specific font, such as Courier Italic.

Fonts are encapsulated by the **Font** class. Several of the methods defined by **Font** are listed in Table 25-2.

The **Font** class defines these protected variables:

Variable	Meaning
String name	Name of the font
float pointSize	Size of the font in points
int size	Size of the font in points
int style	Font style

Several static fields are also defined.

Method	Description
static Font decode(String *str*)	Returns a font given its name.
boolean equals(Object *FontObj*)	Returns **true** if the invoking object contains the same font as that specified by *FontObj*. Otherwise, it returns **false**.
String getFamily( )	Returns the name of the font family to which the invoking font belongs.
static Font getFont(String *property*)	Returns the font associated with the system property specified by *property*. **null** is returned if *property* does not exist.
static Font getFont(String *property*, Font *defaultFont*)	Returns the font associated with the system property specified by *property*. The font specified by *defaultFont* is returned if *property* does not exist.
String getFontName()	Returns the face name of the invoking font.
String getName( )	Returns the logical name of the invoking font.
int getSize( )	Returns the size, in points, of the invoking font.
int getStyle( )	Returns the style values of the invoking font.
int hashCode( )	Returns the hash code associated with the invoking object.
boolean isBold( )	Returns **true** if the font includes the **BOLD** style value. Otherwise, **false** is returned.
boolean isItalic( )	Returns **true** if the font includes the **ITALIC** style value. Otherwise, **false** is returned.
boolean isPlain( )	Returns **true** if the font includes the **PLAIN** style value. Otherwise, **false** is returned.
String toString( )	Returns the string equivalent of the invoking font.

Table 25-2 A Sampling of Methods Defined by **Font**

Determining the Available Fonts

When working with fonts, often you need to know which fonts are available on your machine. To obtain this information, you can use the **getAvailableFontFamilyNames( )** method defined by the **GraphicsEnvironment** class. It is shown here:

String[ ] getAvailableFontFamilyNames( )

This method returns an array of strings that contains the names of the available font families.

In addition, the **getAllFonts( )** method is defined by the **GraphicsEnvironment** class. It is shown here:

Font[ ] getAllFonts( )

This method returns an array of **Font** objects for all of the available fonts.

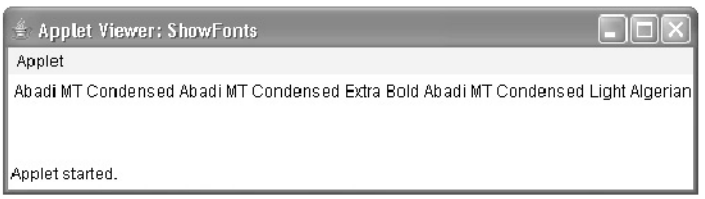
Since these methods are members of **GraphicsEnvironment**, you need a **GraphicsEnvironment** reference to call them. You can obtain this reference by using the **getLocalGraphicsEnvironment( )** static method, which is defined by **GraphicsEnvironment**. It is shown here:

static GraphicsEnvironment getLocalGraphicsEnvironment( )

Here is an applet that shows how to obtain the names of the available font families:

```
// Display Fonts
/*
<applet code="ShowFonts" width=550 height=60>
</applet>
*/
import java.applet.*;
import java.awt.*;

public class ShowFonts extends Applet {
  public void paint(Graphics g) {
    String msg = "";
    String FontList[];

    GraphicsEnvironment ge =
      GraphicsEnvironment.getLocalGraphicsEnvironment();
    FontList = ge.getAvailableFontFamilyNames();
    for(int i = 0; i < FontList.length; i++)
      msg += FontList[i] + " ";

     g.drawString(msg, 4, 16);
  }
}
```

Sample output from this program is shown next. However, when you run this program, you may see a different list of fonts than the one shown in this illustration.



Creating and Selecting a Font

To create a new font, construct a **Font** object that describes that font. One **Font** constructor has this general form:

Font(String *fontName*, int *fontStyle*, int *pointSize*)

Here, *fontName* specifies the name of the desired font. The name can be specified using either the logical or face name. All Java environments will support the following fonts: Dialog, DialogInput, SansSerif, Serif, and Monospaced. Dialog is the font used by your system's dialog boxes. Dialog is also the default if you don't explicitly set a font. You can also use any other fonts supported by your particular environment, but be careful—these other fonts may not be universally available.

The style of the font is specified by *fontStyle*. It may consist of one or more of these three constants: **Font.PLAIN**, **Font.BOLD**, and **Font.ITALIC**. To combine styles, OR them together. For example, **Font.BOLD | Font.ITALIC** specifies a bold, italics style.

The size, in points, of the font is specified by *pointSize*.

To use a font that you have created, you must select it using **setFont( )**, which is defined by **Component**. It has this general form:

void setFont(Font *fontObj*)

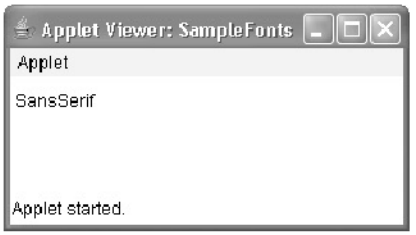
Here, *fontObj* is the object that contains the desired font.

The following program outputs a sample of each standard font. Each time you click the mouse within its window, a new font is selected and its name is displayed.

```
// Show fonts.
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
  <applet code="SampleFonts" width=200 height=100>
  </applet>
*/

public class SampleFonts extends Applet {
  int next = 0;
  Font f;
  String msg;
```

```
  public void init() {
    f = new Font("Dialog", Font.PLAIN, 12);
    msg = "Dialog";
    setFont(f);
    addMouseListener(new MyMouseAdapter(this));
  }

  public void paint(Graphics g) {
    g.drawString(msg, 4, 20);
  }
}

class MyMouseAdapter extends MouseAdapter {
  SampleFonts sampleFonts;

  public MyMouseAdapter(SampleFonts sampleFonts) {
    this.sampleFonts = sampleFonts;
  }

  public void mousePressed(MouseEvent me) {
    // Switch fonts with each mouse click.
    sampleFonts.next++;
    switch(sampleFonts.next) {
    case 0:
      sampleFonts.f = new Font("Dialog", Font.PLAIN, 12);
      sampleFonts.msg = "Dialog";
      break;
    case 1:
      sampleFonts.f = new Font("DialogInput", Font.PLAIN, 12);
      sampleFonts.msg = "DialogInput";
      break;
    case 2:
      sampleFonts.f = new Font("SansSerif", Font.PLAIN, 12);
      sampleFonts.msg = "SansSerif";
      break;
    case 3:
      sampleFonts.f = new Font("Serif", Font.PLAIN, 12);
      sampleFonts.msg = "Serif";
      break;
    case 4:
      sampleFonts.f = new Font("Monospaced", Font.PLAIN, 12);
      sampleFonts.msg = "Monospaced";
      break;
    }

    if(sampleFonts.next == 4) sampleFonts.next = -1;

    sampleFonts.setFont(sampleFonts.f);
    sampleFonts.repaint();
  }
}
```

Sample output from this program is shown here:



Obtaining Font Information

Suppose you want to obtain information about the currently selected font. To do this, you must first get the current font by calling **getFont( )**. This method is defined by the **Graphics** class, as shown here:

```
Font getFont( )
```

Once you have obtained the currently selected font, you can retrieve information about it using various methods defined by **Font**. For example, this applet displays the name, family, size, and style of the currently selected font:

```
// Display font info.
import java.applet.*;
import java.awt.*;
/*
<applet code="FontInfo" width=350 height=60>
</applet>
*/

public class FontInfo extends Applet {
  public void paint(Graphics g) {
    Font f = g.getFont();
    String fontName = f.getName();
    String fontFamily = f.getFamily();
    int fontSize = f.getSize();
    int fontStyle = f.getStyle();

    String msg = "Family: " + fontName;
    msg += ", Font: " + fontFamily;
    msg += ", Size: " + fontSize + ", Style: ";
    if((fontStyle & Font.BOLD) == Font.BOLD)
      msg += "Bold ";
    if((fontStyle & Font.ITALIC) == Font.ITALIC)
      msg += "Italic ";
    if((fontStyle & Font.PLAIN) == Font.PLAIN)
      msg += "Plain ";

    g.drawString(msg, 4, 16);
  }
}
```

Managing Text Output Using FontMetrics

As just explained, Java supports a number of fonts. For most fonts, characters are not all the same dimension—most fonts are proportional. Also, the height of each character, the length of *descenders* (the hanging parts of letters, such as *y*), and the amount of space between horizontal lines vary from font to font. Further, the point size of a font can be changed. That these (and other) attributes are variable would not be of too much consequence except that Java demands that you, the programmer, manually manage virtually all text output.

Given that the size of each font may differ and that fonts may be changed while your program is executing, there must be some way to determine the dimensions and various other attributes of the currently selected font. For example, to write one line of text after another implies that you have some way of knowing how tall the font is and how many pixels are needed between lines. To fill this need, the AWT includes the **FontMetrics** class, which encapsulates various information about a font. Let's begin by defining the common terminology used when describing fonts:

Height	The top-to-bottom size of a line of text
Baseline	The line that the bottoms of characters are aligned to (not counting descent)
Ascent	The distance from the baseline to the top of a character
Descent	The distance from the baseline to the bottom of a character
Leading	The distance between the bottom of one line of text and the top of the next

As you know, we have used the **drawString( )** method in many of the previous examples. It paints a string in the current font and color, beginning at a specified location. However, this location is at the left edge of the baseline of the characters, not at the upper-left corner as is usual with other drawing methods. It is a common error to draw a string at the same coordinate that you would draw a box. For example, if you were to draw a rectangle at coordinate 0,0, you would see a full rectangle. If you were to draw the string "Typesetting" at 0,0, you would only see the tails (or descenders) of the *y*, *p*, and *g*. As you will see, by using font metrics, you can determine the proper placement of each string that you display.

FontMetrics defines several methods that help you manage text output. Several commonly used ones are listed in Table 25-3. These methods help you properly display text in a window. Let's look at some examples.

Displaying Multiple Lines of Text

Perhaps the most common use of **FontMetrics** is to determine the spacing between lines of text. The second most common use is to determine the length of a string that is being displayed. Here, you will see how to accomplish these tasks.

In general, to display multiple lines of text, your program must manually keep track of the current output position. Each time a newline is desired, the Y coordinate must be advanced to the beginning of the next line. Each time a string is displayed, the X coordinate must be set to the point at which the string ends. This allows the next string to be written so that it begins at the end of the preceding one.

Method	Description
int bytesWidth(byte *b*[ ], int *start*, int *numBytes*)	Returns the width of *numBytes* characters held in array *b*, beginning at *start*.
int charWidth(char *c*[ ], int *start*, int *numChars*)	Returns the width of *numChars* characters held in array *c*, beginning at *start*.
int charWidth(char *c*)	Returns the width of *c*.
int charWidth(int *c*)	Returns the width of *c*.
int getAscent( )	Returns the ascent of the font.
int getDescent( )	Returns the descent of the font.
Font getFont( )	Returns the font.
int getHeight( )	Returns the height of a line of text. This value can be used to output multiple lines of text in a window.
int getLeading( )	Returns the space between lines of text.
int getMaxAdvance( )	Returns the width of the widest character. –1 is returned if this value is not available.
int getMaxAscent( )	Returns the maximum ascent.
int getMaxDescent( )	Returns the maximum descent.
int[ ] getWidths( )	Returns the widths of the first 256 characters.
int stringWidth(String *str*)	Returns the width of the string specified by *str*.
String toString( )	Returns the string equivalent of the invoking object.

Table 25-3 A Sampling of Methods Defined by **FontMetrics**

To determine the spacing between lines, you can use the value returned by **getLeading( )**. To determine the total height of the font, add the value returned by **getAscent( )** to the value returned by **getDescent( )**. You can then use these values to position each line of text you output. However, in many cases, you will not need to use these individual values. Often, all that you will need to know is the total height of a line, which is the sum of the leading space and the font's ascent and descent values. The easiest way to obtain this value is to call **getHeight( )**. Simply increment the Y coordinate by this value each time you want to advance to the next line when outputting text.

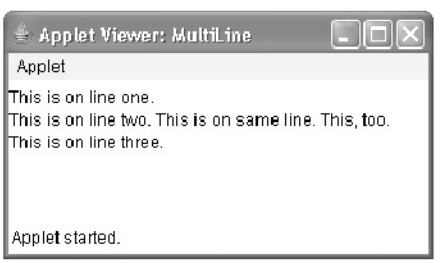
To start output at the end of previous output on the same line, you must know the length, in pixels, of each string that you display. To obtain this value, call **stringWidth( )**. You can use this value to advance the X coordinate each time you display a line.

The following applet shows how to output multiple lines of text in a window. It also displays multiple sentences on the same line. Notice the variables **curX** and **curY**. They keep track of the current text output position.

```
// Demonstrate multiline output.
import java.applet.*;
import java.awt.*;
/*
<applet code="MultiLine" width=300 height=100>
</applet>
*/
```

```
public class MultiLine extends Applet {
  int curX=0, curY=0; // current position

  public void init() {
    Font f = new Font("SansSerif", Font.PLAIN, 12);
    setFont(f);
  }

  public void paint(Graphics g) {
    FontMetrics fm = g.getFontMetrics();

    nextLine("This is on line one.", g);
    nextLine("This is on line two.", g);
    sameLine(" This is on same line.", g);
    sameLine(" This, too.", g);
    nextLine("This is on line three.", g);
    curX = curY = 0; // Reset coordinates for each repaint.
  }

  // Advance to next line.
  void nextLine(String s, Graphics g) {
    FontMetrics fm = g.getFontMetrics();

    curY += fm.getHeight(); // advance to next line
    curX = 0;
    g.drawString(s, curX, curY);
    curX = fm.stringWidth(s); // advance to end of line
  }

  // Display on same line.
  void sameLine(String s, Graphics g) {
    FontMetrics fm = g.getFontMetrics();

    g.drawString(s, curX, curY);
    curX += fm.stringWidth(s); // advance to end of line
  }
}
```

Sample output from this program is shown here:



Centering Text

Here is an example that centers text, left to right, top to bottom, in a window. It obtains the ascent, descent, and width of the string and computes the position at which it must be displayed to be centered.

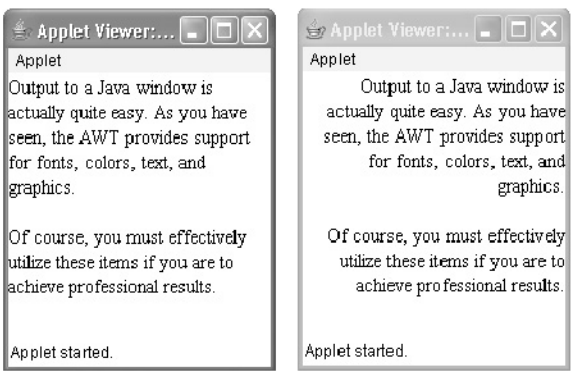
```
// Center text.
import java.applet.*;
import java.awt.*;
/*
  <applet code="CenterText" width=200 height=100>
  </applet>
*/

public class CenterText extends Applet {
  final Font f = new Font("SansSerif", Font.BOLD, 18);

  public void paint(Graphics g) {
    Dimension d = this.getSize();

    g.setColor(Color.white);
    g.fillRect(0, 0, d.width,d.height);
    g.setColor(Color.black);
    g.setFont(f);
    drawCenteredString("This is centered.", 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);
  }
}
```

Following is a sample output from this program:

Multiline Text Alignment

When using a word processor, it is common for text to be aligned so that one or more of the edges of the text make a straight line. For example, most word processors can left-justify and/or right-justify text. Most can also center text. In the following program, you will see how to accomplish these actions.

In the program, the string to be justified is broken into individual words. For each word, the program keeps track of its length in the current font and automatically advances to the next line if the word will not fit on the current line. Each completed line is displayed in the window in the currently selected alignment style. Each time you click the mouse in the applet's window, the alignment style is changed. Sample output from this program is shown here:




```
// Demonstrate text alignment.
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
/* <title>Text Layout</title>
   <applet code="TextLayout" width=200 height=200>
   <param name="text" value="Output to a Java window is actually
      quite easy.
      As you have seen, the AWT provides support for
      fonts, colors, text, and graphics. <P> Of course,
      you must effectively utilize these items
      if you are to achieve professional results.">
   <param name="fontname" value="Serif">
   <param name="fontSize" value="14">
  </applet>
*/

public class TextLayout extends Applet {
  final int LEFT = 0;
  final int RIGHT = 1;
  final int CENTER = 2;
  final int LEFTRIGHT =3;
  int align;
```

```
Dimension d;
Font f;
FontMetrics fm;
int fontSize;
int fh, bl;
int space;
String text;

public void init() {
  setBackground(Color.white);
  text = getParameter("text");
  try {
    fontSize = Integer.parseInt(getParameter("fontSize"));}
  catch (NumberFormatException e) {
    fontSize=14;
  }
  align = LEFT;
  addMouseListener(new MyMouseAdapter(this));
}

public void paint(Graphics g) {
  update(g);
}

public void update(Graphics g) {
  d = getSize();
  g.setColor(getBackground());
  g.fillRect(0,0,d.width, d.height);
  if(f==null) f = new Font(getParameter("fontname"),
                           Font.PLAIN, fontSize);
  g.setFont(f);
  if(fm == null) {
      fm = g.getFontMetrics();
      bl = fm.getAscent();
      fh = bl + fm.getDescent();
      space = fm.stringWidth(" ");
  }

  g.setColor(Color.black);
  StringTokenizer st = new StringTokenizer(text);
  int x = 0;
  int nextx;
  int y = 0;
  String word, sp;
  int wordCount = 0;
  String line = "";
  while (st.hasMoreTokens()) {
    word = st.nextToken();
    if(word.equals("<P>")) {
      drawString(g, line, wordCount,
                 fm.stringWidth(line), y+bl);
      line = "";
      wordCount = 0;
```

```
      x = 0;
      y = y + (fh * 2);
    }
    else {
      int w = fm.stringWidth(word);
      if(( nextx = (x+space+w)) > d.width ) {
        drawString(g, line, wordCount,
                   fm.stringWidth(line), y+bl);
        line = "";
        wordCount = 0;
        x = 0;
        y = y + fh;
      }
      if(x!=0) {sp = " ";} else {sp = "";}
      line = line + sp + word;
      x = x + space + w;
      wordCount++;
    }
  }
  drawString(g, line, wordCount, fm.stringWidth(line), y+bl);
}

public void drawString(Graphics g, String line,
                       int wc, int lineW, int y) {
  switch(align) {
    case LEFT: g.drawString(line, 0, y);
      break;
    case RIGHT: g.drawString(line, d.width-lineW,y);
      break;
    case CENTER: g.drawString(line, (d.width-lineW)/2, y);
      break;
    case LEFTRIGHT:
      if(lineW < (int)(d.width*.75)) {
         g.drawString(line, 0, y);
      }
      else {
        int toFill = (d.width - lineW)/wc;
        int nudge = d.width - lineW - (toFill*wc);
        int s = fm.stringWidth(" ");
        StringTokenizer st = new StringTokenizer(line);
        int x = 0;
        while(st.hasMoreTokens()) {
          String word = st.nextToken();
          g.drawString(word, x, y);
          if(nudge>0) {
            x = x + fm.stringWidth(word) + space + toFill + 1;
            nudge--;
          } else {
            x = x + fm.stringWidth(word) + space + toFill;
          }
        }
      }
      break;
```

```
    }
  }
}

class MyMouseAdapter extends MouseAdapter {
  TextLayout tl;

  public MyMouseAdapter(TextLayout tl) {
    this.tl = tl;
  }

  public void mouseClicked(MouseEvent me) {
    tl.align = (tl.align + 1) % 4;
    tl.repaint();
  }
}
```

Let's take a closer look at how this applet works. The applet first creates several constants that will be used to determine the alignment style, and then declares several variables. The **init( )** method obtains the text that will be displayed. It then initializes the font size in a **try-catch** block, which will set the font size to 14 if the **fontSize** parameter is missing from the HTML. The **text** parameter is a long string of text, with the HTML tag **<P>** as a paragraph separator.

The **update( )** method is the engine for this example. It sets the font and gets the baseline and font height from a font metrics object. Next, it creates a **StringTokenizer** and uses it to retrieve the next token (a string separated by whitespace) from the string specified by **text**. If the next token is **<P>**, it advances the vertical spacing. Otherwise, **update( )** checks to see if the length of this token in the current font will go beyond the width of the column. If the line is full of text or if there are no more tokens, the line is output by a custom version of **drawString( )**.

The first three cases in **drawString( )** are simple. Each aligns the string that is passed in **line** to the left or right edge or to the center of the column, depending upon the alignment style. The **LEFTRIGHT** case aligns both the left and right sides of the string. This means that we need to calculate the remaining whitespace (the difference between the width of the string and the width of the column) and distribute that space between each of the words. The last method in this class advances the alignment style each time you click the mouse on the applet's window.

CHAPTER

26 Using AWT Controls, Layout Managers, and Menus

This chapter continues our overview of the Abstract Window Toolkit (AWT). It begins with a look at several of the AWT's controls and layout managers. It then discusses menus and the menu bar. The chapter also includes a discussion of two high-level components: the dialog box and the file dialog box. It concludes with another look at event handling.

Controls are components that allow a user to interact with your application in various ways—for example, a commonly used control is the push button. A *layout manager* automatically positions components within a container. Thus, the appearance of a window is determined by a combination of the controls that it contains and the layout manager used to position them.

In addition to the controls, a frame window can also include a standard-style *menu bar*. Each entry in a menu bar activates a drop-down menu of options from which the user can choose. This constitutes the *main menu* of an application. As a general rule, a menu bar is positioned at the top of a window. Although different in appearance, menu bars are handled in much the same way as are the other controls.

While it is possible to manually position components within a window, doing so is quite tedious. The layout manager automates this task. For the first part of this chapter, which introduces various controls, the default layout manager will be used. This displays components in a container using left-to-right, top-to-bottom organization. Once the controls have been covered, several layout managers will be examined. There, you will see ways to better manage the positioning of controls.

Before continuing, it is important to emphasize that today you will seldom create GUIs based solely on the AWT because more powerful GUI frameworks (Swing and JavaFX) have been developed for Java. However, the material presented here remains important for the following reasons. First, much of the information and many of the techniques related to controls and event handling are generalizable to the other Java GUI frameworks. (As mentioned in the previous chapter, Swing is built upon the AWT.) Second, the layout managers described here can also be used by Swing. Third, for some small applications, the AWT components might be the appropriate choice. Finally, and perhaps most importantly, you may need to maintain or upgrade legacy code that uses the AWT. Therefore, a basic understanding of the AWT is important for all Java programmers.

AWT Control Fundamentals

The AWT supports the following types of controls:

- Labels
- Push buttons
- Check boxes
- Choice lists
- Lists
- Scroll bars
- Text Editing

These controls are subclasses of **Component**. Although this is not a particularly rich set of controls, it is sufficient for simple applications. (Note that both Swing and JavaFX provide a substantially larger, more sophisticated set of controls.)

Adding and Removing Controls

To include a control in a window, you must add it to the window. To do this, you must first create an instance of the desired control and then add it to a window by calling **add( )**, which is defined by **Container**. The **add( )** method has several forms. The following form is the one that is used for the first part of this chapter:

Component add(Component *compRef*)

Here, *compRef* is a reference to an instance of the control that you want to add. A reference to the object is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed.

Sometimes you will want to remove a control from a window when the control is no longer needed. To do this, call **remove( )**. This method is also defined by **Container**. Here is one of its forms:

void remove(Component *compRef*)

Here, *compRef* is a reference to the control you want to remove. You can remove all controls by calling **removeAll( )**.

Responding to Controls

Except for labels, which are passive, all other controls generate events when they are accessed by the user. For example, when the user clicks on a push button, an event is sent that identifies the push button. In general, your program simply implements the appropriate interface and then registers an event listener for each control that you need to monitor. As explained in Chapter 24, once a listener has been installed, events are automatically sent to it. In the sections that follow, the appropriate interface for each control is specified.

The HeadlessException

Most of the AWT controls described in this chapter have constructors that can throw a **HeadlessException** when an attempt is made to instantiate a GUI component in a non-interactive environment (such as one in which no display, mouse, or keyboard is present). You can use this exception to write code that can adapt to non-interactive environments. (Of course, this is not always possible.) This exception is not handled by the programs in this chapter because an interactive environment is required to demonstrate the AWT controls.

Labels

The easiest control to use is a label. A *label* is an object of type **Label**, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user. **Label** defines the following constructors:

Label( ) throws HeadlessException
Label(String *str*) throws HeadlessException
Label(String *str*, int *how*) throws HeadlessException

The first version creates a blank label. The second version creates a label that contains the string specified by *str*. This string is left-justified. The third version creates a label that contains the string specified by *str* using the alignment specified by *how*. The value of *how* must be one of these three constants: **Label.LEFT**, **Label.RIGHT**, or **Label.CENTER**.

You can set or change the text in a label by using the **setText( )** method. You can obtain the current label by calling **getText( )**. These methods are shown here:

void setText(String *str*)
String getText( )

For **setText( )**, *str* specifies the new label. For **getText( )**, the current label is returned.

You can set the alignment of the string within the label by calling **setAlignment( )**. To obtain the current alignment, call **getAlignment( )**. The methods are as follows:

void setAlignment(int *how*)
int getAlignment( )

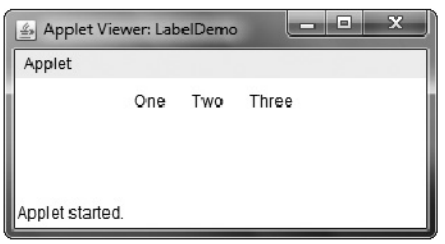
Here, *how* must be one of the alignment constants shown earlier.

The following example creates three labels and adds them to an applet window:

```
// Demonstrate Labels
import java.awt.*;
import java.applet.*;
/*
<applet code="LabelDemo" width=300 height=200>
</applet>
*/

public class LabelDemo extends Applet {
  public void init() {
    Label one = new Label("One");
    Label two = new Label("Two");
    Label three = new Label("Three");
```

```
    // add labels to applet window
    add(one);
    add(two);
    add(three);
  }
}
```

Here is sample output from the **LabelDemo** applet. Notice that the labels are organized in the window by the default layout manager. Later, you will see how to control more precisely the placement of the labels.



Using Buttons

Perhaps the most widely used control is the push button. A *push button* is a component that contains a label and that generates an event when it is pressed. Push buttons are objects of type **Button**. **Button** defines these two constructors:

Button( ) throws HeadlessException
Button(String *str*) throws HeadlessException

The first version creates an empty button. The second creates a button that contains *str* as a label.

After a button has been created, you can set its label by calling **setLabel( )**. You can retrieve its label by calling **getLabel( )**. These methods are as follows:

void setLabel(String *str*)
String getLabel( )

Here, *str* becomes the new label for the button.

Handling Buttons

Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered an interest in receiving action event notifications from that component. Each listener implements the **ActionListener** interface. That interface defines the **actionPerformed( )** method, which is called when an event occurs. An **ActionEvent** object is supplied as the argument to this method. It contains both a reference to the button that generated the event and a reference to the *action command string* associated with the button. By default, the action command string is the label of the button. Either the button reference or the action command string can be used to identify the button. (You will soon see examples of each approach.)

Here is an example that creates three buttons labeled "Yes", "No", and "Undecided". Each time one is pressed, a message is displayed that reports which button has been pressed. In this version, the action command of the button (which, by default, is its label) is used to determine which button has been pressed. The label is obtained by calling the **getActionCommand( )** method on the **ActionEvent** object passed to **actionPerformed( )**.

```
// Demonstrate Buttons
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="ButtonDemo" width=250 height=150>
  </applet>
*/

public class ButtonDemo extends Applet implements ActionListener {
  String msg = "";
  Button yes, no, maybe;

  public void init() {
    yes = new Button("Yes");
    no = new Button("No");
    maybe = new Button("Undecided");

    add(yes);
    add(no);
    add(maybe);

    yes.addActionListener(this);
    no.addActionListener(this);
    maybe.addActionListener(this);
  }

  public void actionPerformed(ActionEvent ae) {
    String str = ae.getActionCommand();

    if(str.equals("Yes")) {
      msg = "You pressed Yes.";
    }
    else if(str.equals("No")) {
      msg = "You pressed No.";
    }
    else {
      msg = "You pressed Undecided.";
    }

    repaint();
  }

  public void paint(Graphics g) {
     g.drawString(msg, 6, 100);
  }
}
```

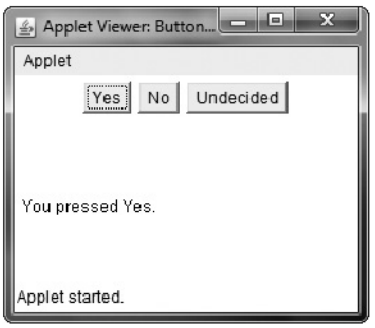


Figure 26-1 Sample output from the **ButtonDemo** applet

Sample output from the **ButtonDemo** program is shown in Figure 26-1.

As mentioned, in addition to comparing button action command strings, you can also determine which button has been pressed by comparing the object obtained from the **getSource( )** method to the button objects that you added to the window. To do this, you must keep a list of the objects when they are added. The following applet shows this approach:

```
// Recognize Button objects.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="ButtonList" width=250 height=150>
  </applet>
*/

public class ButtonList extends Applet implements ActionListener {
  String msg = "";
  Button bList[] = new Button[3];

  public void init() {
    Button yes = new Button("Yes");
    Button no = new Button("No");
    Button maybe = new Button("Undecided");

    // store references to buttons as added
    bList[0] = (Button) add(yes);
    bList[1] = (Button) add(no);
    bList[2] = (Button) add(maybe);

    // register to receive action events
    for(int i = 0; i < 3; i++) {
      bList[i].addActionListener(this);
    }
  }
```

```
  public void actionPerformed(ActionEvent ae) {
    for(int i = 0; i < 3; i++) {
      if(ae.getSource() == bList[i]) {
        msg = "You pressed " + bList[i].getLabel();
      }
    }
    repaint();
  }

  public void paint(Graphics g) {
     g.drawString(msg, 6, 100);
  }
}
```

In this version, the program stores each button reference in an array when the buttons are added to the applet window. (Recall that the **add( )** method returns a reference to the button when it is added.) Inside **actionPerformed( )**, this array is then used to determine which button has been pressed.

For simple programs, it is usually easier to recognize buttons by their labels. However, in situations in which you will be changing the label inside a button during the execution of your program, or using buttons that have the same label, it may be easier to determine which button has been pushed by using its object reference. It is also possible to set the action command string associated with a button to something other than its label by calling **setActionCommand( )**. This method changes the action command string, but does not affect the string used to label the button. Thus, setting the action command enables the action command and the label of a button to differ.

In some cases, you can handle the action events generated by a button (or some other type of control) by use of an anonymous inner class (as described in Chapter 24) or a lambda expression (discussed in Chapter 15). For example, assuming the previous programs, here is a set of action event handlers that use lambda expressions:

```
// Use lambda expressions to handle action events.
yes.addActionListener((ae) -> {
  msg = "You pressed " + ae.getActionCommand();
  repaint();
});

no.addActionListener((ae) -> {
  msg = "You pressed " + ae.getActionCommand();
  repaint();
});

maybe.addActionListener((ae) -> {
  msg = "You pressed " + ae.getActionCommand();
  repaint();
});
```

This code works because **ActionListener** defines a functional interface, which is an interface with exactly one abstract method. Thus, it can be used by a lambda expression. In general, you can use a lambda expression to handle an AWT event when its listener defines a functional interface. For example, **ItemListener** is also a functional interface. Of course,

whether you use the traditional approach, an anonymous inner class, or a lambda expression will be determined by the precise nature of your application. The remaining examples in this chapter use the traditional approach to event handling so that they can be compiled by nearly any version of Java. However, you might find it interesting to try converting the event handlers to lambda expressions or anonymous inner classes, where appropriate.

Applying Check Boxes

A *check box* is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents. You change the state of a check box by clicking on it. Check boxes can be used individually or as part of a group. Check boxes are objects of the **Checkbox** class.

Checkbox supports these constructors:

```
Checkbox( ) throws HeadlessException
Checkbox(String str) throws HeadlessException
Checkbox(String str, boolean on) throws HeadlessException
Checkbox(String str, boolean on, CheckboxGroup cbGroup) throws HeadlessException
Checkbox(String str, CheckboxGroup cbGroup, boolean on) throws HeadlessException
```

The first form creates a check box whose label is initially blank. The state of the check box is unchecked. The second form creates a check box whose label is specified by *str*. The state of the check box is unchecked. The third form allows you to set the initial state of the check box. If *on* is **true**, the check box is initially checked; otherwise, it is cleared. The fourth and fifth forms create a check box whose label is specified by *str* and whose group is specified by *cbGroup*. If this check box is not part of a group, then *cbGroup* must be **null**. (Check box groups are described in the next section.) The value of *on* determines the initial state of the check box.

To retrieve the current state of a check box, call **getState( )**. To set its state, call **setState( )**. You can obtain the current label associated with a check box by calling **getLabel( )**. To set the label, call **setLabel( )**. These methods are as follows:

```
boolean getState( )
void setState(boolean on)
String getLabel( )
void setLabel(String str)
```

Here, if *on* is **true**, the box is checked. If it is **false**, the box is cleared. The string passed in *str* becomes the new label associated with the invoking check box.

Handling Check Boxes

Each time a check box is selected or deselected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the **ItemListener** interface. That interface defines the **itemStateChanged( )** method. An **ItemEvent** object is supplied as the argument to this method. It contains information about the event (for example, whether it was a selection or deselection).

The following program creates four check boxes. The initial state of the first box is checked. The status of each check box is displayed. Each time you change the state of a check box, the status display is updated.

```
// Demonstrate check boxes.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="CheckboxDemo" width=240 height=200>
  </applet>
*/

public class CheckboxDemo extends Applet implements ItemListener {
  String msg = "";
  Checkbox windows, android, solaris, mac;

  public void init() {
    windows = new Checkbox("Windows", null, true);
    android = new Checkbox("Android");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("Mac OS");

    add(windows);
    add(android);
    add(solaris);
    add(mac);

    windows.addItemListener(this);
    android.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
  }

  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Display current state of the check boxes.
  public void paint(Graphics g) {
    msg = "Current state: ";
    g.drawString(msg, 6, 80);
    msg = "  Windows: " + windows.getState();
    g.drawString(msg, 6, 100);
    msg = "  Android: " + android.getState();
    g.drawString(msg, 6, 120);
    msg = "  Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = "  Mac OS: " + mac.getState();
    g.drawString(msg, 6, 160);
  }
}
```

Sample output is shown in Figure 26-2.

Figure 26-2 Sample output from the **CheckboxDemo** applet

CheckboxGroup

It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called *radio buttons*, because they act like the station selector on a car radio—only one station can be selected at any one time. To create a set of mutually exclusive check boxes, you must first define the group to which they will belong and then specify that group when you construct the check boxes. Check box groups are objects of type **CheckboxGroup**. Only the default constructor is defined, which creates an empty group.

You can determine which check box in a group is currently selected by calling **getSelectedCheckbox( )**. You can set a check box by calling **setSelectedCheckbox( )**. These methods are as follows:

Checkbox getSelectedCheckbox( )
void setSelectedCheckbox(Checkbox *which*)

Here, *which* is the check box that you want to be selected. The previously selected check box will be turned off.

Here is a program that uses check boxes that are part of a group:

```
// Demonstrate check box group.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="CBGroup" width=240 height=200>
  </applet>
*/

public class CBGroup extends Applet implements ItemListener {
  String msg = "";
  Checkbox windows, android, solaris, mac;
  CheckboxGroup cbg;
```

```
  public void init() {
    cbg = new CheckboxGroup();
    windows = new Checkbox("Windows", cbg, true);
    android = new Checkbox("Android", cbg, false);
    solaris = new Checkbox("Solaris", cbg, false);
    mac = new Checkbox("Mac OS", cbg, false);

    add(windows);
    add(android);
    add(solaris);
    add(mac);

    windows.addItemListener(this);
    android.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
  }

  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Display current state of the check boxes.
  public void paint(Graphics g) {
    msg = "Current selection: ";
    msg += cbg.getSelectedCheckbox().getLabel();
    g.drawString(msg, 6, 100);
  }
}
```

Sample output generated by the **CBGroup** applet is shown in Figure 26-3. Notice that the check boxes are now circular in shape.

Figure 26-3 Sample output from the **CBGroup** applet

Choice Controls

The **Choice** class is used to create a *pop-up list* of items from which the user may choose. Thus, a **Choice** control is a form of menu. When inactive, a **Choice** component takes up only enough space to show the currently selected item. When the user clicks on it, the whole list of choices pops up, and a new selection can be made. Each item in the list is a string that appears as a left-justified label in the order it is added to the **Choice** object. **Choice** defines only the default constructor, which creates an empty list.

To add a selection to the list, call **add( )**. It has this general form:

void add(String *name*)

Here, *name* is the name of the item being added. Items are added to the list in the order in which calls to **add( )** occur.

To determine which item is currently selected, you may call either **getSelectedItem( )** or **getSelectedIndex( )**. These methods are shown here:

String getSelectedItem( )
int getSelectedIndex( )

The **getSelectedItem( )** method returns a string containing the name of the item. **getSelectedIndex( )** returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected.

To obtain the number of items in the list, call **getItemCount( )**. You can set the currently selected item using the **select( )** method with either a zero-based integer index or a string that will match a name in the list. These methods are shown here:

int getItemCount( )
void select(int *index*)
void select(String *name*)

Given an index, you can obtain the name associated with the item at that index by calling **getItem( )**, which has this general form:

String getItem(int *index*)

Here, *index* specifies the index of the desired item.

Handling Choice Lists

Each time a choice is selected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the **ItemListener** interface. That interface defines the **itemStateChanged( )** method. An **ItemEvent** object is supplied as the argument to this method.

Here is an example that creates two **Choice** menus. One selects the operating system. The other selects the browser.

```
// Demonstrate Choice lists.
import java.awt.*;
import java.awt.event.*;
```

```
import java.applet.*;
/*
  <applet code="ChoiceDemo" width=300 height=180>
  </applet>
*/

public class ChoiceDemo extends Applet implements ItemListener {
  Choice os, browser;
  String msg = "";

  public void init() {
    os = new Choice();
    browser = new Choice();

    // add items to os list
    os.add("Windows");
    os.add("Android");
    os.add("Solaris");
    os.add("Mac OS");

    // add items to browser list
    browser.add("Internet Explorer");
    browser.add("Firefox");
    browser.add("Chrome");

    // add choice lists to window
    add(os);
    add(browser);

    // register to receive item events
    os.addItemListener(this);
    browser.addItemListener(this);
  }

  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Display current selections.
  public void paint(Graphics g) {
    msg = "Current OS: ";
    msg += os.getSelectedItem();
    g.drawString(msg, 6, 120);
    msg = "Current Browser: ";
    msg += browser.getSelectedItem();
    g.drawString(msg, 6, 140);
  }
}
```

Sample output is shown in Figure 26-4.

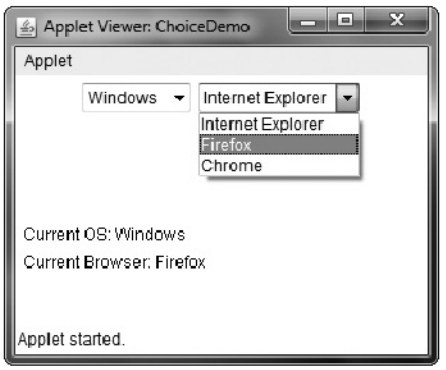


Figure 26-4 Sample output from the **ChoiceDemo** applet

Using Lists

The **List** class provides a compact, multiple-choice, scrolling selection list. Unlike the **Choice** object, which shows only the single selected item in the menu, a **List** object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections. **List** provides these constructors:

List( ) throws HeadlessException
List(int *numRows*) throws HeadlessException
List(int *numRows*, boolean *multipleSelect*) throws HeadlessException

The first version creates a **List** control that allows only one item to be selected at any one time. In the second form, the value of *numRows* specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed). In the third form, if *multipleSelect* is **true**, then the user may select two or more items at a time. If it is **false**, then only one item may be selected.

To add a selection to the list, call **add( )**. It has the following two forms:

void add(String *name*)
void add(String *name*, int *index*)

Here, *name* is the name of the item added to the list. The first form adds items to the end of the list. The second form adds the item at the index specified by *index*. Indexing begins at zero. You can specify –1 to add the item to the end of the list.

For lists that allow only single selection, you can determine which item is currently selected by calling either **getSelectedItem( )** or **getSelectedIndex( )**. These methods are shown here:

String getSelectedItem( )
int getSelectedIndex( )

The **getSelectedItem( )** method returns a string containing the name of the item. If more than one item is selected, or if no selection has yet been made, **null** is returned. **getSelectedIndex( )** returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, –1 is returned.

For lists that allow multiple selection, you must use either **getSelectedItems( )** or **getSelectedIndexes( )**, shown here, to determine the current selections:

String[ ] getSelectedItems( )
int[ ] getSelectedIndexes( )

getSelectedItems( ) returns an array containing the names of the currently selected items. **getSelectedIndexes( )** returns an array containing the indexes of the currently selected items.

To obtain the number of items in the list, call **getItemCount( )**. You can set the currently selected item by using the **select( )** method with a zero-based integer index. These methods are shown here:

int getItemCount( )
void select(int *index*)

Given an index, you can obtain the name associated with the item at that index by calling **getItem( )**, which has this general form:

String getItem(int *index*)

Here, *index* specifies the index of the desired item.

Handling Lists

To process list events, you will need to implement the **ActionListener** interface. Each time a **List** item is double-clicked, an **ActionEvent** object is generated. Its **getActionCommand( )** method can be used to retrieve the name of the newly selected item. Also, each time an item is selected or deselected with a single click, an **ItemEvent** object is generated. Its **getStateChange( )** method can be used to determine whether a selection or deselection triggered this event. **getItemSelectable( )** returns a reference to the object that triggered this event.

Here is an example that converts the **Choice** controls in the preceding section into **List** components, one multiple choice and the other single choice:

```
// Demonstrate Lists.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="ListDemo" width=300 height=180>
  </applet>
*/

public class ListDemo extends Applet implements ActionListener {
  List os, browser;
  String msg = "";

  public void init() {
    os = new List(4, true);
    browser = new List(4, false);

    // add items to os list
    os.add("Windows");
```

```
    os.add("Android");
    os.add("Solaris");
    os.add("Mac OS");

    // add items to browser list
    browser.add("Internet Explorer");
    browser.add("Firefox");
    browser.add("Chrome");

    browser.select(1);

    // add lists to window
    add(os);
    add(browser);

    // register to receive action events
    os.addActionListener(this);
    browser.addActionListener(this);
  }

  public void actionPerformed(ActionEvent ae) {
    repaint();
  }

  // Display current selections.
  public void paint(Graphics g) {
    int idx[];

    msg = "Current OS: ";
    idx = os.getSelectedIndexes();
    for(int i=0; i<idx.length; i++)
      msg += os.getItem(idx[i]) + "  ";
    g.drawString(msg, 6, 120);
    msg = "Current Browser: ";
    msg += browser.getSelectedItem();
    g.drawString(msg, 6, 140);
  }
}
```

Sample output generated by the **ListDemo** applet is shown in Figure 26-5.

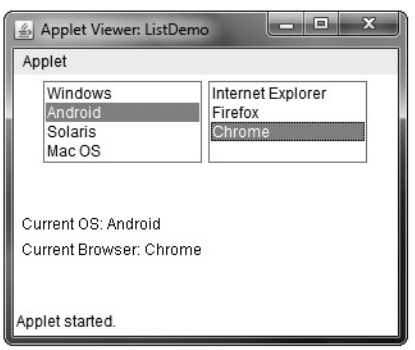


Figure 26-5 Sample output from the **ListDemo** applet

Managing Scroll Bars

Scroll bars are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. A scroll bar is actually a composite of several individual parts. Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow. The current value of the scroll bar relative to its minimum and maximum values is indicated by the *slider box* (or *thumb*) for the scroll bar. The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value. In the background space on either side of the thumb, the user can click to cause the thumb to jump in that direction by some increment larger than 1. Typically, this action translates into some form of page up and page down. Scroll bars are encapsulated by the **Scrollbar** class.

Scrollbar defines the following constructors:

```
Scrollbar( ) throws HeadlessException
Scrollbar(int style) throws HeadlessException
Scrollbar(int style, int initialValue, int thumbSize, int min, int max)
    throws HeadlessException
```

The first form creates a vertical scroll bar. The second and third forms allow you to specify the orientation of the scroll bar. If *style* is **Scrollbar.VERTICAL**, a vertical scroll bar is created. If *style* is **Scrollbar.HORIZONTAL**, the scroll bar is horizontal. In the third form of the constructor, the initial value of the scroll bar is passed in *initialValue*. The number of units represented by the height of the thumb is passed in *thumbSize*. The minimum and maximum values for the scroll bar are specified by *min* and *max*.

If you construct a scroll bar by using one of the first two constructors, then you need to set its parameters by using **setValues( )**, shown here, before it can be used:

```
void setValues(int initialValue, int thumbSize, int min, int max)
```

The parameters have the same meaning as they have in the third constructor just described.

To obtain the current value of the scroll bar, call **getValue( )**. It returns the current setting. To set the current value, call **setValue( )**. These methods are as follows:

```
int getValue( )
void setValue(int newValue)
```

Here, *newValue* specifies the new value for the scroll bar. When you set a value, the slider box inside the scroll bar will be positioned to reflect the new value.

You can also retrieve the minimum and maximum values via **getMinimum( )** and **getMaximum( )**, shown here:

```
int getMinimum( )
int getMaximum( )
```

They return the requested quantity.

By default, 1 is the increment added to or subtracted from the scroll bar each time it is scrolled up or down one line. You can change this increment by calling **setUnitIncrement( )**. By default, page-up and page-down increments are 10. You can change this value by calling **setBlockIncrement( )**. These methods are shown here:

```
void setUnitIncrement(int newIncr)
void setBlockIncrement(int newIncr)
```

Handling Scroll Bars

To process scroll bar events, you need to implement the **AdjustmentListener** interface. Each time a user interacts with a scroll bar, an **AdjustmentEvent** object is generated. Its **getAdjustmentType( )** method can be used to determine the type of the adjustment. The types of adjustment events are as follows:

BLOCK_DECREMENT	A page-down event has been generated.
BLOCK_INCREMENT	A page-up event has been generated.
TRACK	An absolute tracking event has been generated.
UNIT_DECREMENT	The line-down button in a scroll bar has been pressed.
UNIT_INCREMENT	The line-up button in a scroll bar has been pressed.

The following example creates both a vertical and a horizontal scroll bar. The current settings of the scroll bars are displayed. If you drag the mouse while inside the window, the coordinates of each drag event are used to update the scroll bars. An asterisk is displayed at the current drag position. Notice the use of **setPreferredSize( )** to set the size of the scrollbars.

```
// Demonstrate scroll bars.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="SBDemo" width=300 height=200>
  </applet>
*/

public class SBDemo extends Applet
  implements AdjustmentListener, MouseMotionListener {
  String msg = "";
  Scrollbar vertSB, horzSB;

  public void init() {
    int width = Integer.parseInt(getParameter("width"));
    int height = Integer.parseInt(getParameter("height"));

    vertSB = new Scrollbar(Scrollbar.VERTICAL,
                           0, 1, 0, height);
    vertSB.setPreferredSize(new Dimension(20, 100));

    horzSB = new Scrollbar(Scrollbar.HORIZONTAL,
                           0, 1, 0, width);
    horzSB.setPreferredSize(new Dimension(100, 20));

    add(vertSB);
    add(horzSB);

    // register to receive adjustment events
```

```
    vertSB.addAdjustmentListener(this);
    horzSB.addAdjustmentListener(this);

    addMouseMotionListener(this);
  }

  public void adjustmentValueChanged(AdjustmentEvent ae) {
    repaint();
  }

  // Update scroll bars to reflect mouse dragging.
  public void mouseDragged(MouseEvent me) {
    int x = me.getX();
    int y = me.getY();
    vertSB.setValue(y);
    horzSB.setValue(x);
    repaint();
  }

  // Necessary for MouseMotionListener
  public void mouseMoved(MouseEvent me) {
  }

  // Display current value of scroll bars.
  public void paint(Graphics g) {
     msg = "Vertical: " + vertSB.getValue();
     msg += ",  Horizontal: " + horzSB.getValue();
     g.drawString(msg, 6, 160);

     // show current mouse drag position
     g.drawString("*", horzSB.getValue(),
                  vertSB.getValue());
  }
}
```

Sample output from the **SBDemo** applet is shown in Figure 26-6.

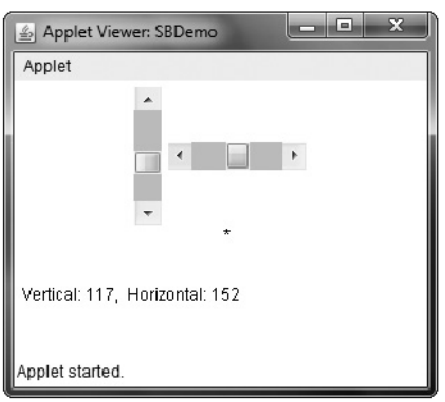


Figure 26-6 Sample output from the **SBDemo** applet

Using a TextField

The **TextField** class implements a single-line text-entry area, usually called an *edit control*. Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections. **TextField** is a subclass of **TextComponent**. **TextField** defines the following constructors:

```
TextField( ) throws HeadlessException
TextField(int numChars) throws HeadlessException
TextField(String str) throws HeadlessException
TextField(String str, int numChars) throws HeadlessException
```

The first version creates a default text field. The second form creates a text field that is *numChars* characters wide. The third form initializes the text field with the string contained in *str*. The fourth form initializes a text field and sets its width.

TextField (and its superclass **TextComponent**) provides several methods that allow you to utilize a text field. To obtain the string currently contained in the text field, call **getText( )**. To set the text, call **setText( )**. These methods are as follows:

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

Here, *str* is the new string.

The user can select a portion of the text in a text field. Also, you can select a portion of text under program control by using **select( )**. Your program can obtain the currently selected text by calling **getSelectedText( )**. These methods are shown here:

```
String getSelectedText( )
void select(int startIndex, int endIndex)
```

getSelectedText( ) returns the selected text. The **select( )** method selects the characters beginning at *startIndex* and ending at *endIndex* –1.

You can control whether the contents of a text field may be modified by the user by calling **setEditable( )**. You can determine editability by calling **isEditable( )**. These methods are shown here:

```
boolean isEditable( )
void setEditable(boolean canEdit)
```

isEditable( ) returns **true** if the text may be changed and **false** if not. In **setEditable( )**, if *canEdit* is **true**, the text may be changed. If it is **false**, the text cannot be altered.

There may be times when you will want the user to enter text that is not displayed, such as a password. You can disable the echoing of the characters as they are typed by calling **setEchoChar( )**. This method specifies a single character that the **TextField** will display when characters are entered (thus, the actual characters typed will not be shown). You can check a text field to see if it is in this mode with the **echoCharIsSet( )** method. You can retrieve the echo character by calling the **getEchoChar( )** method. These methods are as follows:

```
void setEchoChar(char ch)
boolean echoCharIsSet( )
char getEchoChar( )
```

Here, *ch* specifies the character to be echoed. If *ch* is zero, then normal echoing is restored.

Handling a TextField

Since text fields perform their own editing functions, your program generally will not respond to individual key events that occur within a text field. However, you may want to respond when the user presses ENTER. When this occurs, an action event is generated.

Here is an example that creates the classic user name and password screen:

```
// Demonstrate text field.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="TextFieldDemo" width=380 height=150>
  </applet>
*/

public class TextFieldDemo extends Applet
  implements ActionListener {

  TextField name, pass;

  public void init() {
    Label namep = new Label("Name: ", Label.RIGHT);
    Label passp = new Label("Password: ", Label.RIGHT);
    name = new TextField(12);
    pass = new TextField(8);
    pass.setEchoChar('?');

    add(namep);
    add(name);
    add(passp);
    add(pass);

    // register to receive action events
    name.addActionListener(this);
    pass.addActionListener(this);
  }

  // User pressed Enter.
  public void actionPerformed(ActionEvent ae) {
    repaint();
  }

  public void paint(Graphics g) {
    g.drawString("Name: " + name.getText(), 6, 60);
    g.drawString("Selected text in name: "
                 + name.getSelectedText(), 6, 80);
    g.drawString("Password: " + pass.getText(), 6, 100);
  }
}
```

Sample output from the **TextFieldDemo** applet is shown in Figure 26-7.

Figure 26-7 Sample output from the **TextFieldDemo** applet

Using a TextArea

Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called **TextArea**. Following are the constructors for **TextArea**:

TextArea( ) throws HeadlessException
TextArea(int *numLines*, int *numChars*) throws HeadlessException
TextArea(String *str*) throws HeadlessException
TextArea(String *str*, int *numLines*, int *numChars*) throws HeadlessException
TextArea(String *str*, int *numLines*, int *numChars*, int *sBars*) throws HeadlessException

Here, *numLines* specifies the height, in lines, of the text area, and *numChars* specifies its width, in characters. Initial text can be specified by *str*. In the fifth form, you can specify the scroll bars that you want the control to have. *sBars* must be one of these values:

SCROLLBARS_BOTH	SCROLLBARS_NONE
SCROLLBARS_HORIZONTAL_ONLY	SCROLLBARS_VERTICAL_ONLY

TextArea is a subclass of **TextComponent**. Therefore, it supports the **getText( )**, **setText( )**, **getSelectedText( )**, **select( )**, **isEditable( )**, and **setEditable( )** methods described in the preceding section.

TextArea adds the following editing methods:

void append(String *str*)
void insert(String *str*, int *index*)
void replaceRange(String *str*, int *startIndex*, int *endIndex*)

The **append( )** method appends the string specified by *str* to the end of the current text. **insert( )** inserts the string passed in *str* at the specified index. To replace text, call **replaceRange( )**. It replaces the characters from *startIndex* to *endIndex*–1, with the replacement text passed in *str*.

Text areas are almost self-contained controls. Your program incurs virtually no management overhead. Normally, your program simply obtains the current text when it is needed. You can, however, listen for **TextEvent**s, if you choose.

The following program creates a **TextArea** control:

```
// Demonstrate TextArea.
import java.awt.*;
import java.applet.*;
/*
<applet code="TextAreaDemo" width=300 height=250>
</applet>
*/

public class TextAreaDemo extends Applet {
  public void init() {
    String val =
      "Java 8 is the latest version of the most\n" +
      "widely-used computer language for Internet programming.\n" +
      "Building on a rich heritage, Java has advanced both\n" +
      "the art and science of computer language design.\n\n" +
      "One of the reasons for Java's ongoing success is its\n" +
      "constant, steady rate of evolution. Java has never stood\n" +
      "still. Instead, Java has consistently adapted to the\n" +
      "rapidly changing landscape of the networked world.\n" +
      "Moreover, Java has often led the way, charting the\n" +
      "course for others to follow.";

    TextArea text = new TextArea(val, 10, 30);
    add(text);
  }
}
```

Here is sample output from the **TextAreaDemo** applet:

Understanding Layout Managers

All of the components that we have shown so far have been positioned by the default layout manager. As we mentioned at the beginning of this chapter, a layout manager automatically arranges your controls within a window by using some type of algorithm. If you have

programmed for other GUI environments, such as Windows, then you may have laid out your controls by hand. While it is possible to lay out Java controls by hand, too, you generally won't want to, for two main reasons. First, it is very tedious to manually lay out a large number of components. Second, sometimes the width and height information is not yet available when you need to arrange some control, because the native toolkit components haven't been realized. This is a chicken-and-egg situation; it is pretty confusing to figure out when it is okay to use the size of a given component to position it relative to another.

Each **Container** object has a layout manager associated with it. A layout manager is an instance of any class that implements the **LayoutManager** interface. The layout manager is set by the **setLayout( )** method. If no call to **setLayout( )** is made, then the default layout manager is used. Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it.

The **setLayout( )** method has the following general form:

```
void setLayout(LayoutManager layoutObj)
```

Here, *layoutObj* is a reference to the desired layout manager. If you wish to disable the layout manager and position components manually, pass **null** for *layoutObj*. If you do this, you will need to determine the shape and position of each component manually, using the **setBounds( )** method defined by **Component**. Normally, you will want to use a layout manager.

Each layout manager keeps track of a list of components that are stored by their names. The layout manager is notified each time you add a component to a container. Whenever the container needs to be resized, the layout manager is consulted via its **minimumLayoutSize( )** and **preferredLayoutSize( )** methods. Each component that is being managed by a layout manager contains the **getPreferredSize( )** and **getMinimumSize( )** methods. These return the preferred and minimum size required to display each component. The layout manager will honor these requests if at all possible, while maintaining the integrity of the layout policy. You may override these methods for controls that you subclass. Default values are provided otherwise.

Java has several predefined **LayoutManager** classes, several of which are described next. You can use the layout manager that best fits your application.

FlowLayout

FlowLayout is the default layout manager. This is the layout manager that the preceding examples have used. **FlowLayout** implements a simple layout style, which is similar to how words flow in a text editor. The direction of the layout is governed by the container's component orientation property, which, by default, is left to right, top to bottom. Therefore, by default, components are laid out line-by-line beginning at the upper-left corner. In all cases, when a line is filled, layout advances to the next line. A small space is left between each component, above and below, as well as left and right. Here are the constructors for **FlowLayout**:

```
FlowLayout( )
FlowLayout(int how)
FlowLayout(int how, int horz, int vert)
```

The first form creates the default layout, which centers components and leaves five pixels of space between each component. The second form lets you specify how each line is aligned. Valid values for *how* are as follows:

FlowLayout.LEFT
FlowLayout.CENTER
FlowLayout.RIGHT
FlowLayout.LEADING
FlowLayout.TRAILING

These values specify left, center, right, leading edge, and trailing edge alignment, respectively. The third constructor allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

Here is a version of the **CheckboxDemo** applet shown earlier in this chapter, modified so that it uses left-aligned flow layout:

```
// Use left-aligned flow layout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="FlowLayoutDemo" width=240 height=200>
  </applet>
*/

public class FlowLayoutDemo extends Applet
  implements ItemListener {

  String msg = "";
  Checkbox windows, android, solaris, mac;

  public void init() {
    // set left-aligned flow layout
    setLayout(new FlowLayout(FlowLayout.LEFT));

    windows = new Checkbox("Windows", null, true);
    android = new Checkbox("Android");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("Mac OS");

    add(windows);
    add(android);
    add(solaris);
    add(mac);

    // register to receive item events
    windows.addItemListener(this);
    android.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
  }
```

```
  // Repaint when status of a check box changes.
  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Display current state of the check boxes.
  public void paint(Graphics g) {

    msg = "Current state: ";
    g.drawString(msg, 6, 80);
    msg = "  Windows: " + windows.getState();
    g.drawString(msg, 6, 100);
    msg = "  Android: " + android.getState();
    g.drawString(msg, 6, 120);
    msg = "  Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = "  Mac: " + mac.getState();
    g.drawString(msg, 6, 160);
  }
}
```

Here is sample output generated by the **FlowLayoutDemo** applet. Compare this with the output from the **CheckboxDemo** applet, shown earlier in Figure 26-2.

BorderLayout

The **BorderLayout** class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north, south, east, and west. The middle area is called the center. Here are the constructors defined by **BorderLayout**:

BorderLayout( )
BorderLayout(int *horz*, int *vert*)

The first form creates a default border layout. The second allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

BorderLayout defines the following constants that specify the regions:

BorderLayout.CENTER	BorderLayout.SOUTH
BorderLayout.EAST	BorderLayout.WEST
BorderLayout.NORTH	

When adding components, you will use these constants with the following form of **add( )**, which is defined by **Container**:

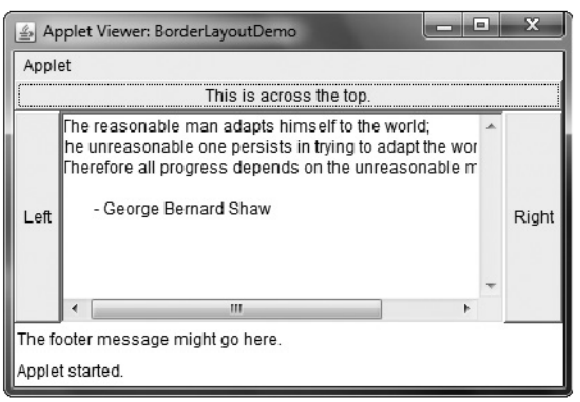
void add(Component *compRef*, Object *region*)

Here, *compRef* is a reference to the component to be added, and *region* specifies where the component will be added.

Here is an example of a **BorderLayout** with a component in each layout area:

```
// Demonstrate BorderLayout.
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code="BorderLayoutDemo" width=400 height=200>
</applet>
*/

public class BorderLayoutDemo extends Applet {
  public void init() {
    setLayout(new BorderLayout());

    add(new Button("This is across the top."),
        BorderLayout.NORTH);
    add(new Label("The footer message might go here."),
        BorderLayout.SOUTH);
    add(new Button("Right"), BorderLayout.EAST);
    add(new Button("Left"), BorderLayout.WEST);

    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 man.\n\n" +
      "        - George Bernard Shaw\n\n";

    add(new TextArea(msg), BorderLayout.CENTER);
  }
}
```

Sample output from the **BorderLayoutDemo** applet is shown here:



Using Insets

Sometimes you will want to leave a small amount of space between the container that holds your components and the window that contains it. To do this, override the **getInsets( )** method that is defined by **Container**. This method returns an **Insets** object that contains the top, bottom, left, and right inset to be used when the container is displayed. These values are used by the layout manager to inset the components when it lays out the window. The constructor for **Insets** is shown here:

Insets(int *top*, int *left*, int *bottom*, int *right*)

The values passed in *top*, *left*, *bottom*, and *right* specify the amount of space between the container and its enclosing window.

The **getInsets( )** method has this general form:

Insets getInsets( )

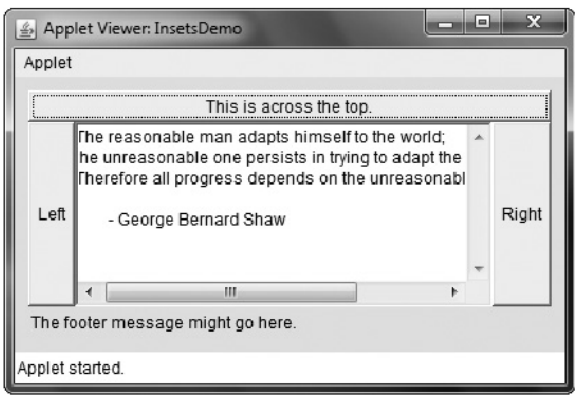
When overriding this method, you must return a new **Insets** object that contains the inset spacing you desire.

Here is the preceding **BorderLayout** example modified so that it insets its components ten pixels from each border. The background color has been set to cyan to help make the insets more visible.

```
// Demonstrate BorderLayout with insets.
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code="InsetsDemo" width=400 height=200>
</applet>
*/

public class InsetsDemo extends Applet {
  public void init() {
    // set background color so insets can be easily seen
    setBackground(Color.cyan);

    setLayout(new BorderLayout());

    add(new Button("This is across the top."),
        BorderLayout.NORTH);
    add(new Label("The footer message might go here."),
        BorderLayout.SOUTH);
    add(new Button("Right"), BorderLayout.EAST);
    add(new Button("Left"), BorderLayout.WEST);

    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 man.\n\n" +
      "        - George Bernard Shaw\n\n";
```

```
    add(new TextArea(msg), BorderLayout.CENTER);
  }

  // add insets
  public Insets getInsets() {
    return new Insets(10, 10, 10, 10);
  }
}
```

Sample output from the **InsetsDemo** applet is shown here:



GridLayout

GridLayout lays out components in a two-dimensional grid. When you instantiate a **GridLayout**, you define the number of rows and columns. The constructors supported by **GridLayout** are shown here:

GridLayout( )
GridLayout(int *numRows*, int *numColumns*)
GridLayout(int *numRows*, int *numColumns*, int *horz*, int *vert*)

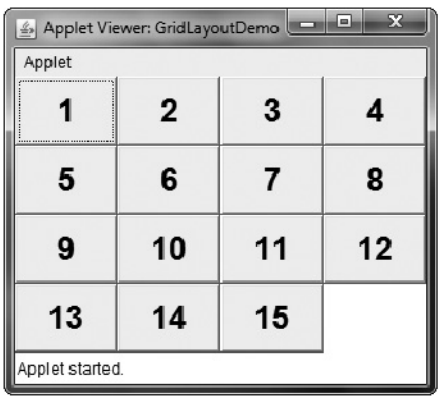
The first form creates a single-column grid layout. The second form creates a grid layout with the specified number of rows and columns. The third form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. Either *numRows* or *numColumns* can be zero. Specifying *numRows* as zero allows for unlimited-length columns. Specifying *numColumns* as zero allows for unlimited-length rows.

Here is a sample program that creates a 4×4 grid and fills it in with 15 buttons, each labeled with its index:

```
// Demonstrate GridLayout
import java.awt.*;
import java.applet.*;
/*
<applet code="GridLayoutDemo" width=300 height=200>
</applet>
*/
```

```
public class GridLayoutDemo extends Applet {
  static final int n = 4;
  public void init() {
    setLayout(new GridLayout(n, n));

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

    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));
      }
    }
  }
}
```

Following is sample output generated by the **GridLayoutDemo** applet:



TIP You might try using this example as the starting point for a 15-square puzzle.

CardLayout

The **CardLayout** class is unique among the other layout managers in that it stores several different layouts. Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time. This can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input. You can prepare the other layouts and have them hidden, ready to be activated when needed.

CardLayout provides these two constructors:

CardLayout( )
CardLayout(int *horz*, int *vert*)

The first form creates a default card layout. The second form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

Use of a card layout requires a bit more work than the other layouts. The cards are typically held in an object of type **Panel**. This panel must have **CardLayout** selected as its layout manager. The cards that form the deck are also typically objects of type **Panel**. Thus, you must create a panel that contains the deck and a panel for each card in the deck. Next, you add to the appropriate panel the components that form each card. You then add these panels to the panel for which **CardLayout** is the layout manager. Finally, you add this panel to the window. Once these steps are complete, you must provide some way for the user to select between cards. One common approach is to include one push button for each card in the deck.

When card panels are added to a panel, they are usually given a name. Thus, most of the time, you will use this form of **add( )** when adding cards to a panel:

void add(Component *panelRef*, Object *name*)

Here, *name* is a string that specifies the name of the card whose panel is specified by *panelRef*.

After you have created a deck, your program activates a card by calling one of the following methods defined by **CardLayout**:

void first(Container *deck*)
void last(Container *deck*)
void next(Container *deck*)
void previous(Container *deck*)
void show(Container *deck*, String *cardName*)

Here, *deck* is a reference to the container (usually a panel) that holds the cards, and *cardName* is the name of a card. Calling **first( )** causes the first card in the deck to be shown. To show the last card, call **last( )**. To show the next card, call **next( )**. To show the previous card, call **previous( )**. Both **next( )** and **previous( )** automatically cycle back to the top or bottom of the deck, respectively. The **show( )** method displays the card whose name is passed in *cardName*.

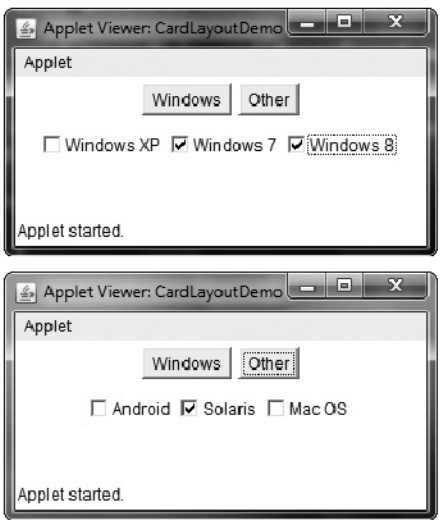
The following example creates a two-level card deck that allows the user to select an operating system. Windows-based operating systems are displayed in one card. Mac OS and Solaris are displayed in the other card.

```
// Demonstrate CardLayout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="CardLayoutDemo" width=300 height=100>
  </applet>
*/

public class CardLayoutDemo extends Applet
  implements ActionListener, MouseListener {
```

```
Checkbox windowsXP, windows7, windows8, android, solaris, mac;
Panel osCards;
CardLayout cardLO;
Button Win, Other;

public void init() {
  Win = new Button("Windows");
  Other = new Button("Other");
  add(Win);
  add(Other);

  cardLO = new CardLayout();
  osCards = new Panel();
  osCards.setLayout(cardLO); // set panel layout to card layout

  windowsXP = new Checkbox("Windows XP", null, true);
  windows7 = new Checkbox("Windows 7", null, false);
  windows8 = new Checkbox("Windows 8", null, false);
  android = new Checkbox("Android");
  solaris = new Checkbox("Solaris");
  mac = new Checkbox("Mac OS");

  // add Windows check boxes to a panel
  Panel winPan = new Panel();
  winPan.add(windowsXP);
  winPan.add(windows7);
  winPan.add(windows8);

  // Add other OS check boxes to a panel
  Panel otherPan = new Panel();
  otherPan.add(android);
  otherPan.add(solaris);
  otherPan.add(mac);

  // add panels to card deck panel
  osCards.add(winPan, "Windows");
  osCards.add(otherPan, "Other");

  // add cards to main applet panel
  add(osCards);

  // register to receive action events
  Win.addActionListener(this);
  Other.addActionListener(this);

  // register mouse events
  addMouseListener(this);
}

// Cycle through panels.
public void mousePressed(MouseEvent me) {
  cardLO.next(osCards);
}
```

```
  // Provide empty implementations for the other MouseListener methods.
  public void mouseClicked(MouseEvent me) {
  }
  public void mouseEntered(MouseEvent me) {
  }
  public void mouseExited(MouseEvent me) {
  }
  public void mouseReleased(MouseEvent me) {
  }

  public void actionPerformed(ActionEvent ae) {
    if(ae.getSource() == Win) {

      cardLO.show(osCards, "Windows");
    }
    else {
      cardLO.show(osCards, "Other");
    }
  }
}
```

Here is sample output generated by the **CardLayoutDemo** applet. Each card is activated by pushing its button. You can also cycle through the cards by clicking the mouse.



GridBagLayout

Although the preceding layouts are perfectly acceptable for many uses, some situations will require that you take a bit more control over how the components are arranged. A good way to do this is to use a grid bag layout, which is specified by the **GridBagLayout** class. What makes the grid bag useful is that you can specify the relative placement of components by specifying their positions within cells inside a grid. The key to the grid bag is that each component can be a different size, and each row in the grid can have a different number