This chapter starts you on the road to writing Java programs that use a graphical user interface (GUI). In particular, you will learn how to write programs that size and locate windows on the screen, display text with multiple fonts in a window, display images, and so on.

1.1 Introducing Swing

- When Java 1.0 was introduced, it contained a class library, which Sun called the Abstract Window Toolkit (AWT), for basic GUI programming.
- In theory, run on any of these platforms, with the “look-and-feel” of the target platform—hence Sun’s trademarked slogan: “Write Once, Run Anywhere.”
- Developers complained that they had to test their applications on each platform—a practice called “write once, debug everywhere.”
- In 1996, Netscape created a GUI library they called the IFC (Internet Foundation Classes) that used an entirely different approach. User interface elements, such as buttons, menus, and so on, were *painted* onto blank windows.
- Sun worked with Netscape to perfect this approach, creating a user interface library with the code name “Swing.” Swing was available as an extension to Java 1.1 and became a part of the standard library in Java SE 1.2.
- Swing is now the official name for the non-peer-based GUI toolkit.
- Swing is part of the Java Foundation Classes (JFC).
- Swing-based user interface elements will be somewhat **slower** to appear on the user’s screen than the peer-based components used by the AWT.

- On the other hand, the reasons to choose Swing are overwhelming:
  - Swing has a rich and suitable set of user interface elements.
  - Swing has few dependencies on the underlying platform; it is therefore less liable to platform-specific bugs.
  - Swing gives a compatible user experience across platforms.

  *Still, the third plus is also a potential drawback: If the user interface elements look the same on all platforms, they look **different** from the native controls, so users will be less familiar with them.*

1.2 Creating a Frame

A top-level window (that is, a window that is not contained inside another window) is called a *frame* in Java. The Abstract Window Toolkit (AWT) library has a class, called `Frame`, for this top level. The Swing version of this class is called `JFrame` and extends the `Frame` class. Thus, the decorations (buttons, title bar, icons, and so on) are drawn by the user’s windowing system, not by Swing.

What is JFrame?

JFrame is a class of javax.swing package extended by java.awt.frame, it adds support for JFC/SWING component architecture. It is the top level window, with border and a title bar. JFrame class has many methods which can be used to customize it.

Creating a JFrame

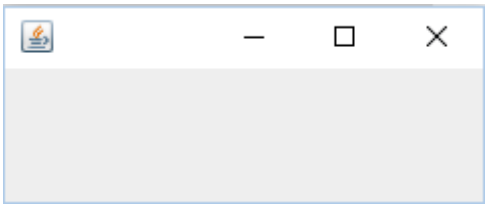
JFrame class has many constructors used to create a JFrame. Following is the description.

- ✓ **JFrame():** creates a frame which is invisible
- ✓ **JFrame(GraphicsConfiguration gc):** creates a frame with a blank title and graphics configuration of screen device.
- ✓ **JFrame(String title):** creates a JFrame with a title.
- ✓ **JFrame(String title, GraphicsConfiguration gc):** creates a JFrame with specific Graphics configuration and specified title.

Here is a simplest example just to create a JFrame.

```
package Example;

import java.awt.GraphicsConfiguration;

import javax.swing.JFrame;

public class JFrameExample {

    static GraphicsConfiguration gc;
    public static void main(String[] args){
        JFrame frame= new JFrame(gc);
        frame.setVisible(true);
    }
}
```

Here is how it will display



Change window size of a JFrame

To resize a frame, JFrame provides a method JFrame.setSize(int width, int height), it takes two parameters width and height. Here is how code looks now

```
package Example;

import java.awt.GraphicsConfiguration;

import javax.swing.JFrame;

public class JFrameExample {

    static GraphicsConfiguration gc;
    public static void main(String[] args){
        JFrame frame= new JFrame(gc);
        frame.setTitle("Welecome to Advance Java Programming");
        frame.setSize(600, 400);
        frame.setVisible(true);
    }
}
```

Resize a JFrame

After setting size of a JFrame you will notice still change it size by just simply putting the cursor at the corners and dragging it. Or if you press resize option next to close at the top right corner, it will maximize to the size of full screen. This happens because resize *is set true by default.* You can simply make false as

> JFrame.setResizable(false), now it will appear according to the dimensions you have given in code and will not resize by the graphical interface.

Change position on the screen

To change position of JFrame on screen JFrame provides a method JFrame.setlocation(int x, int y), it takes two paramters x represents position along x-axis and y represents position along y-axis. The top left corner of your screen is (0,0).

Closing a JFrame

You can easily close your JFrame by clicking on the X(cross) at the top left corner of JFrame. However JFrame.setDefaultCloseOperation(int) is a method provided by JFrame class, you can set the operation that will happen when user clicks on cross. If “0” is given as a parameter, JFrame will not close even after clicking on cross.

The best practice is to use JFrame.EXIT_ON_CLOSE, it exits application (JFrame) and releases memory.

JFrame.HIDE_ON_CLOSE: It does not close JFrame, simply hides it.

JFrame.DISPOSE_ON_CLOSE: It dispose the frame off, but it keeps running and consumes memory.

JFrame.DO_NOTHING_ON_CLOSE: It does nothing when user clicks on close.

Here’s how the final code looks like

```
package Example;

import java.awt.GraphicsConfiguration;

import javax.swing.JFrame;

public class JFrameExample {

    static GraphicsConfiguration gc;
    public static void main(String[] args){
        JFrame frame= new JFrame(gc);
        frame.setTitle("Welecome to Advance Java Programming");
        frame.setSize(600, 400);
        frame.setLocation(200, 200);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setResizable(false);
    }
}
```

Listing 10.1 simpleframe/SimpleFrameTest.java

```
1  package simpleFrame;
2
3  import java.awt.*;
4  import javax.swing.*;
5
6  /**
7   * @version 1.33 2015-05-12
8   * @author Cay Horstmann
9   */
10 public class SimpleFrameTest
11 {
12    public static void main(String[] args)
13    {
14       EventQueue.invokeLater(() ->
15          {
16             SimpleFrame frame = new SimpleFrame();
17             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18             frame.setVisible(true);
19          });
20    }
21 }
22
23 class SimpleFrame extends JFrame
24 {
25    private static final int DEFAULT_WIDTH = 300;
26    private static final int DEFAULT_HEIGHT = 200;
27
28    public SimpleFrame()
29    {
30       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
31    }
32 }
```

There are two technical issues that we need to address in every Swing program:

First, all Swing components must be configured from the *event dispatch thread*, the thread of control that passes events such as mouse clicks and keystrokes to the user interface components. The following code fragment is used to execute statements in the event dispatch thread:

```
EventQueue.invokeLater(() ->
{
statements
});
```

Next, we define what should happen when the user closes the application's frame.

For this particular program, we want the program to exit. To select this behavior, we use the statement

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

In other programs with multiple frames, you would not want the program to exit just because the user closes one of the frames. By default, a frame is hidden when the user closes it, but the program does not terminate.

1.3 Positioning a Frame

The `JFrame` class itself has only a few methods for changing how frames look.
Here are some of the most important methods:

- The `setLocation` and `setBounds` methods for setting the position of the frame
- The `setIconImage` method, which tells the windowing system which icon to display in the title bar, task switcher window, and so on
- The `setTitle` method for changing the text in the title bar
- The `setResizable` method, which takes a `boolean` to determine if a frame will be resizeable by the user

Figure 1.6 illustrates the inheritance hierarchy for the `JFrame` class.

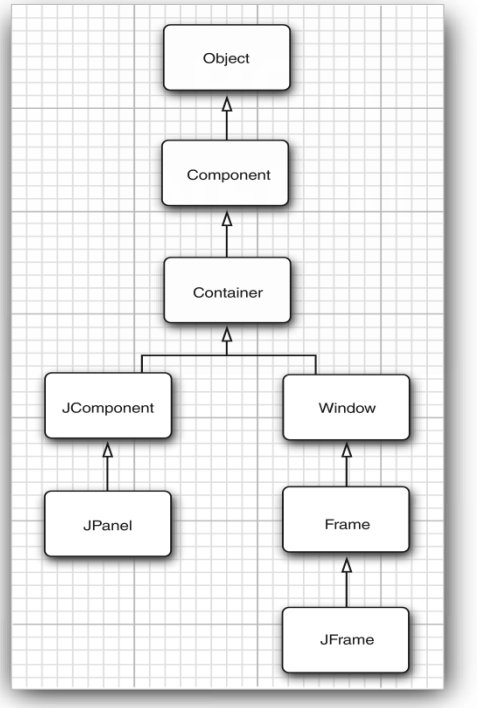


Figure 1.6 Inheritance hierarchy for the frame and component classes in AWT and Swing

`setLocation(x, y)` where (0, 0) is the top left corner of the screen.

Similarly, the `setBounds` method in `Component` lets you resize and relocate a component (in particular, a `JFrame`) in one step, as

```
setBounds(x, y, width, height)
```

Alternatively, you can give the windowing system control over window placement. If you call

```
setLocationByPlatform(true);
```

1.3.1 Frame Properties

Many methods of component classes come in getter/setter pairs, such as the following methods of the `Frame` class:

```
public String getTitle()
public void setTitle(String title)
```

Such a getter/setter pair is called a *property*.

For example, the following two methods define the `locationByPlatform` property:

```
public boolean isLocationByPlatform()
public void setLocationByPlatform(boolean b)
```

1.3.2 Determining a Good Frame Size

To find out the screen size, use the following steps.

```
Toolkit kit = Toolkit.getDefaultToolkit();
Dimension screenSize = kit.getScreenSize();
int screenWidth = screenSize.width;
int screenHeight = screenSize.height;
```

We use 50% of these values for the frame size, and tell the windowing system to position the frame:

```
setSize(screenWidth / 2, screenHeight / 2);
setLocationByPlatform(true);
```

We also supply an icon. The `ImageIcon` class is convenient for loading images. Here is how you use it:

```
Image img = new ImageIcon("icon.gif").getImage();
setIconImage(img);
```

You can maximize a frame by calling

```
frame.setExtendedState(Frame.MAXIMIZED_BOTH);
```

Listing 1.2 `sizedFrame/SizedFrameTest.java`

```
1  package sizedFrame;
2
3  import java.awt.*;
4  import javax.swing.*;
5
6  /**
7   * @version 1.33 2007-05-12
8   * @author Cay Horstmann
9   */
10 public class SizedFrameTest
11 {
12    public static void main(String[] args)
13    {
14       EventQueue.invokeLater(() ->
15          {
16             JFrame frame = new SizedFrame();
17             frame.setTitle("SizedFrame");
18             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19             frame.setVisible(true);
20          });
21    }
22 }
23
24 class SizedFrame extends JFrame
25 {
26    public SizedFrame()
27    {
28       // get screen dimensions
29
30       Toolkit kit = Toolkit.getDefaultToolkit();
31       Dimension screenSize = kit.getScreenSize();
32       int screenHeight = screenSize.height;
33       int screenWidth = screenSize.width;
34
35       // set frame width, height and let platform pick screen location
36
37       setSize(screenWidth / 2, screenHeight / 2);
38       setLocationByPlatform(true);
39
40       // set frame icon
41
42       Image img = new ImageIcon("icon.gif").getImage();
43       setIconImage(img);
44    }
45 }
```

1.4 Displaying Information in a Component

In this section, we will show you how to display information inside a frame.
For example, instead of displaying “Not a Hello World program” in text mode in a console window

Figure 1.7 A frame that displays information

When designing a frame, you add components into the content pane, using code such as the following:

```
Container contentPane = frame.getContentPane();
Component c = . . .;
contentPane.add(c);
```

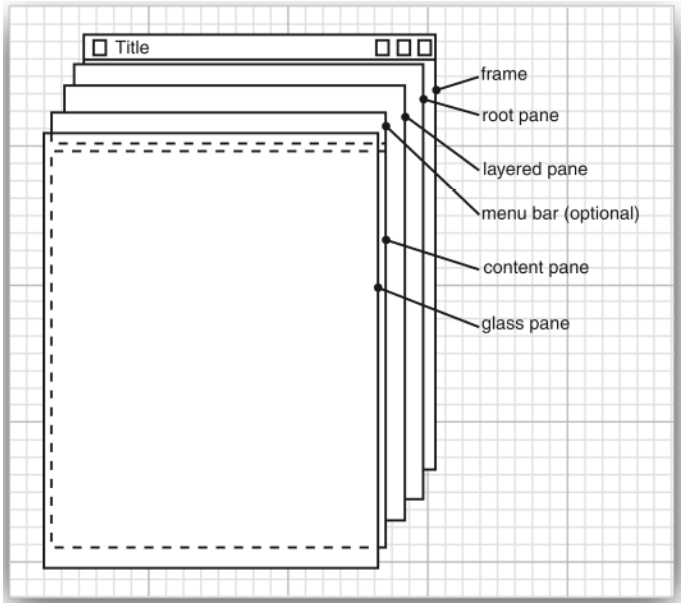


Figure 1.8 Internal structure of a JFrame

Up to Java Standard Edition (SE) 1.4, the `add` method of the `JFrame` class was defined to throw an exception with the message "Do not use `JFrame.add()`. Use `JFrame.getContentPane().add()` instead."

Thus, you can simply use the call

```
frame.add(c);
```

Here's how to make a component onto which we can draw:

```
class MyComponent extends JComponent
{
       public void paintComponent(Graphics g)
       {
       code for drawing
       }
}
```

The `paintComponent` method takes one parameter of type `Graphics`. A `Graphics` object remembers a collection of settings for drawing images and text, such as the font you set or the current color.

Displaying text is considered a special kind of drawing. The `Graphics` class has a `drawString` method that has the following syntax:

```
g.drawString(text, x, y)
```

In our case, we want to draw the string `"Not a Hello World Program"` in our original window, roughly one-quarter of the way across and halfway down.

Thus, our `paintComponent` method looks like this:

```
public class NotHelloWorldComponent extends JComponent
{
       public static final int MESSAGE_X = 75;
       public static final int MESSAGE_Y = 100;
       public void paintComponent(Graphics g)
       {
       g.drawString("Not a Hello World program", MESSAGE_X, MESSAGE_Y);
       }
       . . .
}
```

Finally, a component should tell its users how big it would like to be. Override the `getPreferredSize` method and return an object of the `Dimension` class with the preferred width and height:

```
public class NotHelloWorldComponent extends JComponent
{
```

```
        private static final int DEFAULT_WIDTH = 300;
        private static final int DEFAULT_HEIGHT = 200;
        . . .
        public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
        }
}
```

When you fill a frame with one or more components, and you simply want to use their preferred size, call the `pack` method instead of the `setSize` method:

```
class NotHelloWorldFrame extends JFrame
{
        public NotHelloWorldFrame()
        {
                add(new NotHelloWorldComponent());
                pack();
        }
}
```

Listing 10.3 notHelloWorld/NotHelloWorld.java

```
1  package notHelloWorld;
2
3  import javax.swing.*;
4  import java.awt.*;
5
6  /**
7   * @version 1.33 2015-05-12
8   * @author Cay Horstmann
9   */
10 public class NotHelloWorld
11 {
12    public static void main(String[] args)
13    {
14       EventQueue.invokeLater(() ->
15          {
16             JFrame frame = new NotHelloWorldFrame();
17             frame.setTitle("NotHelloWorld");
18             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19             frame.setVisible(true);
20          });
21    }
22 }
23
24 /**
25  * A frame that contains a message panel
26  */
```

```
27 class NotHelloWorldFrame extends JFrame
28 {
29    public NotHelloWorldFrame()
30    {
31       add(new NotHelloWorldComponent());
32       pack();
33    }
34 }
35
36 /**
37  * A component that displays a message.
38  */
39 class NotHelloWorldComponent extends JComponent
40 {
41    public static final int MESSAGE_X = 75;
42    public static final int MESSAGE_Y = 100;
43
44    private static final int DEFAULT_WIDTH = 300;
45    private static final int DEFAULT_HEIGHT = 200;
46
47    public void paintComponent(Graphics g)
48    {
49       g.drawString("Not a Hello, World program", MESSAGE_X, MESSAGE_Y);
50    }
51
52    public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT); }
53 }
```

1.5 Working with 2D Shapes

Starting with Java 1.0, the `Graphics` class has methods to draw lines, rectangles, ellipses, and so on. For example, you cannot vary the line thickness and cannot rotate the shapes.

Java SE 1.2 introduced the *Java 2D* library, which implements a powerful set of graphical operations.

To draw shapes in the Java 2D library, you need to obtain an object of the `Graphics2D` class. This class is a subclass of the `Graphics` class. Ever since Java SE 2, methods such as `paintComponent` automatically receive an object of the `Graphics2D` class. Simply use a cast, as follows:

```
public void paintComponent(Graphics g)
{
Graphics2D g2 = (Graphics2D) g;
. . .
}
```

The Java 2D library organizes geometric shapes in an object-oriented fashion.

```
Line2D
Rectangle2D
Ellipse2D
```

These classes all implement the `Shape` interface.

To draw a shape, you first create an object of a class that implements the `Shape` interface and then call the `draw` method of the `Graphics2D` class. For example:

```
Rectangle2D rect = . . .;
g2.draw(rect);
```

Using the Java 2D shape classes introduces some complexity.

For example, consider the following statement:

```
float f = 1.2; // Error
```

This statement does not compile because the constant `1.2` has type `double`, and the compiler is nervous about loss of precision. The remedy is to add an `F` suffix to the floating-point constant:

```
float f = 1.2F; // Ok
```

Now consider this statement:

```
Rectangle2D r = . . .
float f = r.getWidth(); // Error
```

This statement does not compile either, for the same reason. The `getWidth` method returns a `double`. This time, the remedy is to provide a cast:

```
float f = (float) r.getWidth(); // OK
```

These suffixes and casts are a bit of a pain, so the designers of the 2D library decided to supply *two versions* of each shape class:

Figure 10.9 shows the inheritance diagram.

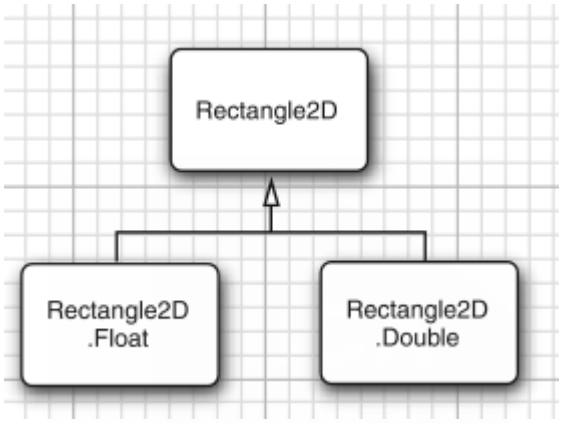


Figure 10.9 2D rectangle classes

It is best to ignore the fact that the two concrete classes are static inner classes:
When you construct both :

```
Rectangle2D.Float floatRect = new Rectangle2D.Float(10.0F, 25.0F, 22.5F, 20.0F);
Rectangle2D.Double doubleRect = new Rectangle2D.Double(10.0, 25.0, 22.5, 20.0);
```

It is extend the common `Rectangle2D` class and the methods in the subclasses simply override those in the `Rectangle2D` superclass, You can simply use `Rectangle2D` variables to hold the rectangle references.

```
Rectangle2D floatRect = new Rectangle2D.Float(10.0F, 25.0F, 22.5F, 20.0F);
Rectangle2D doubleRect = new Rectangle2D.Double(10.0, 25.0, 22.5, 20.0);
```

Furthermore, there is a `Point2D` class with subclasses `Point2D.Float` and `Point2D.Double`. Here is how to make a point object:

```
Point2D p = new Point2D.Double(10, 20);
```

The classes `Rectangle2D` and `Ellipse2D` both inherit from the common superclass `RectangularShape`. Admittedly, ellipses are not rectangular, but they have a *bounding rectangle* (see Figure 10.10).

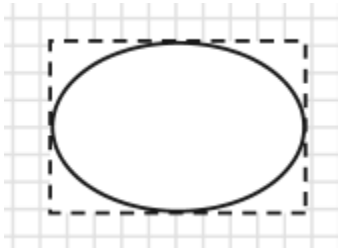


Figure 10.10 The bounding rectangle of an ellipse

`Rectangle2D` and `Ellipse2D` objects are simple to construct. You need to specify

- The *x* and *y* coordinates of the top left corner; and
- The width and height.

For ellipses, these refer to the bounding rectangle. For example,

```
Ellipse2D e = new Ellipse2D.Double(150, 200, 100, 50);
```

constructs an ellipse that is bounded by a rectangle with the top left corner at (150, 200), width of 100, and height of 50.

You can't simply construct a rectangle as

```
Rectangle2D rect = new Rectangle2D.Double(px, py, qx - px, qy - py); // Error
```

In that case, first create a blank rectangle and use the `setFrameFromDiagonal` method, as follows:

```
Rectangle2D rect = new Rectangle2D.Double();
rect.setFrameFromDiagonal(px, py, qx, qy);
```

Or, even better, if you have the corner points as `Point2D` objects `p` and `q`, use

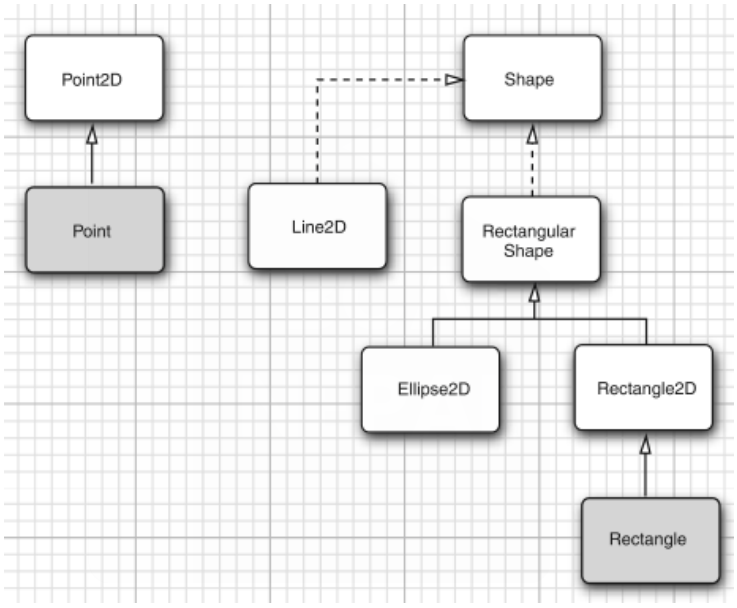


Figure 10.11 Relationships between the shape classes

```
rect.setFrameFromDiagonal(p, q);
```

When constructing an ellipse, you usually know the center, width, and height, but not the corner points of the bounding rectangle (which don't even lie on the ellipse).

The `setFrameFromCenter` method uses the center point, but it still requires one of the four corner points. Thus, you will usually end up constructing an ellipse as follows:

```
Ellipse2D ellipse = new Ellipse2D.Double(centerX - width / 2, centerY - height / 2, width, height);
```

To construct a line, you supply the start and end points, either as `Point2D` objects or as pairs of numbers:

```
Line2D line = new Line2D.Double(start, end);
```

or

```
Line2D line = new Line2D.Double(startX, startY, endX, endY);
```

The program in Listing 10.4 draws a rectangle, the ellipse that is enclosed in the rectangle, a diagonal of the rectangle, and a circle that has the same center as the rectangle. Figure 10.12 shows the result.

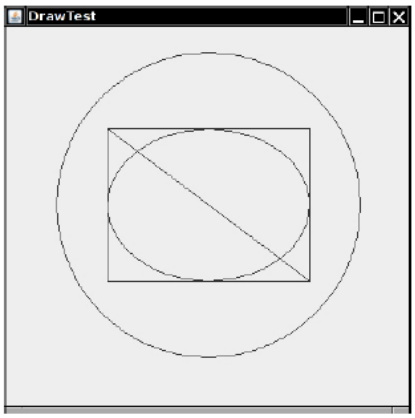


Figure 10.12 Drawing geometric shapes (Page.566)

1.6 Using Color (page. 569)

The `setPaint` method of the `Graphics2D` class lets you select a color that is used for all subsequent drawing operations on the graphics context. For example:

```
g2.setPaint(Color.RED);
g2.drawString("Warning!", 100, 100);
```

You can fill the interiors of closed shapes (such as rectangles or ellipses) with a color. Simply call `fill` instead of `draw`:

```
Rectangle2D rect = . . .;
g2.setPaint(Color.RED);
g2.fill(rect); // fills rect with red
```

Define colors with the `Color` class. The `java.awt.Color` class offers predefined constants for the following 13 standard colors:

```
BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY, MAGENTA, ORANGE, PINK,
RED, WHITE, YELLOW
```

Using a scale of 0–255 (that is, one byte) for the redness, blueness, and greenness, call the `Color` constructor like this:

```
Color(int redness, int greenness, int blueness)
```

Here is an example of setting a custom color:

```
g2.setPaint(new Color(0, 128, 128)); // a dull blue-green
g2.drawString("Welcome!", 75, 125);
```

To set the *background color*, use the `setBackground` method of the `Component` class, an ancestor of `JComponent`.

```
MyComponent p = new MyComponent();
p.setBackground(Color.PINK);
```

There is also a `setForeground` method. It specifies the default color that is used for drawing on the component.

```
p.setBackground(SystemColor.window)  //default color of window
```

Table 10.1 lists the system color names and their meanings.

Table 10.1 System Colors

Name	Purpose
desktop	Background color of desktop
activeCaption	Background color for captions
activeCaptionText	Text color for captions
activeCaptionBorder	Border color for caption text
inactiveCaption	Background color for inactive captions
inactiveCaptionText	Text color for inactive captions
inactiveCaptionBorder	Border color for inactive captions
window	Background for windows
windowBorder	Color of window border frame
windowText	Text color inside windows
menu	Background for menus
menuText	Text color for menus
text	Background color for text
textText	Text color for text
textInactiveText	Text color for inactive controls
textHighlight	Background color for highlighted text
textHighlightText	Text color for highlighted text
control	Background color for controls
controlText	Text color for controls
controlLtHighlight	Light highlight color for controls
controlHighlight	Highlight color for controls
controlShadow	Shadow color for controls
controlDkShadow	Dark shadow color for controls
scrollbar	Background color for scrollbars
info	Background color for spot-help text
infoText	Text color for spot-help text

10.7 Using Special Fonts for Text

To find out which fonts are available on a particular computer, call the `getAvailableFontFamilyNames` method of the `GraphicsEnvironment` class.

The method returns an array of strings containing the names of all available fonts.

To obtain an instance of the `GraphicsEnvironment` class that describes the graphics environment of the user's system, use the static `getLocalGraphicsEnvironment` method.

The following program prints the names of all fonts on your system:

```
import java.awt.*;

public class ListFonts
{
   public static void main(String[] args)
   {
      String[] fontNames = GraphicsEnvironment
         .getLocalGraphicsEnvironment()
         .getAvailableFontFamilyNames();
      for (String fontName : fontNames)
         System.out.println(fontName);
   }
}
```

On one system, the list starts out like this:

```
Abadi MT Condensed Light
Arial
Arial Black
Arial Narrow
Arioso
Baskerville
Binner Gothic
. . .
```

and goes on for another seventy or so fonts.

Font face names can be trademarked, and font designs can be copyrighted in some jurisdictions.

To draw characters in a font, you must first create an object of the class `Font`.

Specify the font face name, the font style, and the point size. Here is an example of how you construct a `Font` object:

```
Font sansbold14 = new Font("SansSerif", Font.BOLD, 14);
```

Specify the style (plain, **bold**, *italic*, or ***bold italic***) by setting the second `Font` constructor argument to one of the following values:

```
Font.PLAIN
Font.BOLD
Font.ITALIC
Font.BOLD + Font.ITALIC
```

You can read font files in TrueType, OpenType, or PostScript Type 1 formats.

You need an input stream for the font—typically from a file or URL.

Then, call the static `Font.createFont` method:

```
URL url = new URL("http://www.fonts.com/Wingbats.ttf");
InputStream in = url.openStream();
Font f1 = Font.createFont(Font.TRUETYPE_FONT, in);
```

The font is plain with a font size of 1 point. Use the `deriveFont` method to get a font of the desired size:

```
Font f = f1.deriveFont(14.0F);
```

The Java fonts contain the usual ASCII characters as well as symbols. For example, if you print the character `'\u2297'` in the Dialog font, you get a (x) character.

Here's the code that displays the string "Hello, World!" in the standard sans serif font on your system, using 14-point bold type:

```
Font sansbold14 = new Font("SansSerif", Font.BOLD, 14);
g2.setFont(sansbold14);
String message = "Hello, World!";
g2.drawString(message, 75, 100);
```

We need to know the width and height of the string in pixels. These dimensions depend on three factors:

- The font used (in our case, sans serif, bold, 14 point);
- The string (in our case, "Hello, World!"); and
- The device on which the font is drawn (in our case, the user's screen).

To obtain an object that represents the font characteristics of the screen device, call the `getFontRenderContext` method of the `Graphics2D` class. It returns an object of the `FontRenderContext` class. Simply pass that object to the `getStringBounds` method of the `Font` class:

```
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D bounds = sansbold14..getStringBounds(message, context);
```

The `getStringBounds` method returns a rectangle that encloses the string.

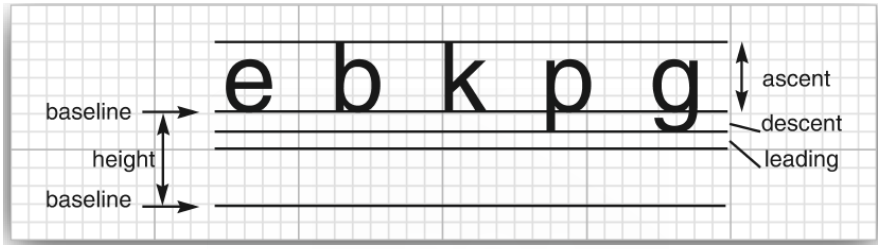


Figure 10.13 Typesetting terms illustrated

Thus, you can obtain string width, height, and ascent as follows:

```
double stringWidth = bounds.getWidth();
double stringHeight = bounds.getHeight();
double ascent = -bounds.getY();
```

If you need to know the descent or leading, use the `getLineMetrics` method of the `Font` class.

```
LineMetrics metrics = f.getLineMetrics(message, context);
float descent = metrics.getDescent();
float leading = metrics.getLeading();
```

The following code uses all this information to center a string in its surrounding component:

```
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D bounds = f.getStringBounds(message, context);
// (x,y) = top left corner of text
double x = (getWidth() - bounds.getWidth()) / 2;
double y = (getHeight() - bounds.getHeight()) / 2;
// add ascent to y to reach the baseline
double ascent = -bounds.getY();
double baseY = y + ascent;
g2.drawString(message, (int) x, (int) baseY);
```

To show that the positioning is accurate, the sample program also draws the baseline and the bounding rectangle. Figure 1.14 shows the screen display;

Figure 1.14 Drawing the baseline and string bounds

Listing 10.5 font/FontTest.java

```
1  package font;
2
3  import java.awt.*;
4  import java.awt.font.*;
5  import java.awt.geom.*;
6  import javax.swing.*;
7
8  /**
9   * @version 1.34 2015-05-12
10  * @author Cay Horstmann
11  */
12 public class FontTest
13 {
14    public static void main(String[] args)
15    {
16       EventQueue.invokeLater(() ->
17          {
18             JFrame frame = new FontFrame();
19             frame.setTitle("FontTest");
20             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21             frame.setVisible(true);
22          });
23    }
24 }
25
26 /**
27  * A frame with a text message component
28  */
```

```
29 class FontFrame extends JFrame
30 {
31    public FontFrame()
32    {
33       add(new FontComponent());
34       pack();
35    }
36 }
37
38 /**
39  * A component that shows a centered message in a box.
40  */
41 class FontComponent extends JComponent
42 {
43    private static final int DEFAULT_WIDTH = 300;
44    private static final int DEFAULT_HEIGHT = 200;
45
46    public void paintComponent(Graphics g)
47    {
48       Graphics2D g2 = (Graphics2D) g;
49
50       String message = "Hello, World!";
51
52       Font f = new Font("Serif", Font.BOLD, 36);
53       g2.setFont(f);
54
55       // measure the size of the message
56
57       FontRenderContext context = g2.getFontRenderContext();
58       Rectangle2D bounds = f.getStringBounds(message, context);
```

```
59
60          // set (x,y) = top left corner of text
61
62          double x = (getWidth() - bounds.getWidth()) / 2;
63          double y = (getHeight() - bounds.getHeight()) / 2;
64
65          // add ascent to y to reach the baseline
66
67          double ascent = -bounds.getY();
68          double baseY = y + ascent;
69
70          // draw the message
71
72          g2.drawString(message, (int) x, (int) baseY);
73
74          g2.setPaint(Color.LIGHT_GRAY);
75
76          // draw the baseline
77
78          g2.draw(new Line2D.Double(x, baseY, x + bounds.getWidth(), baseY));
79
80          // draw the enclosing rectangle
81
82          Rectangle2D rect = new Rectangle2D.Double(x, y, bounds.getWidth(), bounds.getHeight());
83          g2.draw(rect);
84       }
85
86       public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT); }
87    }
```

1.8 Displaying Images

Once images are stored in local files or someplace on the Internet, you can read them into a Java application and display them on `Graphics` objects. There are many ways of reading images. Here, we use the `ImageIcon` class that you already saw:

```
Image image = new ImageIcon(filename).getImage();
```

Now the variable `image` contains a reference to an object that encapsulates the image data. You can display the image with the `drawImage` method of the `Graphics` class.

```
public void paintComponent(Graphics g)
{
. . .
g.drawImage(image, x, y, null);
}
```

Listing 10.6 takes this a little bit further and *tiles* the window with the graphics image. The result looks like the screen shown in Figure 10.15. We do the tiling in the `paintComponent` method. We first draw one copy of the image in the top left corner and then use the `copyArea` call to copy it into the entire window:

```
for (int i = 0; i * imageWidth <= getWidth(); i++)
      for (int j = 0; j * imageHeight <= getHeight(); j++)
            if (i + j > 0)
            g.copyArea(0, 0, imageWidth, imageHeight, i * imageWidth, j * imageHeight);
```

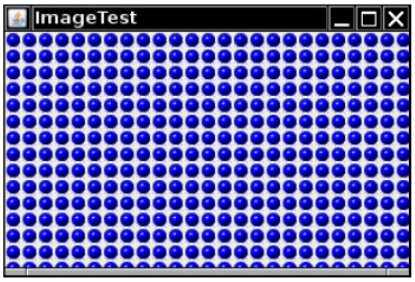


Figure 10.15 Window with tiled graphics image

Listing 10.6 shows the full source code of the image display program.

Listing 10.6 image/ImageTest.java

```
1  package image;
2
3  import java.awt.*;
4  import javax.swing.*;
5
6  /**
7   * @version 1.34 2015-05-12
8   * @author Cay Horstmann
9   */
10 public class ImageTest
11 {
```

```
12    public static void main(String[] args)
13    {
14       EventQueue.invokeLater(() ->
15          {
16             JFrame frame = new ImageFrame();
17             frame.setTitle("ImageTest");
18             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19             frame.setVisible(true);
20          });
21    }
22 }
23
24 /**
25  * A frame with an image component
26  */
27 class ImageFrame extends JFrame
28 {
29    public ImageFrame()
30    {
31       add(new ImageComponent());
32       pack();
33    }
34 }
35
36 /**
37  * A component that displays a tiled image
38  */
39 class ImageComponent extends JComponent
40 {
41    private static final int DEFAULT_WIDTH = 300;
42    private static final int DEFAULT_HEIGHT = 200;
43
44    private Image image;
45
46    public ImageComponent()
47    {
48       image = new ImageIcon("blue-ball.gif").getImage();
49    }
50
51    public void paintComponent(Graphics g)
52    {
53       if (image == null) return;
54
```

```
55        int imageWidth = image.getWidth(this);
56        int imageHeight = image.getHeight(this);
57
58        // draw the image in the upper-left corner
59
60        g.drawImage(image, 0, 0, null);
61
62        // tile the image across the component
63
64        for (int i = 0; i * imageWidth <= getWidth(); i++)
65           for (int j = 0; j * imageHeight <= getHeight(); j++)
66              if (i + j > 0)
67                 g.copyArea(0, 0, imageWidth, imageHeight, i * imageWidth, j * imageHeight);
68     }
69
70     public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT); }
71  }
```


Any operating environment that supports GUIs constantly monitors events such as keystrokes or mouse clicks.

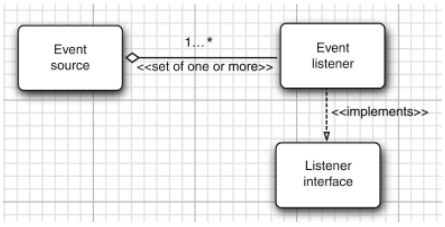
there are subclasses for each event type, such as `ActionEvent` and `WindowEvent`.

Different event sources can produce different kinds of events. For example, a button can send `ActionEvent` objects, whereas a window can send `WindowEvent` objects.

How event handling in the AWT works:

- A listener object is an instance of a class that implements a special interface called a *listener interface*.
- An event source is an object that can register listener objects and send them event objects.
- The event source sends out event objects to all registered listeners when that event occurs.
- The listener objects will then use the information in the event object to determine their reaction to the event.

Figure: shows the relationship between the event handling classes and interfaces.



Here is an example for specifying a listener:

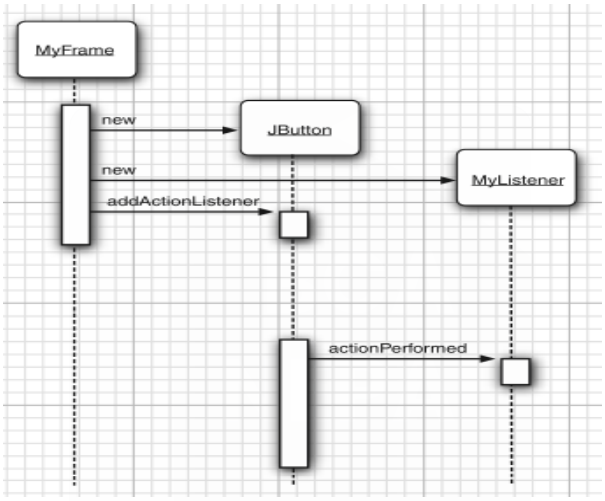
```
ActionListener listener = . . .;
JButton button = new JButton("OK");
button.addActionListener(listener);
```

Now the `listener` object is notified whenever an “action event” occurs in the button.

To implement the `ActionListener` interface, the listener class must have a method called `actionPerformed` that receives an `ActionEvent` object as a parameter.

```
class MyListener implements ActionListener
{
       . . .
       public void actionPerformed(ActionEvent event)
              {
              // reaction to button click goes here
              . . .
              }
}
```

Figure: shows the interaction between the event source, event listener, and event object.



1b.1.1.1 Example: Handling a Button Click

Listing 11.1 button/ButtonFrame.java

```
1  package button;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  /**
8   * A frame with a button panel
9   */
10 public class ButtonFrame extends JFrame
11 {
12    private JPanel buttonPanel;
13    private static final int DEFAULT_WIDTH = 300;
14    private static final int DEFAULT_HEIGHT = 200;
15
16    public ButtonFrame()
17    {
18       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
19
20       // create buttons
21       JButton yellowButton = new JButton("Yellow");
22       JButton blueButton = new JButton("Blue");
23       JButton redButton = new JButton("Red");
24
25       buttonPanel = new JPanel();
26
27       // add buttons to panel
28       buttonPanel.add(yellowButton);
29       buttonPanel.add(blueButton);
30       buttonPanel.add(redButton);
31
32       // add panel to frame
33       add(buttonPanel);
34
35       // create button actions
36       ColorAction yellowAction = new ColorAction(Color.YELLOW);
37       ColorAction blueAction = new ColorAction(Color.BLUE);
38       ColorAction redAction = new ColorAction(Color.RED);
39
40       // associate actions with buttons
41       yellowButton.addActionListener(yellowAction);
42       blueButton.addActionListener(blueAction);
43       redButton.addActionListener(redAction);
44    }
45
46    /**
47     * An action listener that sets the panel's background color.
48     */
49    private class ColorAction implements ActionListener
50    {
51       private Color backgroundColor;
52
53       public ColorAction(Color c)
54       {
55          backgroundColor = c;
56       }
57
58       public void actionPerformed(ActionEvent event)
59       {
60          buttonPanel.setBackground(backgroundColor);
61       }
62    }
63 }
```

// ButtonFrameMain.java

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

public class ButtonFrameMain {

    public static void main(String[] args)
    {
        EventQueue.invokeLater(() ->
        {
            JFrame frame = new ButtonFrame();
            frame.setTitle("Buttom Frame");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
        });
    }
}
```

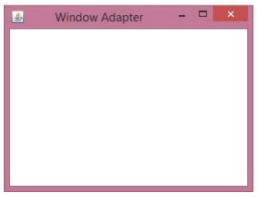
Java adapter classes *provide the default implementation of listener interfaces*.

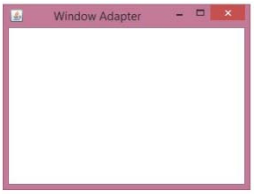
java.awt.event Adapter classes

Adapter class	Listener interface
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener

Java WindowAdapter Example

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  public class AdapterExample{
4.     Frame f;
5.     AdapterExample(){
6.        f=new Frame("Window Adapter");
7.        f.addWindowListener(new WindowAdapter(){
8.           public void windowClosing(WindowEvent e) {
9.              f.dispose();
10.          }
11.       });
12.
13.       f.setSize(400,400);
14.       f.setLayout(null);
15.       f.setVisible(true);
16.    }
17.  public static void main(String[] args) {
18.     new AdapterExample();
19.  }
20.  }
```

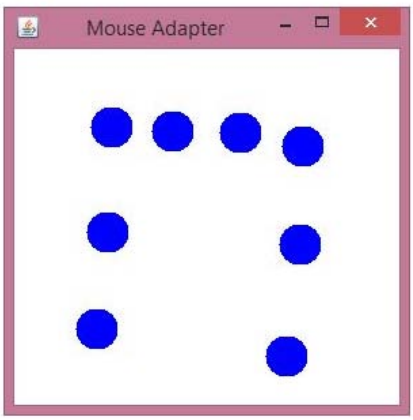




Java MouseAdapter Example

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  public class MouseAdapterExample extends MouseAdapter{
4.     Frame f;
5.     MouseAdapterExample(){
6.        f=new Frame("Mouse Adapter");
7.        f.addMouseListener(this);
8.
9.        f.setSize(300,300);
10.       f.setLayout(null);
11.       f.setVisible(true);
12.    }
13.    public void mouseClicked(MouseEvent e) {
14.       Graphics g=f.getGraphics();
15.       g.setColor(Color.BLUE);
16.       g.fillOval(e.getX(),e.getY(),30,30);
17.    }
18.
19.  public static void main(String[] args) {
20.     new MouseAdapterExample();
21.  }
22.  }
```

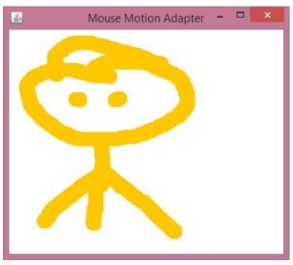
Output:



Java MouseMotionAdapter Example

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  public class MouseMotionAdapterExample extends MouseMotionAdapter{
4.     Frame f;
5.     MouseMotionAdapterExample(){
6.        f=new Frame("Mouse Motion Adapter");
7.        f.addMouseMotionListener(this);
8.
9.        f.setSize(300,300);
10.       f.setLayout(null);
11.       f.setVisible(true);
12.    }
13. public void mouseDragged(MouseEvent e) {
14.    Graphics g=f.getGraphics();
15.    g.setColor(Color.ORANGE);
16.    g.fillOval(e.getX(),e.getY(),20,20);
17. }
18. public static void main(String[] args) {
19.    new MouseMotionAdapterExample();
20. }
21. }
```

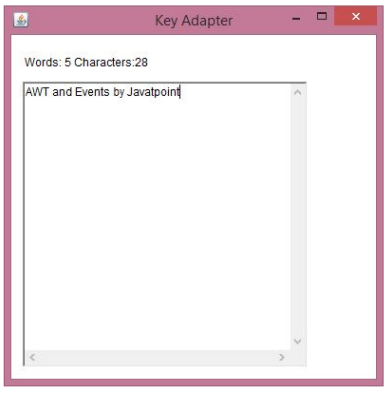
Output:



Java KeyAdapter Example

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  public class KeyAdapterExample extends KeyAdapter{
4.     Label l;
5.     TextArea area;
6.     Frame f;
7.     KeyAdapterExample(){
8.        f=new Frame("Key Adapter");
9.        l=new Label();
10.       l.setBounds(20,50,200,20);
11.       area=new TextArea();
12.       area.setBounds(20,80,300, 300);
13.       area.addKeyListener(this);
14.
15.       f.add(l);f.add(area);
16.       f.setSize(400,400);
17.       f.setLayout(null);
18.       f.setVisible(true);
19.    }
20.    public void keyReleased(KeyEvent e) {
21.       String text=area.getText();
```

Output:



```
22.             String words[]=text.split("\\s");
23.             l.setText("Words: "+words.length+" Characters:"+text.length());
24.         }
25.
26.         public static void main(String[] args) {
27.             new KeyAdapterExample();
28.         }
29.     }
```

1.1.c Swing and the Model-View-Controller Design Pattern

At first we discuss the concept of *design patterns* and then look at the “model-view-controller” pattern that has greatly influenced the design of the Swing framework.

1.1.1 Design Patterns

When solving a problem, you don’t usually figure out a solution from first principles. Instead, you are likely to be guided by your past experience, or you may ask other experts for advice on what has worked for them. Design patterns are a method for presenting this expertise in a structured way.

1.1.2 The Model-View-Controller Pattern

A user interface component such as a button, a checkbox, a text field, or a sophisticated tree control. Every component has three characteristics:

- Its *content*, such as the state of a button (pushed in or not), or the text in a text field
- Its *visual appearance* (color, size, and so on)
- Its *behavior* (reaction to events)

Obviously, the visual appearance of a button depends on the look-and-feel.

To do this, the Swing designers turned to a well-known design pattern: the *model- view- controller* pattern. The model-viewcontroller (MVC) design pattern teaches how to accomplish this. Implement three separate classes:

- The *model,* which stores the content
- The *view,* which displays the content
- The *controller,* which handles user input

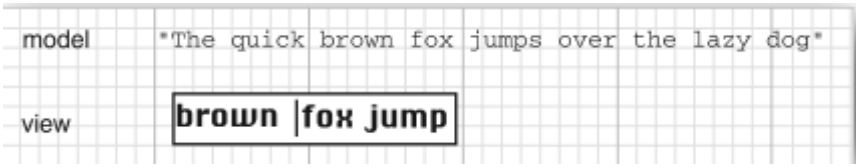


Figure. Model and view of a text field

For example, an HTML editor can offer two *simultaneous* views of the same content: a WYSIWYG view and a “raw tag” view. The controller handles the user-input events, such as mouse clicks and keystrokes. It then decides whether to translate these events into changes in the model or the view.

What You See Is What You Get

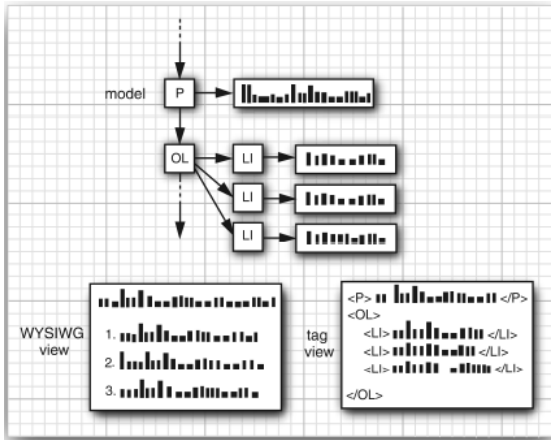


Figure. Two separate views of the same model

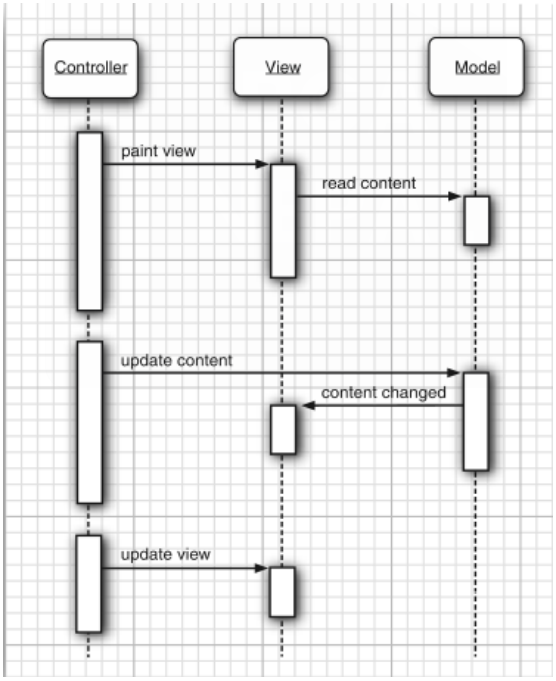


Figure. Interactions among model, view, and controller objects

1.1.3 A Model-View-Controller Analysis of Swing Buttons

Still, buttons are about the simplest user interface elements, so they are a good place to get comfortable with the model-view-controller pattern. We will encounter similar kinds of classes and interfaces for the more sophisticated Swing components.

For most components, the model class implements an interface whose name ends in `Model`; in this case, the interface is called `ButtonModel`. Swing library contains a single class, called `DefaultButtonModel`, that implements this interface.

The Sort of data maintained by a button model by looking at the properties of the `ButtonModel` interface—see Table 1.1.

Table 1.1 Properties of the `ButtonModel` Interface

Property Name	Value
`actionCommand`	The action command string associated with this button
`mnemonic`	The keyboard mnemonic for this button
`armed`	`true` if the button was pressed and the mouse is still over the button
`enabled`	`true` if the button is selectable
`pressed`	`true` if the button was pressed but the mouse button hasn't yet been released
`rollover`	`true` if the mouse is over the button
`selected`	`true` if the button has been toggled on (used for checkboxes and radio buttons)

Each `JButton` object stores a button model object which you can retrieve.

```
JButton button = new JButton("Blue");
ButtonModel model = button.getModel();
```

1.2 Introduction to Layout Management

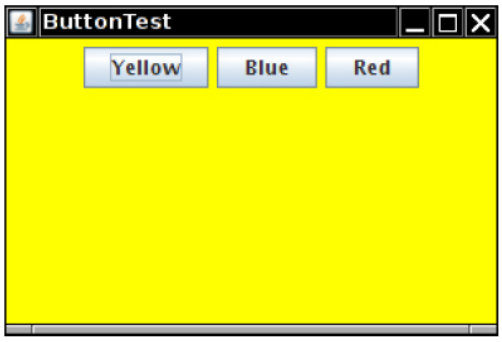


Figure 12.5 A panel with three buttons

Figure 12.6 A panel with six buttons managed by a flow layout

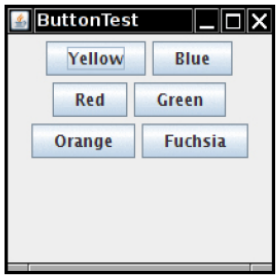
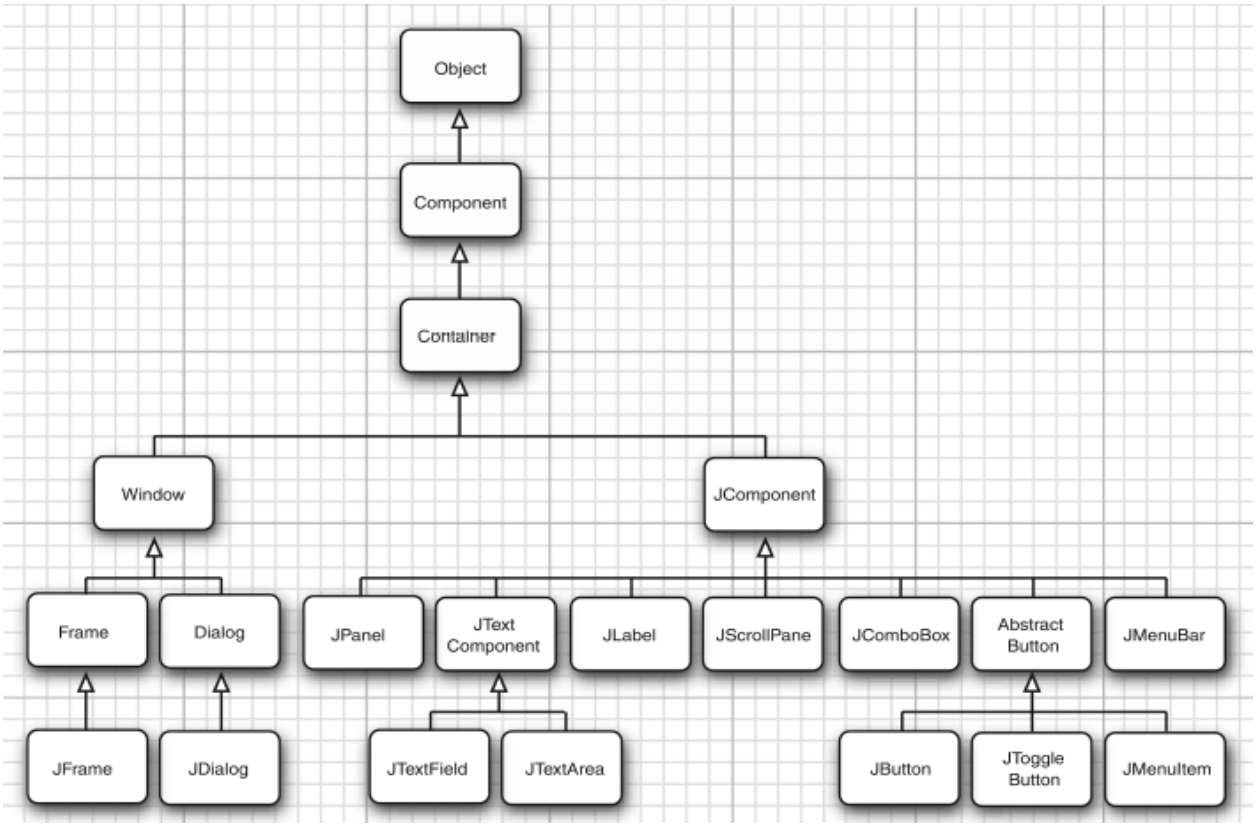


Figure 12.7 Changing the panel size rearranges the buttons automatically.

In general, *components* are placed inside *containers*, and a *layout manager* determines the positions and sizes of components in a container.

Buttons, text fields, and other user interface elements extend the class `Component`. Components can be placed inside containers, such as panels. Containers can themselves be put inside other containers, so the class `Container` extends `Component`.

Figure 1.8 shows the inheritance hierarchy for `Component`.



Each container has a default layout manager, but we can always set our own. For example, the statement uses the `GridLayout` class to lay out the components in four rows and four columns.

```
panel.setLayout(new GridLayout(4, 4));
```

1.2.1 Border Layout

The *border layout manager* is the default layout manager of the content pane of every `JFrame`. We can choose to place the component in the center, north, south, east, or west of the content pane (see Figure 1.9).

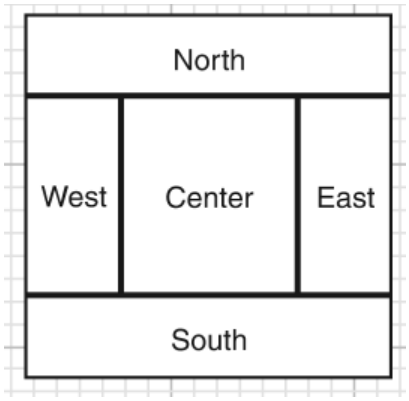


Figure 1.9 Border layout

For example:

```
frame.add(component, BorderLayout.SOUTH);
```

If we don't supply any value, `CENTER` is assumed.

Unlike the flow layout, the border layout grows all components to fill the available space. (The flow layout leaves each component at its preferred size.) This is a problem when you add a button:

```
frame.add(yellowButton, BorderLayout.SOUTH); // don't
```

Figure 1.10 A single button managed by a border layout

To solve this problem, use additional panels. For example, look at Figure 1.11. The three buttons at the bottom of the screen are all contained in a panel. The panel is put into the southern region of the content pane.

Figure 1.11 Panel placed at the southern region of the frame

To achieve this configuration, first create a new `JPanel` object, then add the individual buttons to the panel.

Finally, add the panel to the content pane of the frame.

```
JPanel panel = new JPanel();
panel.add(yellowButton);
panel.add(blueButton);
panel.add(redButton);
frame.add(panel, BorderLayout.SOUTH);
```

1.2.2 Grid Layout

The grid layout arranges all components in rows and columns like a spreadsheet. All components are given the same size. The calculator program in Figure 1.12

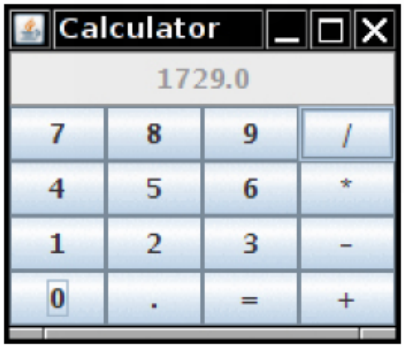


Figure 1.12 A calculator

In the constructor of the grid layout object, you specify how many rows and columns you need.

```
panel.setLayout(new GridLayout(4, 4));
```

Add the components, starting with the first entry in the first row, then the second entry in the first row, and so on.

```
panel.add(new JButton("1"));
panel.add(new JButton("2"));
```

Listing 12.1 `calculator/CalculatorPanel.java`

12.3 Text Input

You can use the `JTextField` and `JTextArea` components for text input. A text field can accept only one line of text; a text area can accept multiple lines of text. A`JPasswordField` accepts one line of text without showing the contents.

All three of these classes inherit from a class called `JTextComponent`.

12.3.1 Text Fields

The usual way to add a text field to a window is to add it to a panel or other container—just as we would add a button:

```
JPanel panel = new JPanel();
JTextField textField = new JTextField("Default input", 20);
panel.add(textField);
```

This code adds a text field and initializes it by placing the string `"Default input"` inside it. The second parameter of this constructor sets the width. In this case, the width is 20 “columns.”

To make a blank text field, just leave out the string as a parameter for the `JTextField` constructor:

```
JTextField textField = new JTextField(20);

textField.setText("Hello!");

String text = textField.getText().trim();
```

To change the font in which the user text appears, use the `setFont` method.

12.3.2 Labels and Labeling Components

Labels are components that hold text. You can use a label to identify components.

To label a component that does not itself come with an identifier:

1. Construct a `JLabel` component with the correct text.
2. Place it close enough to the component you want to identify so that the user can see that the label identifies the correct component.

That interface defines a number of useful constants such as `LEFT`, `RIGHT`, `CENTER`, `NORTH`, `EAST`, and so on. The `JLabel` class is one of several Swing classes that implement this interface. Therefore, you can specify a right-aligned label either as

```
JLabel label = new JLabel("User name: ", SwingConstants.RIGHT);
```

or

```
JLabel label = new JLabel("User name: ", JLabel.RIGHT);
```

The `setText` and `setIcon` methods let you set the text and icon of the label at runtime.

12.3.3 Password Fields

Password fields are a special kind of text field. Each typed character is represented by an *echo character,* typically an asterisk (`*`).Swing supplies a `JPasswordField` class that implements such a text field.

12.3.4 Text Areas

Figure 12.13 Text components

```
textArea = new JTextArea(8, 40); // 8 lines of 40 columns each

textArea.setLineWrap(true); // long lines are wrapped
```

12.3.5 Scroll Panes

In Swing, a text area does not have scrollbars. If you want scrollbars, you have to place the text area inside a *scroll pane.*

```
textArea = new JTextArea(8, 40);
JScrollPane scrollPane = new JScrollPane(textArea);
```

Listing 12.2 text/TextComponentFrame.java

```
1  package text;
2
3  import java.awt.BorderLayout;
4  import java.awt.GridLayout;
5
6  import javax.swing.JButton;
7  import javax.swing.JFrame;
8  import javax.swing.JLabel;
9  import javax.swing.JPanel;
10 import javax.swing.JPasswordField;
11 import javax.swing.JScrollPane;
12 import javax.swing.JTextArea;
13 import javax.swing.JTextField;
14 import javax.swing.SwingConstants;
15
16 /**
17  * A frame with sample text components.
18  */
19 public class TextComponentFrame extends JFrame
20 {
21    public static final int TEXTAREA_ROWS = 8;
22    public static final int TEXTAREA_COLUMNS = 20;
23
24    public TextComponentFrame()
25    {
26       JTextField textField = new JTextField();
27       JPasswordField passwordField = new JPasswordField();
28
29       JPanel northPanel = new JPanel();
30       northPanel.setLayout(new GridLayout(2, 2));
31       northPanel.add(new JLabel("User name: ", SwingConstants.RIGHT));
32       northPanel.add(textField);
33       northPanel.add(new JLabel("Password: ", SwingConstants.RIGHT));
34       northPanel.add(passwordField);
35
36       add(northPanel, BorderLayout.NORTH);
37
38       JTextArea textArea = new JTextArea(TEXTAREA_ROWS, TEXTAREA_COLUMNS);
39       JScrollPane scrollPane = new JScrollPane(textArea);
40
41       add(scrollPane, BorderLayout.CENTER);
42
43       // add button to append text into the text area
44
45       JPanel southPanel = new JPanel();
46
47       JButton insertButton = new JButton("Insert");
```

```
48         southPanel.add(insertButton);
49         insertButton.addActionListener(event ->
50            textArea.append("User name: " + textField.getText() + " Password: "
51               + new String(passwordField.getPassword()) + "\n"));
52
53         add(southPanel, BorderLayout.SOUTH);
54         pack();
55      }
56   }
```

12.4 Choice Components

12.4.1 Checkboxes:

If you want to collect just a “yes” or “no” input, use a checkbox component.

Figure 12.14 Checkboxes

Checkboxes need a label next to them to identify their purpose. Give the label text in the constructor:

```
bold = new JCheckBox("Bold");
```

Use the `setSelected` method to turn a checkbox on or off. For example:

```
bold.setSelected(true);
```

The `isSelected` method then retrieves the current state of each checkbox. It is `false` if unchecked, `true` if checked. In our program, the two checkboxes share the same action listener.

```
ActionListener listener = . . .
bold.addActionListener(listener);
italic.addActionListener(listener);
```

The listener queries the state of the `bold` and `italic` checkboxes and sets the font of the panel to plain, bold, italic, or both bold and italic.

```
ActionListener listener = event -> {
int mode = 0;
if (bold.isSelected()) mode += Font.BOLD;
if (italic.isSelected()) mode += Font.ITALIC;
label.setFont(new Font(Font.SERIF, mode, FONTSIZE));
};
```

Listing 12.3 checkBox/CheckBoxFrame.java

```
1  package checkBox;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  /**
8   * A frame with a sample text label and check boxes for selecting font
9   * attributes.
10  */
11 public class CheckBoxFrame extends JFrame
12 {
13    private JLabel label;
14    private JCheckBox bold;
15    private JCheckBox italic;
16    private static final int FONTSIZE = 24;
17
18    public CheckBoxFrame()
19    {
20       // add the sample text label
21
22       label = new JLabel("The quick brown fox jumps over the lazy dog.");
23       label.setFont(new Font("Serif", Font.BOLD, FONTSIZE));
24       add(label, BorderLayout.CENTER);
25
26       // this listener sets the font attribute of
27       // the label to the check box state
28
29       ActionListener listener = event -> {
30          int mode = 0;
31          if (bold.isSelected()) mode += Font.BOLD;
32          if (italic.isSelected()) mode += Font.ITALIC;
33          label.setFont(new Font("Serif", mode, FONTSIZE));
34       };
35
36       // add the check boxes
37
38       JPanel buttonPanel = new JPanel();
39
40       bold = new JCheckBox("Bold");
41       bold.addActionListener(listener);
42       bold.setSelected(true);
43       buttonPanel.add(bold);
44
45       italic = new JCheckBox("Italic");
46       italic.addActionListener(listener);
47       buttonPanel.add(italic);
48
49       add(buttonPanel, BorderLayout.SOUTH);
50       pack();
51    }
52 }
```

12.4.2 Radio Buttons

In many cases, we want the user to check only one of several boxes. When another box is checked, the previous box is automatically unchecked. Such a group of boxes is often called a *radio button group* because the buttons work like the station selector buttons on a radio.

When you push in one button, the previously depressed button pops out.

Figure 12.15 A radio button group

The button group object is reponsible for turning off the previously set button when a new button is clicked.

```
ButtonGroup group = new ButtonGroup();
JRadioButton smallButton = new JRadioButton("Small", false);
group.add(smallButton);
JRadioButton mediumButton = new JRadioButton("Medium", true);
group.add(mediumButton);
```

In our example program, we define an action listener that sets the font size to a particular value:

```
ActionListener listener = event ->
label.setFont(new Font("Serif", Font.PLAIN, size));
```

We could have a single listener that computes the size as follows:

```
if (smallButton.isSelected()) size = 8;
else if (mediumButton.isSelected()) size = 12;
. . .
```

However, we prefer to use separate action listener objects because they tie the size values more closely to the buttons.

Listing 12.4 radioButton/RadioButtonFrame.java

```
1  package radioButton;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  /**
8   * A frame with a sample text label and radio buttons for selecting font sizes.
9   */
10 public class RadioButtonFrame extends JFrame
11 {
12    private JPanel buttonPanel;
13    private ButtonGroup group;
14    private JLabel label;
15    private static final int DEFAULT_SIZE = 36;
16
17    public RadioButtonFrame()
18    {
19       // add the sample text label
20
21       label = new JLabel("The quick brown fox jumps over the lazy dog.");
22       label.setFont(new Font("Serif", Font.PLAIN, DEFAULT_SIZE));
23       add(label, BorderLayout.CENTER);
24
25       // add the radio buttons
26
27       buttonPanel = new JPanel();
28       group = new ButtonGroup();
29
30       addRadioButton("Small", 8);
31       addRadioButton("Medium", 12);
32       addRadioButton("Large", 18);
33       addRadioButton("Extra large", 36);
34
35        add(buttonPanel, BorderLayout.SOUTH);
36        pack();
37    }
38
39    /**
40     * Adds a radio button that sets the font size of the sample text.
41     * @param name the string to appear on the button
42     * @param size the font size that this button sets
43     */
44    public void addRadioButton(String name, int size)
45    {
46       boolean selected = size == DEFAULT_SIZE;
47       JRadioButton button = new JRadioButton(name, selected);
48       group.add(button);
49       buttonPanel.add(button);
50
51       // this listener sets the label font size
52
53       ActionListener listener = event -> label.setFont(new Font("Serif", Font.PLAIN, size));
54
55       button.addActionListener(listener);
56    }
57 }
```

12.4.3 Borders

If we have multiple groups of radio buttons in a window, We want to visually indicate which buttons are grouped. Swing provides a set of useful *borders* for this purpose. We can apply a border to any component that extends `JComponent`.

We can choose from quite a few borders, but We need to follow the same steps for all of them.

1. Call a static method of the `BorderFactory` to create a border. We can choose among the following styles (see Figure 12.16):
    - Lowered bevel
    - Raised bevel
    - Etched
    - Line
    - Matte
    - Empty (just to create some blank space around the component)

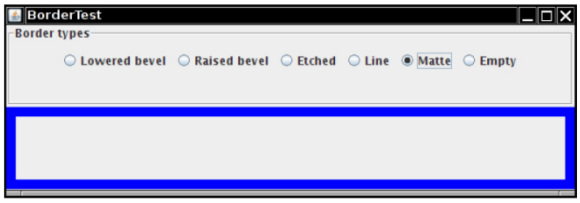


Figure 12.16 Testing border types

2. If We like, add a title to your border by passing your border to `BorderFactory.createTitledBorder`.
3. If We really want to go all out, combine several borders with a call to `BorderFactory.createCompoundBorder`.
4. Add the resulting border to our component by calling the `setBorder` method of the `JComponent` class.

For example, here is how you add an etched border with a title to a panel:

```
Border etched = BorderFactory.createEtchedBorder();
Border titled = BorderFactory.createTitledBorder(etched, "A Title");
panel.setBorder(titled);
```

Listing 12.5 border/BorderFrame.java

```
1  package border;
2
3  import java.awt.*;
4  import javax.swing.*;
5  import javax.swing.border.*;
6
7  /**
8   * A frame with radio buttons to pick a border style.
9   */
10 public class BorderFrame extends JFrame
11 {
12    private JPanel demoPanel;
13    private JPanel buttonPanel;
14    private ButtonGroup group;
15
16    public BorderFrame()
17    {
18       demoPanel = new JPanel();
19       buttonPanel = new JPanel();
20       group = new ButtonGroup();
21
22       addRadioButton("Lowered bevel", BorderFactory.createLoweredBevelBorder());
23       addRadioButton("Raised bevel", BorderFactory.createRaisedBevelBorder());
24       addRadioButton("Etched", BorderFactory.createEtchedBorder());
25       addRadioButton("Line", BorderFactory.createLineBorder(Color.BLUE));
26       addRadioButton("Matte", BorderFactory.createMatteBorder(10, 10, 10, 10, Color.BLUE));
27       addRadioButton("Empty", BorderFactory.createEmptyBorder());
28
29       Border etched = BorderFactory.createEtchedBorder();
```

```
30        Border titled = BorderFactory.createTitledBorder(etched, "Border types");
31        buttonPanel.setBorder(titled);
32
33        setLayout(new GridLayout(2, 1));
34        add(buttonPanel);
35        add(demoPanel);
36        pack();
37     }
38
39     public void addRadioButton(String buttonName, Border b)
40     {
41        JRadioButton button = new JRadioButton(buttonName);
42        button.addActionListener(event -> demoPanel.setBorder(b));
43        group.add(button);
44        buttonPanel.add(button);
45     }
46  }
```

12.4.4 Combo Boxes

If you have more than a handful of alternatives, radio buttons are not a good choice because they take up too much screen space. Instead, you can use a combo box. When the user clicks on this component, a list of choices drops down, and the user can then select one of them (see Figure 12.17).

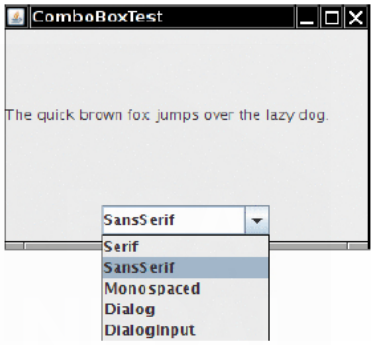


Figure 12.17 A combo box

If the drop-down list box is set to be *editable*, you can edit the current selection as if it were a text field. For that reason, this component is called a *combo box*.

As of Java SE 7, the `JComboBox` class is a generic class. For example, a `JComboBox<String>` holds objects of type `String`, and a `JComboBox<Integer>` holds integers. Call the `setEditable` method to make the combo box editable. Note that editing affects only the selected item. It does not change the list of choices in any way.

If your combo box isn't editable, you are better off calling `combo.getItemAt(combo.getSelectedIndex())` which gives you the selected item with the correct type.

In the example program, the user can choose a font style from a list of styles (Serif, SansSerif, Monospaced, etc.). The user can also type in another font.

Add the choice items with the `addItem` method. In our program, `addItem` is called only in the constructor, but you can call it any time.

```
JComboBox<String> faceCombo = new JComboBox<>();
faceCombo.addItem("Serif");
faceCombo.addItem("SansSerif");
. . .
```

This method adds the string to the end of the list. You can add new items anywhere in the list with the `insertItemAt` method:

```
faceCombo.insertItemAt("Monospaced", 0); // add at the beginning
```

If you need to remove items at runtime, use the `removeItem` or `removeItemAt` method, depending on whether you supply the item to be removed or its position.

```
faceCombo.removeItem("Monospaced");
faceCombo.removeItemAt(0); // remove first item
```

The `removeAllItems` method removes all items at once.

```
ActionListener listener = event ->
label.setFont(new Font(
faceCombo.getItemAt(faceCombo.setSelectedIndex()),
Font.PLAIN,
DEFAULT_SIZE));
```

Listing 12.6 comboBox/ComboBoxFrame.java

```
1  package comboBox;
2
3  import java.awt.BorderLayout;
4  import java.awt.Font;
5
6  import javax.swing.JComboBox;
7  import javax.swing.JFrame;
8  import javax.swing.JLabel;
9  import javax.swing.JPanel;
10
11 /**
12  * A frame with a sample text label and a combo box for selecting font faces.
13  */
14 public class ComboBoxFrame extends JFrame
15 {
16    private JComboBox<String> faceCombo;
17    private JLabel label;
18    private static final int DEFAULT_SIZE = 24;
19
20    public ComboBoxFrame()
21    {
22       // add the sample text label
23
24       label = new JLabel("The quick brown fox jumps over the lazy dog.");
25       label.setFont(new Font("Serif", Font.PLAIN, DEFAULT_SIZE));
26       add(label, BorderLayout.CENTER);
27
28       // make a combo box and add face names
29
30       faceCombo = new JComboBox<>();
31       faceCombo.addItem("Serif");
32       faceCombo.addItem("SansSerif");
33       faceCombo.addItem("Monospaced");
34       faceCombo.addItem("Dialog");
35       faceCombo.addItem("DialogInput");
36
37       // the combo box listener changes the label font to the selected face name
38
39       faceCombo.addActionListener(event ->
40          label.setFont(
41             new Font(faceCombo.getItemAt(faceCombo.getSelectedIndex()),
42                Font.PLAIN, DEFAULT_SIZE)));
43
44       // add combo box to a panel at the frame's southern border
45
46       JPanel comboPanel = new JPanel();
47       comboPanel.add(faceCombo);
48       add(comboPanel, BorderLayout.SOUTH);
49       pack();
50    }
51 }
```

12.4.5 Sliders

Sliders offer a choice from a continuum of values—for example, any number between 1 and 100.

The most common way of constructing a slider is as follows:

```
JSlider slider = new JSlider(min, max, initialValue);
```

If we omit the minimum, maximum, and initial values, they are initialized with `0`, `100`, and `50`, respectively.
Or if we want the slider to be vertical, use the following constructor call:

```
JSlider slider = new JSlider(SwingConstants.VERTICAL, min, max, initialValue);
```

These constructors create a plain slider, such as the top slider in Figure 12.18. You will see presently how to add decorations to a slider.

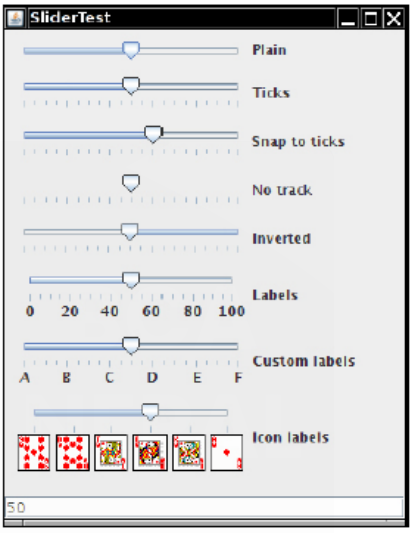


Figure 12.18 Sliders

```
ChangeListener listener = event -> {
JSlider slider = (JSlider) event.getSource();
int value = slider.getValue();
. . .
};
```

For example, in the sample program, the second slider uses the following settings:

```
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
```

These instructions only set the units for the tick marks. To actually have the tick marks appear, call

```
slider.setPaintTicks(true);
```

we can force the slider to *snap to ticks.*

```
slider.setSnapToTicks(true);
```

We can display *tick mark labels* for the major tick marks by calling

```
slider.setPaintLabels(true);
```

We need to fill a hash table with keys of type `Integer` and values of type `Component`.

```
Hashtable<Integer, Component> labelTable = new
Hashtable<Integer, Component>();
labelTable.put(0, new JLabel("A"));
labelTable.put(20, new JLabel("B"));
. . .
labelTable.put(100, new JLabel("F"));
slider.setLabelTable(labelTable);
```

The fourth slider in Figure 12.18 has no track. To suppress the “track” in which the slider moves, call

```
slider.setPaintTrack(false);
```

The fifth slider has its direction reversed by a call to

```
slider.setInverted(true);
```

12.5 Menus:

Swing also supports another type of user interface element—pull-down menus that are familiar from GUI applications. A *menu bar* at the top of a window contains the names of the pull-down menus. Clicking on a name opens the menu containing *menu items* and *submenus.* When the user clicks on a menu item, all menus are closed and a message is sent to the program. Figure 12.19 shows a typical menu with a submenu.

Figure 12.19 A menu with a submenu

12.5.1 Menu Building

Building menus is straightforward. First, create a menu bar:

```
JMenuBar menuBar = new JMenuBar();
```

Normally, We want it to appear at the top of a frame. We can add it there with the `setJMenuBar` method:

```
frame.setJMenuBar(menuBar);
```

For each menu, We create a menu object:

```
JMenu editMenu = new JMenu("Edit");
```

Add the top-level menus to the menu bar:

```
menuBar.add(editMenu);
```

Add menu items, separators, and submenus to the menu object:

```
JMenuItem pasteItem = new JMenuItem("Paste");
editMenu.add(pasteItem);
editMenu.addSeparator();
JMenu optionsMenu = . . .; // a submenu
editMenu.add(optionsMenu);
```

When the user selects a menu, an action event is triggered. we need to install an action listener for each menu item:

```
ActionListener listener = . . .;
pasteItem.addActionListener(listener);
```

The method `JMenu.add(String s)` conveniently adds a menu item to the end of a menu.

For example:

```
editMenu.add("Paste");
```

The `add` method returns the created menu item, so you can capture it and add the listener, as follows:

```
JMenuItem pasteItem = editMenu.add("Paste");
pasteItem.addActionListener(listener);
```

We can define a class that implements the `Action` interface, usually by extending the `AbstractAction` convenience class, specify the menu item label in the constructor of the `AbstractAction` object, and override the `actionPerformed` method to hold the menu action handler. For example:

```
Action exitAction = new AbstractAction("Exit") // menu item text goes here
{
public void actionPerformed(ActionEvent event)
{
// action code goes here
System.exit(0);
}
};
```

You can then add the action to the menu:

```
JMenuItem exitItem = fileMenu.add(exitAction);
```

This is just a convenient shortcut for

```
JMenuItem exitItem = new JMenuItem(exitAction);
                     fileMenu.add(exitItem);
```

12.5.2 Icons in Menu Items

Menu items are very similar to buttons. In fact, the `JMenuItem` class extends the `AbstractButton` class.
We can specify the icon with the `JMenuItem(String, Icon)` or `JMenuItem(Icon)` constructor, or We can set it with the `setIcon` method that the `JMenuItem` class inherits from the `AbstractButton` class. Here is an example:

```
JMenuItem cutItem = new JMenuItem("Cut", new ImageIcon("cut.gif"));

cutItem.setHorizontalTextPosition(SwingConstants.LEFT);
```

moves the menu item text to the left of the icon. We can also add an icon to an action:

```
cutAction.putValue(Action.SMALL_ICON, new ImageIcon("cut.gif"));
```

Alternatively, We can set the icon in the `AbstractAction` constructor:

```
cutAction = new
AbstractAction("Cut", new ImageIcon("cut.gif"))
{
public void actionPerformed(ActionEvent event)
{
. . .
}
};
```

12.5.3 Checkbox and Radio Button Menu Items

Apart from the button decoration, treat these menu items just as We would any others. For example, here is how We create a checkbox menu item:

```
JCheckBoxMenuItem readonlyItem = new JCheckBoxMenuItem("Read-only");
optionsMenu.add(readonlyItem);
```

The radio button menu items work just like regular radio buttons. We must add them to a button group. When one of the buttons in a group is selected, all others are automatically deselected.

```
ButtonGroup group = new ButtonGroup();
JRadioButtonMenuItem insertItem = new JRadioButtonMenuItem("Insert");
insertItem.setSelected(true);
JRadioButtonMenuItem overtypeItem = new JRadioButtonMenuItem("Overtype");
group.add(insertItem);
group.add(overtypeItem);
optionsMenu.add(insertItem);
optionsMenu.add(overtypeItem);
```

12.5.4 Pop-Up Menus

A*pop-up menu* is a menu that is not attached to a menu bar but floats somewhere.
Create a pop-up menu just as you create a regular menu, except that a pop-up menu has no title.

```
JPopupMenu popup = new JPopupMenu();
```

Then, add your menu items as usual:

```
JMenuItem item = new JMenuItem("Cut");
item.addActionListener(listener);
popup.add(item);
```

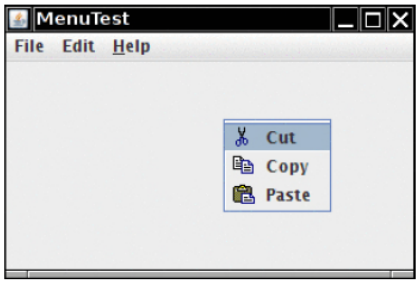


Figure 12.20 A pop-up menu

For example:

```
popup.show(panel, x, y);
```

Usually, you want to pop up a menu when the user clicks a particular mouse button—the so-called *pop-up trigger.*

To pop up a menu when the user clicks on a component, using the pop-up trigger, simply call the method

```
component.setComponentPopupMenu(popup);
```

The child component can inherit the parent component's pop-up menu by calling

```
child.setInheritsPopupMenu(true);
```

12.5.5 Keyboard Mnemonics and Accelerators

It is a real convenience for the experienced user to select menu items by *keyboard mnemonics. We* can create a keyboard mnemonic for a menu item by specifying a mnemonic letter in the menu item constructor:

```
JMenuItem aboutItem = new JMenuItem("About", 'A');
```

For example, if We have a mnemonic "A" for the menu item "Save As," then it makes more sense to underline the second "A" (Save As). We can specify which character you want to have underlined by calling the `setDisplayedMnemonicIndex` method. If We have an `Action` object, We can add the mnemonic as the value of the `Action.MNEMONIC_KEY` key, as follows:

```
cutAction.putValue(Action.MNEMONIC_KEY, new Integer('A'));
```

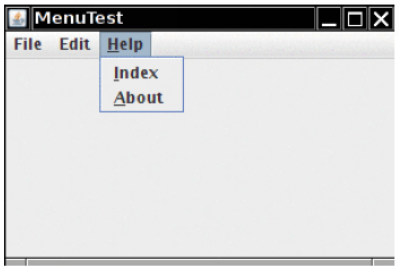


Figure 12.21 Keyboard mnemonics

To attach a mnemonic to a menu, call the `setMnemonic` method:

```
JMenu helpMenu = new JMenu("Help");
helpMenu.setMnemonic('H');
```

For example, the following call attaches the accelerator Ctrl+O to the `openItem` menu item:

```
openItem.setAccelerator(KeyStroke.getKeyStroke("ctrl O"));
```

12.5.6 Enabling and Disabling Menu Items

Occasionally, a particular menu item should be selected only in certain contexts. For example, when a document is opened in read-only mode, the Save menu item is not meaningful. A deactivated menu item is shown in gray, and it cannot be selected (see Figure 12.23).

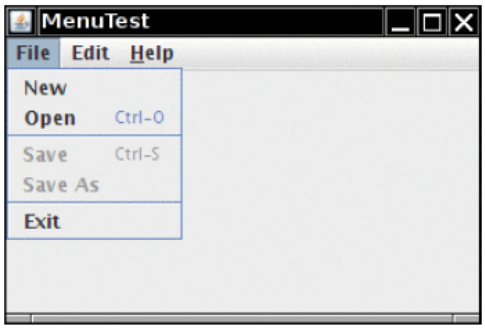


Figure 12.23 Disabled menu items

To enable or disable a menu item, use the `setEnabled` method:

```
saveItem.setEnabled(false);
```

There are two strategies for enabling and disabling menu items. the Save and Save As menu items and disable them. Alternatively, We can disable items just before displaying the menu. To do this, you must register a listener for the "menu selected" event. The `javax.swing.event` package defines a `MenuListener` interface with three methods:

```
void menuSelected(MenuEvent event)
void menuDeselected(MenuEvent event)
void menuCanceled(MenuEvent event)
```

The `menuSelected` method is called *before* the menu is displayed.

The following code shows how to disable the Save and Save As actions whenever the Read Only checkbox menu item is selected:

```
public void menuSelected(MenuEvent event)
{
saveAction.setEnabled(!readonlyItem.isSelected());
saveAsAction.setEnabled(!readonlyItem.isSelected());
}
```

12.5.7 Toolbars

A toolbar is a button bar that gives quick access to the most commonly used commands in a program (see Figure 12.24)

Figure 12.24 A toolbar

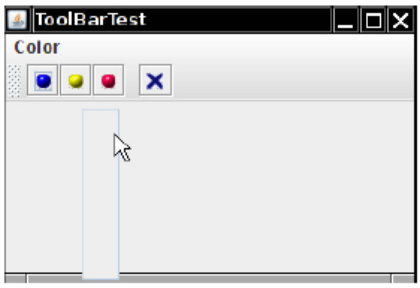


Figure 12.25 Dragging the toolbar

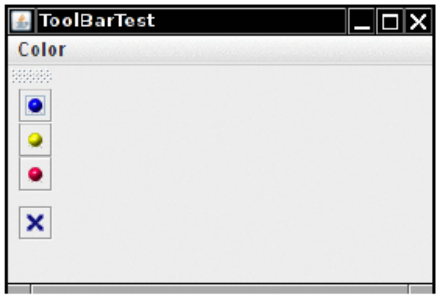


Figure 12.26 The toolbar has been dragged to another border

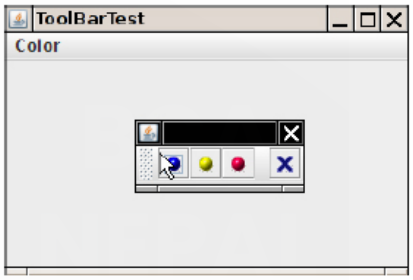


Figure 12.27 Detaching the toolbar

Toolbars are straightforward to program. Add components into the toolbar:

```
JToolBar bar = new JToolBar();
bar.add(blueButton);
```

The `JToolBar` class also has a method to add an `Action` object. Simply populate the toolbar with `Action` objects, like this:

```
bar.add(blueAction);
```

We can separate groups of buttons with a separator:

```
bar.addSeparator();
```

Then, add the toolbar to the frame:

```
add(bar, BorderLayout.NORTH);
```

We can also specify a title for the toolbar that appears when the toolbar is undocked:

```
bar = new JToolBar(titleString);
```

By default, toolbars are initially horizontal. To have a toolbar start out vertical, use

```
bar = new JToolBar(SwingConstants.VERTICAL)
```

or

```
bar = new JToolBar(titleString, SwingConstants.VERTICAL)
```

12.5.8 Tooltips

A disadvantage of toolbars is that users are often mystified by the meanings of the tiny icons in toolbars. To solve this problem, user interface designers invented *tooltips*.

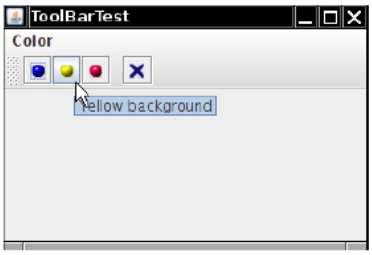


Figure 12.28 A tooltip

In Swing, you can add tooltips to any `JComponent` simply by calling the `setToolTipText` method:

```
exitButton.setToolTipText("Exit");
```

Alternatively, if you use `Action` objects, you associate the tooltip with the `SHORT_DESCRIPTION` key:

```
exitAction.putValue(Action.SHORT_DESCRIPTION, "Exit");
```