

PART

III Introducing GUI Programming with Swing

CHAPTER 31

Introducing Swing

CHAPTER 32

Exploring Swing

CHAPTER 33

Introducing Swing Menus

CHAPTER

31

Introducing Swing

In Part II, you saw how to build very simple user interfaces with the AWT classes. Although the AWT is still a crucial part of Java, its component set is no longer widely used to create graphical user interfaces. Today, programmers typically use Swing for this purpose. Swing is a framework that provides more powerful and flexible GUI components than does the AWT. As a result, it is the GUI that has been widely used by Java programmers for more than a decade.

Coverage of Swing is divided between three chapters. This chapter introduces Swing. It begins by describing Swing's core concepts. It then presents a simple example that shows the general form of a Swing program. This is followed by an example that uses event handling. The chapter concludes by explaining how painting is accomplished in Swing. The next chapter presents several commonly used Swing components. The third chapter introduces Swing-based menus. It is important to understand that the number of classes and interfaces in the Swing packages is quite large, and they can't all be covered in this book. (In fact, full coverage of Swing requires an entire book of its own.) However, these three chapters will give you a basic understanding of this important topic.

NOTE For a comprehensive introduction to Swing, see my book *Swing: A Beginner's Guide* published by McGraw-Hill Professional (2007).

The Origins of Swing

Swing did not exist in the early days of Java. Rather, it was a response to deficiencies present in Java's original GUI subsystem: the Abstract Window Toolkit. The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface. One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents, or *peers*. This means that the look and feel of a component is defined by the platform, not by Java. Because the AWT components use native code resources, they are referred to as *heavyweight*.

The use of native peers led to several problems. First, because of variations between operating systems, a component might look, or even act, differently on different platforms. This potential variability threatened the overarching philosophy of Java: write once, run anywhere. Second, the look and feel of each component was fixed (because it is defined by the platform) and could not be (easily) changed. Third, the use of heavyweight components caused some frustrating restrictions. For example, a heavyweight component was always opaque.

Not long after Java's original release, it became apparent that the limitations and restrictions present in the AWT were sufficiently serious that a better approach was needed. The solution was Swing. Introduced in 1997, Swing was included as part of the Java Foundation Classes (JFC). Swing was initially available for use with Java 1.1 as a separate library. However, beginning with Java 1.2, Swing (and the rest of the JFC) was fully integrated into Java.

Swing Is Built on the AWT

Before moving on, it is necessary to make one important point: although Swing eliminates a number of the limitations inherent in the AWT, Swing *does not* replace it. Instead, Swing is built on the foundation of the AWT. This is why the AWT is still a crucial part of Java. Swing also uses the same event handling mechanism as the AWT. Therefore, a basic understanding of the AWT and of event handling is required to use Swing. (The AWT is covered in [Chapters 25](#) and [26](#). Event handling is described in [Chapter 24](#).)

Two Key Swing Features

As just explained, Swing was created to address the limitations present in the AWT. It does this through two key features: lightweight components and a pluggable look and feel. Together they provide an elegant, yet easy-to-use solution to the problems of the AWT. More than anything else, it is these two features that define the essence of Swing. Each is examined here.

Swing Components Are Lightweight

With very few exceptions, Swing components are *lightweight*. This means that they are written entirely in Java and do not map directly to platform-specific peers. Thus, lightweight components are more efficient and more flexible.

Furthermore, because lightweight components do not translate into native peers, the look and feel of each component is determined by Swing, not by the underlying operating system. As a result, each component will work in a consistent manner across all platforms.

Swing Supports a Pluggable Look and Feel

Swing supports a *pluggable look and feel* (PLAF). Because each Swing component is rendered by Java code rather than by native peers, the look and feel of a component is under the control of Swing. This fact means that it is possible to separate the look and feel of a component from the logic of the component, and this is what Swing does. Separating out the look and feel provides a significant advantage: it becomes possible to change the way that a component is rendered without affecting any of its other aspects. In other words, it is possible to “plug in” a new look and feel for any given component without creating any side effects in the code that uses that component. Moreover, it becomes possible to define entire sets of look-and-feels that represent different GUI styles. To use a specific style, its look and feel is simply “plugged in.” Once this is done, all components are automatically rendered using that style.

Pluggable look-and-feels offer several important advantages. It is possible to define a look and feel that is consistent across all platforms. Conversely, it is possible to create a look and feel that acts like a specific platform. It is also possible to design a custom look and feel. Finally, the look and feel can be changed dynamically at run time.

Java provides look-and-feels, such as metal and Nimbus, that are available to all Swing users. The metal look and feel is also called the *Java look and feel*. It is platform-independent and available in all Java execution environments. It is also the default look and feel. This book uses the default Java look and feel (metal) because it is platform independent.

The MVC Connection

In general, a visual component is a composite of three distinct aspects:

- The way that the component looks when rendered on the screen
- The way that the component reacts to the user
- The state information associated with the component

No matter what architecture is used to implement a component, it must implicitly contain these three parts. Over the years, one component architecture has proven itself to be exceptionally effective: *Model-View-Controller*, or MVC for short.

The MVC architecture is successful because each piece of the design corresponds to an aspect of a component. In MVC terminology, the *model* corresponds to the state information associated with the component. For example, in the case of a check box, the model contains a field that indicates if the box is checked or unchecked. The *view* determines how the component is displayed on the screen, including any aspects of the view that are affected by the current state of the model. The *controller* determines how the component reacts to the user. For example, when the user clicks a check box, the controller reacts by changing the model to reflect the user's choice (checked or unchecked). This then results in the view being updated. By separating a component into a model, a view, and a controller, the specific implementation of each can be changed without affecting the other two. For instance, different view implementations can render the same component in different ways without affecting the model or the controller.

Although the MVC architecture and the principles behind it are conceptually sound, the high level of separation between the view and the controller is not beneficial for Swing components. Instead, Swing uses a modified version of MVC that combines the view and the controller into a single logical entity called the *UI delegate*. For this reason, Swing's approach is called either the *Model-Delegate* architecture or the *Separable Model* architecture. Therefore, although Swing's component architecture is based on MVC, it does not use a classical implementation of it.

Swing's pluggable look and feel is made possible by its Model-Delegate architecture. Because the view (look) and controller (feel) are separate from the model, the look and feel can be changed without affecting how the component is used within a program. Conversely, it is possible to customize the model without affecting the way that the component appears on the screen or responds to user input.

To support the Model-Delegate architecture, most Swing components contain two objects. The first represents the model. The second represents the UI delegate. Models are defined by interfaces. For example, the model for a button is defined by the **ButtonModel** interface. UI delegates are classes that inherit **ComponentUI**. For example, the UI delegate for a button is **ButtonUI**. Normally, your programs will not interact directly with the UI delegate.

Components and Containers

A Swing GUI consists of two key items: *components* and *containers*. However, this distinction is mostly conceptual because all containers are also components. The difference between the two is found in their intended purpose: As the term is commonly used, a *component* is an independent visual control, such as a push button or slider. A container holds a group of components. Thus, a container is a special type of component that is designed to hold other components. Furthermore, in order for a component to be displayed, it must be held within a container. Thus, all Swing GUIs will have at least one container. Because containers are components, a container can also hold other containers. This enables Swing to define what is called a *containment hierarchy*, at the top of which must be a *top-level container*.

Let's look a bit more closely at components and containers.

Components

In general, Swing components are derived from the **JComponent** class. (The only exceptions to this are the four top-level containers, described in the next section.) **JComponent** provides the functionality that is common to all components. For example, **JComponent** supports the pluggable look and feel. **JComponent** inherits the AWT classes **Container** and **Component**. Thus, a Swing component is built on and compatible with an AWT component.

All of Swing's components are represented by classes defined within the package **javax.swing**. The following table shows the class names for Swing components (including those used as containers).

JApplet (Deprecated)	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayer
JLayeredPane	JList	JMenu	JMenuBar
JMenuItem	JOptionPane	JPanel	JPasswordField
JPopupMenu	JProgressBar	JRadioButton	JRadioButtonMenuItem
JRootPane	JScrollBar	JScrollPane	JSeparator
JSlider	JSpinner	JSplitPane	JTabbedPane
JTable	JTextArea	JTextField	JTextPane
JToggleButton	JToolBar	JToolTip	JTree
JViewport	JWindow		

Notice that all component classes begin with the letter **J**. For example, the class for a label is **JLabel**; the class for a push button is **JButton**; and the class for a scroll bar is **JScrollBar**.

Containers

Swing defines two types of containers. The first are top-level containers: **JFrame**, **JApplet**, **JWindow**, and **JDialog**. These containers do not inherit **JComponent**. They do, however, inherit the AWT classes **Component** and **Container**. Unlike Swing's other components, which are lightweight, the top-level containers are heavyweight. This makes the top-level containers a special case in the Swing component library.

As the name implies, a top-level container must be at the top of a containment hierarchy. A top-level container is not contained within any other container. Furthermore, every containment hierarchy must begin with a top-level container. The one most commonly used for applications is **JFrame**. In the past, the one used for applets was **JApplet**. As explained in [Chapter 1](#), beginning with JDK 9 applets have been deprecated. As a result, **JApplet** is deprecated. Furthermore, beginning with JDK 11, applet support has been removed.

The second type of containers supported by Swing are lightweight containers. Lightweight containers *do* inherit **JComponent**. An example of a

lightweight container is **JPanel**, which is a general-purpose container. Lightweight containers are often used to organize and manage groups of related components because a lightweight container can be contained within another container. Thus, you can use lightweight containers such as **JPanel** to create subgroups of related controls that are contained within an outer container.

The Top-Level Container Panes

Each top-level container defines a set of *panes*. At the top of the hierarchy is an instance of **JRootPane**. **JRootPane** is a lightweight container whose purpose is to manage the other panes. It also helps manage the optional menu bar. The panes that comprise the root pane are called the *glass pane*, the *content pane*, and the *layered pane*.

The glass pane is the top-level pane. It sits above and completely covers all other panes. By default, it is a transparent instance of **JPanel**. The glass pane enables you to manage mouse events that affect the entire container (rather than an individual control) or to paint over any other component, for example. In most cases, you won't need to use the glass pane directly, but it is there if you need it.

The layered pane is an instance of **JLayeredPane**. The layered pane allows components to be given a depth value. This value determines which component overlays another. (Thus, the layered pane lets you specify a Z-order for a component, although this is not something that you will usually need to do.) The layered pane holds the content pane and the (optional) menu bar.

Although the glass pane and the layered panes are integral to the operation of a top-level container and serve important purposes, much of what they provide occurs behind the scene. The pane with which your application will interact the most is the content pane, because this is the pane to which you will add visual components. In other words, when you add a component, such as a button, to a top-level container, you will add it to the content pane. By default, the content pane is an opaque instance of **JPanel**.

The Swing Packages

Swing is a very large subsystem and makes use of many packages. At the time of this writing, these are the packages defined by Swing.

javax.swing	javax.swing.plaf.basic	javax.swing.text
javax.swing.border	javax.swing.plaf.metal	javax.swing.text.html
javax.swing.colorchooser	javax.swing.plaf.multi	javax.swing.text.html.parser
javax.swing.event	javax.swing.plaf.nimbus	javax.swing.text.rtf
javax.swing.filechooser	javax.swing.plaf.synth	javax.swing.tree
javax.swing.plaf	javax.swing.table	javax.swing.undo

Beginning the JDK 9, the Swing packages are part of the **java.desktop** module.

The main package is **javax.swing**. This package must be imported into any program that uses Swing. It contains the classes that implement the basic Swing components, such as push buttons, labels, and check boxes.

A Simple Swing Application

Swing programs differ from both the console-based programs and the AWT-based programs shown earlier in this book. For example, they use a different set of components and a different container hierarchy than does the AWT. Swing programs also have special requirements that relate to threading. The best way to understand the structure of a Swing program is to work through an example. Before we begin, it is necessary to point out that in the past there were two types of Java programs in which Swing was typically used. The first is a desktop application. This type of Swing application is widely used, and is the type of Swing program described here. The second is the applet. Because applets are now deprecated and not suitable for use in new code, they are not discussed in this book.

Although quite short, the following program shows one way to write a Swing application. In the process, it demonstrates several key features of Swing. It uses two Swing components: **JFrame** and **JLabel**. **JFrame** is the top-level container that is commonly used for Swing applications. **JLabel** is the Swing component that creates a label, which is a component that displays information. The label is Swing's simplest component because it is passive. That is, a label does not respond to user input. It just displays output. The program uses a **JFrame** container to hold an instance of a **JLabel**. The label displays a short text message.

```
// A simple Swing application.

import javax.swing.*;

class SwingDemo {

    SwingDemo() {

        // Create a new JFrame container.
        JFrame jfrm = new JFrame("A Simple Swing Application");

        // Give the frame an initial size.
        jfrm.setSize(275, 100);

        // Terminate the program when the user closes the application.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Create a text-based label.
        JLabel jlab = new JLabel(" Swing means powerful GUIs.");

        // Add the label to the content pane.
        jfrm.add(jlab);

        // Display the frame.
        jfrm.setVisible(true);
    }

    public static void main(String args[]) {
        // Create the frame on the event dispatching thread.
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new SwingDemo();
            }
        });
    }
}
```

Swing programs are compiled and run in the same way as other Java applications. Thus, to compile this program, you can use this command line:

```
javac SwingDemo.java
```

To run the program, use this command line:

```
java SwingDemo
```

When the program is run, it will produce a window similar to that shown in [Figure 31-1](#).



Figure 31-1 The window produced by the **SwingDemo** program

Because the **SwingDemo** program illustrates several core Swing concepts, we will examine it carefully, line by line. The program begins by importing **javax.swing**. As mentioned, this package contains the components and models defined by Swing. For example, **javax.swing** defines classes that implement labels, buttons, text controls, and menus. It will be included in all programs that use Swing.

Next, the program declares the **SwingDemo** class and a constructor for that class. The constructor is where most of the action of the program occurs. It begins by creating a **JFrame**, using this line of code:

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

This creates a container called **jfrm** that defines a rectangular window complete with a title bar; close, minimize, maximize, and restore buttons; and a system menu. Thus, it creates a standard, top-level window. The title of the window is passed to the constructor.

Next, the window is sized using this statement:

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

The **setSize()** method (which is inherited by **JFrame** from the AWT class **Component**) sets the dimensions of the window, which are specified in pixels. Its general form is shown here:

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

In this example, the width of the window is set to 275 and the height is set to 100.

By default, when a top-level window is closed (such as when the user clicks the close box), the window is removed from the screen, but the application is not terminated. While this default behavior is useful in some situations, it is not what is needed for most applications. Instead, you will usually want the entire application to terminate when its top-level window is closed. There are a couple of ways to achieve this. The easiest way is to call **setDefaultCloseOperation()**, as the program does:

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

After this call executes, closing the window causes the entire application to terminate. The general form of **setDefaultCloseOperation()** is shown here:

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

The value passed in *what* determines what happens when the window is closed. There are several other options in addition to **JFrame.EXIT_ON_CLOSE**. They are shown here:

DISPOSE_ON_CLOSE

HIDE_ON_CLOSE

DO NOTHING ON CLOSE

Their names reflect their actions. These constants are declared in **WindowConstants**, which is an interface declared in **javax.swing** that is implemented by **JFrame**.

The next line of code creates a Swing **JLabel** component:

```
JLabel jlab = new JLabel(" Swing means powerful GUIs.");
```

JLabel is the simplest and easiest-to-use component because it does not accept user input. It simply displays information, which can consist of text, an icon, or a combination of the two. The label created by the program contains only text, which is passed to its constructor.

The next line of code adds the label to the content pane of the frame:

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

As explained earlier, all top-level containers have a content pane in which

components are stored. Thus, to add a component to a frame, you must add it to the frame's content pane. This is accomplished by calling **add()** on the **JFrame** reference (**jfrm** in this case). The general form of **add()** is shown here:

Component **add(Component comp)**

The **add()** method is inherited by **JFrame** from the AWT class **Container**.

By default, the content pane associated with a **JFrame** uses border layout. The version of **add()** just shown adds the label to the center location. Other versions of **add()** enable you to specify one of the border regions. When a component is added to the center, its size is adjusted automatically to fit the size of the center.

Before continuing, an important historical point needs to be made. Prior to JDK 5, when adding a component to the content pane, you could not invoke the **add()** method directly on a **JFrame** instance. Instead, you needed to call **add()** on the content pane of the **JFrame** object. The content pane can be obtained by calling **getContentPane()** on a **JFrame** instance. The **getContentPane()** method is shown here:

Container **getContentPane()**

It returns a **Container** reference to the content pane. The **add()** method was then called on that reference to add a component to a content pane. Thus, in the past, you had to use the following statement to add **jlab** to **jfrm**:

```
jfrm.getContentPane().add(jlab); // old-style
```

Here, **getContentPane()** first obtains a reference to content pane, and then **add()** adds the component to the container linked to this pane. This same procedure was also required to invoke **remove()** to remove a component and **setLayout()** to set the layout manager for the content pane. This is why you will see explicit calls to **getContentPane()** frequently throughout pre-5.0 legacy code. Today, the use of **getContentPane()** is no longer necessary. You can simply call **add()**, **remove()**, and **setLayout()** directly on **JFrame** because these methods have been changed so that they operate on the content pane automatically.

The last statement in the **SwingDemo** constructor causes the window to become visible:

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

The **setVisible()** method is inherited from the AWT **Component** class. If its argument is **true**, the window will be displayed. Otherwise, it will be hidden. By default, a **JFrame** is invisible, so **setVisible(true)** must be called to show it.

Inside **main()**, a **SwingDemo** object is created, which causes the window and the label to be displayed. Notice that the **SwingDemo** constructor is invoked using these lines of code:

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

This sequence causes a **SwingDemo** object to be created on the *event dispatching thread* rather than on the main thread of the application. Here's why. In general, Swing programs are event-driven. For example, when a user interacts with a component, an event is generated. An event is passed to the application by calling an event handler defined by the application. However, the handler is executed on the event dispatching thread provided by Swing and not on the main thread of the application. Thus, although event handlers are defined by your program, they are called on a thread that was not created by your program.

To avoid problems (including the potential for deadlock), all Swing GUI components must be created and updated from the event dispatching thread, not the main thread of the application. However, **main()** is executed on the main thread. Thus, **main()** cannot directly instantiate a **SwingDemo** object. Instead, it must create a **Runnable** object that executes on the event dispatching thread and have this object create the GUI.

To enable the GUI code to be created on the event dispatching thread, you must use one of two methods that are defined by the **SwingUtilities** class. These methods are **invokeLater()** and **invokeAndWait()**. They are shown here:

```
static void invokeLater(Runnable obj)
```

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

Here, `obj` is a **Runnable** object that will have its `run()` method called by the event dispatching thread. The difference between the two methods is that `invokeLater()` returns immediately, but `invokeAndWait()` waits until `obj.run()` returns. You can use one of these methods to call a method that constructs the GUI for your Swing application, or whenever you need to modify the state of the GUI from code not executed by the event dispatching thread. You will normally want to use `invokeLater()`, as the preceding program does. However, when the initial GUI for an applet is constructed, `invokeAndWait()` is required. Thus, you will see its use in legacy applet code.

Event Handling

The preceding example showed the basic form of a Swing program, but it left out one important part: event handling. Because **JLabel** does not take input from the user, it does not generate events, so no event handling was needed. However, the other Swing components *do* respond to user input and the events generated by those interactions need to be handled. Events can also be generated in ways not directly related to user input. For example, an event is generated when a timer goes off. Whatever the case, event handling is a large part of any Swing-based application.

The event handling mechanism used by Swing is the same as that used by the AWT. This approach is called the *delegation event model*, and it is described in [Chapter 24](#). In many cases, Swing uses the same events as does the AWT, and these events are packaged in **java.awt.event**. Events specific to Swing are stored in **javax.swing.event**.

Although events are handled in Swing in the same way as they are with the AWT, it is still useful to work through a simple example. The following program handles the event generated by a Swing push button. Sample output is shown in [Figure 31-2](#).

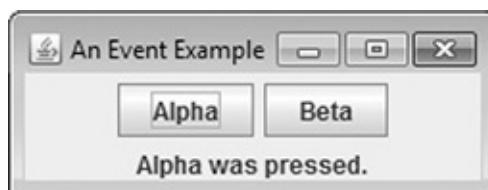


Figure 31-2 Output from the **EventDemo** program

```
// Handle an event in a Swing program.

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

class EventDemo {

    JLabel jlab;

    EventDemo()  {

        // Create a new JFrame container.
        JFrame jfrm = new JFrame("An Event Example");

        // Specify FlowLayout for the layout manager.
        jfrm.setLayout(new FlowLayout());

        // Give the frame an initial size.
        jfrm.setSize(220, 90);
    }
}
```

```
// Terminate the program when the user closes the application.  
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
// Make two buttons.  
JButton jbtnAlpha = new JButton("Alpha");  
JButton jbtnBeta = new JButton("Beta");  
  
// Add action listener for Alpha.  
jbtnAlpha.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent ae) {  
        jlab.setText("Alpha was pressed.");  
    }  
});  
  
// Add action listener for Beta.  
jbtnBeta.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent ae) {  
        jlab.setText("Beta was pressed.");  
    }  
});  
  
// Add the buttons to the content pane.  
jfrm.add(jbtnAlpha);  
jfrm.add(jbtnBeta);  
  
// Create a text-based label.  
jlab = new JLabel("Press a button.");  
  
// Add the label to the content pane.  
jfrm.add(jlab);  
  
// Display the frame.  
jfrm.setVisible(true);  
}  
  
public static void main(String args[]) {  
    // Create the frame on the event dispatching thread.  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            new EventDemo();  
        }  
    });  
}
```

First, notice that the program now imports both the **java.awt** and **java.awt.event** packages. The **java.awt** package is needed because it contains the **FlowLayout** class, which supports the standard flow layout manager used to lay out components in a frame. (See Chapter 26 for coverage of layout managers.) The **java.awt.event** package is needed because it defines the **ActionListener** interface and the **ActionEvent** class.

The **EventDemo** constructor begins by creating a **JFrame** called **jfrm**. It then sets the layout manager for the content pane of **jfrm** to **FlowLayout**. By default, the content pane uses **BorderLayout** as its layout manager. However, for this example, **FlowLayout** is more convenient.

After setting the size and default close operation, **EventDemo()** creates two push buttons, as shown here:

```
 JButton jbtnAlpha = new JButton("Alpha");
 JButton jbtnBeta = new JButton("Beta");
```

The first button will contain the text "Alpha" and the second will contain the text "Beta". Swing push buttons are instances of **JButton**. **JButton** supplies several constructors. The one used here is

```
JButton(String msg)
```

The *msg* parameter specifies the string that will be displayed inside the button.

When a push button is pressed, it generates an **ActionEvent**. Thus, **JButton** provides the **addActionListener()** method, which is used to add an action listener. (**JButton** also provides **removeActionListener()** to remove a listener, but this method is not used by the program.) As explained in [Chapter 24](#), the **ActionListener** interface defines only one method: **actionPerformed()**. It is shown again here for your convenience:

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

This method is called when a button is pressed. In other words, it is the event handler that is called when a button press event has occurred.

Next, event listeners for the button's action events are added by the code shown here:

```

// Add action listener for Alpha.
jbtnAlpha.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("Alpha was pressed.");
    }
});

// Add action listener for Beta.
jbtnBeta.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("Beta was pressed.");
    }
});

```

Here, anonymous inner classes are used to provide the event handlers for the two buttons. Each time a button is pressed, the string displayed in **jlab** is changed to reflect which button was pressed.

Beginning with JDK 8, lambda expressions can also be used to implement some types of event handlers. For example, the event handler for the Alpha button could be written like this:

```
jbtnAlpha.addActionListener( (ae) -> jlab.setText("Alpha was pressed."));
```

As you can see, this code is shorter. Of course, the approach you choose will be determined by the situation and your own preferences.

Next, the buttons are added to the content pane of **jfrm**:

```
jfrm.add(jbtnAlpha);
jfrm.add(jbtnBeta);
```

Finally, **jlab** is added to the content pane and the window is made visible. When you run the program, each time you press a button, a message is displayed in the label that indicates which button was pressed.

One last point: Remember that all event handlers, such as **actionPerformed()**, are called on the event dispatching thread. Therefore, an event handler must return quickly in order to avoid slowing down the application. If your application needs to do something time consuming as the result of an event, it must use a separate thread.

Painting in Swing



Although the Swing component set is quite powerful, you are not limited to using it because Swing also lets you write directly into the display area of a frame, panel, or one of Swing's other components, such as **JLabel**. Although many (perhaps most) uses of Swing will *not* involve drawing directly to the surface of a component, it is available for those applications that need this capability. To write output directly to the surface of a component, you will use one or more drawing methods defined by the AWT, such as **drawLine()** or **drawRect()**. Thus, most of the techniques and methods described in [Chapter 25](#) also apply to Swing. However, there are also some very important differences, and the process is discussed in detail in this section.

Painting Fundamentals

Swing's approach to painting is built on the original AWT-based mechanism, but Swing's implementation offers more finely grained control. Before examining the specifics of Swing-based painting, it is useful to review the AWT-based mechanism that underlies it.

The AWT class **Component** defines a method called **paint()** that is used to draw output directly to the surface of a component. For the most part, **paint()** is not called by your program. (In fact, only in the most unusual cases should it ever be called by your program.) Rather, **paint()** is called by the run-time system whenever a component must be rendered. This situation can occur for several reasons. For example, the window in which the component is displayed can be overwritten by another window and then uncovered. Or, the window might be minimized and then restored. The **paint()** method is also called when a program begins running. When writing AWT-based code, an application will override **paint()** when it needs to write output directly to the surface of the component.

Because **JComponent** inherits **Component**, all Swing's lightweight components inherit the **paint()** method. However, you *will not* override it to paint directly to the surface of a component. The reason is that Swing uses a bit more sophisticated approach to painting that involves three distinct methods: **paintComponent()**, **paintBorder()**, and **paintChildren()**. These methods paint the indicated portion of a component and divide the painting process into its three distinct, logical actions. In a lightweight component, the original AWT method **paint()** simply executes calls to these methods, in the order just shown.

To paint to the surface of a Swing component, you will create a subclass of

the component and then override its **paintComponent()** method. This is the method that paints the interior of the component. You will not normally override the other two painting methods. When overriding **paintComponent()**, the first thing you must do is call **super.paintComponent()**, so that the superclass portion of the painting process takes place. (The only time this is not required is when you are taking complete, manual control over how a component is displayed.) After that, write the output that you want to display. The **paintComponent()** method is shown here:

```
protected void paintComponent(Graphics g)
```

The parameter *g* is the graphics context to which output is written.

To cause a component to be painted under program control, call **repaint()**. It works in Swing just as it does for the AWT. The **repaint()** method is defined by **Component**. Calling it causes the system to call **paint()** as soon as it is possible to do so. Because painting is a time-consuming operation, this mechanism allows the run-time system to defer painting momentarily until some higher-priority task has completed, for example. Of course, in Swing the call to **paint()** results in a call to **paintComponent()**. Therefore, to output to the surface of a component, your program will store the output until **paintComponent()** is called. Inside the overridden **paintComponent()**, you will draw the stored output.

Compute the Paintable Area

When drawing to the surface of a component, you must be careful to restrict your output to the area that is inside the border. Although Swing automatically clips any output that will exceed the boundaries of a component, it is still possible to paint into the border, which will then get overwritten when the border is drawn. To avoid this, you must compute the *paintable area* of the component. This is the area defined by the current size of the component minus the space used by the border. Therefore, before you paint to a component, you must obtain the width of the border and then adjust your drawing accordingly.

To obtain the border width, call **getInsets()**, shown here:

```
Insets getInsets()
```

This method is defined by **Container** and overridden by **JComponent**. It returns an **Insets** object that contains the dimensions of the border. The inset

values can be obtained by using these fields:

```
int top;  
int bottom;  
int left;  
int right;
```

These values are then used to compute the drawing area given the width and the height of the component. You can obtain the width and height of the component by calling **getWidth()** and **getHeight()** on the component. They are shown here:

```
int getWidth()  
int getHeight()
```

By subtracting the value of the insets, you can compute the usable width and height of the component.

A Paint Example

Here is a program that puts into action the preceding discussion. It creates a class called **PaintPanel** that extends **JPanel**. The program then uses an object of that class to display lines whose endpoints have been generated randomly. Sample output is shown in [Figure 31-3](#).

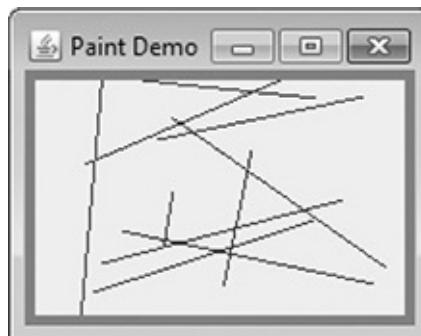


Figure 31-3 Sample output from the **PaintPanel** program

```
// Paint lines to a panel.

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

// This class extends JPanel. It overrides
// the paintComponent() method so that random
// lines are plotted in the panel.
class PaintPanel extends JPanel {
    Insets ins; // holds the panel's insets

    Random rand; // used to generate random numbers

    // Construct a panel.
    PaintPanel() {

        // Put a border around the panel.
        setBorder(
            BorderFactory.createLineBorder(Color.RED, 5));

        rand = new Random();
    }
}
```

```
// Override the paintComponent() method.
protected void paintComponent(Graphics g) {
    // Always call the superclass method first.
    super.paintComponent(g);

    int x, y, x2, y2;

    // Get the height and width of the component.
    int height = getHeight();
    int width = getWidth();

    // Get the insets.
    ins = getInsets();

    // Draw ten lines whose endpoints are randomly generated.
    for(int i=0; i < 10; i++) {
        // Obtain random coordinates that define
        // the endpoints of each line.
        x = rand.nextInt(width-ins.left);
        y = rand.nextInt(height-ins.bottom);
        x2 = rand.nextInt(width-ins.left);
        y2 = rand.nextInt(height-ins.bottom);

        // Draw the line.
        g.drawLine(x, y, x2, y2);
    }
}

// Demonstrate painting directly onto a panel.
class PaintDemo {

    JLabel jlab;
    PaintPanel pp;

    PaintDemo() {

        // Create a new JFrame container.
        JFrame jfrm = new JFrame("Paint Demo");

        // Give the frame an initial size.
        jfrm.setSize(200, 150);

        // Terminate the program when the user closes the application.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Create the panel that will be painted.
        pp = new PaintPanel();

        // Add the panel to the content pane. Because the default
        // border layout is used, the panel will automatically be
        // sized to fit the center region.
        jfrm.add(pp);
    }
}
```

```

        // Display the frame.
        jfrm.setVisible(true);
    }

public static void main(String args[]) {
    // Create the frame on the event dispatching thread.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new PaintDemo();
        }
    });
}
}

```

Let's examine this program closely. The **PaintPanel** class extends **JPanel**. **JPanel** is one of Swing's lightweight containers, which means that it is a component that can be added to the content pane of a **JFrame**. To handle painting, **PaintPanel** overrides the **paintComponent()** method. This enables **PaintPanel** to write directly to the surface of the component when painting takes place. The size of the panel is not specified because the program uses the default border layout and the panel is added to the center. This results in the panel being sized to fill the center. If you change the size of the window, the size of the panel will be adjusted accordingly.

Notice that the constructor also specifies a 5-pixel wide, red border. This is accomplished by setting the border by using the **setBorder()** method, shown here:

`void setBorder(Border border)`

Border is the Swing interface that encapsulates a border. You can obtain a border by calling one of the factory methods defined by the **BorderFactory** class. The one used in the program is **createLineBorder()**, which creates a simple line border. It is shown here:

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

Here, *clr* specifies the color of the border and *width* specifies its width in pixels.

Inside the override of **paintComponent()**, notice that it first calls **super.paintComponent()**. As explained, this is necessary to ensure that the

component is properly drawn. Next, the width and height of the panel are obtained along with the insets. These values are used to ensure the lines lie within the drawing area of the panel. The drawing area is the overall width and height of a component less the border width. The computations are designed to work with differently sized **PaintPanels** and borders. To prove this, try changing the size of the window. The lines will still all lie within the borders of the panel.

The **PaintDemo** class creates a **PaintPanel** and then adds the panel to the content pane. When the application is first displayed, the overridden **paintComponent()** method is called, and the lines are drawn. Each time you resize or hide and restore the window, a new set of lines are drawn. In all cases, the lines fall within the paintable area.

CHAPTER

Exploring Swing

The previous chapter described several of the core concepts relating to Swing and showed the general form of a Swing application. This chapter continues the discussion of Swing by presenting an overview of several Swing components, such as buttons, check boxes, trees, and tables. The Swing components provide rich functionality and allow a high level of customization. Because of space limitations, it is not possible to describe all of their features and attributes. Rather, the purpose of this overview is to give you a feel for the capabilities of the Swing component set.

The Swing component classes described in this chapter are shown here:

JButton	JCheckBox	JComboBox	JLabel
JList	JRadioButton	JScrollPane	JTabbedPane
JTable	JTextField	JToggleButton	JTree

These components are all lightweight, which means that they are all derived from **JComponent**.

Also discussed is the **ButtonGroup** class, which encapsulates a mutually exclusive set of Swing buttons, and **ImageIcon**, which encapsulates a graphics image. Both are defined by Swing and packaged in **javax.swing**.

JLabel and **ImageIcon**

JLabel is Swing's easiest-to-use component. It creates a label and was introduced in the preceding chapter. Here, we will look at **JLabel** a bit more closely. **JLabel** can be used to display text and/or an icon. It is a passive component in that it does not respond to user input. **JLabel** defines several constructors. Here are three of them:

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

Here, *str* and *icon* are the text and icon used for the label. The *align* argument specifies the horizontal alignment of the text and/or icon within the dimensions of the label. It must be one of the following values: **LEFT**, **RIGHT**, **CENTER**, **LEADING**, or **TRAILING**. These constants are defined in the **SwingConstants** interface, along with several others used by the Swing classes.

Notice that icons are specified by objects of type **Icon**, which is an interface defined by Swing. The easiest way to obtain an icon is to use the **ImageIcon** class. **ImageIcon** implements **Icon** and encapsulates an image. Thus, an object of type **ImageIcon** can be passed as an argument to the **Icon** parameter of **JLabel**'s constructor. There are several ways to provide the image, including reading it from a file or downloading it from a URL. Here is the **ImageIcon** constructor used by the example in this section:

```
ImageIcon(String filename)
```

It obtains the image in the file named *filename*.

The icon and text associated with the label can be obtained by the following methods:

```
Icon getIcon()
String getText()
```

The icon and text associated with a label can be set by these methods:

```
void setIcon(Icon icon)
void setText(String str)
```

Here, *icon* and *str* are the icon and text, respectively. Therefore, using **setText()** it is possible to change the text inside a label during program execution.

The following program illustrates how to create and display a label containing both an icon and a string. It begins by creating an **ImageIcon** object for the file **hourglass.png**, which depicts an hourglass. This is used as the second argument to the **JLabel** constructor. The first and last arguments for the **JLabel** constructor are the label text and the alignment. Finally, the label is added to the content pane.

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

public class JLabelDemo {

    public JLabelDemo() {

        // Set up the JFrame.
        JFrame jfrm = new JFrame("JLabelDemo");
        jfrm.setLayout(new FlowLayout());
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jfrm.setSize(260, 210);

        // Create an icon.
        ImageIcon ii = new ImageIcon("hourglass.png");

        // Create a label.
        JLabel jl = new JLabel("Hourglass", ii, JLabel.CENTER);

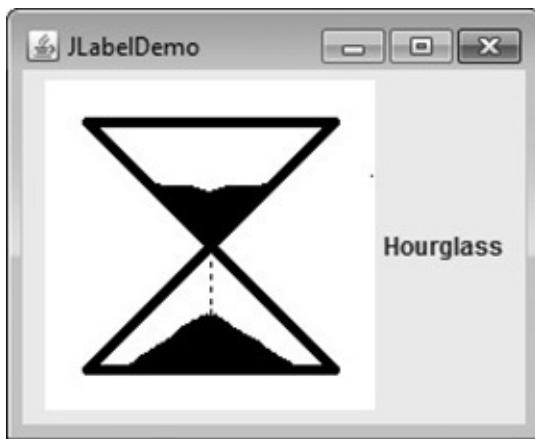
        // Add the label to the content pane.
        jfrm.add(jl);

        // Display the frame.
        jfrm.setVisible(true);
    }

    public static void main(String[] args) {
        // Create the frame on the event dispatching thread.

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

Output from the label example is shown here:



JTextField

JTextField is the simplest Swing text component. It is also probably its most widely used text component. **JTextField** allows you to edit one line of text. It is derived from **JTextComponent**, which provides the basic functionality common to Swing text components. **JTextField** uses the **Document** interface for its model. Three of **JTextField**'s constructors are shown here:

```
JTextField(int cols)  
JTextField(String str, int cols)  
JTextField(String str)
```

Here, *str* is the string to be initially presented, and *cols* is the number of columns in the text field. If no string is specified, the text field is initially empty. If the number of columns is not specified, the text field is sized to fit the specified string.

JTextField generates events in response to user interaction. For example, an **ActionEvent** is fired when the user presses enter. A **CaretEvent** is fired each time the caret (i.e., the cursor) changes position. (**CaretEvent** is packaged in **javax.swing.event**.) Other events are also possible. In many cases, your program will not need to handle these events. Instead, you will simply obtain the string currently in the text field when it is needed. To obtain the text currently in the text field, call **getText()**.

The following example illustrates **JTextField**. It creates a **JTextField** and adds it to the content pane. When the user presses enter, an action event is

generated. This is handled by displaying the text in a label.

```
// Demonstrate JTextField.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JTextFieldDemo {

    public JTextFieldDemo() {

        // Set up the JFrame.
        JFrame jfrm = new JFrame("JTextFieldDemo");
        jfrm.setLayout(new FlowLayout());
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jfrm.setSize(260, 120);

        // Add a text field to content pane.
        JTextField jtf = new JTextField(15);
        jfrm.add(jtf);

        // Add a label.
        JLabel jlab = new JLabel();
        jfrm.add(jlab);

        // Handle action events.
        jtf.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                // Show text when user presses ENTER.
                jlab.setText(jtf.getText());
            }
        });

        // Display the frame.
        jfrm.setVisible(true);
    }

    public static void main(String[] args) {
        // Create the frame on the event dispatching thread.

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

Output from the text field example is shown here:



The Swing Buttons

Swing defines four types of buttons: **JButton**, **JToggleButton**, **JCheckBox**, and **JRadioButton**. All are subclasses of the **AbstractButton** class, which extends **JComponent**. Thus, all buttons share a set of common traits.

AbstractButton contains many methods that allow you to control the behavior of buttons. For example, you can define different icons that are displayed for the button when it is disabled, pressed, or selected. Another icon can be used as a *rollover* icon, which is displayed when the mouse is positioned over a button. The following methods set these icons:

```
void setDisabledIcon(Icon di)
void setPressedIcon(Icon pi)
void setSelectedIcon(Icon si)
void setRolloverIcon(Icon ri)
```

Here, *di*, *pi*, *si*, and *ri* are the icons to be used for the indicated purpose.

The text associated with a button can be read and written via the following methods:

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

Here, *str* is the text to be associated with the button.

The model used by all buttons is defined by the **ButtonModel** interface. A button generates an action event when it is pressed. Other events are possible. Each of the concrete button classes is examined next.

JButton

The **JButton** class provides the functionality of a push button. You have already seen a simple form of it in the preceding chapter. **JButton** allows an icon, a string, or both to be associated with the push button. Three of its constructors are shown here:

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

Here, *str* and *icon* are the string and icon used for the button.

When the button is pressed, an **ActionEvent** is generated. Using the **ActionEvent** object passed to the **actionPerformed()** method of the registered **ActionListener**, you can obtain the *action command* string associated with the button. By default, this is the string displayed inside the button. However, you can set the action command by calling **setActionCommand()** on the button. You can obtain the action command by calling **getActionCommand()** on the event object. It is declared like this:

```
String getActionCommand( )
```

The action command identifies the button. Thus, when using two or more buttons within the same application, the action command gives you an easy way to determine which button was pressed.

In the preceding chapter, you saw an example of a text-based button. The following demonstrates an icon-based button. It displays four push buttons and a label. Each button displays an icon that represents a timepiece. When a button is pressed, the name of that timepiece is displayed in the label.

```
// Demonstrate an icon-based JButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JButtonDemo implements ActionListener {
    JLabel jlab;

    public JButtonDemo() {

        // Set up the JFrame.
        JFrame jfrm = new JFrame("JButtonDemo");
        jfrm.setLayout(new FlowLayout());
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jfrm.setSize(500, 450);

        // Add buttons to content pane.
        ImageIcon hourglass = new ImageIcon("hourglass.png");
        JButton jb = new JButton(hourglass);
        jb.setActionCommand("Hourglass");
        jb.addActionListener(this);
        jfrm.add(jb);

        ImageIcon analog = new ImageIcon("analog.png");
        jb = new JButton(analog);
        jb.setActionCommand("Analog Clock");
        jb.addActionListener(this);
        jfrm.add(jb);

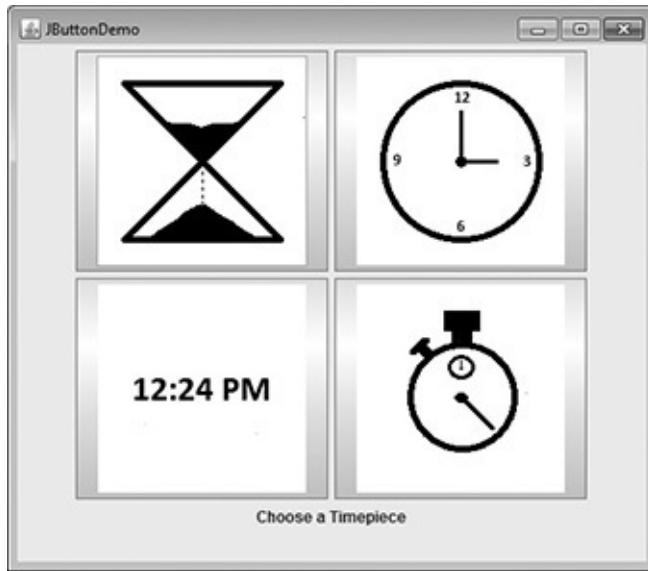
        ImageIcon digital = new ImageIcon("digital.png");
        jb = new JButton(digital);
        jb.setActionCommand("Digital Clock");
        jb.addActionListener(this);
        jfrm.add(jb);

        ImageIcon stopwatch = new ImageIcon("stopwatch.png");
        jb = new JButton(stopwatch);
        jb.setActionCommand("Stopwatch");
        jb.addActionListener(this);
        jfrm.add(jb);
    }

    public void actionPerformed(ActionEvent e) {
        String cmd = e.getActionCommand();
        if (cmd.equals("Hourglass")) {
            jlab.setText("Hourglass");
        } else if (cmd.equals("Analog Clock")) {
            jlab.setText("Analog Clock");
        } else if (cmd.equals("Digital Clock")) {
            jlab.setText("Digital Clock");
        } else if (cmd.equals("Stopwatch")) {
            jlab.setText("Stopwatch");
        }
    }
}
```

```
// Create and add the label to content pane.  
jlab = new JLabel("Choose a Timepiece");  
jfrm.add(jlab);  
  
// Display the frame.  
jfrm.setVisible(true);  
}  
  
// Handle button events.  
public void actionPerformed(ActionEvent ae) {  
    jlab.setText("You selected " + ae.getActionCommand());  
}  
  
public static void main(String[] args) {  
    // Create the frame on the event dispatching thread.  
  
    SwingUtilities.invokeLater(  
        new Runnable() {  
            public void run() {  
                new JButtonDemo();  
            }  
        }  
    );  
}
```

Output from the button example is shown here:



JToggleButton

A useful variation on the push button is called a *toggle button*. A toggle button looks just like a push button, but it acts differently because it has two states: pushed and released. That is, when you press a toggle button, it stays pressed rather than popping back up as a regular push button does. When you press the toggle button a second time, it releases (pops up). Therefore, each time a toggle button is pushed, it toggles between its two states.

Toggle buttons are objects of the **JToggleButton** class. **JToggleButton** implements **AbstractButton**. In addition to creating standard toggle buttons, **JToggleButton** is a superclass for two other Swing components that also represent two-state controls. These are **JCheckBox** and **JRadioButton**, which are described later in this chapter. Thus, **JToggleButton** defines the basic functionality of all two-state components.

JToggleButton defines several constructors. The one used by the example in this section is shown here:

```
JToggleButton(String str)
```

This creates a toggle button that contains the text passed in *str*. By default, the button is in the off position. Other constructors enable you to create toggle buttons that contain images, or images and text.

JToggleButton uses a model defined by a nested class called **JToggleButton.Toggle-ButtonModel**. Normally, you won't need to interact directly with the model to use a standard toggle button.

Like **JButton**, **JToggleButton** generates an action event each time it is pressed. Unlike **JButton**, however, **JToggleButton** also generates an item event. This event is used by those components that support the concept of selection. When a **JToggleButton** is pressed in, it is selected. When it is popped out, it is deselected.

To handle item events, you must implement the **ItemListener** interface. Recall from [Chapter 24](#), that each time an item event is generated, it is passed to the **itemStateChanged()** method defined by **ItemListener**. Inside **itemStateChanged()**, the **getItem()** method can be called on the **ItemEvent** object to obtain a reference to the **JToggleButton** instance that generated the event. It is shown here:

```
Object getItem( )
```

A reference to the button is returned. You will need to cast this reference to **JToggleButton**.

The easiest way to determine a toggle button's state is by calling the **isSelected()** method (inherited from **AbstractButton**) on the button that generated the event. It is shown here:

```
boolean isSelected( )
```

It returns **true** if the button is selected and **false** otherwise.

Here is an example that uses a toggle button. Notice how the item listener works. It simply calls **isSelected()** to determine the button's state.

```
// Demonstrate JToggleButton.  
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
  
public class JToggleButtonDemo {  
  
    public JToggleButtonDemo() {
```

```
// Set up the JFrame.  
JFrame jfrm = new JFrame("JToggledButtonDemo");  
jfrm.setLayout(new FlowLayout());  
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
jfrm.setSize(200, 100);  
  
// Create a label.  
JLabel jlab = new JLabel("Button is off.");  
  
// Make a toggle button.  
JToggleButton jtbn = new JToggleButton("On/Off");  
  
// Add an item listener for the toggle button.  
jtbn.addItemListener(new ItemListener() {  
    public void itemStateChanged(ItemEvent ie) {  
        if(jtbn.isSelected())  
            jlab.setText("Button is on.");  
        else  
            jlab.setText("Button is off.");  
    }  
});  
  
// Add the toggle button and label to the content pane.  
jfrm.add(jtbn);  
jfrm.add(jlab);  
  
// Display the frame.  
jfrm.setVisible(true);  
}  
  
public static void main(String[] args) {  
    // Create the frame on the event dispatching thread.  
  
    SwingUtilities.invokeLater(  
        new Runnable() {  
            public void run() {  
                new JToggledButtonDemo();  
            }  
        }  
    );  
}
```

The output from the toggle button example is shown here:



Check Boxes

The **JCheckBox** class provides the functionality of a check box. Its immediate superclass is **JToggleButton**, which provides support for two-state buttons, as just described. **JCheckBox** defines several constructors. The one used here is

```
JCheckBox(String str)
```

It creates a check box that has the text specified by *str* as a label. Other constructors let you specify the initial selection state of the button and specify an icon.

When the user selects or deselects a check box, an **ItemEvent** is generated. You can obtain a reference to the **JCheckBox** that generated the event by calling **getItem()** on the **ItemEvent** passed to the **itemStateChanged()** method defined by **ItemListener**. The easiest way to determine the selected state of a check box is to call **isSelected()** on the **JCheckBox** instance.

The following example illustrates check boxes. It displays four check boxes and a label. When the user clicks a check box, an **ItemEvent** is generated. Inside the **itemStateChanged()** method, **getItem()** is called to obtain a reference to the **JCheckBox** object that generated the event. Next, a call to **isSelected()** determines if the box was selected or cleared. The **getText()** method gets the text for that check box and uses it to set the text inside the label.

```
// Demonstrate JCheckbox.  
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
  
public class JCheckBoxDemo implements ItemListener {  
    JLabel jlab;  
  
    public JCheckBoxDemo() {  
  
        // Set up the JFrame.  
        JFrame jfrm = new JFrame("JCheckBoxDemo");  
        jfrm.setLayout(new FlowLayout());  
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        jfrm.setSize(250, 100);  
  
        // Add check boxes to the content pane.  
        JCheckBox cb = new JCheckBox("C");  
        cb.addItemListener(this);  
        jfrm.add(cb);  
  
        cb = new JCheckBox("C++");  
        cb.addItemListener(this);  
        jfrm.add(cb);  
  
        cb = new JCheckBox("Java");  
        cb.addItemListener(this);  
        jfrm.add(cb);
```

```
cb = new JCheckBox("Perl");
cb.addItemListener(this);
jfrm.add(cb);

// Create the label and add it to the content pane.
jlab = new JLabel("Select languages");
jfrm.add(jlab);

// Display the frame.
jfrm.setVisible(true);
}

// Handle item events for the check boxes.
public void itemStateChanged(ItemEvent ie) {
    JCheckBox cb = (JCheckBox) ie.getItem();

    if(cb.isSelected())
        jlab.setText(cb.getText() + " is selected");
    else
        jlab.setText(cb.getText() + " is cleared");
}

public static void main(String[] args) {
    // Create the frame on the event dispatching thread.

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

}
```

Output from this example is shown here:



Radio Buttons

Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time. They are supported by the **JRadioButton** class, which extends **JToggleButton**. **JRadioButton** provides several constructors. The one used in the example is shown here:

```
JRadioButton(String str)
```

Here, *str* is the label for the button. Other constructors let you specify the initial selection state of the button and specify an icon.

In order for their mutually exclusive nature to be activated, radio buttons must be configured into a group. Only one of the buttons in the group can be selected at any time. For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected. A button group is created by the **ButtonGroup** class. Its default constructor is invoked for this purpose. Elements are then added to the button group via the following method:

```
void add(AbstractButton ab)
```

Here, *ab* is a reference to the button to be added to the group.

A **JRadioButton** generates action events, item events, and change events each time the button selection changes. Most often, it is the action event that is handled, which means that you will normally implement the **ActionListener** interface. Recall that the only method defined by **ActionListener** is **actionPerformed()**. Inside this method, you can use a number of different ways to determine which button was selected. First, you can check its action command by calling **getActionCommand()**. By default, the action command is the same as the button label, but you can set the action command to something else by calling **setActionCommand()** on the radio button. Second, you can call **getSource()** on the **ActionEvent** object and check that reference against the buttons. Third, you can check each radio button to find out which one is currently selected by calling **isSelected()** on each button. Finally, each

button could use its own action event handler implemented as either an anonymous inner class or a lambda expression. Remember, each time an action event occurs, it means that the button being selected has changed and that one and only one button will be selected.

The following example illustrates how to use radio buttons. Three radio buttons are created. The buttons are then added to a button group. As explained, this is necessary to cause their mutually exclusive behavior. Pressing a radio button generates an action event, which is handled by **actionPerformed()**. Within that handler, the **getActionCommand()** method gets the text that is associated with the radio button and uses it to set the text within a label.

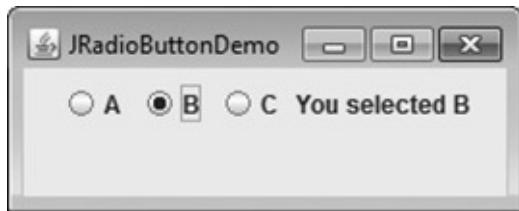
```
// Demonstrate JRadioButton
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JRadioButtonDemo implements ActionListener {
    JLabel jlab;

    public JRadioButtonDemo() {
        // Set up the JFrame.
        JFrame jfrm = new JFrame("JRadioButtonDemo");
        jfrm.setLayout(new FlowLayout());
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jfrm.setSize(250, 100);
```

```
// Create radio buttons and add them to content pane.  
JRadioButton b1 = new JRadioButton("A");  
b1.addActionListener(this);  
jfrm.add(b1);  
  
JRadioButton b2 = new JRadioButton("B");  
b2.addActionListener(this);  
jfrm.add(b2);  
  
JRadioButton b3 = new JRadioButton("C");  
b3.addActionListener(this);  
jfrm.add(b3);  
  
// Define a button group.  
ButtonGroup bg = new ButtonGroup();  
bg.add(b1);  
bg.add(b2);  
bg.add(b3);  
  
// Create a label and add it to the content pane.  
jlab = new JLabel("Select One");  
jfrm.add(jlab);  
  
// Display the frame.  
jfrm.setVisible(true);  
}  
  
// Handle button selection.  
public void actionPerformed(ActionEvent ae) {  
    jlab.setText("You selected " + ae.getActionCommand());  
}  
  
public static void main(String[] args) {  
    // Create the frame on the event dispatching thread.  
  
    SwingUtilities.invokeLater(  
        new Runnable() {  
            public void run() {  
                new JRadioButtonDemo();  
            }  
        }  
    );  
}
```

Output from the radio button example is shown here:



JTabbedPane

JTabbedPane encapsulates a *tabbed pane*. It manages a set of components by linking them with tabs. Selecting a tab causes the component associated with that tab to come to the forefront. Tabbed panes are very common in the modern GUI, and you have no doubt used them many times. Given the complex nature of a tabbed pane, they are surprisingly easy to create and use.

JTabbedPane defines three constructors. We will use its default constructor, which creates an empty control with the tabs positioned across the top of the pane. The other two constructors let you specify the location of the tabs, which can be along any of the four sides. **JTabbedPane** uses the **SingleSelectionModel** model.

Tabs are added by calling **addTab()**. Here is one of its forms:

```
void addTab(String name, Component comp)
```

Here, *name* is the name for the tab, and *comp* is the component that should be added to the tab. Often, the component added to a tab is a **JPanel** that contains a group of related components. This technique allows a tab to hold a set of components.

The general procedure to use a tabbed pane is outlined here:

1. Create an instance of **JTabbedPane**.
2. Add eachn tab by calling **addTab()**.
3. Add the tabbed pane to the content pane.

The following example illustrates a tabbed pane. The first tab is titled "Cities" and contains four buttons. Each button displays the name of a city. The second tab is titled "Colors" and contains three check boxes. Each check box displays the name of a color. The third tab is titled "Flavors" and contains one ~~checkbox box~~. This enables the user to select one of three flavors.

COMBO BOX. THIS CHANGES THE USER TO SELECT ONE OF THREE FLAVORS.

```
// Demonstrate JTabbedPane.  
import javax.swing.*;  
import java.awt.*;  
  
public class JTabbedPaneDemo {  
  
    public JTabbedPaneDemo() {  
  
        // Set up the JFrame.  
        JFrame jfrm = new JFrame("JTabbedPaneDemo");  
        jfrm.setLayout(new FlowLayout());  
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        jfrm.setSize(400, 200);  
  
        // Create the tabbed pane.  
        JTabbedPane jtp = new JTabbedPane();  
        jtp.addTab("Cities", new CitiesPanel());  
        jtp.addTab("Colors", new ColorsPanel());  
        jtp.addTab("Flavors", new FlavorsPanel());  
        jfrm.add(jtp);
```

```
// Display the frame.  
jfrm.setVisible(true);  
}  
  
public static void main(String[] args) {  
    // Create the frame on the event dispatching thread.  
  
    SwingUtilities.invokeLater(  
        new Runnable() {  
            public void run() {  
                new JTabbedPaneDemo();  
            }  
        }  
    );  
}  
  
}  
  
// Make the panels that will be added to the tabbed pane.  
class CitiesPanel extends JPanel {  
  
    public CitiesPanel() {  
        JButton b1 = new JButton("New York");  
        add(b1);  
        JButton b2 = new JButton("London");  
        add(b2);  
        JButton b3 = new JButton("Hong Kong");  
        add(b3);  
        JButton b4 = new JButton("Tokyo");  
        add(b4);  
    }  
}  
  
class ColorsPanel extends JPanel {  
  
    public ColorsPanel() {  
        JCheckBox cb1 = new JCheckBox("Red");  
        add(cb1);  
        JCheckBox cb2 = new JCheckBox("Green");  
        add(cb2);  
        JCheckBox cb3 = new JCheckBox("Blue");  
        add(cb3);  
    }  
}  
  
class FlavorsPanel extends JPanel {  
  
    public FlavorsPanel() {  
        JComboBox<String> jcb = new JComboBox<String>();  
        jcb.addItem("Vanilla");  
        jcb.addItem("Chocolate");  
  
        jcb.addItem("Strawberry");  
        add(jcb);  
    }  
}
```

Output from the tabbed pane example is shown in the following three illustrations:



JScrollPane

JScrollPane is a lightweight container that automatically handles the scrolling of another component. The component being scrolled can be either an individual component, such as a table, or a group of components contained within another lightweight container, such as a **JPanel**. In either case, if the object being scrolled is larger than the viewable area, horizontal and/or vertical scroll bars are automatically provided, and the component can be scrolled through the pane. Because **JScrollPane** automates scrolling, it usually eliminates the need to manage individual scroll bars.

The viewable area of a scroll pane is called the *viewport*. It is a window in which the component being scrolled is displayed. Thus, the viewport displays the visible portion of the component being scrolled. The scroll bars scroll the component through the viewport. In its default behavior, a **JScrollPane** will dynamically add or remove a scroll bar as needed. For example, if the component is taller than the viewport, a vertical scroll bar is added. If the component will completely fit within the viewport, the scroll bars are removed.

JScrollPane defines several constructors. The one used in this chapter is shown here:

`JScrollPane(Component comp)`

The component to be scrolled is specified by *comp*. Scroll bars are automatically displayed when the content of the pane exceeds the dimensions of the viewport.

Here are the steps to follow to use a scroll pane:

1. Create the component to be scrolled.
2. Create an instance of **JScrollPane**, passing to it the object to scroll.
3. Add the scroll pane to the content pane.

The following example illustrates a scroll pane. First, a **JPanel** object is created, and 400 buttons are added to it, arranged into 20 columns. This panel is then added to a scroll pane, and the scroll pane is added to the content pane. Because the panel is larger than the viewport, vertical and horizontal scroll bars appear automatically. You can use the scroll bars to scroll the buttons into view.

```
// Demonstrate JScrollPane.
import java.awt.*;
import javax.swing.*;

public class JScrollPaneDemo {

    public JScrollPaneDemo() {

        // Set up the JFrame.  Use the default BorderLayot.
        JFrame jfrm = new JFrame("JScrollPaneDemo");
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jfrm.setSize(400, 400);

        // Create a panel and add 400 buttons to it.
        JPanel jp = new JPanel();
        jp.setLayout(new GridLayout(20, 20));

        int b = 0;
        for(int i = 0; i < 20; i++) {
            for(int j = 0; j < 20; j++) {
                jp.add(new JButton("Button " + b));
                ++b;
            }
        }

        // Create the scroll pane.
        JScrollPane jsp = new JScrollPane(jp);

        // Add the scroll pane to the content pane.
        // Because the default border layout is used,
        // the scroll pane will be added to the center.
        jfrm.add(jsp, BorderLayout.CENTER);

        // Display the frame.
        jfrm.setVisible(true);
    }
}
```

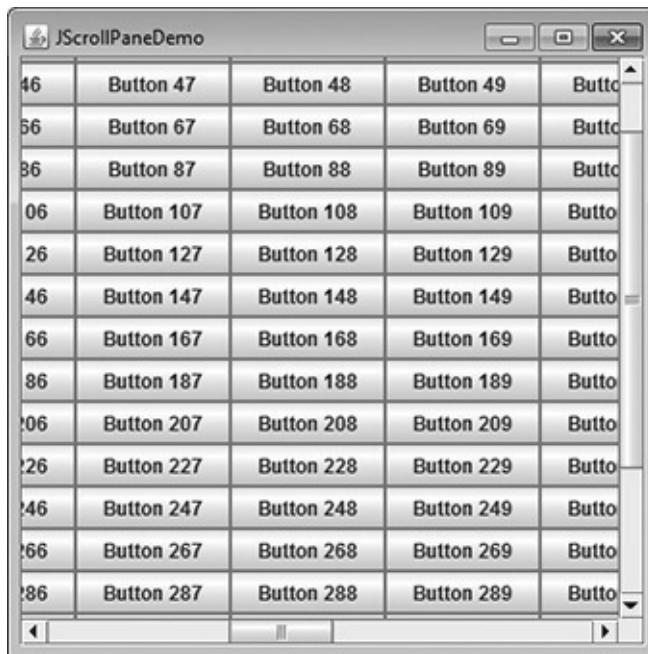
```

public static void main(String[] args) {
    // Create the frame on the event dispatching thread.

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

```

Output from the scroll pane example is shown here:



JList

In Swing, the basic list class is called **JList**. It supports the selection of one or more items from a list. Although the list often consists of strings, it is possible to create a list of just about any object that can be displayed. **JList** is so widely used in Java that it is highly unlikely that you have not seen one before.

In the past, the items in a **JList** were represented as **Object** references.

However, beginning with JDK 7, **JList** was made generic and is now declared like this:

```
class JList<E>
```

Here, **E** represents the type of the items in the list.

JList provides several constructors. The one used here is

```
JList(E[ ] items)
```

This creates a **JList** that contains the items in the array specified by *items*.

JList is based on two models. The first is **ListModel**. This interface defines how access to the list data is achieved. The second model is the **ListSelectionModel** interface, which defines methods that determine what list item or items are selected.

Although a **JList** will work properly by itself, most of the time you will wrap a **JList** inside a **JScrollPane**. This way, long lists will automatically be scrollable, which simplifies GUI design. It also makes it easy to change the number of entries in a list without having to change the size of the **JList** component.

A **JList** generates a **ListSelectionEvent** when the user makes or changes a selection. This event is also generated when the user deselects an item. It is handled by implementing **ListSelectionListener**. This listener specifies only one method, called **valueChanged()**, which is shown here:

```
void valueChanged(ListSelectionEvent le)
```

Here, *le* is a reference to the event. Although **ListSelectionEvent** does provide some methods of its own, normally you will interrogate the **JList** object itself to determine what has occurred. Both **ListSelectionEvent** and **ListSelectionListener** are packaged in **javax.swing.event**.

By default, a **JList** allows the user to select multiple ranges of items within the list, but you can change this behavior by calling **setSelectionMode()**, which is defined by **JList**. It is shown here:

```
void setSelectionMode(int mode)
```

Here, *mode* specifies the selection mode. It must be one of these values defined by **ListSelectionModel**:

SINGLE_SELECTION
SINGLE_INTERVAL_SELECTION
MULTIPLE_INTERVAL_SELECTION

The default, multiple-interval selection, lets the user select multiple ranges of items within a list. With single-interval selection, the user can select one range of items. With single selection, the user can select only a single item. Of course, a single item can be selected in the other two modes, too. It's just that they also allow a range to be selected.

You can obtain the index of the first item selected, which will also be the index of the only selected item when using single-selection mode, by calling **getSelectedIndex()**, shown here:

```
int getSelectedIndex()
```

Indexing begins at zero. So, if the first item is selected, this method will return 0. If no item is selected, -1 is returned.

Instead of obtaining the index of a selection, you can obtain the value associated with the selection by calling **getSelectedValue()**:

```
E getSelectedValue()
```

It returns a reference to the first selected value. If no value has been selected, it returns **null**.

The following program demonstrates a simple **JList**, which holds a list of cities. Each time a city is selected in the list, a **ListSelectionEvent** is generated, which is handled by the **valueChanged()** method defined by **ListSelectionListener**. It responds by obtaining the index of the selected item and displaying the name of the selected city in a label.

```
// Demonstrate JList.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

public class JListDemo {

    // Create an array of cities.
    String Cities[] = { "New York", "Chicago", "Houston",
                        "Denver", "Los Angeles", "Seattle",
                        "London", "Paris", "New Delhi",
                        "Hong Kong", "Tokyo", "Sydney" };

    public JListDemo() {

        // Set up the JFrame.
        JFrame jfrm = new JFrame("JListDemo");
        jfrm.setLayout(new FlowLayout());
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jfrm.setSize(200, 200);

        // Create a JList.
        JList<String> jlst = new JList<String>(Cities);

        // Set the list selection mode to single-selection.
        jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

        // Add the list to a scroll pane.
        JScrollPane jscrln = new JScrollPane(jlst);

        // Set the preferred size of the scroll pane.
        jscrln.setPreferredSize(new Dimension(120, 90));

        // Make a label that displays the selection.
        JLabel jlab = new JLabel("Choose a City");

        // Add selection listener for the list.
        jlst.addListSelectionListener(new ListSelectionListener() {
            public void valueChanged(ListSelectionEvent le) {
                // Get the index of the changed item.
                int idx = jlst.getSelectedIndex();

                // Display selection, if item was selected.
                if(idx != -1)
                    jlab.setText("Current selection: " + Cities[idx]);
            }
        });
    }
}
```

```

        else // Otherwise, reprompt.
        jlab.setText("Choose a City");
    }
});

// Add the list and label to the content pane.
jfrm.add(jscrlp);
jfrm.add(jlab);

// Display the frame.
jfrm.setVisible(true);
}

public static void main(String[] args) {
    // Create the frame on the event dispatching thread.

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

```

Output from the list example is shown here:



JComboBox

Swing provides a *combo box* (a combination of a text field and a drop-down list) through the **JComboBox** class. A combo box normally displays one entry, but it will also display a drop-down list that allows a user to select a different entry. You can also create a combo box that lets the user enter a selection into the text field.

In the past, the items in a **JComboBox** were represented as **Object** references. However, beginning with JDK 7, **JComboBox** was made generic and is now declared like this:

```
class JComboBox<E>
```

Here, **E** represents the type of the items in the combo box.

The **JComboBox** constructor used by the example is shown here:

```
JComboBox(E[ ] items)
```

Here, *items* is an array that initializes the combo box. Other constructors are available.

JComboBox uses the **ComboBoxModel**. Mutable combo boxes (those whose entries can be changed) use the **MutableComboBoxModel**.

In addition to passing an array of items to be displayed in the drop-down list, items can be dynamically added to the list of choices via the **addItem()** method, shown here:

```
void addItem(E obj)
```

Here, *obj* is the object to be added to the combo box. This method must be used only with mutable combo boxes.

JComboBox generates an action event when the user selects an item from the list. **JComboBox** also generates an item event when the state of selection changes, which occurs when an item is selected or deselected. Thus, changing a selection means that two item events will occur: one for the deselected item and another for the selected item. Often, it is sufficient to simply listen for action events, but both event types are available for your use.

One way to obtain the item selected in the list is to call **getSelectedItem()** on the combo box. It is shown here:

```
Object getSelectedItem()
```

```
Object getSelectedItem()
```

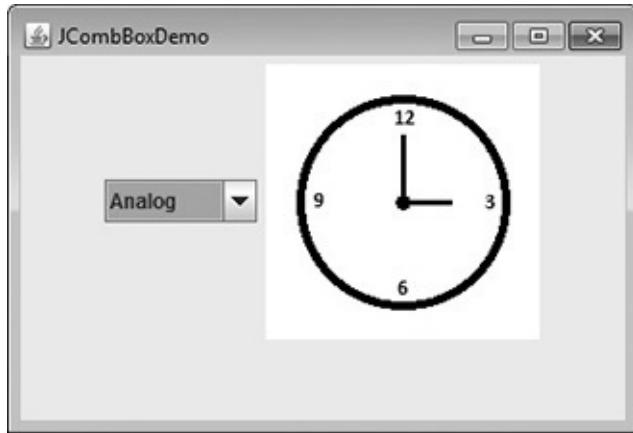
You will need to cast the returned value into the type of object stored in the list.

The following example demonstrates the combo box. The combo box contains entries for "Hourglass", "Analog", "Digital", and "Stopwatch". When a timepiece is selected, an icon-based label is updated to display it. You can see how little code is required to use this powerful component.

```
// Demonstrate JComboBox.  
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
  
public class JComboBoxDemo {  
  
    String timepieces[] = { "Hourglass", "Analog", "Digital", "Stopwatch" };  
  
    public JComboBoxDemo() {  
  
        // Set up the JFrame.  
        JFrame jfrm = new JFrame("JComboBoxDemo");  
        jfrm.setLayout(new FlowLayout());  
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        jfrm.setSize(400, 250);  
  
        // Instantiate a combo box and add it to the content pane.  
        JComboBox<String> jcb = new JComboBox<String>(timepieces);  
        jfrm.add(jcb);
```

```
// Create a label and add it to the content pane.  
JLabel jlab = new JLabel(new ImageIcon("hourglass.png"));  
jfrm.add(jlab);  
  
// Handle selections.  
jcb.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent ae) {  
        String s = (String) jcb.getSelectedItem();  
        jlab.setIcon(new ImageIcon(s + ".png"));  
    }  
});  
  
// Display the frame.  
jfrm.setVisible(true);  
}  
  
public static void main(String[] args) {  
    // Create the frame on the event dispatching thread.  
  
    SwingUtilities.invokeLater(  
        new Runnable() {  
            public void run() {  
                new JComboBoxDemo();  
            }  
        }  
    );  
}
```

Output from the combo box example is shown here:



Trees

A *tree* is a component that presents a hierarchical view of data. The user has the ability to expand or collapse individual subtrees in this display. Trees are implemented in Swing by the **JTree** class. A sampling of its constructors is shown here:

```
JTree(Object obj [ ])
JTree(Vector<?> v)
JTree(TreeNode tn)
```

In the first form, the tree is constructed from the elements in the array *obj*. The second form constructs the tree from the elements of vector *v*. In the third form, the tree whose root node is specified by *tn* specifies the tree.

Although **JTree** is packaged in **javax.swing**, its support classes and interfaces are packaged in **javax.swing.tree**. This is because the number of classes and interfaces needed to support **JTree** is quite large.

JTree relies on two models: **TreeModel** and **TreeSelectionModel**. A **JTree** generates a variety of events, but three relate specifically to trees:

TreeExpansionEvent, **TreeSelectionEvent**, and **TreeModelEvent**.

TreeExpansionEvent events occur when a node is expanded or collapsed. A **TreeSelectionEvent** is generated when the user selects or deselects a node within the tree. A **TreeModelEvent** is fired when the data or structure of the tree changes. The listeners for these events are **TreeExpansionListener**, **TreeSelectionListener**, and **TreeModelListener**, respectively. The tree event classes and listener interfaces are packaged in **javax.swing.event**.

The event handled by the sample program shown in this section is

TreeSelectionEvent. To listen for this event, implement **TreeSelectionListener**. It defines only one method, called **valueChanged()**, which receives the **TreeSelectionEvent** object. You can obtain the path to the selected object by calling **getPath()**, shown here, on the event object:

```
TreePath getPath()
```

It returns a **TreePath** object that describes the path to the changed node. The **TreePath** class encapsulates information about a path to a particular node in a tree. It provides several constructors and methods. In this book, only the **toString()** method is used. It returns a string that describes the path.

The **TreeNode** interface declares methods that obtain information about a tree node. For example, it is possible to obtain a reference to the parent node or an enumeration of the child nodes. The **MutableTreeNode** interface extends **TreeNode**. It declares methods that can insert and remove child nodes or change the parent node.

The **DefaultMutableTreeNode** class implements the **MutableTreeNode** interface. It represents a node in a tree. One of its constructors is shown here:

```
DefaultMutableTreeNode(Object obj)
```

Here, *obj* is the object to be enclosed in this tree node. The new tree node doesn't have a parent or children.

To create a hierarchy of tree nodes, the **add()** method of **DefaultMutableTreeNode** can be used. Its signature is shown here:

```
void add(MutableTreeNode child)
```

Here, *child* is a mutable tree node that is to be added as a child to the current node.

JTree does not provide any scrolling capabilities of its own. Instead, a **JTree** is typically placed within a **JScrollPane**. This way, a large tree can be scrolled through a smaller viewport.

Here are the steps to follow to use a tree:

1. Create an instance of **JTree**.
2. Create a **JScrollPane** and specify the tree as the object to be scrolled.
3. Add the tree to the scroll pane.
4. Add the scroll pane to the content pane.

The following example illustrates how to create a tree and handle selections. The program creates a **DefaultMutableTreeNode** instance labeled "Options". This is the top node of the tree hierarchy. Additional tree nodes are then created, and the **add()** method is called to connect these nodes to the tree. A reference to the top node in the tree is provided as the argument to the **JTree** constructor. The tree is then provided as the argument to the **JScrollPane** constructor. This scroll pane is then added to the content pane. Next, a label is created and added to the content pane. The tree selection is displayed in this label. To receive selection events from the tree, a **TreeSelectionListener** is registered for the tree. Inside the **valueChanged()** method, the path to the current selection is obtained and displayed.

```
// Demonstrate JTree.
import java.awt.*;
import javax.swing.event.*;
import javax.swing.*;
import javax.swing.tree.*;

public class JTreeDemo {

    public JTreeDemo() {

        // Set up the JFrame. Use default BorderLayout.
        JFrame jfrm = new JFrame("JTreeDemo");
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jfrm.setSize(200, 250);

        // Create top node of tree.
        DefaultMutableTreeNode top = new DefaultMutableTreeNode("Options");

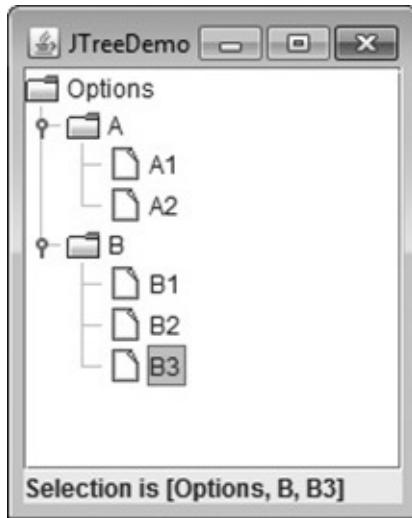
        // Create subtree of "A".
        DefaultMutableTreeNode a = new DefaultMutableTreeNode("A");
        top.add(a);
        DefaultMutableTreeNode a1 = new DefaultMutableTreeNode("A1");
        a.add(a1);
        DefaultMutableTreeNode a2 = new DefaultMutableTreeNode("A2");
        a.add(a2);

        // Create subtree of "B".
        DefaultMutableTreeNode b = new DefaultMutableTreeNode("B");
        top.add(b);
        DefaultMutableTreeNode b1 = new DefaultMutableTreeNode("B1");
        b.add(b1);
        DefaultMutableTreeNode b2 = new DefaultMutableTreeNode("B2");
        b.add(b2);
        DefaultMutableTreeNode b3 = new DefaultMutableTreeNode("B3");
        b.add(b3);

        // Create the tree.
        JTree tree = new JTree(top);
    }
}
```

```
// Add the tree to a scroll pane.  
JScrollPane jsp = new JScrollPane(tree);  
  
// Add the scroll pane to the content pane.  
jfrm.add(jsp);  
  
// Add the label to the content pane.  
JLabel jlab = new JLabel();  
jfrm.add(jlab, BorderLayout.SOUTH);  
  
// Handle tree selection events.  
tree.addTreeSelectionListener(new TreeSelectionListener() {  
    public void valueChanged(TreeSelectionEvent tse) {  
        jlab.setText("Selection is " + tse.getPath());  
    }  
});  
  
// Display the frame.  
jfrm.setVisible(true);  
}  
  
public static void main(String[] args) {  
    // Create the frame on the event dispatching thread.  
  
    SwingUtilities.invokeLater(  
        new Runnable() {  
            public void run() {  
                new JTreeDemo();  
            }  
        }  
    );  
}
```

Output from the tree example is shown here:



The string presented in the text field describes the path from the top tree node to the selected node.

JTable

JTable is a component that displays rows and columns of data. You can drag the cursor on column boundaries to resize columns. You can also drag a column to a new position. Depending on its configuration, it is also possible to select a row, column, or cell within the table, and to change the data within a cell.

JTable is a sophisticated component that offers many more options and features than can be discussed here. (It is perhaps Swing's most complicated component.) However, in its default configuration, **JTable** still offers substantial functionality that is easy to use—especially if you simply want to use the table to present data in a tabular format. The brief overview presented here will give you a general understanding of this powerful component.

Like **JTree**, **JTable** has many classes and interfaces associated with it. These are packaged in **javax.swing.table**.

At its core, **JTable** is conceptually simple. It is a component that consists of one or more columns of information. At the top of each column is a heading. In addition to describing the data in a column, the heading also provides the mechanism by which the user can change the size of a column or change the location of a column within the table. **JTable** does not provide any scrolling capabilities of its own. Instead, you will normally wrap a **JTable** inside a **JScrollPane**.

JTable supplies several constructors. The one used here is

```
JTable(Object data[ ][ ], Object colHeads[ ])
```

Here, *data* is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings.

JTable relies on three models. The first is the table model, which is defined by the **TableModel** interface. This model defines those things related to displaying data in a two-dimensional format. The second is the table column model, which is represented by **TableColumnModel**. **JTable** is defined in terms of columns, and it is **TableColumnModel** that specifies the characteristics of a column. These two models are packaged in **javax.swing.table**. The third model determines how items are selected, and it is specified by the **ListSelectionModel**, which was described when **JList** was discussed.

A **JTable** can generate several different events. The two most fundamental to a table's operation are **ListSelectionEvent** and **TableModelEvent**. A **ListSelectionEvent** is generated when the user selects something in the table. By default, **JTable** allows you to select one or more complete rows, but you can change this behavior to allow one or more columns, or one or more individual cells to be selected. A **TableModelEvent** is fired when that table's data changes in some way. Handling these events requires a bit more work than it does to handle the events generated by the previously described components and is beyond the scope of this book. However, if you simply want to use **JTable** to display data (as the following example does), then you don't need to handle any events.

Here are the steps required to set up a simple **JTable** that can be used to display data:

1. Create an instance of **JTable**.
2. Create a **JScrollPane** object, specifying the table as the object to scroll.
3. Add the table to the scroll pane.
4. Add the scroll pane to the content pane.

The following example illustrates how to create and use a simple table. A one-dimensional array of strings called **colHeads** is created for the column headings. A two-dimensional array of strings called **data** is created for the table cells. You can see that each element in the array is an array of three strings. These arrays are passed to the **JTable** constructor. The table is added to a scroll pane, and then the scroll pane is added to the content pane. The table displays the data in the **data** array. The default table configuration also allows

the contents of a cell to be edited. Changes affect the underlying array, which is **data** in this case.

```
// Demonstrate JTable.
import java.awt.*;
import javax.swing.*;

public class JTableDemo {

    // Initialize column headings.
    String[] colHeads = { "Name", "Extension", "ID#" };

    // Initialize data.
    Object[][] data = {
        { "Gail", "4567", "865" },
        { "Ken", "7566", "555" },
        { "Viviane", "5634", "587" },
        { "Melanie", "7345", "922" },
        { "Anne", "1237", "333" },
        { "John", "5656", "314" },
        { "Matt", "5672", "217" },
        { "Claire", "6741", "444" },
        { "Erwin", "9023", "519" },
        { "Ellen", "1134", "532" },
        { "Jennifer", "5689", "112" },
        { "Ed", "9030", "133" },
        { "Helen", "6751", "145" }
    };

    public JTableDemo() {

        // Set up the JFrame. Use default BorderLayout.
        JFrame jfrm = new JFrame("JTableDemo");
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jfrm.setSize(300, 300);

        // Create the table.
        JTable table = new JTable(data, colHeads);

        // Add the table to a scroll pane.
        JScrollPane jsp = new JScrollPane(table);
    }
}
```

```

// Add the scroll pane to the content pane.
jfrm.add(jsp);

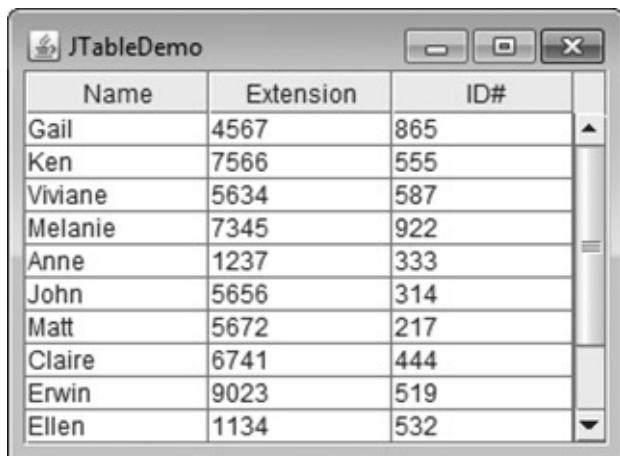
// Display the frame.
jfrm.setVisible(true);
}

public static void main(String[] args) {
    // Create the frame on the event dispatching thread.

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

```

Output from this example is shown here:



The screenshot shows a Java Swing application window titled "JTableDemo". The window contains a scrollable table with three columns: "Name", "Extension", and "ID#". The table has ten rows of data. A vertical scrollbar is visible on the right side of the table.

Name	Extension	ID#
Gail	4567	865
Ken	7566	555
Viviane	5634	587
Melanie	7345	922
Anne	1237	333
John	5656	314
Matt	5672	217
Claire	6741	444
Erwin	9023	519
Ellen	1134	532

CHAPTER

Introducing Swing Menus

This chapter introduces another fundamental aspect of the Swing GUI environment: the menu. Menus form an integral part of many applications because they present the program's functionality to the user. Because of their importance, Swing provides extensive support for menus. They are an area in which Swing's power is readily apparent.

The Swing menu system supports several key elements, including

- The menu bar, which is the main menu for an application.
- The standard menu, which can contain either items to be selected or other menus (submenus).
- The popup menu, which is usually activated by right-clicking the mouse.
- The toolbar, which provides rapid access to program functionality, often paralleling menu items.
- The action, which enables two or more different components to be managed by a single object. Actions are commonly used with menus and toolbars.

Swing menus also support accelerator keys, which enable menu items to be selected without having to activate the menu, and mnemonics, which allow a menu item to be selected by the keyboard once the menu options are displayed.

Menu Basics

The Swing menu system is supported by a group of related classes. The ones used in this chapter are shown in [Table 33-1](#), and they represent the core of the menu system. Although they may seem a bit confusing at first, Swing menus are quite easy to use. Swing allows a high degree of customization, if desired; however, you will normally use the menu classes as-is because they support all of the most needed options. For example, you can easily add images and keyboard shortcuts to a menu.

Class	Description
JMenuBar	An object that holds the top-level menu for the application.
JMenu	A standard menu. A menu consists of one or more JMenuItem s.
JMenuItem	An object that populates menus.
JCheckBoxMenuItem	A check box menu item.
JRadioButtonMenuItem	A radio button menu item
JSeparator	The visual separator between menu items.
JPopupMenu	A menu that is typically activated by right-clicking the mouse.

Table 33-1 The Core Swing Menu Classes

Here is a brief overview of how the classes fit together. To create the top-level menu for an application, you first create a **JMenuBar** object. This class is, loosely speaking, a container for menus. To the **JMenuBar** instance, you will add instances of **JMenu**. Each **JMenu** object defines a menu. That is, each **JMenu** object contains one or more selectable items. The items displayed by a **JMenu** are objects of **JMenuItem**. Thus, a **JMenuItem** defines a selection that can be chosen by the user.

As an alternative or adjunct to menus that descend from the menu bar, you can also create stand-alone, popup menus. To create a popup menu, first create an object of type **JPopupMenu**. Then, add **JMenuItem**s to it. A popup menu is normally activated by clicking the right mouse button when the mouse is over a component for which a popup menu has been defined.

In addition to “standard” menu items, you can also include check boxes and radio buttons in a menu. A check box menu item is created by **JCheckBoxMenuItem**. A radio button menu item is created by **JRadioButtonMenuItem**. Both of these classes extend **JMenuItem**. They can be used in standard menus and popup menus.

JToolBar creates a stand-alone component that is related to the menu. It is often used to provide fast access to functionality contained within the menus of the application. For example, a toolbar might provide fast access to the formatting commands supported by a word processor.

JSeparator is a convenience class that creates a separator line in a menu.

One key point to understand about Swing menus is that each menu item extends **AbstractButton**. Recall that **AbstractButton** is also the superclass of all of Swing’s button components, such as **JButton**. Thus, all menu items are, essentially, buttons. Obviously, they won’t actually look like buttons when used

in a menu, but they will, in many ways, act like buttons. For example, selecting a menu item generates an action event in the same way that pressing a button does.

Another key point is that **JMenuItem** is a superclass of **JMenu**. This allows the creation of submenus, which are, essentially, menus within menus. To create a submenu, you first create and populate a **JMenu** object and then add it to another **JMenu** object. You will see this process in action in the following section.

As mentioned in passing previously, when a menu item is selected, an action event is generated. The action command string associated with that action event will, by default, be the name of the selection. Thus, you can determine which item was selected by examining the action command. Of course, you can also use separate anonymous inner classes or lambda expressions to handle each menu item's action events. In this case, the menu selection is already known, and there is no need to examine the action command string to determine which item was selected.

Menus can also generate other types of events. For example, each time that a menu is activated, selected, or canceled, a **MenuEvent** is generated that can be listened for via a **MenuListener**. Other menu-related events include **MenuKeyEvent**, **MenuDragMouseEvent**, and **PopupMenuEvent**. In many cases, however, you need only watch for action events, and in this chapter, we will use only action events.

An Overview of **JMenuBar**, **JMenu**, and **JMenuItem**

Before you can create a menu, you need to know something about the three core menu classes: **JMenuBar**, **JMenu**, and **JMenuItem**. These form the minimum set of classes needed to construct a main menu for an application. **JMenu** and **JMenuItem** are also used by popup menus. Thus, these classes form the foundation of the menu system.

JMenuBar

As mentioned, **JMenuBar** is essentially a container for menus. Like all components, it inherits **JComponent** (which inherits **Container** and **Component**). It has only one constructor, which is the default constructor.

Therefore, initially the menu bar will be empty, and you will need to populate it with menus prior to use. Each application has one and only one menu bar.

JMenuBar defines several methods, but often you will only need to use one: **add()**. The **add()** method adds a **JMenu** to the menu bar. It is shown here:

```
JMenu add(JMenu menu)
```

Here, *menu* is a **JMenu** instance that is added to the menu bar. A reference to the menu is returned. Menus are positioned in the bar from left to right, in the order in which they are added. If you want to add a menu at a specific location, then use this version of **add()**, which is inherited from **Container**:

```
Component add(Component menu, int idx)
```

Here, *menu* is added at the index specified by *idx*. Indexing begins at 0, with 0 being the left-most menu.

In some cases, you might want to remove a menu that is no longer needed. You can do this by calling **remove()**, which is inherited from **Container**. It has these two forms:

```
void remove(Component menu)
```

```
void remove(int idx)
```

Here, *menu* is a reference to the menu to remove, and *idx* is the index of the menu to remove. Indexing begins at zero.

Another method that is sometimes useful is **getMenuCount()**, shown here:

```
int getMenuCount()
```

It returns the number of elements contained within the menu bar.

JMenuBar defines some other methods that you might find helpful in specialized applications. For example, you can obtain an array of references to the menus in the bar by calling **getSubElements()**. You can determine if a menu is selected by calling **isSelected()**.

Once a menu bar has been created and populated, it is added to a **JFrame** by calling **setJMenuBar()** on the **JFrame** instance. (Menu bars *are not* added to the content pane.) The **setJMenuBar()** method is shown here:

```
void setJMenuBar(JMenuBar mb)
```

Here, *mb* is a reference to the menu bar. The menu bar will be displayed in a position determined by the look and feel. Usually, this is at the top of the window.

JMenu

JMenu encapsulates a menu, which is populated with **JMenuItem**s. As mentioned, it is derived from **JMenuItem**. This means that one **JMenu** can be a selection in another **JMenu**. This enables one menu to be a submenu of another. **JMenu** defines a number of constructors. For example, here is the one used in the examples in this chapter:

`JMenu(String name)`

This constructor creates a menu that has the title specified by *name*. Of course, you don't have to give a menu a name. To create an unnamed menu, you can use the default constructor:

`JMenu()`

Other constructors are also supported. In each case, the menu is empty until menu items are added to it.

JMenu defines many methods. Here is a brief description of some commonly used ones. To add an item to the menu, use the **add()** method, which has a number of forms, including the two shown here:

`JMenuItem add(JMenuItem item)`

`Component add(Component item, int idx)`

Here, *item* is the menu item to add. The first form adds the item to the end of the menu. The second form adds the item at the index specified by *idx*. As expected, indexing starts at zero. Both forms return a reference to the item added. As a point of interest, you can also use **insert()** to add menu items to a menu.

You can add a separator (an object of type **JSeparator**) to a menu by calling **addSeparator()**, shown here:

`void addSeparator()`

The separator is added onto the end of the menu. You can insert a separator into a menu by calling **insertSeparator()**, shown next:

```
void insertSeparator(int idx)
```

Here, *idx* specifies the zero-based index at which the separator will be added.

You can remove an item from a menu by calling **remove()**. Two of its forms are shown here:

```
void remove(JMenuItem menu)
```

```
void remove(int idx)
```

In this case, *menu* is a reference to the item to remove and *idx* is the index of the item to remove.

You can obtain the number of items in the menu by calling **getMenuComponentCount()**, shown here:

```
int getMenuComponentCount()
```

You can get an array of the items in the menu by calling **getMenuComponents()**, shown next:

```
Component[ ] getMenuComponents()
```

An array containing the components is returned.

JMenuItem

JMenuItem encapsulates an element in a menu. This element can be a selection linked to some program action, such as Save or Close, or it can cause a submenu to be displayed. As mentioned, **JMenuItem** is derived from **AbstractButton**, and every item in a menu can be thought of as a special kind of button. Therefore, when a menu item is selected, an action event is generated. (This is similar to the way a **JButton** fires an action event when it is pressed.) **JMenuItem** defines many constructors. The ones used in this chapter are shown here:

```
JMenuItem(String name)
```

`JMenuItem(Icon image)`

`JMenuItem(String name, Icon image)`

`JMenuItem(String name, int mnem)`

`JMenuItem(Action action)`

The first constructor creates a menu item with the name specified by *name*. The second creates a menu item that displays the image specified by *image*. The third creates a menu item with the name specified by *name* and the image specified by *image*. The fourth creates a menu item with the name specified by *name* and uses the keyboard mnemonic specified by *mnem*. This mnemonic enables you to select an item from the menu by pressing the specified key. The last constructor creates a menu item using the information specified in *action*. A default constructor is also supported.

Because menu items inherit **AbstractButton**, you have access to the functionality provided by **AbstractButton**. One such method that is often useful with menus is **setEnabled()**, which you can use to enable or disable a menu item. It is shown here:

```
void setEnabled(boolean enable)
```

If *enable* is **true**, the menu item is enabled. If *enable* is **false**, the item is disabled and cannot be selected.

Create a Main Menu

Traditionally, the most commonly used menu is the *main menu*. This is the menu defined by the menu bar, and it is the menu that defines all (or nearly all) of the functionality of an application. Fortunately, Swing makes creating and managing the main menu easy. This section shows you how to construct a basic main menu. Subsequent sections will show you how to add options to it.

Constructing the main menu requires several steps. First, create the **JMenuBar** object that will hold the menus. Next, construct each menu that will be in the menu bar. In general, a menu is constructed by first creating a **JMenu** object and then adding **JMenuItem**s to it. After the menus have been created, add them to the menu bar. The menu bar, itself, must then be added to the frame by calling **setJMenuBar()**. Finally, for each menu item, you must add

an action listener that handles the action event fired when the menu item is selected.

A good way to understand the process of creating and managing menus is to work through an example. Here is a program that creates a simple menu bar that contains three menus. The first is a standard File menu that contains Open, Close, Save, and Exit selections. The second menu is called Options, and it contains two submenus called Colors and Priority. The third menu is called Help, and it has one item: About. When a menu item is selected, the name of the selection is displayed in a label in the content pane. Sample output is shown in [Figure 33-1](#).

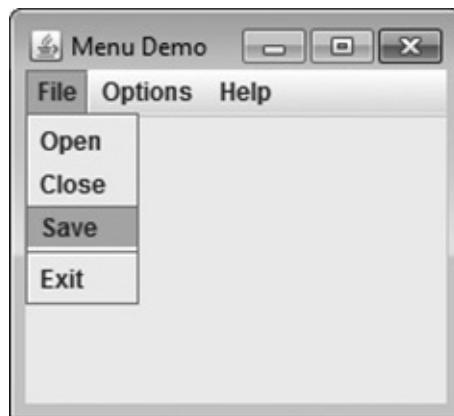


Figure 33-1 Sample output from the **MenuDemo** program

```
// Demonstrate a simple main menu.

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

class MenuDemo implements ActionListener {

    JLabel jlab;

    MenuDemo() {
        // Create a new JFrame container.
        JFrame jfrm = new JFrame("Menu Demo");

```

```
// Specify FlowLayout for the layout manager.  
jfrm.setLayout(new FlowLayout());  
  
// Give the frame an initial size.  
jfrm.setSize(220, 200);  
  
// Terminate the program when the user closes the application.  
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
// Create a label that will display the menu selection.  
jlab = new JLabel();  
  
// Create the menu bar.  
JMenuBar jmb = new JMenuBar();  
  
// Create the File menu.  
JMenu jmFile = new JMenu("File");  
JMenuItem jmiOpen = new JMenuItem("Open");  
JMenuItem jmiClose = new JMenuItem("Close");  
JMenuItem jmiSave = new JMenuItem("Save");  
JMenuItem jmiExit = new JMenuItem("Exit");  
jmFile.add(jmiOpen);  
jmFile.add(jmiClose);  
jmFile.add(jmiSave);  
jmFile.addSeparator();  
jmFile.add(jmiExit);  
jmb.add(jmFile);  
  
// Create the Options menu.  
JMenu jmOptions = new JMenu("Options");  
  
// Create the Colors submenu.  
JMenu jmColors = new JMenu("Colors");  
JMenuItem jmiRed = new JMenuItem("Red");  
JMenuItem jmiGreen = new JMenuItem("Green");  
JMenuItem jmiBlue = new JMenuItem("Blue");  
jmColors.add(jmiRed);  
jmColors.add(jmiGreen);  
jmColors.add(jmiBlue);  
jmOptions.add(jmColors);  
  
// Create the Priority submenu.  
JMenu jmPriority = new JMenu("Priority");  
JMenuItem jmiHigh = new JMenuItem("High");  
JMenuItem jmiLow = new JMenuItem("Low");  
jmPriority.add(jmiHigh);  
jmPriority.add(jmiLow);  
jmOptions.add(jmPriority);  
  
// Create the Reset menu item.  
JMenuItem jmiReset = new JMenuItem("Reset");  
jmOptions.addSeparator();  
jmOptions.add(jmiReset);
```



```
// Finally, add the entire options menu to
// the menu bar
jmb.add(jmOptions);

// Create the Help menu.
JMenu jmHelp = new JMenu("Help");
JMenuItem jmiAbout = new JMenuItem("About");
jmHelp.add(jmiAbout);
jmb.add(jmHelp);

// Add action listeners for the menu items.
jmiOpen.addActionListener(this);
jmiClose.addActionListener(this);
jmiSave.addActionListener(this);
jmiExit.addActionListener(this);
jmiRed.addActionListener(this);
jmiGreen.addActionListener(this);
jmiBlue.addActionListener(this);
jmiHigh.addActionListener(this);
jmiLow.addActionListener(this);
jmiReset.addActionListener(this);
jmiAbout.addActionListener(this);

// Add the label to the content pane.
jfrm.add(jlab);

// Add the menu bar to the frame.
jfrm.setJMenuBar(jmb);

// Display the frame.
jfrm.setVisible(true);
}

// Handle menu item action events.
public void actionPerformed(ActionEvent ae) {
    // Get the action command from the menu selection.
    String comStr = ae.getActionCommand();

    // If user chooses Exit, then exit the program.
    if(comStr.equals("Exit")) System.exit(0);

    // Otherwise, display the selection.
    jlab.setText(comStr + " Selected");
}

public static void main(String args[]) {
    // Create the frame on the event dispatching thread.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new MenuDemo();
        }
    });
}
}
```

Let's examine, in detail, how the menus in this program are created, beginning with the **MenuDemo** constructor. It starts by creating a **JFrame** and setting its layout manager, size, and default close operation. (These operations are described in [Chapter 31](#).) A **JLabel** is then constructed. It will be used to display a menu selection. Next, the menu bar is constructed and a reference to it is assigned to **jmb** by this statement:

```
// Create the menu bar.  
JMenuBar jmb = new JMenuBar();
```

Then, the File menu **jmFile** and its menu entries are created by this sequence:

```
// Create the File menu.  
JMenu jmFile = new JMenu("File");  
JMenuItem jmiOpen = new JMenuItem("Open");  
JMenuItem jmiClose = new JMenuItem("Close");  
JMenuItem jmiSave = new JMenuItem("Save");  
JMenuItem jmiExit = new JMenuItem("Exit");
```

The names Open, Close, Save, and Exit will be shown as selections in the menu. Next, the menu entries are added to the file menu by this sequence:

```
jmFile.add(jmiOpen);  
jmFile.add(jmiClose);  
jmFile.add(jmiSave);  
jmFile.addSeparator();  
jmFile.add(jmiExit);
```

Finally, the File menu is added to the menu bar with this line:

```
jmb.add(jmFile);
```

Once the preceding code sequence completes, the menu bar will contain one entry: File. The File menu will contain four selections in this order: Open, Close, Save, and Exit. However, notice that a separator has been added before Exit. This visually separates Exit from the preceding three selections.

The Options menu is constructed using the same basic process as the File menu. However, the Options menu consists of two submenus, Colors and Priority, and a Reset entry. The submenus are first constructed individually and then added to the Options menu. The Reset item is added last. Then, the Options menu is added to the menu bar. The Help menu is constructed using the ~~same process~~

same process.

Notice that **MenuDemo** implements the **ActionListener** interface and action events generated by a menu selection are handled by the **actionPerformed()** method defined by **MenuDemo**. Therefore, the program adds **this** as the action listener for the menu items. Notice that no listeners are added to the Colors or Priority items because they are not actually selections. They simply activate submenus.

Finally, the menu bar is added to the frame by the following line:

```
jfrm.setJMenuBar(jmb);
```

As mentioned, menu bars are not added to the content pane. They are added directly to the **JFrame**.

The **actionPerformed()** method handles the action events generated by the menu. It obtains the action command string associated with the selection by calling **getActionCommand()** on the event. It stores a reference to this string in **comStr**. Then, it tests the action command against "Exit", as shown here:

```
if(comStr.equals("Exit")) System.exit(0);
```

If the action command is "Exit", then the program terminates by calling **System.exit()**. This method causes the immediate termination of a program and passes its argument as a status code to the calling process, which is usually the operating system. By convention, a status code of zero means normal termination. Anything else indicates that the program terminated abnormally. For all other menu selections, the choice is displayed.

At this point, you might want to experiment a bit with the **MenuDemo** program. Try adding another menu or adding additional items to an existing menu. It is important that you understand the basic menu concepts before moving on because this program will evolve throughout the course of this chapter.

Add Mnemonics and Accelerators to Menu Items

The menu created in the preceding example is functional, but it is possible to make it better. In real applications, a menu usually includes support for keyboard shortcuts because they give an experienced user the ability to select menu items rapidly. Keyboard shortcuts come in two forms: mnemonics and

accelerators. As it applies to menus, a *mnemonic* defines a key that lets you select an item from an active menu by typing the key. Thus, a mnemonic allows you to use the keyboard to select an item from a menu that is already being displayed. An *accelerator* is a key that lets you select a menu item without having to first activate the menu.

A mnemonic can be specified for both **JMenuItem** and **JMenu** objects. There are two ways to set the mnemonic for **JMenuItem**. First, it can be specified when an object is constructed using this constructor:

```
JMenuItem(String name, int mnem)
```

In this case, the name is passed in *name* and the mnemonic is passed in *mnem*. Second, you can set the mnemonic by calling **setMnemonic()**. To specify a mnemonic for **JMenu**, you must call **setMnemonic()**. This method is inherited by both classes from **AbstractButton** and is shown next:

```
void setMnemonic(int mnem)
```

Here, *mnem* specifies the mnemonic. It should be one of the constants defined in **java.awt.event.KeyEvent**, such as **KeyEvent.VK_F** or **KeyEvent.VK_Z**. (There is another version of **setMnemonic()** that takes a **char** argument, but it is considered obsolete.) Mnemonics are not case sensitive, so in the case of **VK_A**, typing either *a* or *A* will work.

By default, the first matching letter in the menu item will be underscored. In cases in which you want to underscore a letter other than the first match, specify the index of the letter as an argument to **setDisplayedMnemonicIndex()**, which is inherited by both **JMenu** and **JMenuItem** from **AbstractButton**. It is shown here:

```
void setDisplayedMnemonicIndex(int idx)
```

The index of the letter to underscore is specified by *idx*.

An accelerator can be associated with a **JMenuItem** object. It is specified by calling **setAccelerator()**, shown next:

```
void setAccelerator(KeyStroke ks)
```

Here, *ks* is the key combination that is pressed to select the menu item. **KeyStroke** is a class that contains several factory methods that construct

various types of keystroke accelerators. The following are three examples:

```
static KeyStroke getKeyStroke(char ch)
```

```
static KeyStroke getKeyStroke(Character ch, int modifier)
```

```
static KeyStroke getKeyStroke(int ch, int modifier)
```

Here, *ch* specifies the accelerator character. In the first version, the character is specified as a **char** value. In the second, it is specified as an object of type **Character**. In the third, it is a value of type **KeyEvent**, previously described. The value of *modifier* must be one or more of the following constants, defined in the **java.awt.event.InputEvent** class:

InputEvent.ALT_DOWN_MASK	InputEvent.ALT_GRAPH_DOWN_MASK
InputEvent.CTRL_DOWN_MASK	InputEvent.META_DOWN_MASK
InputEvent.SHIFT_DOWN_MASK	

Therefore, if you pass **VK_A** for the key character and **InputEvent.CTRL_DOWN_MASK** for the modifier, the accelerator key combination is CTRL-A.

The following sequence adds both mnemonics and accelerators to the File menu created by the **MenuDemo** program in the previous section. After making this change, you can select the File menu by typing ALT-F. Then, you can use the mnemonics O, C, S, or E to select an option. Alternatively, you can directly select a File menu option by pressing CTRL-O, CTRL-C, CTRL-S, or CTRL-E. [Figure 33-2](#) shows how this menu looks when activated.

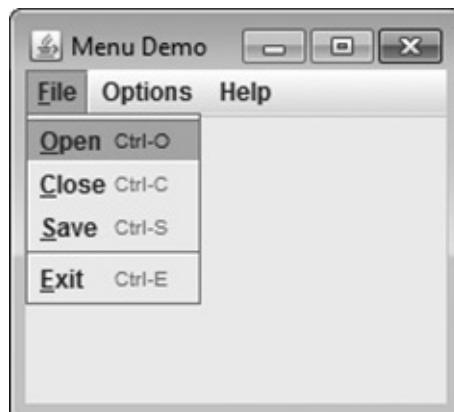


Figure 33-2 The File menu after adding mnemonics and accelerators

```

// Create the File menu with mnemonics and accelerators.
JMenu jmFile = new JMenu("File");
jmFile.setMnemonic(KeyEvent.VK_F);

JMenuItem jmiOpen = new JMenuItem("Open",
                                  KeyEvent.VK_O);
jmiOpen.setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_O,
                          InputEvent.CTRL_DOWN_MASK));

JMenuItem jmiClose = new JMenuItem("Close",
                                   KeyEvent.VK_C);
jmiClose.setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_C,
                          InputEvent.CTRL_DOWN_MASK));

JMenuItem jmiSave = new JMenuItem("Save",
                                 KeyEvent.VK_S);
jmiSave.setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_S,
                          InputEvent.CTRL_DOWN_MASK));

JMenuItem jmiExit = new JMenuItem("Exit",
                                 KeyEvent.VK_E);
jmiExit.setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_E,
                          InputEvent.CTRL_DOWN_MASK));

```

Add Images and Tooltips to Menu Items

You can add images to menu items or use images instead of text. The easiest way to add an image is to specify it when the menu item is being constructed using one of these constructors:

`JMenuItem(Icon image)`

`JMenuItem(String name, Icon image)`

The first creates a menu item that displays the image specified by *image*. The second creates a menu item with the name specified by *name* and the image specified by *image*. For example, here the About menu item is associated with an image when it is created:

```
 ImageIcon icon = new ImageIcon("AboutIcon.gif");
 JMenuItem jmiAbout = new JMenuItem("About", icon);
```

After this addition, the icon specified by **icon** will be displayed next to the text "About" when the Help menu is displayed. This is shown in [Figure 33-3](#). You can also add an icon to a menu item after the item has been created by calling **setIcon()**, which is inherited from **AbstractButton**. You can specify the horizontal alignment of the image relative to the text by calling **setHorizontalTextPosition()**.

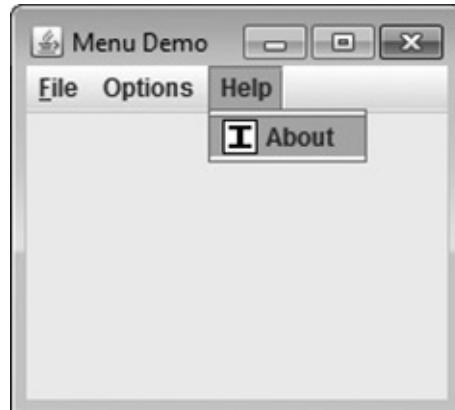


Figure 33-3 The About item with the addition of an icon

You can specify a disabled icon, which is shown when the menu item is disabled, by calling **setDisabledIcon()**. Normally, when a menu item is disabled, the default icon is shown in gray. If a disabled icon is specified, then that icon is displayed when the menu item is disabled.

A *tooltip* is a small message that describes an item. It is automatically displayed if the mouse remains over the item for a moment. You can add a tooltip to a menu item by calling **setToolTipText()** on the item, specifying the text you want displayed. It is shown here:

```
void setToolTipText(String msg)
```

In this case, *msg* is the string that will be displayed when the tooltip is activated. For example, this creates a tooltip for the About item:

```
jmiAbout.setToolTipText("Info about the MenuDemo program.");
```

As a point of interest, **setToolTipText()** is inherited by **JMenuItem** from **JComponent**. This means you can add a tooltip to other types of components, such as a push button. You might want to try this on your own.

Use **JRadioButtonMenuItem** and **JCheckBoxMenuItem**

Although the type of menu items used by the preceding examples are, as a general rule, the most commonly used, Swing defines two others: check boxes and radio buttons. These items can streamline a GUI by allowing a menu to provide functionality that would otherwise require additional, stand-alone components. Also, sometimes, including check boxes or radio buttons in a menu simply seems the most natural place for a specific set of features. Whatever your reason, Swing makes it easy to use check boxes and radio buttons in menus, and both are examined here.

To add a check box to a menu, create a **JCheckBoxMenuItem**. It defines several constructors. This is the one used in this chapter:

JCheckBoxMenuItem(String name)

Here, *name* specifies the name of the item. The initial state of the check box is unchecked. If you want to specify the initial state, you can use this constructor:

JCheckBoxMenuItem(String name, boolean state)

In this case, if *state* is **true**, the box is initially checked. Otherwise, it is cleared. **JCheckBoxMenuItem** also provides constructors that let you specify an icon. Here is one example:

JCheckBoxMenuItem(String name, Icon icon)

In this case, *name* specifies the name of the item and the image associated with the item is passed in *icon*. The item is initially unchecked. Other constructors are also supported.

Check boxes in menus work like stand-alone check boxes. For example, they generate action events and item events when their state changes. Check boxes are especially useful in menus when you have options that can be selected and

you want to display their selected/deselected status.

A radio button can be added to a menu by creating an object of type **JRadioButtonMenuItem**. **JRadioButtonMenuItem** inherits **JMenuItem**. It provides a rich assortment of constructors. The ones used in this chapter are shown here:

`JRadioButtonMenuItem(String name)`

`JRadioButtonMenuItem(String name, boolean state)`

The first constructor creates an unselected radio button menu item that is associated with the name passed in *name*. The second lets you specify the initial state of the button. If *state* is **true**, the button is initially selected. Otherwise, it is deselected. Other constructors let you specify an icon. Here is one example:

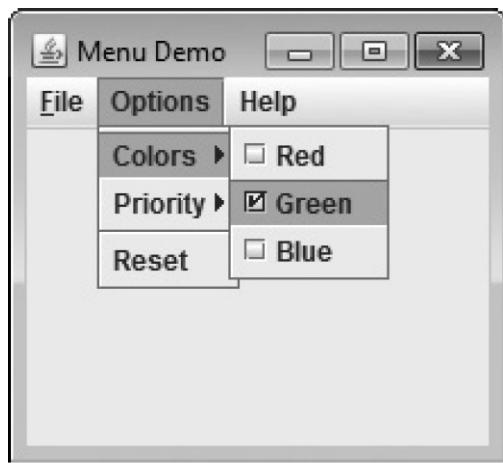
`JRadioButtonMenuItem(String name, Icon icon, boolean state)`

This creates a radio button menu item that is associated with the name passed in *name* and the image passed in *icon*. If *state* is **true**, the button is initially selected. Otherwise, it is deselected. Several other constructors are supported.

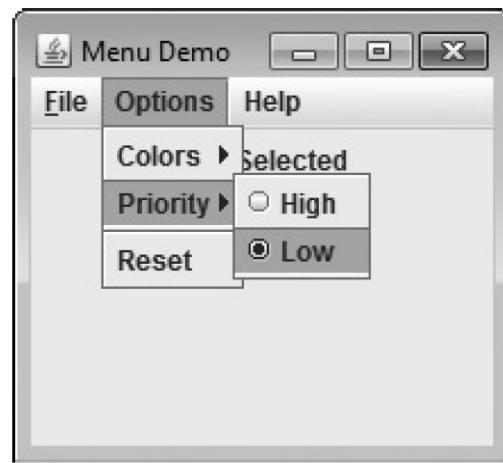
A **JRadioButtonMenuItem** works like a stand-alone radio button, generating item and action events. Like stand-alone radio buttons, menu-based radio buttons must be put into a button group in order for them to exhibit mutually exclusive selection behavior.

Because both **JCheckBoxMenuItem** and **JRadioButtonMenuItem** inherit **JMenuItem**, each has all of the functionality provided by **JMenuItem**. Aside from having the extra capabilities of check boxes and radio buttons, they act like and are used like other menu items.

To try check box and radio button menu items, first remove the code that creates the Options menu in the **MenuDemo** example program. Then substitute the following code sequence, which uses check boxes for the Colors submenu and radio buttons for the Priority submenu. After making the substitution, the Options menu will look like those shown in [Figure 33-4](#).



(a)



(b)

Figure 33-4 The effects of check box (a) and radio button (b) menu items

```
// Create the Options menu.  
JMenu jmOptions = new JMenu("Options");  
  
// Create the Colors submenu.  
JMenu jmColors = new JMenu("Colors");
```

```
// Use check boxes for colors. This allows
// the user to select more than one color.
JCheckBoxMenuItem jmiRed = new JCheckBoxMenuItem("Red");
JCheckBoxMenuItem jmiGreen = new JCheckBoxMenuItem("Green");
JCheckBoxMenuItem jmiBlue = new JCheckBoxMenuItem("Blue");

jmColors.add(jmiRed);
jmColors.add(jmiGreen);
jmColors.add(jmiBlue);
jmOptions.add(jmColors);

// Create the Priority submenu.
JMenu jmPriority = new JMenu("Priority");

// Use radio buttons for the priority setting.
// This lets the menu show which priority is used
// but also ensures that one and only one priority
// can be selected at any one time. Notice that
// the High radio button is initially selected.
JRadioButtonMenuItem jmiHigh =
    new JRadioButtonMenuItem("High", true);
JRadioButtonMenuItem jmiLow =
    new JRadioButtonMenuItem("Low");

jmPriority.add(jmiHigh);
jmPriority.add(jmiLow);
jmOptions.add(jmPriority);

// Create button group for the radio button menu items.
ButtonGroup bg = new ButtonGroup();
bg.add(jmiHigh);
bg.add(jmiLow);

// Create the Reset menu item.
JMenuItem jmiReset = new JMenuItem("Reset");
jmOptions.addSeparator();
jmOptions.add(jmiReset);

// Finally, add the entire options menu to
// the menu bar
jmb.add(jmOptions);
```

Create a Popup Menu

A popular alternative or addition to the menu bar is the popup menu. Typically, a popup menu is activated by clicking the right mouse button when over a component. Popup menus are supported in Swing by the **JPopupMenu** class. **JPopupMenu** has two constructors. In this chapter, only the default constructor is used:

```
JPopupMenu( )
```

It creates a default popup menu. The other constructor lets you specify a title for the menu. Whether this title is displayed is subject to the look and feel.

In general, popup menus are constructed like regular menus. First, create a **JPopupMenu** object, and then add menu items to it. Menu item selections are also handled in the same way: by listening for action events. The main difference between a popup menu and regular menu is the activation process.

Activating a popup menu requires three steps:

1. You must register a listener for mouse events.
2. Inside the mouse event handler, you must watch for the popup trigger.
3. When a popup trigger is received, you must show the popup menu by calling **show()**.

Let's examine each of these steps closely.

A popup menu is normally activated by clicking the right mouse button when the mouse pointer is over a component for which a popup menu is defined. Thus, the *popup trigger* is usually caused by right-clicking the mouse on a popup menu-enabled component. To listen for the popup trigger, implement the **MouseListener** interface and then register the listener by calling the **addMouseListener()** method. As described in [Chapter 24](#), **MouseListener** defines the methods shown here:

```
void mouseClicked(MouseEvent me)
```

```
void mouseEntered(MouseEvent me)
```

```
void mouseExited(MouseEvent me)
```

```
void mousePressed(MouseEvent me)  
void mouseReleased(MouseEvent me)
```

Of these, two are very important relative to the popup menu: **mousePressed()** and **mouseReleased()**. Depending on the installed look and feel, either of these two events can trigger a popup menu. For this reason, it is often easier to use a **MouseAdapter** to implement the **MouseListener** interface and simply override **mousePressed()** and **mouseReleased()**.

The **MouseEvent** class defines several methods, but only four are commonly needed when activating a popup menu. They are shown here:

```
int getX()  
int getY()  
boolean isPopupTrigger()  
Component getComponent()
```

The current X,Y location of the mouse relative to the source of the event is found by calling **getX()** and **getY()**. These are used to specify the upper-left corner of the popup menu when it is displayed. The **isPopupTrigger()** method returns **true** if the mouse event represents a popup trigger and **false** otherwise. You will use this method to determine when to pop up the menu. To obtain a reference to the component that generated the mouse event, call **getComponent()**.

To actually display the popup menu, call the **show()** method defined by **JPopupMenu**, shown next:

```
void show(Component invoker, int upperX, int upperY)
```

Here, *invoker* is the component relative to which the menu will be displayed. The values of *upperX* and *upperY* define the X,Y location of the upper-left corner of the menu, relative to *invoker*. A common way to obtain the invoker is to call **getComponent()** on the event object passed to the mouse event handler.

The preceding theory can be put into practice by adding a popup Edit menu to the **MenuDemo** program shown at the start of this chapter. This menu will have three items called Cut, Copy, and Paste. Begin by adding the following instance variable to **MenuDemo**:

```
JPopupMenu jpu;
```

The **jpu** variable will hold a reference to the popup menu.

Next, add the following code sequence to the **MenuDemo** constructor:

```
// Create an Edit popup menu.  
jpu = new JPopupMenu();  
  
// Create the popup menu items.  
JMenuItem jmiCut = new JMenuItem("Cut");  
JMenuItem jmiCopy = new JMenuItem("Copy");  
JMenuItem jmiPaste = new JMenuItem("Paste");  
  
// Add the menu items to the popup menu.  
jpu.add(jmiCut);  
jpu.add(jmiCopy);  
jpu.add(jmiPaste);  
  
// Add a listener for the popup trigger.  
jfrm.addMouseListener(new MouseAdapter() {  
    public void mousePressed(MouseEvent me) {  
        if(me.isPopupTrigger())  
            jpu.show(me.getComponent(), me.getX(), me.getY());  
    }  
    public void mouseReleased(MouseEvent me) {  
        if(me.isPopupTrigger())  
            jpu.show(me.getComponent(), me.getX(), me.getY());  
    }  
});
```

This sequence begins by constructing an instance of **JPopupMenu** and storing it in **jpu**. Then, it creates the three menu items, Cut, Copy, and Paste, in the usual way, and adds them to **jpu**. This finishes the construction of the popup Edit menu. Popup menus are not added to the menu bar or any other object.

Next, a **MouseListener** is added by creating an anonymous inner class. This class is based on the **MouseAdapter** class, which means that the listener need only override those methods that are relevant to the popup menu: **mousePressed()** and **mouseReleased()**. The adapter provides default

implementations of the other **MouseListener** methods. Notice that the mouse listener is added to **jfrm**. This means that a right-button click inside any part of the content pane will trigger the popup menu.

The **mousePressed()** and **mouseReleased()** methods call **isPopupTrigger()** to determine if the mouse event is a popup trigger event. If it is, the popup menu is displayed by calling **show()**. The invoker is obtained by calling **getComponent()** on the mouse event. In this case, the invoker will be the content pane. The X,Y coordinates of the upper-left corner are obtained by calling **getX()** and **getY()**. This makes the menu pop up with its upper-left corner directly under the mouse pointer.

Finally, you also need to add these action listeners to the program. They handle the action events fired when the user selects an item from the popup menu.

```
jmiCut.addActionListener(this);
jmiCopy.addActionListener(this);
jmiPaste.addActionListener(this);
```

After you have made these additions, the popup menu can be activated by clicking the right mouse button anywhere inside the content pane of the application. [Figure 33-5](#) shows the result.

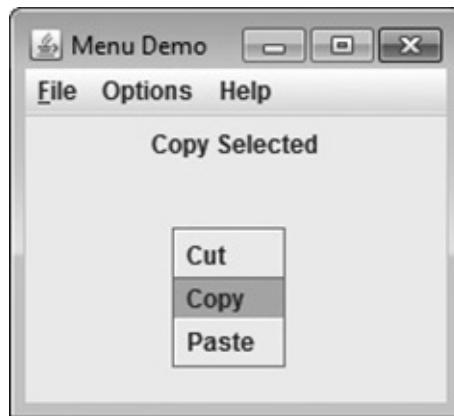


Figure 33-5 A popup Edit menu

One other point about the preceding example. Because the invoker of the popup menu is always **jfrm**, in this case, you could pass it explicitly rather than calling **getComponent()**. To do so, you must make **jfrm** into an instance variable of the **MenuDemo** class (rather than a local variable) so that it is accessible to the inner class. Then you can use this call to **show()** to display the popup menu:

```
jpu.show(jfrm, me.getX(), me.getY());
```

Although this works in this example, the advantage of using **getComponent()** is that the popup menu will automatically pop up relative to the invoking component. Thus, the same code could be used to display any popup menu relative to its invoking object.

Create a Toolbar

A toolbar is a component that can serve as both an alternative and as an adjunct to a menu. A toolbar contains a list of buttons (or other components) that give the user immediate access to various program options. For example, a toolbar might contain buttons that select various font options, such as bold, italics, highlight, or underline. These options can be selected without needing to drop through a menu. Typically, toolbar buttons show icons rather than text, although either or both are allowed. Furthermore, tooltips are often associated with icon-based toolbar buttons. Toolbars can be positioned on any side of a window by dragging the toolbar, or they can be dragged out of the window entirely, in which case they become free floating.

In Swing, toolbars are instances of the **JToolBar** class. Its constructors enable you to create a toolbar with or without a title. You can also specify the layout of the toolbar, which will be either horizontal or vertical. The **JToolBar** constructors are shown here:

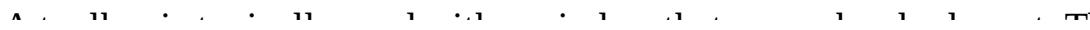
JToolBar()

JToolBar(String *title*)

JToolBar(int *how*)

JToolBar(String *title*, int *how*)

The first constructor creates a horizontal toolbar with no title. The second creates a horizontal toolbar with the title specified by *title*. The title will show only when the toolbar is dragged out of its window. The third creates a toolbar that is oriented as specified by *how*. The value of *how* must be either **JToolBar.VERTICAL** or **JToolBar.HORIZONTAL**. The fourth constructor creates a toolbar that has the title specified by *title* and is oriented as specified by *how*.



A toolbar is typically used with a window that uses a border layout. There are two reasons for this. First, it allows the toolbar to be initially positioned along one of the four border positions. Frequently, the top position is used. Second, it allows the toolbar to be dragged to any side of the window.

In addition to dragging the toolbar to different locations within a window, you can also drag it out of the window. Doing so creates an *undocked* toolbar. If you specify a title when you create the toolbar, then that title will be shown when the toolbar is undocked.

You add buttons (or other components) to a toolbar in much the same way that you add them to a menu bar. Simply call **add()**. The components are shown in the toolbar in the order in which they are added.

Once you have created a toolbar, you *do not* add it to the menu bar (if one exists). Instead, you add it to the window container. As mentioned, typically you will add a toolbar to the top (that is, north) position of a border layout, using a horizontal orientation. The component that will be affected is added to the center of the border layout. Using this approach causes the program to begin running with the toolbar in the expected location. However, you can drag the toolbar to any of the other positions. Of course, you can also drag the toolbar out of the window.

To illustrate the toolbar, we will add one to the **MenuDemo** program. The toolbar will present three debugging options: set a breakpoint, clear a breakpoint, and resume program execution. Three steps are needed to add the toolbar.

First, remove this line from the program:

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

By removing this line, the **JFrame** automatically uses a border layout.

Second, because **BorderLayout** is being used, change the line that adds the label **jlab** to the frame, as shown next:

```
jfrm.add(jlab, BorderLayout.CENTER);
```

This line explicitly adds **jlab** to the center of the border layout. (Explicitly specifying the center position is technically not necessary because, by default, components are added to the center when a border layout is used. However, explicitly specifying the center makes it clear to anyone reading the code that a border layout is being used and that **jlab** goes in the center.)

Next, add the following code, which creates the Debug toolbar:

```
// Create a Debug toolbar.  
JToolBar jtb = new JToolBar("Debug");  
  
// Load the images.  
ImageIcon set = new ImageIcon("setBP.gif");  
ImageIcon clear = new ImageIcon("clearBP.gif");  
ImageIcon resume = new ImageIcon("resume.gif");  
  
// Create the toolbar buttons.  
 JButton jbtnSet = new JButton(set);  
jbtnSet.setActionCommand("Set Breakpoint");  
jbtnSet.setToolTipText("Set Breakpoint");  
  
 JButton jbtnClear = new JButton(clear);  
jbtnClear.setActionCommand("Clear Breakpoint");  
jbtnClear.setToolTipText("Clear Breakpoint");  
  
 JButton jbtnResume = new JButton(resume);  
jbtnResume.setActionCommand("Resume");  
jbtnResume.setToolTipText("Resume");  
  
// Add the buttons to the toolbar.  
jtb.add(jbtnSet);  
jtb.add(jbtnClear);  
jtb.add(jbtnResume);  
  
// Add the toolbar to the north position of  
// the content pane.  
jfrm.add(jtb, BorderLayout.NORTH);
```

Let's look at this code closely. First, a **JToolBar** is created and given the title "Debug". Then, a set of **ImageIcon** objects are created that hold the images for the toolbar buttons. Next, three toolbar buttons are created. Notice that each has an image, but no text. Also, each is explicitly given an action command and a tooltip. The action commands are set because the buttons are not given names when they are constructed. Tooltips are especially useful when applied to icon-based toolbar components because sometimes it's hard to design images that are intuitive to all users. The buttons are then added to the toolbar, and the toolbar is added to the north side of the border layout of the

frame.

Finally, add the action listeners for the toolbar, as shown here:

```
// Add the toolbar action listeners.  
jbtnSet.addActionListener(this);  
jbtnClear.addActionListener(this);  
jbtnResume.addActionListener(this);
```

Each time the user presses a toolbar button, an action event is fired, and it is handled in the same way as the other menu-related events. [Figure 33-6](#) shows the toolbar in action.

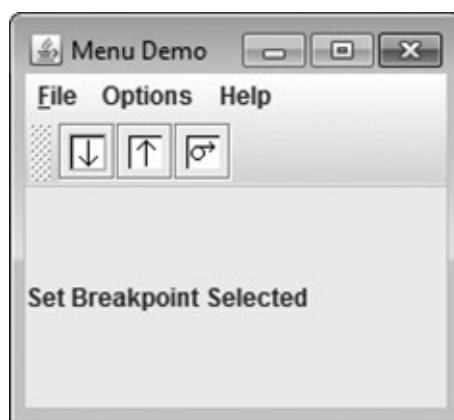


Figure 33-6 The Debug toolbar in action

Use Actions

Often, a toolbar and a menu item contain items in common. For example, the same functions provided by the Debug toolbar in the preceding example might also be offered through a menu selection. In such a case, selecting an option (such as setting a breakpoint) causes the same action to occur, independently of whether the menu or the toolbar was used. Also, both the toolbar button and the menu item would (most likely) use the same icon. Furthermore, when a toolbar button is disabled, the corresponding menu item would also need to be disabled. Such a situation would normally lead to a fair amount of duplicated, interdependent code, which is less than optimal. Fortunately, Swing provides a solution: the *action*.

An action is an instance of the **Action** interface. **Action** extends the **ActionListener** interface and provides a means of combining state information with the **actionPerformed()** event handler. This combination allows one

action to manage two or more components. For example, an action lets you centralize the control and handling of a toolbar button and a menu item. Instead of having to duplicate code, your program need only create an action that automatically handles both components.

Because **Action** extends **ActionListener**, an action must provide an implementation of the **actionPerformed()** method. This handler will process the action events generated by the objects linked to the action.

In addition to the inherited **actionPerformed()** method, **Action** defines several methods of its own. One of particular interest is **putValue()**. It sets the value of the various properties associated with an action and is shown here:

```
void putValue(String key, Object val)
```

It assigns *val* to the property specified by *key* that represents the desired property. Although not used by the example that follows, it is helpful to note that **Action** also supplies the **getValue()** method that obtains a specified property. It is shown here:

```
Object getValue(String key)
```

It returns a reference to the property specified by *key*.

The key values used by **putValue()** and **getValue()** include those shown here:

Key Value	Description
static final String ACCELERATOR_KEY	Represents the accelerator property. Accelerators are specified as KeyStroke objects.
static final String ACTION_COMMAND_KEY	Represents the action command property. An action command is specified as a string.
static final String DISPLAYED_MNEMONIC_INDEX_KEY	Represents the index of the character displayed as the mnemonic. This is an Integer value.
static final String LARGE_ICON_KEY	Represents the large icon associated with the action. The icon is specified as an object of type Icon .
static final String LONG_DESCRIPTION	Represents a long description of the action. This description is specified as a string.
static final String MNEMONIC_KEY	Represents the mnemonic property. A mnemonic is specified as a KeyEvent constant.
static final String NAME	Represents the name of the action (which also becomes the name of the button or menu item to which the action is linked). The name is specified as a string.
static final String SELECTED_KEY	Represents the selection status. If set, the item is selected. The state is represented by a Boolean value.
static final String SHORT_DESCRIPTION	Represents the tooltip text associated with the action. The tooltip text is specified as a string.
static final String SMALL_ICON	Represents the icon associated with the action. The icon is specified as an object of type Icon .

For example, to set the mnemonic to the letter *X*, use this call to **putValue()**:

```
action0b.putValue(MNEMONIC_KEY, KeyEvent.VK_X);
```

One **Action** property that is not accessible through **putValue()** and **getValue()** is the enabled/disabled status. For this, you use the **setEnabled()** and **isEnabled()** methods. They are shown here:

```
void setEnabled(boolean enabled)
```

```
boolean isEnabled()
```

For **setEnabled()**, if *enabled* is **true**, the action is enabled. Otherwise, it is disabled. If the action is enabled, **isEnabled()** returns **true**. Otherwise, it returns **false**.

Although you can implement all of the **Action** interface yourself, you won't usually need to. Instead, Swing provides a partial implementation called **AbstractAction** that you can extend. By extending **AbstractAction**, you need implement only one method: **actionPerformed()**. The other **Action** methods are provided for you. **AbstractAction** provides three constructors. The one used in this chapter is shown here:

```
AbstractAction(String name, Icon image)
```

It constructs an **AbstractAction** that has the name specified by *name* and the icon specified by *image*.

Once you have created an action, it can be added to a **JToolBar** and used to construct a **JMenuItem**. To add an action to a **JToolBar**, use this version of **add()**:

```
JButton add(Action actObj)
```

Here, *actObj* is the action that is being added to the toolbar. The properties defined by *actObj* are used to create a toolbar button. To create a menu item from an action, use this **JMenuItem** constructor:

```
JMenuItem(Action actObj)
```

Here, *actObj* is the action used to construct a menu item according to its properties.

NOTE In addition to **JToolBar** and **JMenuItem**, actions are also supported by several other Swing components, such as **JPopupMenu**, **JButton**, **JRadioButton**, and **JCheckBox**. **JRadioButtonMenuItem** and **JCheckBoxMenuItem** also support actions.

To illustrate the benefit of actions, we will use them to manage the Debug toolbar created in the previous section. We will also add a Debug submenu under the Options main menu. The Debug submenu will contain the same selections as the Debug toolbar: Set Breakpoint, Clear Breakpoint, and Resume. The same actions that support these items in the toolbar will also support these items in the menu. Therefore, instead of having to create duplicate code to handle both the toolbar and menu, both are handled by the actions.

Begin by creating an inner class called **DebugAction** that extends **AbstractAction**, as shown here:

```
// A class to create an action for the Debug menu
// and toolbar.
class DebugAction extends AbstractAction {
    public DebugAction(String name, Icon image, int mnem,
                       int accel, String tTip) {
        super(name, image);
        putValue(ACCELERATOR_KEY,
                 KeyStroke.getKeyStroke(accel,
                                       InputEvent.CTRL_DOWN_MASK));
        putValue(MNEMONIC_KEY, mnem);
        putValue(SHORT_DESCRIPTION, tTip);
    }

    // Handle events for both the toolbar and the
    // Debug menu.
    public void actionPerformed(ActionEvent ae) {
        String comStr = ae.getActionCommand();

        jlab.setText(comStr + " Selected");

        // Toggle the enabled status of the
        // Set and Clear Breakpoint options.
        if(comStr.equals("Set Breakpoint")) {
            clearAct.setEnabled(true);
            setAct.setEnabled(false);
        } else if(comStr.equals("Clear Breakpoint")) {
            clearAct.setEnabled(false);
            setAct.setEnabled(true);
        }
    }
}
```

DebugAction extends **AbstractAction**. It creates an action class that will be used to define the properties associated with the Debug menu and toolbar. Its constructor has five parameters that let you specify the following items:

- Name

- Icon
- Mnemonic
- Accelerator
- Tooltip

The first two are passed to **AbstractAction**'s constructor via **super**. The other three properties are set through calls to **putValue()**.

The **actionPerformed()** method of **DebugAction** handles events for the action. This means that when an instance of **DebugAction** is used to create a toolbar button and a menu item, events generated by either of those components are handled by the **actionPerformed()** method in **DebugAction**. Notice that this handler displays the selection in **jlab**. In addition, if the Set Breakpoint option is selected, then the Clear Breakpoint option is enabled and the Set Breakpoint option is disabled. If the Clear Breakpoint option is selected, then the Set Breakpoint option is enabled and the Clear Breakpoint option is disabled. This illustrates how an action can be used to enable or disable a component. When an action is disabled, it is disabled for all uses of that action. In this case, if Set Breakpoint is disabled, then it is disabled both in the toolbar and in the menu.

Next, add these **DebugAction** instance variables to **MenuDemo**:

```
DebugAction setAct;
DebugAction clearAct;
DebugAction resumeAct;
```

Next, create three **ImageIcons** that represent the Debug options, as shown here:

```
// Load the images for the actions.
ImageIcon setIcon = new ImageIcon("setBP.gif");
ImageIcon clearIcon = new ImageIcon("clearBP.gif");
ImageIcon resumeIcon = new ImageIcon("resume.gif");
```

Now, create the actions that manage the Debug options, as shown here:

```

// Create actions.
setAct =
    new DebugAction("Set Breakpoint",
                    setIcon,
                    KeyEvent.VK_S,
                    KeyEvent.VK_B,
                    "Set a break point.");
```

```

clearAct =
    new DebugAction("Clear Breakpoint",
                    clearIcon,
                    KeyEvent.VK_C,
                    KeyEvent.VK_L,
                    "Clear a break point.");
```

```

resumeAct =
    new DebugAction("Resume",
                    resumeIcon,
                    KeyEvent.VK_R,
                    KeyEvent.VK_R,
                    "Resume execution after breakpoint.");
```

```

// Initially disable the Clear Breakpoint option.
clearAct.setEnabled(false);
```

Notice that the accelerator for Set Breakpoint is B and the accelerator for Clear Breakpoint is L. The reason these keys are used rather than S and C is that these keys are already allocated by the File menu for Save and Close. However, they can still be used as mnemonics because each mnemonic is localized to its own menu. Also notice that the action that represents Clear Breakpoint is initially disabled. It will be enabled only after a breakpoint has been set.

Next, use the actions to create buttons for the toolbar and then add those buttons to the toolbar, as shown here:

```

// Create the toolbar buttons by using the actions.
JButton jbtnSet = new JButton(setAct);
JButton jbtnClear = new JButton(clearAct);
JButton jbtnResume = new JButton(resumeAct);

// Create a Debug toolbar.
```

```

JToolBar jtb = new JToolBar("Breakpoints");

// Add the buttons to the toolbar.
jtb.add(jbtnSet);
jtb.add(jbtnClear);
jtb.add(jbtnResume);

// Add the toolbar to the north position of
// the content pane.
jfrm.add(jtb, BorderLayout.NORTH);

```

Finally, create the Debug menu, as shown next:

```

// Now, create a Debug menu that goes under the Options
// menu bar item. Use the actions to create the items.
JMenu jmDebug = new JMenu("Debug");
JMenuItem jmiSetBP = new JMenuItem(setAct);
JMenuItem jmiClearBP = new JMenuItem(clearAct);
JMenuItem jmiResume = new JMenuItem(resumeAct);
jmDebug.add(jmiSetBP);
jmDebug.add(jmiClearBP);
jmDebug.add(jmiResume);
jmOptions.add(jmDebug);

```

After making these changes and additions, the actions that you created will be used to manage both the Debug menu and the toolbar. Thus, changing a property in the action (such as disabling it) will affect all uses of that action. The program will now look as shown in [Figure 33-7](#).

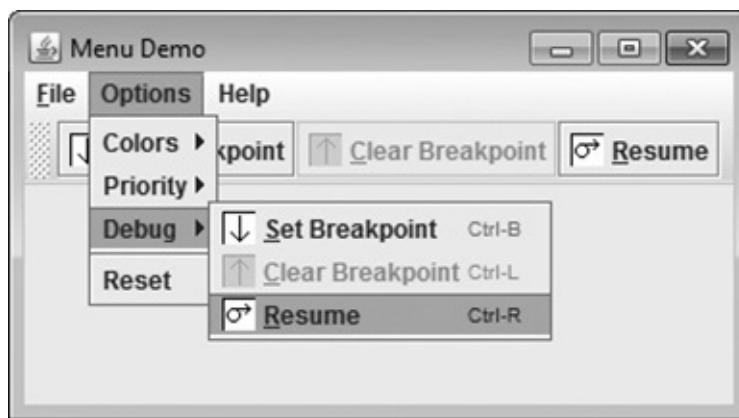


Figure 33-7 Using actions to manage the Debug toolbar and menu

Put the Entire MenuDemo Program Together

Throughout the course of this discussion, many changes and additions have been made to the **MenuDemo** program shown at the start of the chapter. Before concluding, it will be helpful to assemble all the pieces. Doing so not only eliminates any ambiguity about the way the pieces fit together, but it also gives you a complete menu demonstration program that you can experiment with.

The following version of **MenuDemo** includes all of the additions and enhancements described in this chapter. For clarity, the program has been reorganized, with separate methods being used to construct the various menus and toolbar. Notice that several of the menu-related variables, such as **jmb**, **jmFile**, and **jtb**, have been made into instance variables.

```
// The complete MenuDemo program.

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

class MenuDemo implements ActionListener {

    JLabel jlab;

    JMenuBar jmb;

    JToolBar jtb;

    JPopupMenu jpu;

    DebugAction setAct;
    DebugAction clearAct;
    DebugAction resumeAct;

    MenuDemo() {
        // Create a new JFrame container.
        JFrame jfrm = new JFrame("Complete Menu Demo");

        // Use default border layout.

        // Give the frame an initial size.
        jfrm.setSize(360, 200);

        // Terminate the program when the user closes the application.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Create a label that will display the menu selection.
        jlab = new JLabel();

        // Create the menu bar.
        jmb = new JMenuBar();

        // Make the File menu.
        makeFileMenu();
```

```
// Construct the Debug actions.  
makeActions();  
  
// Make the toolbar.  
makeToolBar();  
  
// Make the Options menu.  
makeOptionsMenu();  
  
// Make the Help menu.  
makeHelpMenu();  
  
// Make the Edit popup menu.  
makeEditPUMenu();  
  
// Add a listener for the popup trigger.  
jfrm.addMouseListener(new MouseAdapter() {  
    public void mousePressed(MouseEvent me) {  
        if(me.isPopupTrigger())  
            jpu.show(me.getComponent(), me.getX(), me.getY());  
    }  
    public void mouseReleased(MouseEvent me) {  
        if(me.isPopupTrigger())  
            jpu.show(me.getComponent(), me.getX(), me.getY());  
    }  
});  
  
// Add the label to the center of the content pane.  
jfrm.add(jlab, SwingConstants.CENTER);  
  
// Add the toolbar to the north position of  
// the content pane.  
jfrm.add(jtb, BorderLayout.NORTH);  
  
// Add the menu bar to the frame.  
jfrm.setJMenuBar(jmb);  
  
// Display the frame.  
jfrm.setVisible(true);  
}  
  
// Handle menu item action events.  
// This does NOT handle events generated  
// by the Debug options.  
public void actionPerformed(ActionEvent ae) {  
    // Get the action command from the menu selection.  
    String comStr = ae.getActionCommand();  
  
    // If user chooses Exit, then exit the program.  
    if(comStr.equals("Exit")) System.exit(0);  
  
    // Otherwise, display the selection.  
    jlab.setText(comStr + " Selected");  
}
```



```
JMenuItem jmiExit = new JMenuItem("Exit",
                                  KeyEvent.VK_E);
jmiExit.setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_E,
                          InputEvent.CTRL_DOWN_MASK));

jmFile.add(jmiOpen);
jmFile.add(jmiClose);
jmFile.add(jmiSave);
jmFile.addSeparator();
jmFile.add(jmiExit);
jmb.add(jmFile);

// Add the action listeners for the File menu.
jmiOpen.addActionListener(this);
jmiClose.addActionListener(this);
jmiSave.addActionListener(this);
jmiExit.addActionListener(this);
}

// Create the Options menu.
void makeOptionsMenu() {
    JMenu jmOptions = new JMenu("Options");

    // Create the Colors submenu.
    JMenu jmColors = new JMenu("Colors");

    // Use check boxes for colors. This allows
    // the user to select more than one color.
    JCheckBoxMenuItem jmiRed = new JCheckBoxMenuItem("Red");
    JCheckBoxMenuItem jmiGreen = new JCheckBoxMenuItem("Green");
    JCheckBoxMenuItem jmiBlue = new JCheckBoxMenuItem("Blue");

    // Add the items to the Colors menu.
    jmColors.add(jmiRed);
    jmColors.add(jmiGreen);
    jmColors.add(jmiBlue);
    jmOptions.add(jmColors);

    // Create the Priority submenu.
    JMenu jmPriority = new JMenu("Priority");

    // Use radio buttons for the priority setting.
    // This lets the menu show which priority is used
    // but also ensures that one and only one priority
    // can be selected at any one time. Notice that
    // the High radio button is initially selected.
    JRadioButtonMenuItem jmiHigh =
        new JRadioButtonMenuItem("High", true);
    JRadioButtonMenuItem jmiLow =
        new JRadioButtonMenuItem("Low");

    // Add the items to the Priority menu.
    jmPriority.add(jmiHigh);
```



```
jmPriority.add(jmiLow);
jmOptions.add(jmPriority);

// Create a button group for the radio button
// menu items.
ButtonGroup bg = new ButtonGroup();
bg.add(jmiHigh);
bg.add(jmiLow);

// Now, create a Debug submenu that goes under
// the Options menu bar item. Use actions to
// create the items.
JMenu jmDebug = new JMenu("Debug");
JMenuItem jmiSetBP = new JMenuItem(setAct);
JMenuItem jmiClearBP = new JMenuItem(clearAct);
JMenuItem jmiResume = new JMenuItem(resumeAct);

// Add the items to the Debug menu.
jmDebug.add(jmiSetBP);
jmDebug.add(jmiClearBP);
jmDebug.add(jmiResume);
jmOptions.add(jmDebug);

// Create the Reset menu item.
JMenuItem jmiReset = new JMenuItem("Reset");
jmOptions.addSeparator();
jmOptions.add(jmiReset);

// Finally, add the entire options menu to
// the menu bar
jmb.add(jmOptions);

// Add the action listeners for the Options menu,
// except for those supported by the Debug menu.
jmiRed.addActionListener(this);
jmiGreen.addActionListener(this);
jmiBlue.addActionListener(this);
jmiHigh.addActionListener(this);
jmiLow.addActionListener(this);
jmiReset.addActionListener(this);
}

// Create the Help menu.
void makeHelpMenu() {
    JMenu jmHelp = new JMenu("Help");

    // Add an icon to the About menu item.
    ImageIcon icon = new ImageIcon("AboutIcon.gif");

    JMenuItem jmiAbout = new JMenuItem("About", icon);
    jmiAbout.setToolTipText("Info about the MenuDemo program.");
    jmHelp.add(jmiAbout);
    jmb.add(jmHelp);
```



```
// Add action listener for About.  
jmiAbout.addActionListener(this);  
}  
  
// Construct the actions needed by the Debug menu  
// and toolbar.  
void makeActions() {  
    // Load the images for the actions.  
    ImageIcon setIcon = new ImageIcon("setBP.gif");  
    ImageIcon clearIcon = new ImageIcon("clearBP.gif");  
    ImageIcon resumeIcon = new ImageIcon("resume.gif");  
  
    // Create actions.  
    setAct =  
        new DebugAction("Set Breakpoint",  
                        setIcon,  
                        KeyEvent.VK_S,  
                        KeyEvent.VK_B,  
                        "Set a break point.");  
  
    clearAct =  
        new DebugAction("Clear Breakpoint",  
                        clearIcon,  
                        KeyEvent.VK_C,  
                        KeyEvent.VK_L,  
                        "Clear a break point.");  
  
    resumeAct =  
        new DebugAction("Resume",  
                        resumeIcon,  
                        KeyEvent.VK_R,  
                        KeyEvent.VK_R,  
                        "Resume execution after breakpoint.");  
  
    // Initially disable the Clear Breakpoint option.  
    clearAct.setEnabled(false);  
}  
  
// Create the Debug toolbar.  
void makeToolBar() {  
    // Create the toolbar buttons by using the actions.  
    JButton jbtnSet = new JButton(setAct);  
    JButton jbtnClear = new JButton(clearAct);  
    JButton jbtnResume = new JButton(resumeAct);  
  
    // Create the Debug toolbar.  
    jtb = new JToolBar("Breakpoints");  
  
    // Add the buttons to the toolbar.  
    jtb.add(jbtnSet);  
    jtb.add(jbtnClear);  
    jtb.add(jbtnResume);  
}
```

```

// Create the Edit popup menu.
void makeEditPUMenu() {
    jpu = new JPopupMenu();

    // Create the popup menu items
    JMenuItem jmiCut = new JMenuItem("Cut");
    JMenuItem jmiCopy = new JMenuItem("Copy");
    JMenuItem jmiPaste = new JMenuItem("Paste");

    // Add the menu items to the popup menu.
    jpu.add(jmiCut);
    jpu.add(jmiCopy);
    jpu.add(jmiPaste);

    // Add the Edit popup menu action listeners.
    jmiCut.addActionListener(this);
    jmiCopy.addActionListener(this);
    jmiPaste.addActionListener(this);
}

public static void main(String args[]) {
    // Create the frame on the event dispatching thread.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new MenuDemo();
        }
    });
}
}

```

Continuing Your Exploration of Swing

Swing defines a very large GUI toolkit. It has many more features that you will want to explore on your own. For example, it supplies dialog classes, such as **JOptionPane** and **JDialog**, that you can use to streamline the construction of dialog windows. It also provides additional controls beyond those introduced in [Chapter 31](#). Two you will want to explore are **JSpinner** (which creates a spin

control) and **JFormattedTextField** (which supports formatted text). You will also want to experiment with defining your own models for the various components. Frankly, the best way to become familiar with Swing's capabilities is to experiment with it.

PART

IV

Applying Java

CHAPTER 34

Java Beans

CHAPTER 35

Introducing Servlets

CHAPTER

Java Beans

This chapter provides an overview of creating Java Beans. Beans are important because they allow you to build complex systems from software components. These components may be provided by you or supplied by one or more different vendors. Java Beans use an architecture called *JavaBeans* that specifies how these building blocks can operate together.

To better understand the value of Beans, consider the following. Hardware designers have a wide variety of components that can be integrated together to construct a system. Resistors, capacitors, and inductors are examples of simple building blocks. Integrated circuits provide more advanced functionality. All of these different parts can be reused. It is not necessary or possible to rebuild these capabilities each time a new system is needed. Also, the same pieces can be used in different types of circuits. This is possible because the behavior of these components is understood and documented.

The software industry also sought the benefits of reusability and interoperability of a component-based approach. To realize these benefits, a component architecture is needed that allows programs to be assembled from software building blocks, perhaps provided by different vendors. It must also be possible for a designer to select a component, understand its capabilities, and incorporate it into an application. When a new version of a component becomes available, it should be easy to incorporate this functionality into existing code. JavaBeans provides just such an architecture.

What Is a Java Bean?

A *Java Bean* is a software component that has been designed to be reusable in a variety of different environments. There is no restriction on the capability of a Bean. It may perform a simple function, such as obtaining an inventory value, or a complex function, such as forecasting the performance of a stock portfolio. A Bean may be visible to an end user. One example of this is a button on a graphical user interface. A Bean may also be invisible to a user. Software to decode a stream of multimedia information in real time is an example of this type of building block. Finally, a Bean may be designed to work autonomously

on a user's workstation or to work in cooperation with a set of other distributed components. Software to generate a pie chart from a set of data points is an example of a Bean that can execute locally. However, a Bean that provides real-time price information from a stock or commodities exchange would need to work in cooperation with other distributed software to obtain its data.

Advantages of Beans

The following list enumerates some of the benefits that JavaBeans technology provides for a component developer:

- A Bean obtains all the benefits of Java's "write-once, run-anywhere" paradigm.
- The properties, events, and methods of a Bean that are exposed to another application can be controlled.
- Auxiliary software can be provided to help configure a Bean. This software is only needed when the design-time parameters for that component are being set. It does not need to be included in the run-time environment.
- The state of a Bean can be saved in persistent storage and restored at a later time.
- A Bean may register to receive events from other objects and can generate events that are sent to other objects.

Introspection

At the core of Bean programming is *introspection*. This is the process of analyzing a Bean to determine its capabilities. This is an essential feature of the JavaBeans API because it allows another application, such as a design tool, to obtain information about a component. Without introspection, the JavaBeans technology could not operate.

There are two ways in which the developer of a Bean can indicate which of its properties, events, and methods should be exposed. With the first method, simple naming conventions are used. These allow the introspection mechanisms to infer information about a Bean. In the second way, an additional class that extends the **BeanInfo** interface is provided that explicitly supplies this information. Both approaches are examined here.

Design Patterns for Properties

A *property* is a subset of a Bean's state. The values assigned to the properties determine the behavior and appearance of that component. A property is set through a *setter* method. A property is obtained by a *getter* method. There are two types of properties: simple and indexed.

Simple Properties

A simple property has a single value. It can be identified by the following design patterns, where **N** is the name of the property and **T** is its type:

```
public T getN()  
public void setN(T arg)
```

A read/write property has both of these methods to access its values. A read-only property has only a get method. A write-only property has only a set method.

Here are three read/write simple properties along with their getter and setter methods:

```

private double depth, height, width;

public double getDepth( ) {
    return depth;
}

public void setDepth(double d) {
    depth = d;
}

public double getHeight( ) {
    return height;
}

public void setHeight(double h) {
    height = h;
}

public double getWidth( ) {
    return width;
}

public void setWidth(double w) {
    width = w;
}

```

NOTE For a **boolean** property, a method of the form **isPropertyName()** can also be used as an accessor.

Indexed Properties

An indexed property consists of multiple values. It can be identified by the following design patterns, where **N** is the name of the property and **T** is its type:

```

public T getN(int index);
public void setN(int index, T value);
public T[ ] getN( );
public void setN(T values[ ]);

```

Here is an indexed property called **data** along with its getter and setter methods:

```

private double data[ ];

public double getData(int index) {
    return data[index];
}

public void setData(int index, double value) {
    data[index] = value;
}

public double[ ] getData( ) {
    return data;
}

public void setData(double[ ] values) {
    data = new double[values.length];
    System.arraycopy(values, 0, data, 0, values.length);
}

```

Design Patterns for Events

Beans use the delegation event model that was discussed earlier in this book. Beans can generate events and send them to other objects. These can be identified by the following design patterns, where **T** is the type of the event:

```

public void addTListener(TListener eventListener)
public void addTListener(TListener eventListener)
                    throws java.util.TooManyListenersException
public void removeTListener(TListener eventListener)

```

These methods are used to add or remove a listener for the specified event. The version of **addTListener()** that does not throw an exception can be used to *multicast* an event, which means that more than one listener can register for the event notification. The version that throws **TooManyListenersException** *unicasts* the event, which means that the number of listeners can be restricted to one. In either case, **removeTListener()** is used to remove the listener. For example, assuming an event interface type called **TemperatureListener**, a Bean that monitors temperature might supply the following methods:

```

public void addTemperatureListener(TemperatureListener t1) {
...
}

public void removeTemperatureListener(TemperatureListener t1) {

```

```
...  
}
```

Methods and Design Patterns

Design patterns are not used for naming nonproperty methods. The introspection mechanism finds all of the public methods of a Bean. Protected and private methods are not presented.

Using the BeanInfo Interface

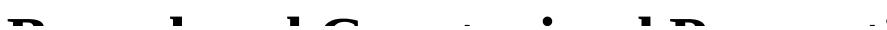
As the preceding discussion shows, design patterns *implicitly* determine what information is available to the user of a Bean. The **BeanInfo** interface enables you to *explicitly* control what information is available. The **BeanInfo** interface defines several methods, including these:

```
PropertyDescriptor[ ] getPropertyDescriptors( )  
EventSetDescriptor[ ] getEventSetDescriptors( )  
MethodDescriptor[ ] getMethodDescriptors( )
```

They return arrays of objects that provide information about the properties, events, and methods of a Bean. The classes **PropertyDescriptor**, **EventSetDescriptor**, and **MethodDescriptor** are defined within the **java.beans** package, and they describe the indicated elements. By implementing these methods, a developer can designate exactly what is presented to a user, bypassing introspection based on design patterns.

When creating a class that implements **BeanInfo**, you must call that class *bnameBeanInfo*, where *bname* is the name of the Bean. For example, if the Bean is called **MyBean**, then the information class must be called **MyBeanBeanInfo**.

To simplify the use of **BeanInfo**, JavaBeans supplies the **SimpleBeanInfo** class. It provides default implementations of the **BeanInfo** interface, including the three methods just shown. You can extend this class and override one or more of the methods to explicitly control what aspects of a Bean are exposed. If you don't override a method, then design-pattern introspection will be used. For example, if you don't override **getPropertyDescriptors()**, then design patterns are used to discover a Bean's properties. You will see **SimpleBeanInfo** in action later in this chapter.



Bound and Constrained Properties

A Bean that has a *bound* property generates an event when the property is changed. The event is of type **PropertyChangeEvent** and is sent to objects that previously registered an interest in receiving such notifications. A class that handles this event must implement the **PropertyChangeListener** interface.

A Bean that has a *constrained* property generates an event when an attempt is made to change its value. It also generates an event of type **PropertyChangeEvent**. It too is sent to objects that previously registered an interest in receiving such notifications. However, those other objects have the ability to veto the proposed change by throwing a **PropertyVetoException**. This capability allows a Bean to operate differently according to its run-time environment. A class that handles this event must implement the **VetoableChangeListener** interface.

Persistence

Persistence is the ability to save the current state of a Bean, including the values of a Bean's properties and instance variables, to nonvolatile storage and to retrieve them at a later time. The object serialization capabilities provided by the Java class libraries are used to provide persistence for Beans.

The easiest way to serialize a Bean is to have it implement the **java.io.Serializable** interface, which is simply a marker interface. Implementing **java.io.Serializable** makes serialization automatic. Your Bean need take no other action. Automatic serialization can also be inherited. Therefore, if any superclass of a Bean implements **java.io.Serializable**, then automatic serialization is obtained.

When using automatic serialization, you can selectively prevent a field from being saved through the use of the **transient** keyword. Thus, data members of a Bean specified as **transient** will not be serialized.

If a Bean does not implement **java.io.Serializable**, you must provide serialization yourself, such as by implementing **java.io.Externalizable**. Otherwise, containers cannot save the configuration of your component.

Customizers

A Bean developer can provide a *customizer* that helps another developer

configure the Bean. A customizer can provide a step-by-step guide through the process that must be followed to use the component in a specific context. Online documentation can also be provided. A Bean developer has great flexibility to develop a customizer that can differentiate his or her product in the marketplace.

The JavaBeans API

The JavaBeans functionality is provided by a set of classes and interfaces in the **java.beans** package. Beginning with JDK 9, this package is in the **java.desktop** module. This section provides a brief overview of its contents. [Table 34-1](#) lists the interfaces in **java.beans** and provides a brief description of their functionality. [Table 34-2](#) lists the classes in **java.beans**.

Interface	Description
AppletInitializer	Methods in this interface are used to initialize Beans that are also applets. (Deprecated by JDK 9.)
BeanInfo	This interface allows a designer to specify information about the properties, events, and methods of a Bean.
Customizer	This interface allows a designer to provide a graphical user interface through which a Bean may be configured.
DesignMode	Methods in this interface determine if a Bean is executing in design mode.
ExceptionListener	A method in this interface is invoked when an exception has occurred.
PropertyChangeListener	A method in this interface is invoked when a bound property is changed.
PropertyEditor	Objects that implement this interface allow designers to change and display property values.
VetoableChangeListener	A method in this interface is invoked when a constrained property is changed.
Visibility	Methods in this interface allow a Bean to execute in environments where a graphical user interface is not available.

Table 34-1 The Interfaces in **java.beans**

Class	Description
BeanDescriptor	This class provides information about a Bean. It also allows you to associate a customizer with a Bean.
Beans	This class is used to obtain information about a Bean.
DefaultPersistenceDelegate	A concrete subclass of PersistenceDelegate .
Encoder	Encodes the state of a set of Beans. Can be used to write this information to a stream.
EventHandler	Supports dynamic event listener creation.
EventSetDescriptor	Instances of this class describe an event that can be generated by a Bean.
Expression	Encapsulates a call to a method that returns a result.
FeatureDescriptor	This is the superclass of the PropertyDescriptor , EventSetDescriptor , and MethodDescriptor classes, among others.

IndexedPropertyChangeEvent	A subclass of PropertyChangeEvent that represents a change to an indexed property.
IndexedPropertyDescriptor	Instances of this class describe an indexed property of a Bean.
IntrospectionException	An exception of this type is generated if a problem occurs when analyzing a Bean.
Introspector	This class analyzes a Bean and constructs a BeanInfo object that describes the component.
MethodDescriptor	Instances of this class describe a method of a Bean.
ParameterDescriptor	Instances of this class describe a method parameter.
PersistenceDelegate	Handles the state information of an object.
PropertyChangeEvent	This event is generated when bound or constrained properties are changed. It is sent to objects that registered an interest in these events and that implement either the PropertyChangeListener or VetoableChangeListener interfaces.
PropertyChangeListenerProxy	Extends EventListenerProxy and implements PropertyChangeListener .
PropertyChangeSupport	Beans that support bound properties can use this class to notify PropertyChangeListener objects.
PropertyDescriptor	Instances of this class describe a property of a Bean.
PropertyEditorManager	This class locates a PropertyEditor object for a given type.
PropertyEditorSupport	This class provides functionality that can be used when writing property editors.
PropertyVetoException	An exception of this type is generated if a change to a constrained property is vetoed.
SimpleBeanInfo	This class provides functionality that can be used when writing BeanInfo classes.
Statement	Encapsulates a call to a method.
VetoableChangeListenerProxy	Extends EventListenerProxy and implements VetoableChangeListener .
VetoableChangeSupport	Beans that support constrained properties can use this class to notify VetoableChangeListener objects.
XMLDecoder	Used to read a Bean from an XML document.
XMLEncoder	Used to write a Bean to an XML document.

Table 34-2 The Classes in **java.beans**

Although it is beyond the scope of this chapter to discuss all of the classes, four are of particular interest: **Introspector**, **PropertyDescriptor**,

EventSetDescriptor, and **MethodDescriptor**. Each is briefly examined here.

Introspector

The **Introspector** class provides several static methods that support introspection. Of most interest is **getBeanInfo()**. This method returns a **BeanInfo** object that can be used to obtain information about the Bean. The **getBeanInfo()** method has several forms, including the one shown here:

```
static BeanInfo getBeanInfo(Class<?> bean) throws IntrospectionException
```

The returned object contains information about the Bean specified by *bean*.

PropertyDescriptor

The **PropertyDescriptor** class describes the characteristics of a Bean property. It supports several methods that manage and describe properties. For example, you can determine if a property is bound by calling **isBound()**. To determine if a property is constrained, call **isConstrained()**. You can obtain the name of a property by calling **getName()**.

EventSetDescriptor

The **EventSetDescriptor** class represents a set of Bean events. It supports several methods that obtain the methods that a Bean uses to add or remove event listeners, and to otherwise manage events. For example, to obtain the method used to add listeners, call **getAddListenerMethod()**. To obtain the method used to remove listeners, call **getRemoveListenerMethod()**. To obtain the type of a listener, call **getListenerType()**. You can obtain the name of an event set by calling **getName()**.

MethodDescriptor

The **MethodDescriptor** class represents a Bean method. To obtain the name of the method, call **getName()**. You can obtain information about the method by calling **getMethod()**, shown here:

```
Method getMethod()
```

An object of type **Method** that describes the method is returned.

A Bean Example

This chapter concludes with an example that illustrates various aspects of Bean programming, including introspection and using a **BeanInfo** class. It also makes use of the **Introspector**, **PropertyDescriptor**, and **EventSetDescriptor** classes. The example uses three classes. The first is a Bean called **Colors**, shown here:

```
// A simple Bean.  
import java.awt.*;  
import java.awt.event.*;  
import java.io.Serializable;  
  
public class Colors extends Canvas implements Serializable {  
    transient private Color color; // not persistent  
    private boolean rectangular; // is persistent
```

```
public Colors() {
    addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent me) {
            change();
        }
    });
    rectangular = false;
    setSize(200, 100);
    change();
}

public boolean getRectangular() {
    return rectangular;
}

public void setRectangular(boolean flag) {
    this.rectangular = flag;
    repaint();
}

public void change() {
    color = randomColor();
    repaint();
}

private Color randomColor() {
    int r = (int)(255*Math.random());
    int g = (int)(255*Math.random());
    int b = (int)(255*Math.random());
    return new Color(r, g, b);
}

public void paint(Graphics g) {
    Dimension d = getSize();
    int h = d.height;
    int w = d.width;
    g.setColor(color);
    if(rectangular) {
        g.fillRect(0, 0, w-1, h-1);
    }
    else {
        g.fillOval(0, 0, w-1, h-1);
    }
}
```

The **Colors** Bean displays a colored object within a frame. The color of the component is determined by the private **Color** variable **color**, and its shape is determined by the private **boolean** variable **rectangular**. The constructor defines an anonymous inner class that extends **MouseAdapter** and overrides its **mousePressed()** method. The **change()** method is invoked in response to mouse presses. It selects a random color and then repaints the component. The **getRectangular()** and **setRectangular()** methods provide access to the one property of this Bean. The **change()** method calls **randomColor()** to choose a color and then calls **repaint()** to make the change visible. Notice that the **paint()** method uses the **rectangular** and **color** variables to determine how to present the Bean.

The next class is **ColorsBeanInfo**. It is a subclass of **SimpleBeanInfo** that provides explicit information about **Colors**. It overrides **getPropertyDescriptors()** in order to designate which properties are presented to a Bean user. In this case, the only property exposed is **rectangular**. The method creates and returns a **PropertyDescriptor** object for the **rectangular** property. The **PropertyDescriptor** constructor that is used is shown here:

```
PropertyDescriptor(String property, Class<?> beanCls)  
throws IntrospectionException
```

Here, the first argument is the name of the property, and the second argument is the class of the Bean.

```
// A Bean information class.  
import java.beans.*;  
public class ColorsBeanInfo extends SimpleBeanInfo {  
    public PropertyDescriptor[] getPropertyDescriptors() {  
        try {  
            PropertyDescriptor rectangular = new  
                PropertyDescriptor("rectangular", Colors.class);  
            PropertyDescriptor pd[] = {rectangular};  
            return pd;  
        }  
        catch(Exception e) {  
            System.out.println("Exception caught. " + e);  
        }  
        return null;  
    }  
}
```

The final class is called **IntrospectorDemo**. It uses introspection to display the properties and events that are available within the **Colors** Bean.

```
// Show properties and events.  
import java.awt.*;  
import java.beans.*;  
  
public class IntrospectorDemo {  
    public static void main(String args[]) {  
        try {  
            Class<?> c = Class.forName("Colors");  
            BeanInfo beanInfo = Introspector.getBeanInfo(c);  
  
            System.out.println("Properties:");  
            PropertyDescriptor propertyDescriptor[] =  
                beanInfo.getPropertyDescriptors();  
            for(int i = 0; i < propertyDescriptor.length; i++) {  
                System.out.println("\t" + propertyDescriptor[i].getName());  
            }  
        }
```

```

        System.out.println("Events:");
        EventSetDescriptor eventSetDescriptor[] =
            beanInfo.getEventSetDescriptors();
        for(int i = 0; i < eventSetDescriptor.length; i++) {
            System.out.println("\t" + eventSetDescriptor[i].getName());
        }
    }
    catch(Exception e) {
        System.out.println("Exception caught. " + e);
    }
}
}

```

The output from this program is the following:

```

Properties:
    rectangular
Events:
    mouseWheel
    mouse
    mouseMotion
    component
    hierarchyBounds
    focus
    hierarchy
    propertyChange
    inputMethod
    key

```

Notice two things in the output. First, because **ColorsBeanInfo** overrides **getPropertyDescriptors()** such that the only property returned is **rectangular**, only the **rectangular** property is displayed. However, because **getEventSetDescriptors()** is not overridden by **ColorsBeanInfo**, design-pattern introspection is used, and all events are found, including those in **Colors**' superclass, **Canvas**. Remember, if you don't override one of the "get" methods defined by **SimpleBeanInfo**, then the default, design-pattern introspection is used. To observe the difference that **ColorsBeanInfo** makes, erase its class file and then run **IntrospectorDemo** again. This time it will report more properties.

CHAPTER

35

Introducing Servlets

This chapter presents an introduction to *servlets*. Servlets are small programs that execute on the server side of a web connection. The topic of servlets is quite large, and it is beyond the scope of this chapter to cover it all. Instead, we will focus on the core concepts, interfaces, and classes, and develop several examples.

Background

In order to understand the advantages of servlets, you must have a basic understanding of how web browsers and servers cooperate to provide content to a user. Consider a request for a static web page. A user enters a Uniform Resource Locator (URL) into a browser. The browser generates an HTTP request to the appropriate web server. The web server maps this request to a specific file. That file is returned in an HTTP response to the browser. The HTTP header in the response indicates the type of the content. The Multipurpose Internet Mail Extensions (MIME) are used for this purpose. For example, ordinary ASCII text has a MIME type of text/plain. The Hypertext Markup Language (HTML) source code of a web page has a MIME type of text/html.

Now consider dynamic content. Assume that an online store uses a database to store information about its business. This would include items for sale, prices, availability, orders, and so forth. It wishes to make this information accessible to customers via web pages. The contents of those web pages must be dynamically generated to reflect the latest information in the database.

In the early days of the Web, a server could dynamically construct a page by creating a separate process to handle each client request. The process would open connections to one or more databases in order to obtain the necessary information. It communicated with the web server via an interface known as the Common Gateway Interface (CGI). CGI allowed the separate process to read data from the HTTP request and write data to the HTTP response. A variety of different languages were used to build CGI programs. These included C, C++, and Perl.

However, CGI suffered serious performance problems. It was expensive in terms of processor and memory resources to create a separate process for each client request. It was also expensive to open and close database connections for each client request. In addition, the CGI programs were not platform-independent. Therefore, other techniques were introduced. Among these are servlets.

Servlets offer several advantages in comparison with CGI. First, performance is significantly better. Servlets execute within the address space of a web server. It is not necessary to create a separate process to handle each client request. Second, servlets are platform-independent because they are written in Java. Third, the Java security manager on the server enforces a set of restrictions to protect the resources on a server machine. Finally, the full functionality of the Java class libraries is available to a servlet. It can communicate with other software via the sockets and RMI mechanisms that you have seen already.

The Life Cycle of a Servlet

Three methods are central to the life cycle of a servlet. These are **init()**, **service()**, and **destroy()**. They are implemented by every servlet and are invoked at specific times by the server. Let us consider a typical user scenario to understand when these methods are called.

First, assume that a user enters a Uniform Resource Locator (URL) to a web browser. The browser then generates an HTTP request for this URL. This request is then sent to the appropriate server.

Second, this HTTP request is received by the web server. The server maps this request to a particular servlet. The servlet is dynamically retrieved and loaded into the address space of the server.

Third, the server invokes the **init()** method of the servlet. This method is invoked only when the servlet is first loaded into memory. It is possible to pass initialization parameters to the servlet so it may configure itself.

Fourth, the server invokes the **service()** method of the servlet. This method is called to process the HTTP request. You will see that it is possible for the servlet to read data that has been provided in the HTTP request. It may also formulate an HTTP response for the client.

The servlet remains in the server's address space and is available to process any other HTTP requests received from clients. The **service()** method is called for each HTTP request.

Finally, the server may decide to unload the servlet from its memory. The algorithms by which this determination is made are specific to each server. The server calls the `destroy()` method to relinquish any resources such as file handles that are allocated for the servlet. Important data may be saved to a persistent store. The memory allocated for the servlet and its objects can then be garbage collected.

Servlet Development Options

To experiment with servlets, you will need access to a servlet container/server. Two popular ones are Glassfish and Apache Tomcat. Glassfish is an open source project sponsored by Oracle and is provided with the Java EE SDK. It is supported by NetBeans. Apache Tomcat is an open-source product maintained by the Apache Software Foundation. It can also be used by NetBeans. Both Tomcat and Glassfish can also be used with other IDEs, such as Eclipse.

Although IDEs such as NetBeans and Eclipse are very useful and can streamline the creation of servlets, they are not used in this chapter. The way you develop and deploy servlets differs among IDEs, and it is simply not possible for this book to address each environment. Furthermore, many readers will be using the command-line tools rather than an IDE. Therefore, if you are using an IDE, you must refer to the instructions for that environment for information concerning the development and deployment of servlets. For this reason, the instructions given here and elsewhere in this chapter assume that only the command-line tools are employed. Thus, they will work for nearly any reader.

This chapter uses Tomcat in the examples. It provides a simple, yet effective way to experiment with servlets using only the command line tools. It is also widely available in various programming environments. Furthermore, since only command-line tools are used, you don't need to download and install an IDE just to experiment with servlets. Understand, however, that even if you are developing in an environment that uses a different servlet container, the concepts presented here still apply. It is just that the mechanics of preparing a servlet for testing will be slightly different.

REMEMBER The instructions for developing and deploying servlets in this chapter are based on Tomcat and use only command-line tools. If you are using an IDE and/or a different servlet container/server, consult the documentation for your environment.

Using Tomcat

Tomcat contains the class libraries, documentation, and run-time support that you will need to create and test servlets. At the time of this writing, several versions of Tomcat are available. The instructions that follow use 8.5.31. This version of Tomcat is used here because it will work for a very wide range of readers. You can download Tomcat from tomcat.apache.org. You should choose a version appropriate to your environment.

The examples in this chapter assume a 64-bit Windows environment. Assuming that a 64-bit version of Tomcat 8.5.31 was unpacked from the root directly, the default location is

```
C:\apache-tomcat-8.5.31-windows-x64\apache-tomcat-8.5.31\
```

This is the location assumed by the examples in this book. If you load Tomcat in a different location (or use a different version of Tomcat), you will need to make appropriate changes to the examples. You may need to set the environmental variable **JAVA_HOME** to the top-level directory in which the Java Development Kit is installed.

NOTE All of the directories shown in this section assume Tomcat 8.5.31. If you install a different version of Tomcat, then you will need to adjust the directory names and paths to match those used by the version you installed.

Once installed, you start Tomcat by selecting **startup.bat** from the **bin** directly under the **apache-tomcat-8.5.31** directory. To stop Tomcat, execute **shutdown.bat**, also in the **bin** directory.

The classes and interfaces needed to build servlets are contained in **servlet-api.jar**, which is in the following directory:

```
C:\apache-tomcat-8.5.31-windows-x64\apache-tomcat-8.5.31\lib
```

To make **servlet-api.jar** accessible, update your **CLASSPATH** environment variable so that it includes

```
C:\apache-tomcat-8.5.31-windows-x64\apache-tomcat-8.5.31\lib\servlet-api.jar
```

Alternatively, you can specify this file when you compile the servlets. For example, the following command compiles the first servlet example:

```
javac HelloServlet.java -classpath "C:\apache-tomcat-8.5.31-
windows-
x64\apache-tomcat-8.5.31\lib\servlet-api.jar"
```

Once you have compiled a servlet, you must enable Tomcat to find it. For our purposes, this means putting it into a directory under Tomcat's **webapps** directory and entering its name into a **web.xml** file. To keep things simple, the examples in this chapter use the directory and **web.xml** file that Tomcat supplies for its own example servlets. This way, you won't have to create any files or directories just to experiment with the sample servlets. Here is the procedure that you will follow.

First, copy the servlet's class file into the following directory:

```
C:\apache-tomcat-8.5.31-windows-x64\apache-tomcat-
8.5.31\webapps\
examples\WEB-INF\classes
```

Next, add the servlet's name and mapping to the **web.xml** file in the following directory:

```
C:\apache-tomcat-8.5.31-windows-x64\apache-tomcat-
8.5.31\webapps\
examples\WEB-INF
```

For instance, assuming the first example, called **HelloServlet**, you will add the following lines in the section that defines the servlets:

```
<servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>HelloServlet</servlet-class>
</servlet>
```

Next, you will add the following lines to the section that defines the servlet mappings:

```
<servlet-mapping>
  <servlet-name>HelloServlet</servlet-name>
  <url-pattern>/servlets/servlet/HelloServlet</url-pattern>
</servlet-mapping>
```

Follow this same general procedure for all of the examples.

A Simple Servlet

To become familiar with the key servlet concepts, we will begin by building and testing a simple servlet. The basic steps are the following:

1. Create and compile the servlet source code. Then, copy the servlet's class file to the proper directory, and add the servlet's name and mappings to the proper **web.xml** file.
2. Start Tomcat.
3. Start a web browser and request the servlet.

Let us examine each of these steps in detail.

Create and Compile the Servlet Source Code

To begin, create a file named **HelloServlet.java** that contains the following program:

```
import java.io.*;
import javax.servlet.*;

public class HelloServlet extends GenericServlet {

    public void service(ServletRequest request,
                        ServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>Hello!");
        pw.close();
    }
}
```

Let's look closely at this program. First, note that it imports the **javax.servlet** package. This package contains the classes and interfaces required to build servlets. You will learn more about these later in this chapter. Next, the program defines **HelloServlet** as a subclass of **GenericServlet**. The **GenericServlet** class provides functionality that simplifies the creation of a servlet. For example, it provides versions of **init()** and **destroy()**, which may be used as is. You need supply only the **service()** method.

Inside **HelloServlet**, the **service()** method (which is inherited from **GenericServlet**) is overridden. This method handles requests from a client. Notice that the first argument is a **ServletRequest** object. This enables the servlet to read data that is provided via the client request. The second argument is a **ServletResponse** object. This enables the servlet to formulate a response for the client.

The call to **setContentType()** establishes the MIME type of the HTTP response. In this program, the MIME type is `text/html`. This indicates that the browser should interpret the content as HTML source code.

Next, the **getWriter()** method obtains a **PrintWriter**. Anything written to this stream is sent to the client as part of the HTTP response. Then **println()** is used to write some simple HTML source code as the HTTP response.

Compile this source code and place the **HelloServlet.class** file in the proper Tomcat directory as described in the previous section. Also, add **HelloServlet** to the **web.xml** file, as described earlier.

Start Tomcat

Start Tomcat as explained earlier. Tomcat must be running before you try to execute a servlet.

Start a Web Browser and Request the Servlet

Start a web browser and enter the URL shown here:

```
http://localhost:8080/examples/servlets/servlet/HelloServlet
```

Alternatively, you may enter the URL shown here:

```
http://127.0.0.1:8080/examples/servlets/servlet/HelloServlet
```

This can be done because 127.0.0.1 is defined as the IP address of the local machine.

You will observe the output of the servlet in the browser display area. It will contain the string **Hello!** in bold type.

The Servlet API

Two packages contain the classes and interfaces that are required to build the

servlets described in this chapter. These are **javax.servlet** and **javax.servlet.http**. They constitute the core of the Servlet API. Keep in mind that these packages are not part of the Java core packages. Therefore, they are not included with Java SE. Instead, they are provided by Tomcat. They are also provided by Java EE.

The Servlet API has been in a process of ongoing development and enhancement. The servlet specification supported by Tomcat 8.5.31 is version 3.1. (As a point of interest, Tomcat 9 supports servlet specification 4.) This chapter discusses the core of the Servlet API, which will be available to most readers and works with all modern versions of the servlet specification.

The **javax.servlet** Package

The **javax.servlet** package contains a number of interfaces and classes that establish the framework in which servlets operate. The following table summarizes several key interfaces that are provided in this package. The most significant of these is **Servlet**. All servlets must implement this interface or extend a class that implements the interface. The **ServletRequest** and **ServletResponse** interfaces are also very important.

Interface	Description
Servlet	Declares life cycle methods for a servlet.
ServletConfig	Allows servlets to get initialization parameters.
ServletContext	Enables servlets to log events and access information about their environment.
ServletRequest	Used to read data from a client request.
ServletResponse	Used to write data to a client response.

The following table summarizes the core classes that are provided in the **javax.servlet** package:

Class	Description
GenericServlet	Implements the Servlet and ServletConfig interfaces.
ServletInputStream	Encapsulates an input stream for reading requests from a client.
ServletOutputStream	Encapsulates an output stream for writing responses to a client.
ServletException	Indicates a servlet error occurred.
UnavailableException	Indicates a servlet is unavailable.

Let us examine these interfaces and classes in more detail.

The Servlet Interface

All servlets must implement the **Servlet** interface. It declares the **init()**, **service()**, and **destroy()** methods that are called by the server during the life cycle of a servlet. A method is also provided that allows a servlet to obtain any initialization parameters. The methods defined by **Servlet** are shown in [Table 35-1](#).

Method	Description
void destroy()	Called when the servlet is unloaded.
ServletConfig getServletConfig()	Returns a ServletConfig object that contains any initialization parameters.
String getServletInfo()	Returns a string describing the servlet.
void init(ServletConfig sc) throws ServletException	Called when the servlet is initialized. Initialization parameters for the servlet can be obtained from <i>sc</i> . A ServletException should be thrown if the servlet cannot be initialized.
void service(ServletRequest <i>req</i> , ServletResponse <i>res</i>) throws ServletException, IOException	Called to process a request from a client. The request from the client can be read from <i>req</i> . The response to the client can be written to <i>res</i> . An exception is generated if a servlet or IO problem occurs.

Table 35-1 The Methods Defined by **Servlet**

The **init()**, **service()**, and **destroy()** methods are the life cycle methods of the servlet. These are invoked by the server. The **getServletConfig()** method is called by the servlet to obtain initialization parameters. A servlet developer overrides the **getServletInfo()** method to provide a string with useful information (for example, the version number). This method is also invoked by the server.

The ServletConfig Interface

The **ServletConfig** interface allows a servlet to obtain configuration data when it is loaded. The methods declared by this interface are summarized here:

Method	Description
ServletContext getServletContext()	Returns the context for this servlet.
String getInitParameter(String <i>param</i>)	Returns the value of the initialization parameter named <i>param</i> .
Enumeration<String> getInitParameterNames()	Returns an enumeration of all initialization parameter names.
String getServletName()	Returns the name of the invoking servlet.

The ServletContext Interface

The **ServletContext** interface enables servlets to obtain information about their environment. Several of its methods are summarized in [Table 35-2](#).

Method	Description
Object getAttribute(String <i>attr</i>)	Returns the value of the server attribute named <i>attr</i> .
String getMimeType(String <i>file</i>)	Returns the MIME type of <i>file</i> .
String getRealPath(String <i>vpath</i>)	Returns the real (i.e., absolute) path that corresponds to the relative path <i>vpath</i> .
String getServerInfo()	Returns information about the server.
void log(String <i>s</i>)	Writes <i>s</i> to the servlet log.
void log(String <i>s</i> , Throwable <i>e</i>)	Writes <i>s</i> and the stack trace for <i>e</i> to the servlet log.
void setAttribute(String <i>attr</i> , Object <i>val</i>)	Sets the attribute specified by <i>attr</i> to the value passed in <i>val</i> .

Table 35-2 Various Methods Defined by **ServletContext**

The ServletRequest Interface

The **ServletRequest** interface enables a servlet to obtain information about a client request. Several of its methods are summarized in [Table 35-3](#).

Method	Description
Object getAttribute(String <i>attr</i>)	Returns the value of the attribute named <i>attr</i> .
String getCharacterEncoding()	Returns the character encoding of the request.
int getContentLength()	Returns the size of the request. The value -1 is returned if the size is unavailable.
String getContentType()	Returns the type of the request. A null value is returned if the type cannot be determined.
ServletInputStream getInputStream() throws IOException	Returns a ServletInputStream that can be used to read binary data from the request. An IllegalStateException is thrown if getReader() has been previously invoked on this object.
String getParameter(String <i>pname</i>)	Returns the value of the parameter named <i>pname</i> .
Enumeration<String> getParameterNames()	Returns an enumeration of the parameter names for this request.
String[] getParameterValues(String <i>name</i>)	Returns an array containing values associated with the parameter specified by <i>name</i> .
String getProtocol()	Returns a description of the protocol.
BufferedReader getReader() throws IOException	Returns a buffered reader that can be used to read text from the request. An IllegalStateException is thrown if getInputStream() has been previously invoked on this object.
String getRemoteAddr()	Returns the string equivalent of the client IP address.
String getRemoteHost()	Returns the string equivalent of the client host name.
String getScheme()	Returns the transmission scheme of the URL used for the request (for example, "http", "ftp").
String getServerName()	Returns the name of the server.
int getServerPort()	Returns the port number.

Table 35-3 Various Methods Defined by **ServletRequest**

The **ServletResponse** Interface

The **ServletResponse** interface enables a servlet to formulate a response for a client. Several of its methods are summarized in [Table 35-4](#).

Method	Description
<code>String getCharacterEncoding()</code>	Returns the character encoding for the response.
<code>ServletOutputStream getOutputStream()</code> <code>throws IOException</code>	Returns a <code>ServletOutputStream</code> that can be used to write binary data to the response. An <code>IllegalStateException</code> is thrown if <code>getWriter()</code> has been previously invoked on this object.
<code>PrintWriter getWriter()</code> <code>throws IOException</code>	Returns a <code>PrintWriter</code> that can be used to write character data to the response. An <code>IllegalStateException</code> is thrown if <code>getOutputStream()</code> has been previously invoked on this object.
<code>void setContentLength(int size)</code>	Sets the content length for the response to <code>size</code> .
<code>void setContentType(String type)</code>	Sets the content type for the response to <code>type</code> .

Table 35-4 Various Methods Defined by `ServletResponse`

The GenericServlet Class

The **GenericServlet** class provides implementations of the basic life cycle methods for a servlet. **GenericServlet** implements the **Servlet** and **ServletConfig** interfaces. In addition, a method to append a string to the server log file is available. The signatures of this method are shown here:

```
void log(String s)
void log(String s, Throwable e)
```

Here, `s` is the string to be appended to the log, and `e` is an exception that occurred.

The ServletInputStream Class

The **ServletInputStream** class extends **InputStream**. It is implemented by the servlet container and provides an input stream that a servlet developer can use to read the data from a client request. In addition to the input methods inherited from **InputStream**, a method is provided to read bytes from the stream. It is shown here:

```
int readLine(byte[ ] buffer, int offset, int size) throws IOException
```

Here, `buffer` is the array into which `size` bytes are placed starting at `offset`. The method returns the actual number of bytes read or `-1` if an end-of-stream condition is encountered.

The **ServletOutputStream** Class

The **ServletOutputStream** class extends **OutputStream**. It is implemented by the servlet container and provides an output stream that a servlet developer can use to write data to a client response. In addition to the output methods provided by **OutputStream**, it also defines the **print()** and **println()** methods, which output data to the stream.

The Servlet Exception Classes

javax.servlet defines two exceptions. The first is **ServletException**, which indicates that a servlet problem has occurred. The second is **UnavailableException**, which extends **ServletException**. It indicates that a servlet is unavailable.

Reading Servlet Parameters

The **ServletRequest** interface includes methods that allow you to read the names and values of parameters that are included in a client request. We will develop a servlet that illustrates their use. The example contains two files. A web page is defined in **PostParameters.html**, and a servlet is defined in **PostParametersServlet.java**.

The HTML source code for **PostParameters.html** is shown in the following listing. It defines a table that contains two labels and two text fields. One of the labels is Employee and the other is Phone. There is also a submit button. Notice that the action parameter of the form tag specifies a URL. The URL identifies the servlet to process the HTTP POST request.

```
<html>
<body>
<center>
<form name="Form1"
      method="post"
      action="http://localhost:8080/examples/servlets/
              servlet/PostParametersServlet">
<table>
<tr>
  <td><B>Employee</td>
  <td><input type= textbox name="e" size="25" value=""></td>
</tr>
<tr>
  <td><B>Phone</td>
  <td><input type= textbox name="p" size="25" value=""></td>
</tr>
</table>
<input type=submit value="Submit">
</body>
</html>
```

The source code for **PostParametersServlet.java** is shown in the following listing. The **service()** method is overridden to process client requests. The **getParameterNames()** method returns an enumeration of the parameter names. These are processed in a loop. You can see that the parameter name and value are output to the client. The parameter value is obtained via the **getParameter()** method.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;

public class PostParametersServlet
extends GenericServlet {

    public void service(ServletRequest request,
                        ServletResponse response)
    throws ServletException, IOException {

        // Get print writer.
        PrintWriter pw = response.getWriter();

        // Get enumeration of parameter names.
        Enumeration<String> e = request.getParameterNames();

        // Display parameter names and values.
        while(e.hasMoreElements()) {
            String pname = e.nextElement();
            pw.print(pname + " = ");
            String pvalue = request.getParameter(pname);
            pw.println(pvalue);
        }
        pw.close();
    }
}
```

Compile the servlet. Next, copy it to the appropriate directory, and update the **web.xml** file, as previously described. Then, perform these steps to test this example:

1. Start Tomcat (if it is not already running).
2. Display the web page in a browser.
3. Enter an employee name and phone number in the text fields.
4. Submit the web page.

After following these steps, the browser will display a response that is dynamically generated by the servlet.

The javax.servlet.http Package

The preceding examples have used the classes and interfaces defined in **javax.servlet**, such as **ServletRequest**, **ServletResponse**, and **GenericServlet**, to illustrate the basic functionality of servlets. However, when working with HTTP, you will normally use the interfaces and classes in **javax.servlet.http**. As you will see, its functionality makes it easy to build servlets that work with HTTP requests and responses.

The following table summarizes the interfaces used in this chapter:

Interface	Description
HttpServletRequest	Enables servlets to read data from an HTTP request.
HttpServletResponse	Enables servlets to write data to an HTTP response.
HttpSession	Allows session data to be read and written.

The following table summarizes the classes used in this chapter. The most important of these is **HttpServlet**. Servlet developers typically extend this class in order to process HTTP requests.

Class	Description
Cookie	Allows state information to be stored on a client machine.
HttpServlet	Provides methods to handle HTTP requests and responses.

The HttpServletRequest Interface

The **HttpServletRequest** interface enables a servlet to obtain information about a client request. Several of its methods are shown in [Table 35-5](#).

Method	Description
<code>String getAuthType()</code>	Returns authentication scheme.
<code>Cookie[] getCookies()</code>	Returns an array of the cookies in this request.
<code>long getDateHeader(String field)</code>	Returns the value of the date header field named <i>field</i> .
<code>String getHeader(String field)</code>	Returns the value of the header field named <i>field</i> .
<code>Enumeration<String> getHeaderNames()</code>	Returns an enumeration of the header names.
<code>int getIntHeader(String field)</code>	Returns the <code>int</code> equivalent of the header field named <i>field</i> .
<code>String getMethod()</code>	Returns the HTTP method for this request.
<code>String getPathInfo()</code>	Returns any path information that is located after the servlet path and before a query string of the URL.
<code>String getPathTranslated()</code>	Returns any path information that is located after the servlet path and before a query string of the URL after translating it to a real path.
<code>String getQueryString()</code>	Returns any query string in the URL.
<code>String getRemoteUser()</code>	Returns the name of the user who issued this request.
<code>String getRequestedSessionId()</code>	Returns the ID of the session.
<code>String getRequestURI()</code>	Returns the URI.
<code>StringBuffer getRequestURL()</code>	Returns the URL.
<code>String getServletPath()</code>	Returns that part of the URL that identifies the servlet.
<code>HttpSession getSession()</code>	Returns the session for this request. If a session does not exist, one is created and then returned.
<code>HttpSession getSession(boolean new)</code>	If <i>new</i> is <code>true</code> and no session exists, creates and returns a session for this request. Otherwise, returns the existing session for this request.
<code>boolean isRequestedSessionIdFromCookie()</code>	Returns <code>true</code> if a cookie contains the session ID. Otherwise, returns <code>false</code> .
<code>boolean isRequestedSessionIdFromURL()</code>	Returns <code>true</code> if the URL contains the session ID. Otherwise, returns <code>false</code> .
<code>boolean isRequestedSessionIdValid()</code>	Returns <code>true</code> if the requested session ID is valid in the current session context.

Table 35-5 Various Methods Defined by `HttpServletRequest`

The `HttpServletResponse` Interface

The `HttpServletResponse` interface enables a servlet to formulate an HTTP response to a client. Several constants are defined. These correspond to the

different status codes that can be assigned to an HTTP response. For example, **SC_OK** indicates that the HTTP request succeeded, and **SC_NOT_FOUND** indicates that the requested resource is not available. Several methods of this interface are summarized in [Table 35-6](#).

Method	Description
<code>void addCookie(Cookie cookie)</code>	Adds <i>cookie</i> to the HTTP response.
<code>boolean containsHeader(String field)</code>	Returns true if the HTTP response header contains a field named <i>field</i> .
<code>String encodeURL(String url)</code>	Determines if the session ID must be encoded in the URL identified as <i>url</i> . If so, returns the modified version of <i>url</i> . Otherwise, returns <i>url</i> . All URLs generated by a servlet should be processed by this method.
<code>String encodeRedirectURL(String url)</code>	Determines if the session ID must be encoded in the URL identified as <i>url</i> . If so, returns the modified version of <i>url</i> . Otherwise, returns <i>url</i> . All URLs passed to <code>sendRedirect()</code> should be processed by this method.
<code>void sendError(int c) throws IOException</code>	Sends the error code <i>c</i> to the client.
<code>void sendError(int c, String s) throws IOException</code>	Sends the error code <i>c</i> and message <i>s</i> to the client.
<code>void sendRedirect(String url) throws IOException</code>	Redirects the client to <i>url</i> .
<code>void setDateHeader(String field, long msec)</code>	Adds <i>field</i> to the header with date value equal to <i>msec</i> (milliseconds since midnight, January 1, 1970, GMT).
<code>void setHeader(String field, String value)</code>	Adds <i>field</i> to the header with value equal to <i>value</i> .
<code>void setIntHeader(String field, int value)</code>	Adds <i>field</i> to the header with value equal to <i>value</i> .
<code>void setStatus(int code)</code>	Sets the status code for this response to <i>code</i> .

Table 35-6 Various Methods Defined by `HttpServletResponse`

The HttpSession Interface

The **HttpSession** interface enables a servlet to read and write the state information that is associated with an HTTP session. Several of its methods are summarized in [Table 35-7](#). All of these methods throw an **IllegalStateException** if the session has already been invalidated.

Method	Description
Object getAttribute(String <i>attr</i>)	Returns the value associated with the name passed in <i>attr</i> . Returns null if <i>attr</i> is not found.
Enumeration<String> getAttributeNames()	Returns an enumeration of the attribute names associated with the session.
long getCreationTime()	Returns the creation time (in milliseconds since midnight, January 1, 1970, GMT) of the invoking session.
String getId()	Returns the session ID.
long getLastAccessedTime()	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the client last made a request on the invoking session.
void invalidate()	Invalidates this session and removes it from the context.
boolean isNew()	Returns true if the server created the session and it has not yet been accessed by the client.
void removeAttribute(String <i>attr</i>)	Removes the attribute specified by <i>attr</i> from the session.
void setAttribute(String <i>attr</i> , Object <i>val</i>)	Associates the value passed in <i>val</i> with the attribute name passed in <i>attr</i> .

Table 35-7 Various Methods Defined by **HttpSession**

The Cookie Class

The **Cookie** class encapsulates a cookie. A *cookie* is stored on a client and contains state information. Cookies are valuable for tracking user activities. For example, assume that a user visits an online store. A cookie can save the user's name, address, and other information. The user does not need to enter this data each time he or she visits the store.

A servlet can write a cookie to a user's machine via the **addCookie()** method of the **HttpServletResponse** interface. The data for that cookie is then included in the header of the HTTP response that is sent to the browser.

The names and values of cookies are stored on the user's machine. Some of the information that can be saved for each cookie includes the following:

- The name of the cookie
- The value of the cookie
- The expiration date of the cookie
- The domain and path of the cookie

The expiration date determines when this cookie is deleted from the user's

machine. If an expiration date is not explicitly assigned to a cookie, it is deleted when the current browser session ends.

The domain and path of the cookie determine when it is included in the header of an HTTP request. If the user enters a URL whose domain and path match these values, the cookie is then supplied to the web server. Otherwise, it is not.

There is one constructor for **Cookie**. It has the signature shown here:

`Cookie(String name, String value)`

Here, the name and value of the cookie are supplied as arguments to the constructor. The methods of the **Cookie** class are summarized in [Table 35-8](#).

Method	Description
<code>Object clone()</code>	Returns a copy of this object.
<code>String getComment()</code>	Returns the comment.
<code>String getDomain()</code>	Returns the domain.
<code>int getMaxAge()</code>	Returns the maximum age (in seconds).
<code>String getName()</code>	Returns the name.
<code>String getPath()</code>	Returns the path.
<code>boolean getSecure()</code>	Returns true if the cookie is secure. Otherwise, returns false .
<code>String getValue()</code>	Returns the value.
<code>int getVersion()</code>	Returns the version.
<code>boolean isHttpOnly()</code>	Returns true if the cookie has the HttpOnly attribute.
<code>void setComment(String c)</code>	Sets the comment to <i>c</i> .
<code>void setDomain(String d)</code>	Sets the domain to <i>d</i> .
<code>void setHttpOnly(boolean httpOnly)</code>	If <i>httpOnly</i> is true , then the HttpOnly attribute is added to the cookie. If <i>httpOnly</i> is false , the HttpOnly attribute is removed.
<code>void setMaxAge(int secs)</code>	Sets the maximum age of the cookie to <i>secs</i> . This is the number of seconds after which the cookie is deleted.
<code>void setPath(String p)</code>	Sets the path to <i>p</i> .
<code>void setSecure(boolean secure)</code>	Sets the security flag to <i>secure</i> .
<code>void setValue(String v)</code>	Sets the value to <i>v</i> .
<code>void setVersion(int v)</code>	Sets the version to <i>v</i> .

Table 35-8 The Methods Defined by **Cookie**

The HttpServlet Class

The **HttpServlet** class extends **GenericServlet**. It is commonly used when developing servlets that receive and process HTTP requests. The methods defined by the **HttpServlet** class are summarized in [Table 35-9](#).

Method	Description
<code>void doDelete(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</code>	Handles an HTTP DELETE request.
<code>void doGet(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</code>	Handles an HTTP GET request.
<code>void doHead(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</code>	Handles an HTTP HEAD request.
<code>void doOptions(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</code>	Handles an HTTP OPTIONS request.
<code>void doPost(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</code>	Handles an HTTP POST request.
<code>void doPut(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</code>	Handles an HTTP PUT request.
<code>void doTrace(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</code>	Handles an HTTP TRACE request.
<code>long getLastModified(HttpServletRequest req)</code>	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the requested resource was last modified.
<code>void service(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</code>	Called by the server when an HTTP request arrives for this servlet. The arguments provide access to the HTTP request and response, respectively.

Table 35-9 The Methods Defined by **HttpServlet**

Handling HTTP Requests and Responses

The **HttpServlet** class provides specialized methods that handle the various

types of HTTP requests. A servlet developer typically overrides one of these methods. These methods are **doDelete()**, **doGet()**, **doHead()**, **doOptions()**, **doPost()**, **doPut()**, and **doTrace()**. A complete description of the different types of HTTP requests is beyond the scope of this book. However, the GET and POST requests are commonly used when handling form input. Therefore, this section presents examples of these cases.

Handling HTTP GET Requests

Here we will develop a servlet that handles an HTTP GET request. The servlet is invoked when a form on a web page is submitted. The example contains two files. A web page is defined in **ColorGet.html**, and a servlet is defined in **ColorGetServlet.java**. The HTML source code for **ColorGet.html** is shown in the following listing. It defines a form that contains a select element and a submit button. Notice that the action parameter of the form tag specifies a URL. The URL identifies a servlet to process the HTTP GET request.

```
<html>
<body>
<center>
<form name="Form1"
      action="http://localhost:8080/examples/servlets/servlet/ColorGetServlet">
<B>Color:</B>
<select name="color" size="1">
<option value="Red">Red</option>
<option value="Green">Green</option>

<option value="Blue">Blue</option>
</select>
<br><br>
<input type=submit value="Submit">
</form>
</body>
</html>
```

The source code for **ColorGetServlet.java** is shown in the following listing. The **doGet()** method is overridden to process any HTTP GET requests that are sent to this servlet. It uses the **getParameter()** method of **HttpServletRequest** to obtain the selection that was made by the user. A response is then formulated.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ColorGetServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {

        String color = request.getParameter("color");
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>The selected color is: ");
        pw.println(color);
        pw.close();
    }
}
```

Compile the servlet. Next, copy it to the appropriate directory, and update the **web.xml** file, as previously described. Then, perform these steps to test this example:

1. Start Tomcat, if it is not already running.
2. Display the web page in a browser.
3. Select a color.
4. Submit the web page.

After completing these steps, the browser will display the response that is dynamically generated by the servlet.

One other point: Parameters for an HTTP GET request are included as part of the URL that is sent to the web server. Assume that the user selects the red option and submits the form. The URL sent from the browser to the server is

`http://localhost:8080/examples/servlets/servlet/ColorGetServlet?
color=Red`

The characters to the right of the question mark are known as the *query string*.

Handling HTTP POST Requests

Here we will develop a servlet that handles an HTTP POST request. The servlet is invoked when a form on a web page is submitted. The example contains two files. A web page is defined in **ColorPost.html**, and a servlet is defined in **ColorPostServlet.java**.

The HTML source code for **ColorPost.html** is shown in the following listing. It is identical to **ColorGet.html** except that the method parameter for the form tag explicitly specifies that the POST method should be used, and the action parameter for the form tag specifies a different servlet.

```
<html>
<body>
<center>
<form name="Form1"
      method="post"
      action="http://localhost:8080/examples/servlets/servlet/ColorPostServlet">
<B>Color:</B>
<select name="color" size="1">
<option value="Red">Red</option>
<option value="Green">Green</option>
<option value="Blue">Blue</option>
</select>
<br><br>
<input type=submit value="Submit">
</form>
</body>
</html>
```

The source code for **ColorPostServlet.java** is shown in the following listing. The **doPost()** method is overridden to process any HTTP POST requests that are sent to this servlet. It uses the **getParameter()** method of **HttpServletRequest** to obtain the selection that was made by the user. A response is then formulated.

```

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ColorPostServlet extends HttpServlet {

    public void doPost(HttpServletRequest request,
                        HttpServletResponse response)
        throws ServletException, IOException {

        String color = request.getParameter("color");
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>The selected color is: ");
        pw.println(color);
        pw.close();
    }
}

```

Compile the servlet and perform the same steps as described in the previous section to test it.

NOTE Parameters for an HTTP POST request are not included as part of the URL that is sent to the web server. In this example, the URL sent from the browser to the server is <http://localhost:8080/examples/servlets/servlet/ColorPostServlet>. The parameter names and values are sent in the body of the HTTP request.

Using Cookies

Now, let's develop a servlet that illustrates how to use cookies. The servlet is invoked when a form on a web page is submitted. The example contains three files as summarized here:

File	Description
AddCookie.html	Allows a user to specify a value for the cookie named MyCookie .
AddCookieServlet.java	Processes the submission of AddCookie.html .
GetCookiesServlet.java	Displays cookie values.

The HTML source code for **AddCookie.html** is shown in the following

listing. This page contains a text field in which a value can be entered. There is also a submit button on the page. When this button is pressed, the value in the text field is sent to **AddCookieServlet** via an HTTP POST request.

```
<html>
<body>
<center>
<form name="Form1"
      method="post"
      action="http://localhost:8080/examples/servlets/servlet/AddCookieServlet">
<B>Enter a value for MyCookie:</B>
<input type= textbox name="data" size=25 value="">
<input type=submit value="Submit">
</form>
</body>
</html>
```

The source code for **AddCookieServlet.java** is shown in the following listing. It gets the value of the parameter named "data". It then creates a **Cookie** object that has the name "MyCookie" and contains the value of the "data" parameter. The cookie is then added to the header of the HTTP response via the **addCookie()** method. A feedback message is then written to the browser.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class AddCookieServlet extends HttpServlet {

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
```

```
// Get parameter from HTTP request.  
String data = request.getParameter("data");  
  
// Create cookie.  
Cookie cookie = new Cookie("MyCookie", data);  
  
// Add cookie to HTTP response.  
response.addCookie(cookie);  
  
// Write output to browser.  
response.setContentType("text/html");  
PrintWriter pw = response.getWriter();  
pw.println("<B>MyCookie has been set to");  
pw.println(data);  
pw.close();  
}  
}
```

The source code for **GetCookiesServlet.java** is shown in the following listing. It invokes the **getCookies()** method to read any cookies that are included in the HTTP GET request. The names and values of these cookies are then written to the HTTP response. Observe that the **getName()** and **getValue()** methods are called to obtain this information.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class GetCookiesServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {

        // Get cookies from header of HTTP request.
        Cookie[] cookies = request.getCookies();

        // Display these cookies.
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>");
        for(int i = 0; i < cookies.length; i++) {
            String name = cookies[i].getName();
            String value = cookies[i].getValue();
            pw.println("name = " + name +
                       "; value = " + value);
        }
        pw.close();
    }
}
```

Compile the servlets. Next, copy them to the appropriate directory, and update the **web.xml** file, as previously described. Then, perform these steps to test this example:

1. Start Tomcat, if it is not already running.
2. Display **AddCookie.html** in a browser.
3. Enter a value for **MyCookie**.
4. Submit the web page.

After completing these steps, you will observe that a feedback message is displayed by the browser.

Next, request the following URL via the browser:

`http://localhost:8080/examples/servlets/servlet/GetCookiesServlet`

Observe that the name and value of the cookie are displayed in the browser.

In this example, an expiration date is not explicitly assigned to the cookie via the `setMaxAge()` method of **Cookie**. Therefore, the cookie expires when the browser session ends. You can experiment by using `setMaxAge()` and observe that the cookie is then saved on the client machine.

Session Tracking

HTTP is a stateless protocol. Each request is independent of the previous one. However, in some applications, it is necessary to save state information so that information can be collected from several interactions between a browser and a server. Sessions provide such a mechanism.

A session can be created via the `getSession()` method of **HttpServletRequest**. An **HttpSession** object is returned. This object can store a set of bindings that associate names with objects. The `setAttribute()`, `getAttribute()`, `getAttributeNames()`, and `removeAttribute()` methods of **HttpSession** manage these bindings. Session state is shared by all servlets that are associated with a client.

The following servlet illustrates how to use session state. The `getSession()` method gets the current session. A new session is created if one does not already exist. The `getAttribute()` method is called to obtain the object that is bound to the name "date". That object is a **Date** object that encapsulates the date and time when this page was last accessed. (Of course, there is no such binding when the page is first accessed.) A **Date** object encapsulating the current date and time is then created. The `setAttribute()` method is called to bind the name "date" to this object.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DateServlet extends HttpServlet {
```

```
public void doGet(HttpServletRequest request,
    HttpServletResponse response)
throws ServletException, IOException {

    // Get the HttpSession object.
    HttpSession hs = request.getSession(true);

    // Get writer.
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.print("<B>");

    // Display date/time of last access.
    Date date = (Date)hs.getAttribute("date");
    if(date != null) {
        pw.print("Last access: " + date + "<br>");
    }

    // Display current date/time.
    date = new Date();
    hs.setAttribute("date", date);
    pw.println("Current date: " + date);
}
}
```

When you first request this servlet, the browser displays one line with the current date and time information. On subsequent invocations, two lines are displayed. The first line shows the date and time when the servlet was last accessed. The second line shows the current date and time.

PART

Appendixes

APPENDIX A

Using Java's Documentation Comments

APPENDIX B

Introducing JShell

APPENDIX C

Compile and Run Simple Single-File Programs in One Step

APPENDIX

Using Java's Documentation Comments

As explained in [Part I](#), Java supports three types of comments. The first two are the `//` and the `/* */`. The third type is called a *documentation comment*. It begins with the character sequence `/**`. It ends with `*/`. Documentation comments allow you to embed information about your program into the program itself. You can then use the **javadoc** utility program (supplied with the JDK) to extract the information and put it into an HTML file. Documentation comments make it convenient to document your programs. You have almost certainly seen documentation that uses such comments because that is the way the Java API library was documented. Beginning with JDK 9, **javadoc** includes support for modules.

The **javadoc** Tags

The **javadoc** utility recognizes several tags, including those shown here:

Tag	Meaning
<code>@author</code>	Identifies the author.
<code>{@code}</code>	Displays information as-is, without processing HTML styles, in code font.
<code>@deprecated</code>	Specifies that a program element is deprecated.
<code>{@docRoot}</code>	Specifies the path to the root directory of the current documentation.
<code>@exception</code>	Identifies an exception thrown by a method or constructor.
<code>@hidden</code>	Prevents an element from appearing in the documentation.
<code>{@index}</code>	Specifies a term for indexing.
<code>{@inheritDoc}</code>	Inherits a comment from the immediate superclass.
<code>{@link}</code>	Inserts an in-line link to another topic.
<code>{@linkplain}</code>	Inserts an in-line link to another topic, but the link is displayed in a plain-text font.
<code>{@literal}</code>	Displays information as-is, without processing HTML styles.
<code>@param</code>	Documents a parameter.

Tag	Meaning
@provides	Documents a service provided by a module.
@return	Documents a method's return value.
@see	Specifies a link to another topic.
@serial	Documents a default serializable field.
@serialData	Documents the data written by the <code>writeObject()</code> or <code>writeExternal()</code> methods.
@serialField	Documents an ObjectStreamField component.
@since	States the release when a specific change was introduced.
{@summary}	Documents a summary of an item. (Added by JDK 10.)
@throws	Same as <code>@exception</code> .
@uses	Documents a service needed by a module.
{@value}	Displays the value of a constant, which must be a static field.
@version	Specifies the version of a program element.

Document tags that begin with an “at” sign (@) are called *stand-alone* tags (also called *block* tags), and they must be used on their own line. Tags that begin with a brace, such as {@code}, are called *in-line* tags, and they can be used within a larger description. You may also use other, standard HTML tags in a documentation comment. However, some tags, such as headings, should not be used because they disrupt the look of the HTML file produced by **javadoc**.

As it relates to documenting source code, you can use documentation comments to document classes, interfaces, fields, constructors, methods, and modules. In all cases, the documentation comment must immediately precede the item being documented. Some tags, such as `@see`, `@since`, and `@deprecated`, can be used to document any element. Other tags apply only to the relevant elements. Each tag is examined next.

NOTE Documentation comments can also be used for documenting a package and preparing an overview, but the procedures differ from those used to document source code. See the **javadoc** documentation for details on these uses. Beginning with JDK 9, **javadoc** can also document a **module-info.java** file.

@author

The **@author** tag documents the author of a program element. It has the following syntax:

`@author description`

Here, *description* will usually be the name of the author. You will need to specify the **-author** option when executing **javadoc** in order for the `@author` field to be included in the HTML documentation.

`{@code}`

The `{@code}` tag enables you to embed text, such as a snippet of code, into a comment. That text is then displayed as-is in code font, without any further processing, such as HTML rendering. It has the following syntax:

`{@code code-snippet}`

`@deprecated`

The `@deprecated` tag specifies that a program element is deprecated. It is recommended that you include `@see` or `{@link}` tags to inform the programmer about available alternatives. The syntax is the following:

`@deprecated description`

Here, *description* is the message that describes the deprecation. The `@deprecated` tag can be used in documentation for fields, methods, constructors, classes, modules, and interfaces.

`{@docRoot}`

`{@docRoot}` specifies the path to the root directory of the current documentation.

`@exception`

The `@exception` tag describes an exception to a method. It has the following syntax:

`@exception exception-name explanation`

Here, the fully qualified name of the exception is specified by *exception-name*, and *explanation* is a string that describes how the exception can occur. The

@exception tag can only be used in documentation for a method or constructor.

@hidden

The **@hidden** tag prevents an element from appearing in the documentation.

{@index}

The **{@index}** tag specifies an item that will be indexed, and thus found when using the search feature. It has the following syntax:

```
{ @index term usage-str }
```

Here, *term* is the item (which can be a quoted string) to be indexed. *usage-str* is optional. Thus, in the following **@exception** tag, **{@index}** causes the term “error” to be added to the index:

```
@exception IOException On input {@index error}.
```

Note that the word “error” is still displayed as part of the description. It’s just that now it is also indexed. If you include the optional *usage-str*, then that description will be shown in the index and in the search box to indicate how the term is used. For example, **{@index error Serious execution failure}** will show “Serious execution failure” under “error” in the index and in the search box.

{@inheritDoc}

This tag inherits a comment from the immediate superclass.

{@link}

The **{@link}** tag provides an in-line link to additional information. It has the following syntax:

```
{@link pkg.class#member text}
```

Here, *pkg.class#member* specifies the name of a class or method to which a link is added, and *text* is the string that is displayed.

{@linkplain}

Inserts an in-line link to another topic. The link is displayed in plain-text font. Otherwise, it is similar to **{@link}**.

{@literal}

The **{@literal}** tag enables you to embed text into a comment. That text is then displayed as-is, without any further processing, such as HTML rendering. It has the following syntax:

`{@literal description}`

Here, *description* is the text that is embedded.

@param

The **@param** tag documents a parameter. It has the following syntax:

`@param parameter-name explanation`

Here, *parameter-name* specifies the name of a parameter. The meaning of that parameter is described by *explanation*. The **@param** tag can be used only in documentation for a method or constructor, or a generic class or interface.

@provides

The **@provides** tag documents a service provided by a module. It has the following syntax:

`@provides type explanation`

Here, *type* specifies a service provider type and *explanation* describes the service provider.

@return

The **@return** tag describes the return value of a method. It has the following syntax:

`@return explanation`

Here, *explanation* describes the type and meaning of the value returned by a method. The **@return** tag can be used only in documentation for a method.

@see

The **@see** tag provides a reference to additional information. Two commonly used forms are shown here:

```
@see anchor  
@see pkg.class#member text
```

In the first form, *anchor* is a link to an absolute or relative URL. In the second form, *pkg.class#member* specifies the name of the item, and *text* is the text displayed for that item. The *text* parameter is optional, and if not used, then the item specified by *pkg.class#member* is displayed. The member name, too, is optional. Thus, you can specify a reference to a package, class, or interface in addition to a reference to a specific method or field. The name can be fully qualified or partially qualified. However, the dot that precedes the member name (if it exists) must be replaced by a hash character.

@serial

The **@serial** tag defines the comment for a default serializable field. It has the following syntax:

```
@serial description
```

Here, *description* is the comment for that field.

@serialData

The **@serialData** tag documents the data written by the **writeObject()** and **writeExternal()** methods. It has the following syntax:

```
@serialData description
```

Here, *description* is the comment for that data.

@serialField

For a class that implements **Serializable**, the **@serialField** tag provides comments for an **ObjectStreamField** component. It has the following syntax:

@serialField *name type description*

Here, *name* is the name of the field, *type* is its type, and *description* is the comment for that field.

@since

The **@since** tag states that an element was introduced in a specific release. It has the following syntax:

@since *release*

Here, *release* is a string that designates the release or version in which this feature became available.

{@summary}

The **{@summary}** tag explicitly specifies the summary text that will be used for an item. It must be the first tag in the documentation for the item. It has the following syntax:

@summary *explanation*

Here, *explanation* provides a summary of the tagged item, which can span multiple lines. This tag was added by JDK 10. Without the use of **{@summary}**, the first line in an item's documentation comment is used as the summary.

@throws

The **@throws** tag has the same meaning as the **@exception** tag.

@uses

The **@uses** tag documents a service provider needed by a module. It has the following syntax:

`@uses type explanation`

Here, `type` specifies a service provider type and `explanation` describes the service.

{@value}

`{@value}` has two forms. The first displays the value of the constant that it precedes, which must be a **static** field. It has this form:

`{@value}`

The second form displays the value of a specified **static** field. It has this form:

`{@value pkg.class#field}`

Here, `pkg.class#field` specifies the name of the **static** field.

@version

The **@version** tag specifies the version of a program element. It has the following syntax:

`@version info`

Here, `info` is a string that contains version information, typically a version number, such as 2.2. You will need to specify the **-version** option when executing **javadoc** in order for the **@version** field to be included in the HTML documentation.

The General Form of a Documentation Comment

After the beginning `/**`, the first line or lines become the main description of your class, interface, field, constructor, method, or module. After that, you can include one or more of the various `@` tags. Each `@` tag must start at the beginning of a new line or follow one or more asterisks (*) that are at the start of a line. Multiple tags of the same type should be grouped together. For example, if you have three `@see` tags, put them one after the other. In-line tags

(those that begin with a brace) can be used within any description.

Here is an example of a documentation comment for a class:

```
/**  
 * This class draws a bar chart.  
 * @author Herbert Schildt  
 * @version 3.2  
 */
```

What javadoc Outputs

The **javadoc** program takes as input your Java program's source file and outputs several HTML files that contain the program's documentation. Information about each class will be in its own HTML file. **javadoc** will also output an index and a hierarchy tree. Other HTML files can be generated. Beginning with JDK 9, a search box feature is also included.

An Example that Uses Documentation Comments

Following is a sample program that uses documentation comments. Notice the way each comment immediately precedes the item that it describes. After being processed by **javadoc**, the documentation about the **SquareNum** class will be found in **SquareNum.html**.

```
import java.io.*;
/**
 * This class demonstrates documentation comments.
 * @author Herbert Schildt
 * @version 1.2
 */
public class SquareNum {
    /**
     * This method returns the square of num.
     * This is a multiline description. You can use
     * as many lines as you like.
     * @param num The value to be squared.
     * @return num squared.
    */
    public double square(double num) {
        return num * num;
    }

    /**
     * This method inputs a number from the user.
     * @return The value input as a double.
     * @exception IOException On input error.
     * @see IOException
    */
    public double getNumber() throws IOException {
        // create a BufferedReader using System.in
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader inData = new BufferedReader(isr);
        String str;
```

```
    str = inData.readLine();
    return (new Double(str)).doubleValue();
}
/** 
 * This method demonstrates square().
 * @param args Unused.
 * @exception IOException On input error.
 * @see IOException
*/
public static void main(String args[])
    throws IOException
{
    SquareNum ob = new SquareNum();
    double val;

    System.out.println("Enter value to be squared: ");
    val = ob.getNumber();
    val = ob.square(val);

    System.out.println("Squared value is " + val);
}
}
```

APPENDIX

Introducing JShell

Beginning with JDK 9, Java has included a tool called JShell. It provides an interactive environment that enables you to quickly and easily experiment with Java code. JShell implements what is referred to as *read-evaluate-print loop* (REPL) execution. Using this mechanism, you are prompted to enter a fragment of code. This fragment is then read and evaluated. Next, JShell displays output related to the code, such as the output produced by a `println()` statement, the result of an expression, or the current value of a variable. JShell then prompts for the next piece of code, and the process continues (i.e., loops). In the language of JShell, each code sequence you enter is called a *snippet*.

A key point to understand about JShell is that you do not need to enter a complete Java program to use it. Each snippet you enter is simply evaluated as you enter it. This is possible because JShell handles many of the details associated with a Java program for you automatically. This lets you concentrate on a specific feature without having to write a complete program, which makes JShell especially helpful when you are first learning Java.

As you might expect, JShell can also be useful to experienced programmers. Because JShell stores state information, it is possible to enter multiline code sequences and run them inside JShell. This makes JShell quite useful when you need to prototype a concept because it lets you interactively experiment with your code without having to develop and compile a complete program.

This appendix introduces JShell and explores several of its key features, with the primary focus being on those features most useful to beginning Java programmers.

JShell Basics

JShell is a command-line tool. Thus, it runs in a command-prompt window. To start a JShell session, execute `jshell` from the command line. After doing so, you will see the JShell prompt:

```
jshell>
```

When this prompt is displayed you can enter a code snippet or a JShell

When the prompt is displayed, you can enter a code snippet or a command.

In its simplest form, JShell lets you enter an individual statement and immediately see the result. To begin, think back to the first example Java program in this book. It is shown again here:

```
class Example {  
    // Your program begins with a call to main().  
    public static void main(String args[]) {  
        System.out.println("This is a simple Java program.");  
    }  
}
```

In this program, only the **println()** statement actually performs an action, which is displaying its message on the screen. The rest of the code simply provides the required class and method declarations. In JShell, it is not necessary to explicitly specify the class or method in order to execute the **println()** statement. JShell can execute it directly on its own. To see how, enter the following line at the JShell prompt:

```
System.out.println("This is a simple Java program.");
```

Then, press enter. This output is displayed:

```
This is a simple Java program.
```

```
jshell>
```

As you can see, the call to **println()** is evaluated and its string argument is output. Then, the prompt is redisplayed.

Before moving on, it is useful to explain why JShell can execute a single statement, such as the call to **println()**, when the Java compiler, **javac**, requires a complete program. JShell is able to evaluate a single statement because JShell automatically provides the necessary program framework for you, behind the scenes. This consists of a *synthetic class* and a *synthetic method*. Thus, in this case, the **println()** statement is embedded in a synthetic method that is part of a synthetic class. As a result, the preceding code is still part of a valid Java program even though you don't see all of the details. This approach provides a very fast and convenient way to experiment with Java code.

Next, let's look at how variables are supported. In JShell, you can declare a

variable, assign the variable a value, and use it in any valid expressions. For example, enter the following line at the prompt:

```
int count;
```

After doing so you will see the following response:

```
count ==> 0
```

This indicates that **count** has been added to the synthetic class and initialized to zero. Furthermore, it has been added as a **static** variable of the synthetic class.

Next, give **count** the value 10 by entering this statement:

```
count = 10;
```

You will see this response:

```
count ==> 10
```

As you can see, **count**'s value is now 10. Because **count** is **static**, it can be used without reference to an object.

Now that **count** has been declared, it can be used in an expression. For example, enter this **println()** statement:

```
System.out.println("Reciprocal of count: " + 1.0 / count);
```

JShell responds with:

```
Reciprocal of count: 0.1
```

Here, the result of the expression **1.0 / count** is 0.1 because **count** was previously assigned the value 10.

In addition to demonstrating the use of a variable, the preceding example illustrates another important aspect of JShell: it maintains state information. In this case, **count** is assigned the value 10 in one statement and then this value is used in the expression **1.0 / count** in the subsequent call to **println()** in a second statement. Between these two statements, JShell stores **count**'s value. In general, JShell maintains the current state and effect of the code snippets that you enter. This lets you experiment with larger code fragments that span multiple lines.

Before moving on, let's try one more example. In this case, we will create a

for loop that uses the **count** variable. Begin by entering this line at the prompt:

```
for(count = 0; count < 5; count++)
```

At this point, JShell responds with the following prompt:

```
...>
```

This indicates that additional code is required to finish the statement. In this case, the target of the **for** loop must be provided. Enter the following:

```
System.out.println(count);
```

After entering this line, the **for** statement is complete and both lines are executed. You will see the following output:

```
0  
1  
2  
3  
4
```

In addition to statements and variable declarations, JShell lets you declare classes and methods, and use import statements. Examples are shown in the following sections. One other point: Any code that is valid for JShell will also be valid for compilation by **javac**, assuming the necessary framework is provided to create a complete program. Thus, if a code fragment can be executed by JShell, then that fragment represents valid Java code. In other words, JShell code *is* Java code.

List, Edit, and Rerun Code

JShell supports a large number of commands that let you control the operation of JShell. At this point, three are of particular interest because they let you list the code that you have entered, edit a line of code, and rerun a code snippet. As the subsequent examples become longer, you will find these commands to be very helpful.

In JShell, all commands start with a / followed by the command. Perhaps the most commonly used command is **/list**, which lists the code that you have entered. Assuming that you have followed along with the examples shown in

the preceding section, you can list your code by entering `/list` at this time. Your JShell session will respond with a numbered list of the snippets you entered. Pay special attention to the entry that shows the `for` loop. Although it consists of two lines, it constitutes one statement. Thus, only one snippet number is used. In the language of JShell, the snippet numbers are referred to as *snippet IDs*. In addition to the basic form of `/list` just shown, other forms are supported, including those that let you list specific snippets by name or number. For example, you can list the `count` declaration by using `/list count`.

You can edit a snippet by using the `/edit` command. This command causes an edit window to open in which you can modify your code. Here are three forms of the `/edit` command that you will find helpful at this time. First, if you specify `/edit` by itself, the edit window contains all of the lines you have entered and lets you edit any part of it. Second, you can specify a specific snippet to edit by using `/edit n`, where *n* specifies the snippet's number. For example, to edit snippet 3, use `/edit 3`. Finally, you can specify a named element, such as a variable. For example, to change the value of `count`, use `/edit count`.

As you have seen, JShell executes code as you enter it. However, you can also rerun what you have entered. To rerun the last fragment that you entered, use `!.` To rerun a specific snippet, specify its number using this form: `/n`, where *n* specifies the snippet to run. For example, to rerun the fourth snippet, enter `/4`. You can rerun a snippet by specifying its position relative to the current fragment by use of a negative offset. For example, to rerun a fragment that is three snippets before the current one, use `/-3`.

Before moving on, it is helpful to point out that several commands, including those just shown, allow you to specify a list of names or numbers. For example, to edit lines 2 and 4, you could use `/edit 2 4`. For recent versions of JShell, several commands allow you specify a range of snippets. These include the `/list`, `/edit`, and `/n` commands just described. For example, to list snippets 4 through 6, you would use `/list 4-6`.

There is one other important command that you need to know about now: `/exit`. This terminates JShell.

Add a Method

As explained in [Chapter 6](#), methods occur within classes. However, when using JShell it is possible to experiment with a method without having to *explicitly* declare it within a class. As mentioned earlier, this is because JShell

automatically wraps code fragments within a synthetic class. As a result, you can easily and quickly write a method without having to provide a class framework. You can also call the method without having to create an object. This feature of JShell is especially beneficial when learning the basics of methods in Java or when prototyping new code. To understand the process, we will work through an example.

To begin, start a new JShell session and enter the following method at the prompt:

```
double reciprocal(double val) {  
    return 1.0/val;  
}
```

This creates a method that returns the reciprocal of its argument. After you enter this, JShell responds with the following:

```
| created method reciprocal(double)
```

This indicates the method has been added to JShell's synthetic class and is ready for use.

To call **reciprocal()**, simply specify its name, without any object or class reference. For example, try this:

```
System.out.println(reciprocal(4.0));
```

JShell responds by displaying 0.25.

You might be wondering why you can call **reciprocal()** without using the dot operator and an object reference. Here is the answer. When you create a stand-alone method in JShell, such as **reciprocal()**, JShell automatically makes that method a **static** member of the synthetic class. As you know from [Chapter 7](#), **static** methods are called relative to their class, not on a specific object. So, no object is required. This is similar to the way that stand-alone variables become **static** variables of the synthetic class, as described earlier.

Another important aspect of JShell is its support for a *forward reference* inside a method. This feature lets one method call another method, even if the second method has not yet been defined. This enables you to enter a method that depends on another method without having to worry about which one you enter first. Here is a simple example. Enter this line in JShell:

```
void myMeth() { myMeth2(); }
```

JShell responds with the following:

```
| created method myMeth(), however, it cannot be invoked until myMeth2()  
| is declared
```

As you can see, JShell knows that **myMeth2()** has not yet been declared, but it still lets you define **myMeth()**. As you would expect, if you try to call **myMeth()** at this time, you will see an error message since **myMeth2()** is not yet defined, but you are still able to enter the code for **myMeth()**.

Next, define **myMeth2()** like this:

```
void myMeth2() { System.out.println("JShell is powerful."); }
```

Now that **myMeth2()** has been defined, you can call **myMeth()**.

In addition to its use in a method, you can use a forward reference in a field initializer in a class.

Create a Class

Although JShell automatically supplies a synthetic class that wraps code snippets, you can also create your own class in JShell. Furthermore, you can instantiate objects of your class. This allows you to experiment with classes inside JShell's interactive environment. The following example illustrates the process.

Start a new JShell session and enter the following class, line by line:

```
class MyClass {  
    double v;  
  
    MyClass(double d) { v = d; }  
  
    // Return the reciprocal of v.  
    double reciprocal() { return 1.0 / v; }  
}
```

When you finish entering the code, JShell will respond with

```
| created class MyClass
```

Now that you have added **MyClass**, you can use it. For example, you can

create a **MyClass** object with the following line:

```
MyClass ob = new MyClass(10.0);
```

JShell will respond by telling you that it added **ob** as a variable of type **MyClass**. Next, try the following line:

```
System.out.println(ob.reciprocal());
```

JShell responds by displaying the value 0.1.

As a point of interest, when you add a class to JShell, it becomes a **static** nested member of a synthetic class.

Use an Interface

Interfaces are supported by JShell in the same way as classes. Therefore, you can declare an interface and implement it by a class within JShell. Let's work through a simple example. Before beginning, start a new JShell session.

The interface that we will use declares a method called **isLegalVal()** that is used to determine if a value is valid for some purpose. It returns **true** if the value is legal and **false** otherwise. Of course, what constitutes a legal value will be determined by each class that implements the interface. Begin by entering the following interface into JShell:

```
interface MyIF {  
    boolean isLegalVal(double v);  
}
```

JShell responds with

```
| created interface MyIf
```

Next, enter the following class, which implements MyIF:

```
class MyClass implements MyIF {  
  
    double start;  
    double end;  
  
    MyClass(double a, double b) { start = a; end = b; }  
  
    // Determine if v is within the range start to end, inclusive.  
    public boolean isLegalVal(double v) {  
        if((v >= start) && (v <= end)) return true;  
        return false;  
    }  
}
```

JShell responds with

```
| created class MyClass
```

Notice that **MyClass** implements **isLegalVal()** by determining if the value **v** is within the range (inclusive) of the values in the **MyClass** instance variables **start** and **end**.

Now that both **MyIF** and **MyClass** have been added, you can create a **MyClass** object and call **isLegalVal()** on it, as shown here:

```
MyClass ob = new MyClass(0.0, 10.0);  
  
System.out.println(ob.isLegalVal(5.0));
```

In this case, the value **true** is displayed because 5 is within the range 0 through 10.

Because **MyIF** has been added to JShell, you can also create a reference to an object of type **MyIF**. For example, the following is also valid code:

```
MyIF ob2 = new MyClass(1.0, 3.0);  
boolean result = ob2.isLegalVal(1.1);
```

In this case, the value of **result** will be **true** and will be reported as such by JShell.

One other point: enumerations and annotations are supported in JShell in the same way as classes and interfaces.

Evaluate Expressions and Use Built-in Variables

JShell includes the ability to directly evaluate an expression without it needing to be part of a full Java statement. This is especially useful when you are experimenting with code and don't need to execute the expression in a larger context. Here is a simple example. Using a new JShell session, enter the following at the prompt:

```
3.0 / 16.0
```

JShell responds with:

```
$1 ==> 0.1875
```

As you can see, the result of the expression is computed and displayed. However, note that this value is also assigned to a temporary variable called **\$1**. In general, each time an expression is evaluated directly, its result is stored in a temporary variable of the proper type. Temporary variable names all begin with a \$ followed by a number, which is increased each time a new temporary variable is needed. You can use these temporary variables like any other variable. For example, the following displays the value of **\$1**, which is 0.1875 in this case.

```
System.out.println($1);
```

Here is another example:

```
double v = $1 * 2;
```

Here, the value **\$1** times 2 is assigned to **v**. Thus, **v** will contain 0.375.

You can change the value of a temporary variable. For example, this reverses the sign of **\$1**:

```
$1 = -$1
```

JShell responds with

```
$1 ==> -0.1875
```

Expressions are not limited to numeric values. For example, here is one that concatenates a **String** with the value returned by **Math.abs(\$1)**.

```
"The absolute value of $1 is " + Math.abs($1)
```

This results in a temporary variable that contains the string

```
The absolute value of $1 is 0.1875
```

Importing Packages

As described in [Chapter 9](#), an **import** statement is used to bring members of a package into view. Furthermore, any time you use a package other than **java.lang**, you must import it. The situation is much the same in JShell except that by default, JShell imports several commonly used packages automatically. These include **java.io** and **java.util**, among several others. Since these packages are already imported, no explicit **import** statement is required to use them.

For example, because **java.io** is automatically imported, the following statement can be entered:

```
FileInputStream fin = new FileInputStream("myfile.txt");
```

Recall that **FileInputStream** is packaged in **java.io**. Since **java.io** is automatically imported, it can be used without having to include an explicit **import** statement. Assuming that you actually have a file called **myfile.txt** in the current directory, JShell will respond by adding the variable **fin** and opening the file. You can then read and display the file by entering these statements:

```
int i;
do {
    i = fin.read();
    if(i != -1) System.out.print((char) i);
} while(i != -1);
```

This is the same basic code that was discussed in [Chapter 13](#), but no explicit **import java.io** statement is required.

Keep in mind that JShell automatically imports only a handful of packages. If you want to use a package not automatically imported by JShell, then you

must explicitly import it as you do with a normal Java program. One other point: you can see a list of the current imports by using the **/imports** command.

Exceptions

In the I/O example shown in the preceding section on imports, the code snippets also illustrate another very important aspect of JShell. Notice that there are no **try/catch** blocks that handle I/O exceptions. If you look back at the similar code in [Chapter 13](#), the code that opens the file catches a **FileNotFoundException**, and the code that reads the file watches for an **IOException**. The reason that you don't need to catch these exceptions in the snippets shown earlier is because JShell automatically handles them for you. More generally, JShell will automatically handle checked exceptions in many cases.

Some More JShell Commands

In addition to the commands discussed earlier, JShell supports several others. One command that you will want to try immediately is **/help**. It displays a list of the commands. You can also use **/?** to obtain help. Some of the more commonly used commands are examined here.

You can reset JShell by using the **/reset** command. This is especially useful when you want to change to a new project. By use of **/reset** you avoid the need to exit and then restart JShell. Be aware, however, that **/reset** resets the entire JShell environment, so all state information is lost.

You can save a session by using **/save**. Its simplest form is shown here:

```
/save filename
```

Here, *filename* specifies the name of the file to save into. By default, **/save** saves your current source code, but it supports several options, of which two are of particular interest. By specifying **-all** you save all lines that you enter, including those that you entered incorrectly. You can use the **-history** option to save your session history (i.e., the list of the commands that you have entered).

You can load a saved session by using **/open**. Its form is shown next:

```
/open filename
```

Here, *filename* is the name of the file to load.

JShell provides several commands that let you list various elements of your work. They are shown here:

Command	Effect
/types	Shows classes, interfaces, and enums.
/imports	Shows import statements.
/methods	Shows methods.
/vars	Shows variables.

For example, if you entered the following lines:

```
int start = 0;  
int end = 10;  
int count = 5;
```

and then entered the **/vars** command, you would see

```
| int start = 0;  
| int end = 10;  
| int count = 5;
```

Another often useful command is **/history**. It lets you view the history of the current session. The history contains a list of what you have typed at the command prompt.

Exploring JShell Further

The best way to get proficient with JShell is to work with it. Try entering several different Java constructs and watching the way that JShell responds. As you experiment with JShell, you will find the usage patterns that work best for you. This will enable you to find effective ways to integrate JShell into your learning or development process. Also, keep in mind that JShell is not just for beginners. It also excels when prototyping code. Thus, even if you are an experienced pro, you will still find JShell helpful whenever you need to explore new areas.

Simply put: JShell is an important tool that further enhances the overall Java development experience.

APPENDIX

Compile and Run Simple Single-File Programs in One Step

In [Chapter 2](#), you were shown how to compile a Java program into bytecode using the **javac** compiler and then run the resulting **.class** file(s) using the Java launcher **java**. This is how Java programs have been compiled and run since Java's beginning, and it is the method that you will use when developing applications. However, beginning with JDK 11, it is possible to compile and run some types of simple Java programs directly from the source file without having to first invoke **javac**. To do this, pass the name of the source file, using the **.java** file extension, to **java**. This causes **java** to automatically invoke the compiler and execute the program.

For example, the following automatically compiles and runs the first example in this book:

```
java Example.java
```

In this case, the **Example** class is compiled and then run in a single step. There is no need to use **javac**. Be aware, however, that no **.class** file is created. Instead, the compilation is done behind the scenes. As a result, to rerun the program, you must execute the source file again. You can't execute its **.class** file because one won't be created. One other point: if you have already used **javac** to compile **Example.java**, then you must erase the resulting **.class** file **Example.class** before trying the source-file launch feature. When running a source file, there cannot already be a **.class** file with the same name as the class that contains **main()** in the source file.

One use of the source-file launch capability is to facilitate the use of Java programs in script files. It can also be useful for short one-time-use programs. In some cases, it makes it a little easier to run simple example programs when you are experimenting with Java. It is not, however, a general-purpose substitute for Java's normal compilation/execution process.

Although this new ability to launch a Java program directly from its source file is appealing, it comes with several restrictions. First, the entire program must be contained in a single source file. However, most real-world programs

use multiple source files. Second, it will always execute the first class it finds in the file, and that class must contain a **main()** method. If the first class in the file does not contain a **main()** method, the launch will fail. This means that you must follow a strict organization for your code, even if you would prefer to organize it otherwise. Third, because no **.class** files are created, using **java** to run a single-file program does not result in a class file that can be reused, possibly by other programs. Finally, as a general rule, if there is already a **.class** file that has the same name as a class in the source file, then the single-file launch will fail. As a result of these restrictions, using **java** to run a single-file source program can be useful, but it constitutes what is, essentially, a special-case technique.

As it relates to this book, it is possible to use the single source-file launch feature to try many of the examples; just be sure that the class with the **main()** method is first in your file. That said, it is not, however, applicable or appropriate in all cases. Furthermore, the discussions (and many of the examples) in the book assume that you are using the normal compilation process of invoking **javac** to compile a source file into bytecode and then using **java** to run that bytecode. This is the mechanism used for real-world development, and understanding this process is an important part of learning Java. It is imperative that you are thoroughly familiar with it. For these reasons, when trying the examples in this book, it is strongly recommended that in all cases you use the normal approach to compiling and running a Java program. Doing so ensures that you have a solid foundation in the way Java works. Of course, you might find it fun to experiment with the single source-file launch option!

Index

Please note that index links point to page beginnings from the print edition. Locations are approximate in e-readers, and you may need to page down one or more times after clicking a link to get to the indexed material.

&

- bitwise AND, [72](#), [73](#), [74–75](#)
- Boolean logical AND, [81](#), [82](#), [83](#)
- and bounded type declarations, [357](#)

&& (short-circuit AND), [81](#), [83](#)

*

- and glob syntax, [754](#)
- multiplication operator, [31](#), [67](#), [68](#)
- regular expression quantifier, [997](#)
- used in import statement, [206](#), [341](#)

** (glob syntax), [754](#)

@

- annotation syntax, [37](#), [295](#)
- used with tags (javadoc), [1148](#), [1152](#)

|

- bitwise OR, [72](#), [73](#), [74–75](#)
- Boolean logical OR, [81](#), [82](#)

|| (short-circuit OR), [81](#), [83](#)

[], [37](#), [55](#), [56](#), [58](#), [62](#), [64](#), [67](#)

- character class specification, [997](#), [1001](#)

^

- bitwise exclusive OR (XOR), [72](#), [73](#), [74–75](#)
- Boolean logical exclusive OR (XOR), [81](#), [82](#)
- character class specification, [997](#)

: (used with a label), [111](#)

:: constructor reference, 37, 412, 416
method reference, 37, 404, 410
, (comma), 37, 101, 395
format flag, 652, 654
{ }, 29, 30, 34, 37, 49, 50, 57, 60, 87, 88, 95, 99, 231, 306, 396, 420
used with javadoc tags, 1148
{ ... } anonymous inner class syntax, 813
\$ used in temporary variable name, 1162
=, 31, 48, 81, 83–84
== (Boolean logical operator), 81
== (relational operator), 32, 80, 81, 278, 284
versus equals(), 458–459
!, 81, 82
!=, 80, 81
/, 67, 68
/* */, 28
/** */, 37, 1147
//, 29
<, 32, 80
argument index syntax, 655–656
< >
diamond operator (type inference), 380–381
and generic type parameter, 348
<?>, 296, 298, 358
<<, 72, 75–77
<=, 80
-, 67, 68
format flag, 652
-> lambda expression arrow operator, 18, 67, 85, 390
--, 34, 67, 70–71
%
used in format conversion specifier syntax, 643
modulus operator, 67, 69
(format flag, 652, 654
(), 29, 37, 86, 121, 131

used in a lambda expression, [390](#), [394](#), [395](#)
used to raise the precedence of operations, [37](#), [45](#), [85](#), [454](#)

. (period)
and calling static interface methods, [223](#)
dot operator, [85](#), [119](#), [125](#), [154](#), [180](#), [206](#), [223](#)
in import statement, [206](#)
in multileveled package statement, [200](#), [206](#)
and nested interfaces, [213](#)
regular expression wildcard character, [997](#), [1000](#)
separator, [37](#)

...

and enabling/disabling assertions syntax, [338](#)–[339](#)
variable-length argument syntax, [37](#), [164](#), [167](#)

+

addition operator, [67](#), [68](#)
concatenation operator, [31](#), [160](#)–[161](#), [453](#)–[454](#)
format flag, [652](#), [653](#)
regular expression quantifier, [997](#), [999](#)–[1001](#)
unary plus, [67](#), [68](#)

++, [34](#), [67](#), [70](#)–[72](#)

format flag, [652](#), [654](#)

?

regular expression quantifier, [997](#), [1000](#)–[1001](#)
wildcard argument specifier, [358](#), [361](#), [364](#), [378](#), [387](#), [754](#)

?: (ternary if-then-else operator), [82](#), [84](#)

>, [32](#), [80](#)

>>, [72](#), [77](#)–[78](#)

>>>, [72](#), [78](#)–[79](#)

>=, [80](#)

;(semicolon), [30](#), [37](#), [96](#), [209](#)
used in try-with-resources statement, [331](#), [688](#)

~(bitwise unary NOT operator), [72](#), [73](#), [74](#)–[75](#)

_ (underscore), [36](#), [46](#), [47](#)

A

`abs()`, 139–140, 517
Abstract method(s), 191–194
 and lambda expressions, 390, 391, 392, 393
abstract type modifier, 192, 195, 212, 391
Abstract Window Toolkit. *See* AWT (Abstract Window Toolkit)
AbstractAction interface, 1095, 1096
AbstractButton class, 1047, 1050, 1074, 1077, 1082, 1085
AbstractCollection class, 551, 553, 560
AbstractList class, 551, 603
AbstractMap class, 578, 580, 582
AbstractQueue class, 551, 559
AbstractSequentialList class, 551, 555
AbstractSet class, 551, 556, 558, 561
`accept()`, 566, 674, 675, 676, 685, 686, 754, 773, 976, 989, 991
Access control, 149–152
 and default access, 203, 209
 example program, 203–206
 and inheritance, 150, 152, 173–174
 and modules, 150
 and packages, 150, 199, 202–206
Access modifiers, 29, 150, 202–203
`acquire()`, 922–925
Action (Swing), 1073, 1093–1098
Action interface, 1093, 1094, 1095
ActionEvent class, 786–787, 847, 871, 884, 1036, 1037, 1045, 1047, 1054
ActionListener interface, 797, 847, 851, 859, 884, 1036, 1037, 1047, 1054,
 1081, 1093
`actionPerformed()`, 797, 847, 850, 1037, 1038, 1047, 1054, 1081, 1082, 1093,
 1095, 1096–1097
`adapt()`, 966
Adapter classes, 804, 808–810
`add()`, 541, 542, 543, 544, 556, 563, 628, 819, 844, 850, 855, 858, 871, 875,
 884, 989, 1008, 1009, 1033–1034, 1054, 1066, 1067, 1075, 1076,
 1091, 1095
`addActionListener()`, 1037
`addAll()`, 541, 542, 543, 544, 989

`addCookie()`, [1134](#), [1140](#)
`addElement()`, [604](#)
`addExports()`, [528](#)
`addFirst()`, [549](#), [550](#), [555](#), [989](#)
`addItem()`, [1064](#)
`addKeyListener()`, [785](#)
`addLast()`, [549](#), [550](#), [555](#), [556](#)
`addMouseListener()`, [803](#)–[804](#), [813](#), [1088](#)
`addMouseMotionListener()`, [785](#), [803](#)–[804](#)
`addOpens()`, [528](#)
`addReads()`, [528](#)
Address, Internet, [760](#), [761](#)–[763](#), [766](#)–[767](#)
`addSeparator()`, [1076](#)
`addTab()`, [1056](#)
`addTListener()` [1112](#)
`addTypeListener()`, [784](#), [785](#)
`addUses()`, [528](#)
AdjustmentEvent class, [787](#), [788](#), [862](#)
AdjustmentListener interface, [797](#), [798](#), [862](#)
`adjustmentValueChanged()`, [797](#)
Algorithms, collection, [539](#), [591](#)–[596](#), [602](#)
`allMatch()`, [994](#)
`allocate()`, [729](#), [739](#), [741](#)
anchor constraint field, [879](#)–[880](#)
AND operator
 bitwise (`&`), [72](#), [73](#), [74](#)–[75](#)
 Boolean logical (`&`), [81](#), [82](#), [83](#)
 and bounded type declarations (`&`), [357](#)
 short-circuit (`&&`), [81](#), [83](#)
AnnotatedElement interface, [301](#), [303](#), [313](#), [536](#)
Annotation interface, [295](#), [301](#), [535](#)
Annotation member, [295](#)
 assigning a value to an, [295](#), [304](#), [305](#)
 default values for an, [302](#)–[303](#), [305](#)
 obtaining the value of an, [297](#), [298](#)
Annotation(s), [16](#), [294](#)–[314](#), [535](#)

built-in, 305–307
container, 312, 313
declaration example, 295
and JShell, 1161
marker, 303–304
member. *See* Annotation member
obtaining all, 300–301
reflection to obtain, using, 296–301
repeated, 301, 312–314
restrictions on, 314
retention policies, 295–296
single-member, 304–305
type, 307–312
annotationType(), 295
anyMatch(), 994
Apache Software Foundation, 1122
Apache Tomcat, 1122. *See also* Tomcat
API library, compact profiles of the, 344
APPEND, 505
append(), 471, 533, 710, 867
Appendable interface, 533, 645, 703, 710, 717
appendTo(), 505
Applet, 8–9
 deprecated, 11, 19
 removal of support for the, 9, 11, 19, 1029
Applet class, 818
Applet, Swing, 1029, 1030, 1035
Application launcher (java). *See* java (Java application launcher)
apply(), 417, 674, 675, 676, 977–978, 982
applyAsDouble(), 674, 675, 676, 985
AreaAveragingScaleFilter class, 904
areFieldsSet Calendar class instance variable, 628
Argument(s), 124, 128
 command-line, 29, 162–163
 index, 655–656
 lambda expressions passed as, 399–401

and overloaded constructors, 142
passing, 144–146
type. *See* Type argument(s)
variable-length. *See* Varargs
wildcard. *See* Wildcard arguments

Arithmetic operators, 67–72

ArithmeticException, 229, 230, 240, 518

Array class, 536, 1004

Array(s), 29, 55–62, 155, 196

- boundary checks, 57
- and collections, 597
- constructor reference for, 416
- converting collections into, 542, 553–554
- copying with arraycopy(), 506, 508
- declaration syntax, alternative, 62
- declaration using var, 64, 65
- dynamic, 551–553, 560, 603–607
- and the for-each loop, 103–107
- and generics, 386–387
- implemented as objects, 155
- indexes, 55, 56
- initializing, 57, 60–61
- length instance variable of, 155–157
- multidimensional, 58–62, 105–106
- one-dimensional, 55–58
- and spliterators, 600
- and a stream API stream, 973
- string using a byte, initializing a, 451
- of strings, 65, 162
- and valueOf(), 465
- and varargs, 164

ArrayList class, 551–554, 570, 603, 604, 973

example using an, 564–565
examples using a stream API stream, 973–977, 982–986, 987–988, 990–993
Arrays class, 597–602, 973
ArrayStoreException, 240, 598
arrive(), 934–935
arriveAndAwaitAdvance(), 934, 935, 937, 940
arriveAndDeregister(), 935, 937
Arrow operator (\rightarrow), 18, 67, 85, 390
ASCII character set, 43, 44, 47, 460
 and strings on the Internet, 451
asIterator(), 603
asList(), 597
Assembly language, 4, 5
assert statement, 15, 336–339
Assertions, 336–339
AssertionError, 336, 337
Assignment operator
 =, 31, 81, 83–84
 arithmetic compound (op=), 67, 69–70
 bitwise compound, 72, 79–80
 Boolean logical, 81
Atomic operations, 950–951
AtomicInteger class, 921, 951
AtomicLong class, 921, 951
AttributeView interface, 738
Autoboxing/unboxing, 16, 287, 289–294, 349–350
 Boolean and Character values, 292–293
 and collections, 554
 definition of, 289
 and error prevention, 293–294
 and expressions, 291–292
 and methods, 290–291
Autocloseable interface, 324, 330, 534, 656, 663, 686, 688, 689, 693, 703, 706, 708, 709, 710, 717, 721, 723, 730, 739, 753, 764, 774, 970
Automatic resource management (ARM), 228, 329–333, 534, 656, 766
available(), 691–692, 723, 724

availableProcessors(), 962
await(), 928, 929–930, 931, 949
awaitAdvance(), 940
awaitAdvanceInterruptibly(), 940
AWT (Abstract Window Toolkit), 315, 783, 800, 815–816, 843
 classes, table of some, 816–818
 color system, 829
 controls. *See* Controls, AWT
 and fonts, 833–839
 layout managers. *See* Layout manager(s)
 support for imaging, 893
 support for text and graphics, 824
 and Swing, 815, 843, 1025–1026
AWTEvent class, 786, 816

B

B, 4
Base64 class, 672
BaseStream interface, 970–971, 972, 979, 981
 methods, table of, 970
BASIC, 4
Basic multilingual plane (BMP), 494
BasicFileAttributes class, 736, 737, 738, 751
 methods, table of, 737
BasicFileAttributeView interface, 738
BCP 49, 636
BCPL, 4
BeanInfo interface, 1110, 1112–1113, 1114, 1116
 naming convention for a class implementing the, 1112
Beans, Java. *See* Java Beans
Bell curve, 636
Bell Laboratories, 6
Berkeley UNIX, 759
Berners-Lee, Tim, 766
Beyond Photography, The Digital Darkroom (Holzmann), 900

BiConsumer functional interface, [674](#), [989](#)
BiFunction functional interface, [674](#), [977](#)
Binary
 exponent, [47](#)
 literals, [46](#)
 numbers and integers, [72](#)–[73](#)
BinaryOperator<T> predefined functional interface, [417](#), [674](#), [977](#)
binarySearch()
 algorithm defined by Collections, [591](#)
 Arrays method, [597](#)
BitSet class, [621](#)–[623](#)
 methods, table of, [621](#)–[622](#)
Bitwise operators, [72](#)–[80](#)
Block lambdas, [390](#), [395](#)–[397](#). *See also* Lambda expression(s)
BLOCKED, [274](#)
Blocks of code. *See* Code blocks
Body handlers, HTTP Client API, [778](#), [779](#), [780](#), [781](#), [782](#)
body(), [779](#), [780](#), [781](#)
Boolean, [39](#)
 expression, [32](#), [33](#)
 literals, [47](#)
 logical operators, [81](#)–[83](#)
Boolean class, [287](#), [495](#)
 and autoboxing/unboxing, [292](#)–[293](#)
 methods, table of, [497](#)
boolean data type, [39](#), [44](#)–[45](#), [47](#), [52](#)
 and relational operators, [44](#), [45](#), [80](#)–[81](#)
booleanValue(), [287](#), [497](#)
Border interface, [1042](#)
BorderFactory class, [1042](#)
BorderLayout class, [816](#), [870](#)–[872](#), [1037](#), [1092](#)
 example with insets, [872](#)–[873](#)
boxed(), [972](#)
Boxing, [289](#)
break statement, [90](#), [91](#)–[92](#), [109](#)–[113](#)
 and the for-each loop, [104](#)–[105](#)

as form of goto, 111–113

Buffer class, 728–730
methods, table of, 728–729

Buffer, NIO, 728–730

BufferedImage class, 895

BufferedInputStream class, 317, 698–700, 749

BufferedOutputStream class, 317, 698, 700, 749

BufferedReader class, 318, 319, 320–322, 714–716, 973

BufferedWriter class, 318, 716

Buffering, double, 896–899

build(), 503, 778, 779

bulkRegister(), 941

Button class, AWT, 816, 847

Button(s)
as event sources, 787, 796, 847
group, 1054, 1086
push. *See* Push buttons, AWT; Push buttons, JavaFX; Push buttons, Swing
radio. *See* Radio buttons
Swing, 1047–1055
Swing menu items as, 1074
toggle. *See* Toggle button, Swing

ButtonGroup class, 1043, 1054

ButtonModel interface, 1028, 1047

ButtonUI, 1028

Byte class, 287, 288, 477, 483, 490, 491
methods defined by, table of, 484

byte data type, 39, 40–41, 45
and automatic type conversion, 52
and automatic type promotion, 54, 75–76, 78–79

ByteArrayInputStream class, 317, 695–696

ByteArrayOutputStream class, 317, 696–698

ByteBuffer class, 729, 738, 739, 743
get() and put() methods, table of, 730

Bytecode, 10–11, 12, 14, 15, 20, 28, 336, 343, 520, 1121, 1122

BYTES, 479, 490, 492

byteValue(), 288, 478, 480, 481, 484, 485, 486, 488

C

C, 6, 8

history of, 4–5

and Java, 3, 5, 7, 13

C Programming Language, The (Kernighan and Ritchie), 5

C++

history of, 5–6

and Java, 3, 7–8, 13

C# and Java, 8

Calendar class, 626, 627, 628–631, 632, 636, 1015

constants, 630

methods defined by, table of a sampling of, 628–630

Call-by-reference, 144, 145–146

Call-by-value, 144–145, 146

call(), 944, 966

Callable interface, 921, 943–946, 966

CallSite class, 535

cancel(), 639, 640, 965, 966

canRead(), 528

canUse(), 528

Canvas class, AWT, 816, 819, 894

capacity(), 469, 604, 728

capacityIncrement Vector data member, 604

Card layouts, 875–878

CardLayout class, 816, 875–878

CaretEvent class, 1045–1046

Case sensitivity and Java, 27, 29, 36, 200

case statement, 90–95

Casts, 52–54, 346, 349, 350, 352

and casting one instance of a generic class into another, 378

and erasure, 349, 382

using instanceof with, 333–335

using a type intersection with, 357

catch clause(s), 227, 228, 230–234, 236, 237–238, 246

displaying exception description within, 232

and the more precise (final) rethrow feature, 245, 246
multi-catch feature of, 245–246
using multiple, 232–233
and nested try statements, 231, 234

CGI (Common Gateway Interface), 12, 1121–1122

Channel interface, 730

Channel(s), NIO, 728, 730–731. *See also* NIO and channel-based I/O

char data type, 39, 43–44, 46, 72, 451
and arithmetic operators, 67
and automatic type conversion, 52
and automatic type promotion, 54

Character class, 287, 477, 492–495
and autoboxing/unboxing, 292–293
methods, table of various, 493, 496
support for 34-bit Unicode, 494–495

Character(s), 39, 43–44
basic multilingual plane (BMP), 494
changing case of, 465–466, 492, 493
classes (regular expressions), 997, 1001
code point, 494
escape sequences, 47, 48
extraction from String objects, 455–456
formatting an individual, 645
literals, 47
supplemental, 494

Character.Subset class, 494

Character.UnicodeBlock class, 494

characteristics(), 567, 568

CharArrayReader class, 318, 712–713

CharArrayWriter class, 318, 713–714

charAt(), 161, 455, 470, 532

CharBuffer class, 729

CharSequence interface, 449, 466, 472, 532, 996
methods defined by, table of, 532

Charset, 452, 704, 718, 731

charValue(), 287, 492

Check boxes

AWT, 844, 851–853
as event sources, 787, 791, 796, 852
Swing, 1052–1053
and Swing menus, 1085, 1086–1087
Checkbox class, AWT, 816, 851–853
CheckboxGroup class, 816, 853–855
CheckboxMenuItem class, 816, 883, 884
checked... methods, 591–592, 595
checkedCollection(), 591, 595
checkedList(), 591, 595
checkedMap(), 591, 595
checkedSet(), 592, 595
Choice class, 816, 855–858
Choice controls, 796, 844, 855–861
as event source, 787, 791, 796
Class class, 296–297, 298, 300, 301, 348, 434–435, 495, 512–515, 737, 1004, 1006
methods, table of some, 512–514
.class file, 120, 200, 201, 1165, 1166
.class filename extension, 28
class keyword, 29, 117
CLASS retention policy, 295–296
Class(es), 28–29, 117–136
abstract, 191–194, 195, 212
access levels of, 202–203
adapter, 804, 808–810
anonymous, 18, 63. *See also* Inner classes
character, regular expression, 997, 1001
and code, 27, 117, 202
in collections, storing user-defined, 568–570
constructor, 297, 300, 301, 536, 1004
controlling access to. *See* Access control
as a data type, 117, 119, 121, 122, 123, 134
definition of the term, 23
encapsulation achieved through, 23, 134

final, 195
general form of, 117–118
generic. *See* Generic class
hierarchy, 24, 181–184, 208
inner. *See* Inner classes
instance of a, 23, 117, 119, 122
and interfaces, 199, 208, 210–213, 369
JShell to experiment with, using, 1160
libraries, 27, 38
literal, 298, 435
member. *See* Member, class
name and source file name, 27, 28
nested, 157–159
packages as containers for, 199, 202, 206
path, 428
public, 203
scope defined by a, 50
and state information, 220, 223
synthetic, 1156, 1159, 1160
type for bounded types, using a, 355–357
value-based, 624
ClassCastException, 240, 540–541, 543, 546, 548, 550, 571, 574, 575, 583, 595, 597, 599, 600
ClassDefinition class, 535
ClassFileTransformer interface, 535
ClassLoader class, 515
classModifiers(), 1007
ClassNotFoundException, 241, 723
CLASSPATH, 200, 201, 1010
–classpath option, 201
ClassValue class, 532
clear(), 541, 542, 572, 611, 621, 629, 728
Client/server model, 8–9, 11, 759
 and sockets, 763–766
clone(), 197, 509–512, 531, 604, 611, 621, 627, 629, 633, 1136
Cloneable interface, 509–512

`CloneNotSupportedException`, 241, 509, 531

Cloning

and enumerations, 531

potential dangers of, 510, 512

`close()`, 324, 326–328, 329, 330, 331, 333, 534, 643, 656, 658, 663, 667, 686, 687, 688, 690, 691, 694, 695, 696, 706, 709, 710, 712, 713, 721, 722, 723, 724, 739, 764, 765, 766, 774, 780, 970

within a finally block, calling, 326–329, 687

`Closeable` interface, 324, 330, 663, 686, 689, 693, 703, 706, 708, 709, 710, 717, 730

Closures, 389

COBOL, 4

Code blocks, 32, 34–35, 49, 88–89

and the break statement, 111–113

and scopes, 49, 50–51

static, 153

synchronized, 263–265, 968

Code point, definition of, 494

Code

snippet, 1155, 1157, 1158, 1160, 1163

unreachable, 115, 233

`codePointAt()`, 467, 474, 495, 496

`codePointBefore()`, 467, 474, 496

`codePointCount()`, 467, 474

`collect()`, 971, 986–989

`Collection` interface, 540–543, 548, 553, 571, 973, 975, 979

methods defined by, table of, 541–542

Collection-view, 539, 570, 571, 612

Collection(s), 345, 538–618

algorithms, 539, 591–596, 602

into arrays, converting, 542, 553–554

and autoboxing, 554

classes, 345, 550–561

concurrent, 920, 947–948

cycling through, 539, 561–568, 607

dynamically typesafe view of a, 595

and the for-each version of the for loop, 103, 107, 565–566, 607
Framework. *See* Collections Framework
interfaces, 539–550
and iterators, 539, 543, 561–565, 566–568
and legacy classes and interfaces, 602–618
modifiable versus unmodifiable, 540
and primitive types, 478, 554
random access to, 570
storing user-defined classes in, 568–570
and the stream API, 618, 969, 970, 973, 975, 988
stream API stream to obtain a, using a, 986–989
and synchronization, 550, 595, 602
and type safety, 595
when to use, 618

Collections class, 410, 539, 595, 602
algorithms defined by, table of, 591–595

Collections Framework, 15, 103, 107, 289, 537–618, 948
advantages of generics as applied to the, 345
legacy classes and interfaces, 602–618
and method references, 410–412
overview, 538–539

Collector interface, 986

Collectors class, 986

Color class, AWT, 816, 821–822, 829–831
constants to specify colors, list of, 822
methods to manipulate colors, 829–830

Color, working with, 829–831

Combo boxes, Swing, 1063–1065

ComboBoxModel interface, 1064

Comment(s), 28, 29
documentation, 36–37, 1147–1154

Common Gateway interface (CGI), 12, 1121–1122

commonPool(), 955, 959

Compact profiles, 344

Comparable interface, 366, 369, 459, 532–533, 627, 683

Comparable<Path> interface, 732

Comparator interface, 411, 412, 540, 575, 580, 583, 975
comparator(), 546, 560, 575
Comparators, 558, 559, 560, 580, 581, 582–590
 using a lambda expression with, 586–587, 590
compare(), 411–412, 480, 481, 484, 485, 486, 488, 497, 532, 583, 585–586,
 601, 975
compareAndSet(), 921, 951
compareTo(), 283–285, 459–460, 480, 481, 484, 485, 486, 488, 494, 497, 503,
 531, 532, 586, 627, 683
compareToIgnoreCase(), 460
compareToOptional(), 503
compareUnsigned(), 601
comparing(), 584–585
comparingByKey(), 575
comparingByValue(), 575
comparingDouble(), 585
comparingInt(), 585
comparingLong(), 585
Compilation unit, 27
compile(), 996
Compiler
 ahead-of-time, 11
 Just-in-Time (JIT), 10, 14
Compiler class, 520
Compiler, Java, 27–28
 and main(), 29
Component class, 785, 796, 816, 818–819, 821, 822, 823, 824, 836, 844, 869,
 892, 894, 1028, 1029, 1032, 1034, 1038, 1039, 1075
ComponentAdapter class, 809
componentAdded(), 798
ComponentEvent class, 787, 788–789, 790, 796
componentHidden(), 798
ComponentListener interface, 797, 798, 809
componentMoved(), 798
componentRemoved(), 798
componentResized(), 798

Components, AWT, 1025–1026, 1028
 lightweight versus heavyweight, 892
 and overriding `paint()`, 892
Components and Java Beans, software, 1109
Components, Swing, 1028–1029, 1043–1071
 architecture, 1027–1028
 class names for, table of, 1028–1029
 and the Swing event dispatching thread, 1034
 heavyweight, 1029
 lightweight, 1026, 1043
 painting, 1038–1042
 and pluggable look and feel, 1026–1027, 1028
 and tabbed panes, 1056–1058
`componentShown()`, 798
ComponentUI, 1028
`compute()`, 954, 959, 963, 965, 968
`concat()`, 463
Concurrency utilities, 16, 919–968
 versus traditional multithreading and synchronization, 919, 968
Concurrent API, 919–920
 packages, 920–921
Concurrent collection classes, 920, 947–948
 list of, 947
Concurrent program, definition of the term, 919
ConcurrentHashMap class, 921, 947
ConcurrentLinkedDeque, 947
ConcurrentLinkedQueue class, 921, 947
ConcurrentSkipListMap class, 947
ConcurrentSkipListSet class, 947
Condition class, 949
`connect()`, 764
Console class, 718–720
 methods, table of, 719
Console I/O, 30, 99, 315, 319–323, 718–720
`console()`, 506, 719
const keyword, 38

Constants, 36
 using an interface to define shared, 216–218

Constructor class, 297, 300, 301, 536, 1004

Constructor reference, 412–416
 for an array, 416
 to generic classes, 413–416

Constructor(s), 121, 129–132
 in class hierarchy, order of execution of, 184–185
 default, 121, 131
 enumeration, 281–283
 factory methods versus overloaded, 761
 generic, 367
 object parameters for, 143–144
 overloading, 140–142
 parameterized, 131–132
 reference. *See* Constructor reference(s)
 and super(), 177–180, 184, 344
 this() and overloaded, 341–344

constructorModifiers(), 1007

Consumer<T> predefined functional interface, 417, 533, 566, 674, 976, 991

Container class, 816, 819, 821, 844, 869, 871, 872, 1028, 1029, 1033, 1039, 1075

Container(s), Swing, 1028, 1029
 lightweight versus heavyweight, 1029
 panes, 1029–1030. *See also* Content pane
 top-level, 1028, 1029

ContainerAdapter class, 809

ContainerEvent class, 787, 789

ContainerListener interface, 797, 798, 809

Containment hierarchy, 1028, 1029

contains(), 467, 541, 542, 556, 604, 611

containsAll(), 541, 542

Content pane, 1029, 1030, 1033–1034, 1042, 1056, 1059, 1067, 1070, 1076, 1081
 adding a component to a, 1033–1034
 default layout manager of a JFrame, 1033, 1036–1037

Context switching, 247, 263, 276
 rules for, 249
CONTINUE, 756
continue statement, 113–114
Control statements. *See* Statements, control
Control(s), AWT, 843, 844–868
 action events, using an anonymous inner class or lambda expression to handle, 850–851
 adding, 844
 definition of an, 843
 removing, 844
 responding to, 844–845
convert(), 947
ConvolveOp built-in convolution filter, 915
Convolution filters, 907, 913, 915
Cookie class, 1132, 1134–1135, 1140, 1142
 methods, table of, 1136
CookieHandler class, 772
CookieManager class, 772
CookiePolicy interface, 772
Cookies, 772, 1134–1135
 example servlet using, 1140–1142
CookieStore interface, 772
copy(), 734, 747
copyOf(), 544, 546, 561, 562, 572, 597–598
copyOfRange(), 598
CopyOnWriteArrayList class, 921, 947
CopyOnWriteArraySet class, 947
cos(), 42, 515
count(), 971, 977, 994
countDown(), 928–929
CountDownLatch class, 920, 922, 927–929
CountedCompleter class, 953
countStackFrames(), 522
createImage(), 894, 899, 900, 904
createLineBorder(), 1042

CropImageFilter class, 904–906
Currency class, 641–642
 methods, table of, 641
currentThread(), 251, 521
currentTimeMillis(), 505, 506, 507–508
CyclicBarrier class, 920, 922, 929–931, 934

D

Data types, 31. *See also* Type(s); Types, primitive
DatagramPacket class, 773, 775
 methods, list of some, 775
Datagrams, 760, 773–777
 server/client example, 775–777
DatagramSocket class, 730, 773–774, 776
DataInput interface, 706, 707, 708, 723
DataInputStream class, 317, 706, 678–708
DataOutput interface, 706, 708, 679, 721
DataOutputStream class, 317, 706–708
Date and time. *See* Time and date; Time and date API
Date class, 626–628, 1012, 1013
 methods, table of, 627
DateFormat class, 626, 636, 1011–1013, 1018
DateTimeFormatter class, 1017–1020
Deadlock, 270–271, 522, 1034
Decoder class, 672
Decrement operator (–), 34, 67, 70–71
decrementAndGet(), 921, 951
deepEquals(), 598–599
deepHashCode(), 601
deepToString(), 601
default
 clause for annotation member, 302–303
 to declare a default interface method, using, 220
 statement, 90–91
DefaultMutableTreeNode class, 1066, 1067

defaults Properties instance variable, 613
DelayQueue class, 947
Delegation event model, 784–785
 and Beans, 1112
 and event listeners, 784, 785, 797–800
 and event sources, 784–785, 796–797
 and Swing, 1035
 using, 800–808
delete operator, 133
delete(), 473, 683, 734
deleteCharAt(), 473
deleteOnExit(), 683
delimiter(), 666
Delimiters, 619–620
 Scanner class, 657, 665–666
@Deprecated built-in annotation, 305, 307
Deprecated, definition of the term, 11
Deque interface, 540, 549–550, 555, 560
 methods, table of, 549–550
descendingIterator(), 547, 549, 550
Deserialization and security, 720
destroy(), 498, 501, 522, 523
 and servlets, 1122, 1125, 1127
Dialog boxes, 888–892
Dialog class, 816, 888
Dialog windows, 1105
Diamond operator (<>), 380–381
Dictionary class, 538, 602, 609–610
 abstract methods, table of, 609
digit(), 494
Dimension class, 816, 820, 827
 reflection example using the, 1004–1005
Directories as File objects, 681, 683–684
 creating, 686
Directories and packages, file system, 200
Directory, listing the contents of a

using `list()`, 683–685
using `listFiles()`, 685–686
using NIO, 752–755
Directory tree, obtaining a list of files in a, 756–758
DirectoryStream<Path> class, 752, 753
DirectoryStream.Filter interface, 754
DISCARD, 505
`dispose()`, 888
`distinct()`, 994
do-while loop, 96–99
 and continue, 113
Document interface, 1045
@Documented built-in annotation, 305, 306
`doDelete()`, 1136, 1137
`doGet()`, 1136, 1137, 1138
`doHead()`, 1136, 1137
Domain name, 760, 761
Domain Naming Service (DNS), 760
`doOptions()`, 1136, 1137
`doPost()`, 1137, 1139
`doPut()`, 1137
DosFileAttributes class, 736, 737, 752
DosFileAttributeView interface, 738
Dot operator `(.)`, 85, 119, 125, 154, 180, 206, 223
`doTrace()`, 1137
Double buffering, 896–899
Double class, 287, 288, 478–483, 490
 methods, table of, 481–482
double data type, 39, 42–43, 46
 and automatic type conversion, 52
 and automatic type promotion, 54–55
DoubleAccumulator class, 951
DoubleAdder class, 951
DoubleBinaryOperator functional interface, 600
DoubleBuffer class, 729
`doubles()`, 638

DoubleStream interface, 972, 973
DoubleSummaryStatistics class, 672
doubleValue(), 288, 355, 478, 480, 481, 484, 485, 486, 488
drawArc(), 825
drawImage(), 895, 897, 898
drawLine(), 824, 1038
drawOval(), 825
drawPolygon(), 825–826
drawRect(), 824–825, 1038
drawRoundRect(), 825
drawString(), 804, 821, 822, 839, 840
Duration class, 947, 1021
Dynamic method
 dispatch, 188–191
 lookup, 211
 resolution, 208, 211, 212, 216

E

E (Math constant), 515
Early binding, 194
echoCharIsSet(), 865
Eclipse IDE, 1122, 1123
Edit control, 864
element(), 548
elementAt(), 604
elementCount Vector data member, 604
elementData Vector data member, 604
elements(), 604, 609, 610, 611
ElementType enumeration, 306, 535
ElementType.FIELD, 311
ElementType.METHOD, 311
ElementType.TYPE_USE, 308, 311
else, 87–90
empty(), 607, 608, 624, 626
EMPTY_LIST static variable, 595

EMPTY_MAP static variable, 595
EMPTY_SET static variable, 595
EmptyStackException, 607, 609
Encapsulation, 5, 22–23, 24, 26–27, 134, 177
 and access control, 149
 and scope rules, 50
Encoder class, 672
end(), 996–997
endsWith(), 458, 733
ensureCapacity(), 469, 553, 604
entrySet(), 571, 572, 575, 579, 612
enum, 278, 531, 561, 582
Enum class, 283, 531
 methods, table of, 531
enumerate(), 521, 523, 527
Enumeration interface, 602–603, 605–607, 610, 619, 620, 702
Enumeration(s), 16, 277–286, 531
 == relational operator and, 278, 254
 as a class type in Java, 277, 281–283
 constants, 277, 278, 279, 280, 281, 282, 283
 constructor, 281–283
 and JShell, 1161
 restrictions, 283
 values in switch statements, using, 278–280
 variable, declaring an, 278
EnumMap class, 578, 581
EnumSet class, 551, 561
 factory methods, table of, 562
Environment properties, list of, 509
equals(), 161, 197, 284–285, 295, 457, 479, 480, 481, 484, 485, 486, 488, 494, 497, 503, 510, 530, 531, 541, 543, 572, 577, 583, 586, 598, 610, 621, 624, 627, 629, 762, 834
 versus ==, 458–459
equalsIgnoreCase(), 457
equalsIgnoreOptional(), 503
Erasure, 349, 382–384

and ambiguity errors, 384–385
bridge methods and, 382–384
`err`, 318, 505. *See also* `System.err`
Error class, 228, 229, 237, 244, 718
Errors
 ambiguity, 384–385
 assertions to check for, using, 336–338
 autoboxing/unboxing and prevention of, 293–294
 automatic type promotions and compile-time, 54
 compile-time versus run-time, 352
 generics and prevention of, 350–352
 raw types and run-time, 372
 run-time, 13–14, 227, 333. *See also* Exception handling
 unreachable code, 115, 233

Event
 definition of an, 784
 design patterns for a Java Bean, 1112
 dispatching thread and Swing, 1034, 1038, 1074
 driven programs, 783, 1034
 listeners, 784–785, 797–800
 loop with polling, 248, 265
 model, delegation. *See* Delegation event model
 multicasting and unicasting, 785, 1112
 sources, 784–785, 796–797
 timestamp, 787

Event handling, 783–813. *See also* Delegation event model
 and adapter classes, 808–810
 event classes for, 785–796
 and immediate and quick event processing, 785, 800, 822, 1038
 and inner classes, 159, 810–813, 850, 851
 keyboard, 804–808
 and lambda expressions, 850–851, 1037–1038
 mouse, 799, 801–804
 and Swing, 786, 1026, 1035–1038

Event listener interfaces, 797–800
 and adapter classes, 808–810

table of commonly used, 797
EventListener interface, 672
EventListenerProxy class, 672
EventObject class, 672, 786
EventSetDescriptor class, 1112, 1114, 1116
Exception class, 228, 229, 241, 243, 244
Exception classes
 and generics, 387
 hierarchy of the built-in, 228–229
Exception handling, 14, 99, 109, 227–246, 326, 327–329
 block, general form of, 228
 and chained exceptions, 15, 244–245
 and creating custom exceptions, 241–243
 and the default exception handler, 229–230, 236
 and lambdas, 402–403
 and the more precise (final) rethrow feature, 245, 246
 multi-catch, 245–246
 and suppressed exceptions, 242, 332
 and uncaught exceptions, 229–230, 534
Exception(s)
 definition of the term, 227
 and JShell, 1163
Exceptions, built-in, 240–241
 checked, table of, 241
 run-time, constructors for, 237
 unchecked, table of, 240
Exceptions, I/O, 687
exchange(), 931, 932, 933
Exchanger class, 920, 922, 931–933
exec(), 495, 499, 501
execute(), 941, 955, 965
Execution point, 530
Executor interface, 921, 941
Executors, 920, 921
 using, 941, 943
Executors class, 921, 941

ExecutorService interface, 921, 941, 942, 944
exists(), 681, 734, 750
exit(). *See* System.exit()
exitValue(), 498, 501
exports keyword, 37, 420
exports statement, 420, 424, 426
 and qualified export, 428–429
Expression lambda, 395. *See also* Lambda expression(s)
Expressions
 and autoboxing/unboxing, 291–292
 automatic type promotion in, 54–55
 JShell to evaluate, using, 1162
 regular. *See* Regular expressions
extends, 171, 173, 218, 355, 360, 373
 and bounded wildcard arguments, 361, 364
Externalizable interface, 721, 1113

F

false, 38, 44, 45, 47, 81, 82, 83, 131
FALSE, 495
FAT file system, 736, 738, 752
Feature release, 12, 502
feature(), 502–503
Field class, 297, 300, 301, 536, 1004
Field, final, 154–155
fieldModifiers(), 1007
fields array (Calendar class), 628
File attribute(s)
 File to access, using, 681–686, 750
 interfaces, 736–738
 NIO to access, using, 737–738, 750–752
 view interfaces, 738
File class, 657, 681–686, 689, 693, 704, 708, 710, 718, 750
 instance into a Path instance, converting a, 683, 732, 750
 methods, 681–686, 690

File system directories and packages, 200
file(), 505
File(s)
 to a buffer, map a, 731, 742–743, 746–747
 close() to close a, using, 324, 326–328, 329, 332, 694
 I/O, 323–332, 681–686. *See also* NIO; NIO and channel-based I/O
 path to a, obtaining a, 732, 735–736, 739
 pointer, 708
 source. *See* Source file(s)
 system, accessing the, 738
 try-with-resources to automatically close a, using, 324, 329–333, 694
FileChannel class, 731, 739, 742, 743, 744, 746
FileDialog class, 816
FileFilter interface, 686
FileInputStream class, 317, 323–324, 689–692, 730, 731
FilenameFilter interface, 684–685, 686
FileNotFoundException, 324, 327, 687, 689, 693, 710
FileOutputStream class, 317, 323–324, 328, 693–695, 730, 731
FileReader class, 318, 657, 710–711
Files class, 681, 731, 732–735, 737, 738, 747, 748, 750, 751, 752, 756
 methods, table of a sampling of, 734–735
FileStore class, 738
FileSystem class, 738
FileSystems class, 738
FileVisitor interface, 756–757
FileVisitResult enumeration, 756
FileWriter class, 318, 711–712
fill()
 algorithm defined by Collections, 592
 Arrays method, 599
fillArc(), 825
fillOval(), 825
fillPolygon(), 825–826
fillRect(), 824–825
fillRoundRect(), 825
filter(), 624, 626, 971, 972, 976–977, 984

FilteredImageSource class, 899, 904
FilterInputStream class, 317, 698, 706
FilterOutputStream class, 317, 698, 706
FilterReader class, 318
FilterWriter class, 318
final, 154–155, 245, 246
 to prevent class inheritance, 195
 to prevent method overriding, 194
finalize(), 197, 510
finally block, 227, 228, 238–239, 326–327, 687
find(), 733, 996, 998–999
findAll(), 667
findInLine(), 666–667
findWithinHorizon(), 667
Finger protocol, 766
first(), 546, 876
firstElement(), 604
firstKey(), 575
flatMap(), 624, 626, 986
flatMapToDouble(), 986
flatMapToInt(), 986
flatMapToLong(), 986
flip(), 621, 728, 745
Float class, 287, 288, 478–481, 482–483, 490
 methods, table of, 480–481
float data type, 39, 42, 46
 and type promotion, 54–55
Floating-point(s), 39, 42–43
 literals, 46–47
 strictfp and, 335
FloatBuffer class, 729
floatValue(), 288, 478, 480, 481, 484, 485, 486, 488
Flow class, 921
Flow subsystem to control data flow, 921
Flow.Processor interface, 921
Flow.Publisher interface, 780, 921

Flow.Subscriber interface, [921](#)
Flow.Subscription interface, [921](#)
FlowLayout class, [817](#), [846](#), [868](#), [869–870](#), [1036–1037](#)
flush(), [643](#), [686](#), [691](#), [700](#), [710](#), [719](#), [721](#), [722](#)
Flushable interface, [686](#), [689](#), [693](#), [703](#), [706](#), [710](#), [717](#), [718](#)
FocusAdapter class, [809](#)
FocusEvent class, [787](#), [789–790](#)
FocusEvent.Cause enumeration, [790](#)
focusGained(), [798](#)
FocusListener interface, [797](#), [798](#), [809](#)
focusLost(), [798](#)
followRedirects(), [778](#)
Font class, AWT, [817](#), [834](#), [835](#), [836](#), [838](#)
 methods, table of some, [834](#)
Font(s), [833–839](#)
 creating and selecting, [836–838](#)
 determining available, [835–836](#)
 information, obtaining, [838–839](#)
 metrics to manage text output, using, [839–842](#)
 terminology used to describe, [839](#)
FontMetrics class, [817](#), [839–842](#)
 methods, table of some, [840](#)
for loop, [33–34](#), [44](#), [99–108](#)
 and continue, [113](#)
 enhanced. *See* For-each version of the for loop
 and local variable type inference, [107–108](#)
 variations, [102–103](#)
For-each version of the for loop, [16](#), [99](#), [103–107](#)
 and arrays, [103–107](#)
 and the break statement, [104–105](#)
 and collections, [103](#), [107](#), [540](#), [565–566](#)
 general form, [103](#)
 and the Iterable interface, [533](#), [540](#), [565](#)
 and maps, [570](#)
forceTermination(), [941](#)
forDigit(), [494](#)

forEach(), 533, 563, 971, 972, 976, 981
forEachOrdered(), 981
forEachRemaining(), 562, 563, 566–568, 992
Fork/Join Framework, 17–18, 249, 276, 673, 919–920, 921, 941, 952–968
 advantages to using the, 952, 968
 classes, main, 952–956
 tips for using the, 968
Fork/Join Framework divide-and-conquer strategy, 954, 956–959, 968
 and the sequential processing threshold interaction with the level of parallelism, 959–962
Fork/Join Framework tasks, 953
 asynchronous execution of, 965
 cancelling, 965
 completion status of, 966
 and the parallelism level, 955, 968
 restarting, 966
 starting, 955, 965
 and subtasks, 955, 956
 tags, 967
 that do not return a result, 954, 958, 962
 that return a result, 954, 962–965
fork(), 953, 955, 959, 963, 964, 966
ForkJoinPool class, 921, 941, 952, 953, 954–956, 957, 958, 959, 962, 965, 967–968
 common pool, 954, 955, 959, 962, 968
 and work stealing, 955, 967
ForkJoinTask class, 921, 952, 953, 954, 955, 958, 959, 965, 966–967, 968
ForkJoinTask<T> class, 956
Format flags, 652–654
Format specifiers (conversions), 642, 643–656
 argument index with, using an, 655–656
 and format flags, 652–654
 and specifying minimum field width, 649–650
 and specifying precision, 651
 suffixes for the time and date, table of, 647–648
 table of, 644

uppercase versions of, 654–655
format(), 467, 643–645, 655, 705, 706, 718, 719, 1012, 1017
FormatStyle enumeration, 1018
Formattable interface, 672
FormattableFlags class, 672
Formatted input, using Scanner to read, 657–667
Formatter class, 642–657, 704, 705. *See also* Format specifiers
 closing an instance of the, 656–657
 constructors, 642–643
 methods, table of, 643
 forName(), 512, 1004
FORTRAN, 4, 5
Forward reference, 1159
Frame class, 796, 797, 801, 803, 817, 818, 819, 821, 870
Frame window(s), 819–824
 and creating a Frame-based application, 823–824
 default layout manager for, 870
 methods used to work with a, 820–822
 requesting repainting of a, 822–823
Frank, Ed, 6
freeMemory(), 498, 499, 500
from(), 505, 627, 632
FTP (File Transfer Protocol), 760, 766
Function<T,R> predefined functional interface, 417, 584, 654, 982
Functional interfaces, 18, 307, 389, 390, 391–392, 394, 851
 and their abstract methods, table of, 674–676
 generic, 397–399
 and lambda expressions passed as arguments, 399–401
 predefined, 416–417
 and public Object methods, 390
@FunctionalInterface built-in annotation, 305, 307
Future interface, 921, 944–946

G

Garbage collection, 14, 133, 147, 497–498, 535, 1122

`gc()`, 498, 499, 500
Generic class
 and casting, 378
 definition of the term, 346
 example program with one type parameter, 346–350
 example program with two type parameters, 353–354
 general form of a, 354
 hierarchies, 372–380
 and instanceof, 376–378
 overriding methods in a, 379–380
 and raw types, 370–372
 and type inference, 380–381
Generic constructors, 367
Generic interfaces, 346, 368–370
 and classes, 369
Generic method, 346, 358, 364–367, 386
Generics, 16, 289, 345–387
 and annotations, 314
 and ambiguity errors, 384–385
 and arrays, 386–387
 and casts, 346, 349, 350, 352
 and the Collections Framework, 345
 and compatibility with pre-generics code, 370–372, 382
 and exception classes, 387
 restrictions when using, 385–387
 type checking and safety, 349, 350–352, 371, 387
GenericServlet class, 1125, 1127, 1130, 1135
`get()`, 543, 544, 556, 571, 572, 578, 609, 610, 611, 621, 624, 625, 629, 676, 735–736, 738–739, 743, 772, 944, 946–947, 951, 975, 976, 989
 and buffers, 729, 730, 741
`getActionCommand()`, 787, 847, 859, 1048, 1054, 1082
`getActiveThreadCount()`, 967
`getAddListenerMethod()`, 1116
`getAddress()`, 762, 775
`getAdjustable()`, 788
`getAdjustmentType()`, 788, 862

`getAlignment()`, 845
`getAllByName()`, 761, 762
`getAllFonts()`, 835
`getAndSet()`, 921, 951
`getAnnotation()`, 297, 301, 312–313, 512, 527, 529
`getAnnotations()`, 300–301, 512, 527
`getAnnotationsByType()`, 301, 313, 512, 527
`getArrivedParties()`, 941
`getAsDouble()`, 626
`getAscent()`, 840, 841
`getAsInt()`, 626
`getAsLong()`, 626
`getAttribute()`, 1128, 1129, 1135, 1142
`getAttributeNames()`, 1135, 1142
`getAvailableFontFamilyNames()`, 835
`getBackground()`, 822
`getBeanInfo()`, 1116
`getBlue()`, 830
`getButton()`, 794
`getByAddress()`, 762
`getByName()`, 761, 762
`getBytes()`, 456, 693
`getCause()`, 242, 244–245, 790
`getChannel()`, 730–731
`getChars()`, 455–456, 471, 711
`getChild()`, 789
`getClass()`, 197, 296–297, 348, 510, 512, 514–515, 1006
`getClassLoader()`, 529
`getClickCount()`, 793
`getColor()`, 830
`getCommonPoolParallelism()`, 962
`getComponent()`, 789, 1088, 1089, 1090
`getConstructor()`, 297, 513
`getConstructors()`, 513, 1004
`getContainer()`, 789
`getContentPane()`, 1033–1034

`getContents()`, 670
`getContentType()`, 768, 769
`getCookies()`, 1133, 1141
`getDate()`, 768, 769
`getDateInstance()`, 1011–1012
`getDateTimeInstance()`, 1013
`getDeclaredAnnotation()`, 301, 528
`getDeclaredAnnotations()`, 301, 513, 528, 529
`getDeclaredAnnotationsByType()`, 301, 313, 513, 528
`getDeclaredMethods()`, 513, 1006
`getDefault()`, 633, 636
`getDescent()`, 840, 841
`getDescriptor()`, 529
`getDirectionality()`, 494
`getDisplayCountry()`, 635
`getDisplayLanguage()`, 635
`getDisplayName()`, 635, 641
`getEchoChar()`, 865
`getEventSetDescriptors()`, 1112, 1119
GetField inner class, 723–724
`getField()`, 297, 513
`getFields()`, 513, 1004
`getFileAttributeView()`, 738
`getFirst()`, 549, 555
`getFont()`, 834, 838, 840
`getForeground()`, 822
`getForkJoinTaskTag()`, 967
`getFreeSpace()`, 683
`getGraphics()`, 822, 896
`getGreen()`, 830
`getHeaderField()`, 769
`getHeaderFields()`, 769, 772
`getHeight()`, 840, 841, 1040
`getHostAddress()`, 763
`getHostName()`, 763
`getHour()`, 1020

`getIcon()`, 1044
`getID()` AWTEvent class method, 786
`getInetAddress()`, 764, 774
`getInitParameter()`, 1128
`getInitParameterNames()`, 1128
`getInputStream()`, 498, 501, 764, 769
`getInsets()`, 828, 872, 1039–1040
`getInstance()`, 629, 631, 641
`getISOCountries()`, 636
`getItem()`, 791, 856, 859, 884, 1050, 1052
`getItemCount()`, 856, 859
`getItemSelectable()`, 792, 859
`getKey()`, 577, 579
`getKeyChar()`, 792
`getKeyCode()`, 792
`getKeyStroke()`, 1083
`getLabel()`, 847, 851, 883–884
`getLast()`, 549, 555
`getLayer()`, 529
`getLeading()`, 840, 841
`getListenerType()`, 1116
`getLocalGraphicsEnvironment()`, 835
`getLocalHost()`, 761, 762
`getLocalPort()`, 764, 774
`getLocationOnScreen()`, 794
`getLogger()`, 506, 509
`getMaximum()`, 862
`getMenuComponentCount()`, 1077
`getMenuComponents()`, 1077
`getMenuCount()`, 1075
`getMessage()`, 237, 242
`getMethod()`, 297, 299, 513, 1116, 1133
`getMethodDescriptors()`, 1112
`getMethods()`, 514, 1004
`getMinimum()`, 862
`getMinimumSize()`, 869

getModifiers(), 787, 791, 1006
getModifiersEx(), 791
getModule(), 512, 514, 529
getMonth(), 1020
getN() getter method design pattern, 1110, 1111
getName(), 250, 252, 348, 514, 521, 523, 528, 529, 681, 732, 733, 750, 834,
1006, 1116, 1136, 1141
getNameCount(), 732, 733
getNestHost(), 515
getNestMembers(), 515
getNewState(), 796
getObjectInputFilter(), 724
getOldState(), 796
getOppositeComponent(), 790
getOppositeWindow(), 796
getOutputStream(), 498, 501, 764, 1129
getPackages(), 529
getParallelism(), 962
getParameter(), 1129, 1131, 1138, 1139
getParameterNames(), 1129, 1131
getParent(), 523, 681, 733, 750, 941
getPath(), 1066, 1136
getPhase(), 935, 937
getPoint(), 793
getPoolSize(), 967
getPort(), 764, 774, 775
getPreciseWheelRotation(), 795
getPreferredSize(), 869
getPriority(), 250, 261, 521
getProperties(), 506, 613
getProperty(), 506, 509, 614–615
getPropertyDescriptors(), 1112, 1113, 1118, 1119
getQueuedTaskCount(), 966
getRed(), 830
getRegisteredParties(), 941
getRemoveListenerMethod(), 1116

`getResourceAsStream()`, 529
`getRGB()`, 830
`getRuntime()`, 496–497, 499
`getScript()`, 636
`getScrollAmount()`, 795
`getScrollType()`, 795
`getSecurityManager()`, 506, 529
`getSelectedCheckbox()`, 854
`getSelectedIndex()`, 856, 858, 1061
`getSelectedIndexes()`, 859
`getSelectedItem()`, 856, 858, 1064
`getSelectedItems()`, 859
`getSelectedText()`, 864–865, 867
`getSelectedValue()`, 1061
`getServletConfig()`, 1127
`getServletContext()`, 1128
`getServletInfo()`, 1127
`getServletName()`, 1128
`getSession()`, 1133, 1142
`getSize()`, 820, 827, 834
`getSource()`, 786, 849, 1054
`getStackTrace()`, 242, 521, 530
`getState()`, 274–275, 521, 851, 884
`getStateChange()`, 792, 859
`getSubElements()`, 1076
`getSuperclass()`, 514–515
`getSuppressed()`, 242, 332
`getSurplusQueuedTaskCount()`, 966–967
`getText()`, 845, 864, 867, 1044, 1046, 1047, 1052
`getTimeInstance()`, 1012–1013
`getUnarrivedParties()`, 941
`getTotalSpace()`, 683
`getUsableSpace()`, 683
`getValue()`, 577, 579, 788, 861, 1094–1095, 1136, 1141
`getWheelRotation()`, 794–795
`getWhen()`, 787

`getWidth()`, [1040](#)
`getWindow()`, [796](#)
`getWriter()`, [1125](#), [1129](#)
`getX()`, [793](#), [1088](#), [1090](#)
`getXOnScreen()`, [794](#), [1088](#), [1090](#)
`getY()`, [793](#)
`getYear()`, [1020](#)
`getYOnScreen()`, [794](#)
GIF image format, [893](#)–[894](#)
Glass pane, [1029](#), [1030](#)
Glassfish, [1122](#)
Glob, [754](#)
Gosling, James, [6](#)
goto keyword, [37](#)
Goto statement, using labeled break as form of, [111](#)–[113](#)
`grabPixels()`, [901](#), [902](#)
Graphical User Interface. *See* GUI (Graphical User Interface)
Graphics
 context, [804](#), [821](#), [823](#), [824](#)
 sizing, [827](#)–[829](#)
Graphics class, [804](#), [817](#), [821](#), [824](#), [830](#), [838](#), [895](#), [896](#)
 drawing methods, [824](#)–[827](#)
Graphics2D class, [824](#)
GraphicsEnvironment class, [817](#), [835](#)
Greedy behavior (regular expression pattern matching), [1000](#), [1001](#)
GregorianCalendar class, [628](#), [631](#)–[633](#), [636](#), [1015](#)
Grid bag layouts, [878](#)–[883](#)
GridBagConstraints class, [817](#), [878](#)–[880](#)
 constraint fields, table of, [879](#)
GridLayout class, [817](#), [879](#), [880](#), [883](#)
gridheight constraint field, [879](#), [880](#)
GridLayout class, [817](#), [874](#)–[875](#)
gridwidth constraint field, [879](#), [880](#)
group() Matcher class method, [996](#)
GIU (Graphical User Interface), [315](#), [815](#), [843](#)
 frameworks, Java's three, [800](#)–[801](#)

programs, handling events generated by, 783–813, 822
GZIP file format, 677

H

hasCharacteristics(), 567, 568
Hash code, 556, 610
Hash table, 556, 610
hashCode(), 197, 295, 480, 481, 484, 485, 486, 488, 494, 497, 510, 528, 530, 531, 541, 572, 577, 601, 610, 621, 624, 627, 834
Hashing, 556, 557
HashMap class, 578–580, 581, 582, 610
HashSet class, 551, 556–557, 578, 973
 from a stream API stream, obtaining a, 989
Hashtable class, 551, 601, 610–613
 and iterators, 612–613
 legacy methods, table of, 611
hasMoreElements(), 603, 620
hasMoreTokens(), 620
hasNext(), 562, 563, 566, 990, 991
hasNextX() Scanner methods, 658, 661
 table of, 659
Headers, 769
headers(), 779–780
HeadlessException, 820, 845
headMap(), 574, 575, 576
headSet(), 546, 547
Hexadecimals, 45, 47
 as character values, 47
 and string literals, 48
Hierarchical abstraction and classification, 22
 and inheritance, 23, 171
High surrogate char, 495
Histogram, 902–903
Hoare, C.A.R., 250
Holzmann, Gerard J., 900
HotSpot technology.10

HSB (hue-saturation-brightness) color model, 829
HSBtoRGB(), 830
HTML (Hypertext Markup Language), 1121, 1125
 and javadoc, 1147, 1148, 1153
HTTP, 760, 767
 GET requests, handling, 1137–1138
 and HttpURLConnection class, 770
 port, 760
 POST requests, handling, 1137, 1139–1140
 requests, 1121, 1122, 1132–1133, 1137
 response, 1121, 1122, 1125, 1132–1133, 1134
 and URLConnection class, 768
HTTP Client API, 19, 759, 777–782
 and asynchronous communication, 777, 782
 and bidirectional communication, 777, 782
HTTP session
 stateful, 772
 tracking, 1142–1143
HttpClient class, 777–778
HttpClient.Builder interface, 778
HttpClient.Redirect enumeration, 778
HttpCookie class, 772
HttpHeaders class, 780
HttpRequest class, 777, 779
HttpRequest.Builder, 779
HttpResponse interface, 777, 778, 779–780
HttpResponse.BodyHandler interface, 780
HttpResponse.BodyHandlers class, 778, 780
HttpServlet class, 1132, 1135, 1137
 methods, table of, 1136–1137
HttpServletRequest interface, 1132, 1138, 1139, 1142
 methods, table of several, 1133
HttpServletResponse interface, 1132–1133, 1134
 methods, table of, 1134
HttpSession interface, 1132, 1133, 1142

methods, table of several, 1135
HttpURLConnection class, 770–772, 777
methods, sampling of, 771

I

Icon interface, 1044
Icons
 Swing button, 1047, 1091
 Swing label, 1044
Identifiers, 29, 36, 38, 48, 49
IdentityHashMap class, 578, 582
if statement, 32–33, 34, 44, 45, 87–90
 Boolean object used to control the, 293
 boolean variable used to control the, 88, 293
 nested, 89
 and recursive methods, 148
 switch statement versus, 94–95
if-else-if ladder, 89–90
IllegalAccessException, 237, 241
IllegalArgumentException, 240, 541–542, 543, 546, 548, 550, 561, 571, 574, 575, 598
IllegalFormatException, 645
IllegalStateException, 240, 542, 548, 550, 996, 997, 1133
Image class, AWT, 817, 893, 894, 895, 896, 899, 901
ImageConsumer interface, 901–903, 904
ImageFilter class, 904, 906
ImageIcon class, 1043, 1044, 1092
ImageIO class, 894
Image observer, 895
ImageObserver interface, 895
ImageProducer interface, 894, 899, 901, 904
Images (AWT), 893–918
 creating, loading, displaying, 894–896
 double buffering and, 896–899
 file formats for web, 893–894

filters for, 904–918
stream model for, 904

Imaging, 893

implements clause, 210
and generic interfaces, 369, 370

import statement, 206–207, 1163
and static import, 339–341

in, 318, 501, 505, 657, 718. *See also* System.in

Increment operator (++), 34, 67, 70–72

indexOf(), 460–462, 474, 475, 543, 544, 604

IndexOutOfBoundsException, 240, 543

Inet4Address class, 763

Inet6Address class, 763

InetAddress class, 761–763, 774

InetSocketAddress class, 774

infinity (IEEE floating-point specification value), 482

inForkJoinPool(), 966

INHERIT, 505

InheritableThreadLocal class, 527

Inheritance, 5, 22, 23–24, 26–27, 150, 152, 171–195
and annotations, 314
and enumerations, 283
final and, 194–195
and interfaces, 199, 208, 218–219, 222–223, 224
and local variable type inference, 195–196
multilevel, 181–184
and multiple superclasses, 173, 199

@Inherited built-in annotation, 305, 306

init() and servlets, 1122, 1125, 1127

initCause(), 242, 244

Inline method calls, 194

Inner classes, 157–159, 810–813
anonymous, 159, 812–813, 850, 851, 940, 1037, 1054, 1075, 1089–1090

InputEvent class, 787, 790–791, 792, 793, 1083

InputMismatchException, 661

InputStream class, 316, 317, 319, 657, 688, 689, 690, 695, 698, 699, 701, 702,

706, 723, 726, 748, 780, 781, 1130
methods, table of, 690
objects, concatenating, 702
InputStreamReader class, 318, 319
insert(), 472, 867, 1076
insertSeparator(), 1076
Insets
 definition of the term, 827–828
 example program, 872–873
Insets class, 817, 828, 872–873, 1039
Instance of a class, 23, 117, 119, 122. *See also* Object(s)
Instance variables
 accessing, 119, 124, 125, 128
 default values of, 131
 definition of the term, 23, 118
 hiding, 133
 and interfaces, 220
 static, 153–154
 transient, 333
 as unique to their object, 119, 120–121
 using super to access hidden, 180–181
instanceof operator, 67, 333–335, 570
 and generic classes, 376–378
Instant class, 627, 1021
Instrumentation interface, 535
int, 31, 39, 40, 41
 and automatic type conversion, 52
 and automatic type promotion, 54–55, 75–76, 78–79
 and integer literals, 45
IntBuffer class, 729
Integer class, 287, 288, 289, 292, 477, 483, 490–491, 975
 constructors, 288
 methods, table of, 486–488
Integer(s), 39, 40–42, 72–73
 literals, 45–46, 52
interface keyword, 199, 208

and annotations, 295

Interface methods

- default, 18, 19, 209, 219–223, 389, 391
- extension, 219
- private, 19, 209, 224–225
- static, 209, 223–224
- traditional, 208, 210, 391

Interface(s), 199, 208–225

- functional. *See* Functional interfaces
- general form of, 209
- generic. *See* Generic interfaces
- implementing, 210–212
- inheritance hierarchy, 208
- inheritance of, 218–219, 223
- and JShell, 1160–1161
- member, 212
- methods. *See* Interface methods
- nested, 212–213
- reference variables, 210–212, 216
- and state information, 220, 223
- types for bounded types, using, 357
- variables, 209, 216–218

interfaceModifiers(), 1007

interim(), 502–503

Internet, 3, 6, 7, 8, 9, 15, 20, 759

- addresses, obtaining, 762–763
- addressing scheme, 760
- and portability, 7, 8, 9–10
- and security, 8, 9

Internet Engineering Task Force (IETF) BCP 49, 636

Internet of Things (IoT), 419

Internet Protocol (IP)

- addresses, 760
- definition of the term, 759

InterNIC, 764, 765

InterruptedException, 241, 251–252, 902

Introspection, 1110–1113, 1116, 1118–1119
Introspector class, 1115, 1116
ints(), 638
IntStream interface, 972, 973, 985
IntSummaryStatistics class, 672
intValue(), 288, 289, 478, 480, 481, 484, 485, 486, 489
InvalidPathException, 735
invoke(), 953, 955, 959, 965, 967
invokeAll(), 953, 959, 963, 966
invokeAndWait(), 1034, 1035
invokeLater(), 1034
I/O, 30, 315–332, 679–726
 channel-based, 15, 316, 727. *See also* NIO; NIO and channel-based I/O
 classes, list of, 680
 console, 30, 99, 315, 319–323, 718–720
 error handling, 326–329
 exceptions, 687
 file, 323–332, 681–686
 formatted. *See* I/O, formatted
 interfaces, list of, 680
 and Java’s GUI frameworks, 315
 new. *See* NIO
 redirection, 505
 streams. *See* Streams, I/O
I/O, formatted, 16
 format specifiers. *See* Format specifiers
 using Formatter, 642–657. *See also* Formatter class
 using printf(), 163, 704–706
 using Scanner, 657–667. *See also* Scanner class
io package. *See* java.io package
IOException, 99, 319, 324, 327, 328, 687, 689, 690, 694, 701, 709, 710, 711, 717, 722, 723, 733, 752, 756, 764, 768, 773
ipadx constraint field, 879, 880
ipady constraint field, 879, 880
IPv4 (Internet Protocol, version 4), 760, 761, 762, 763

IPv6 (Internet Protocol, version 6), [760](#), [761](#), [762](#), [763](#)
isAbsolute(), [682](#), [733](#)
isAlive(), [250](#), [258–260](#), [496](#), [498](#), [521](#)
isAltDown(), [791](#)
isAltGraphDown(), [791](#)
isAnnotationPresent(), [301](#), [303](#), [528](#)
isBound(), [764](#), [774](#), [1116](#)
isCancelled(), [965](#), [966](#)
isClosed(), [764](#)
isCompletedAbnormally(), [966](#)
isCompletedNormally(), [966](#)
isConnected(), [764](#), [774](#)
isConstrained(), [1116](#)
isControlDown(), [791](#)
isDigit(), [492](#), [493](#), [495](#)
isDirectory(), [683–684](#), [734](#), [737](#)
isEditable(), [865](#), [867](#)
isEmpty(), [467](#), [541](#), [542](#), [572](#), [605](#), [609](#), [610](#), [611](#), [622](#), [625](#)
isEnabled(), [883](#), [1095](#)
isExecutable(), [734](#), [750](#)
isExported(), [529](#)
isFile(), [682](#)
isHidden(), [683](#), [734](#), [736](#), [738](#), [750](#)
isInfinite(), [480](#), [482–483](#)
isLeapYear(), [632](#)
isLetter(), [492](#), [493](#), [495](#)
isLightweight(), [892](#)
isLowercase(), [492](#), [493](#)
isMetaDown(), [791](#)
isMulticastAddress(), [763](#)
isNamed(), [529](#)
isNaN(), [480](#), [482–483](#)
isNestMateOf(), [515](#)
ISO-Latin-1 character set, [43](#), [47](#)
isOpen(), [529](#)
isPopupTrigger(), [793](#), [1088](#), [1090](#)

isPresent(), 625, 975
isPropertyName(), 1111
isPublic(), 1006
isQuiescent(), 967
isReadable(), 734, 750
isSelected(), 1050, 1052, 1054, 1076
isSet array (Calendar class), 628
isShiftDown(), 791
isShutdown(), 968
isTemporary(), 790
isTerminated(), 968
isTimeSet (Calendar class instance variable), 628
isUppercase(), 492, 493
isWhitespace(), 492, 493
isWritable(), 734, 738, 750
ItemEvent class, 787, 791–792, 852, 856, 859, 884, 1050, 1052
ItemListener interface, 797, 798, 851, 852, 856, 884, 1050, 1052
ItemSelectable interface, 792
itemStateChanged(), 798, 852, 856, 1050, 1052
Iterable interface, 467, 533, 540, 565, 570, 603
Iterable<Path> interface, 732, 753
Iteration statements, 87, 95–106
Iterator interface, 539, 540, 561, 563–565, 566, 602, 603, 990, 991
 methods, table of, 562
iterator(), 533, 541, 543, 563, 753, 970, 990
Iterator(s), 539, 543, 561–565, 566–568
 and maps, 570
 obtaining an, 563
 and primitive types, 539
 and PriorityQueue, 560
 and stream API streams, 990–993
 and synchronized collections, 595

J

J2SE 5, features added by, 16

JApplet class, [1029](#)
JAR files. *See* Java Archive (JAR) files
jar tool, [445](#)
Java
 and C, [3, 5, 7, 13](#)
 and C++, [3, 7–8, 13](#)
 and C#, [8](#)
 design features (buzzwords), [12–15](#)
 history of, [3, 6–8, 15–20](#)
 and the Internet, [3, 6, 7–10, 14, 20, 759](#)
 as an interpreted language, [10, 14](#)
 keywords. *See* Keywords
 look and feels, [1027](#)
 open-source implementations of, [20, 27](#)
 release schedule, [11–12, 19](#)
 as a strongly typed language, [13, 39, 45](#)
 versions of, [15–20, 502–503](#)
 and the World Wide Web, [7, 13](#)
Java Archive (JAR) files, [446, 673](#)
 modular, [445](#)
Java Beans, [514, 536, 995, 1004, 1109–1119](#)
 advantages of, [1110](#)
 API, [1114–1116](#)
 customizers, [1113](#)
 definition of the term, [1109](#)
 demonstration program, [1116–1119](#)
 events, design pattern for, [1112](#)
 introspection, [1110–1113, 1116, 1118–1119](#)
 persistence, [1113](#)
 properties. *See* Property, Java Bean
 serialization, [1113](#)
Java Community Process (JCP), [20](#)
Java EE SDK, [1122, 1126](#)
.java filename extension, [27, 1165](#)
Java Foundation Classes (JFC), [1026](#)
java (Java application launcher), [28, 29, 201, 324, 446, 1165, 1166](#)

and main(), 29
and modules, 419, 425, 446
Java Native Interface (JNI), 336
java package, 201, 207
Java Runtime Environment (JRE), 10, 11, 427
Java SE 7, 17–18
Java SE 8, 18
Java SE 9, 18–19
Java SE 10, 19
Java SE 11, 19
Java Virtual Machine (JVM), 10, 11, 14, 15, 20, 28, 29, 497, 520, 535
Java Web Start, 11, 19
java.awt package, 783, 786, 816, 894, 1036
 classes, tables of some, 816–818
java.awt.Dimension class, reflection example using the, 1004–1005
java.awt.event package, 783, 785, 786, 797, 808, 1035, 1036
 adapter classes, table of commonly used, 809
 event classes, table of commonly used, 787
 interfaces, table of commonly used, 797
java.awt.event.InputEvent class. *See* InputEvent class
java.awt.event.KeyEvent class. *See* KeyEvent class
java.awt.image package, 893, 899, 904, 915, 918
java.base module, 427, 477, 534, 537, 673, 727, 761, 783, 816, 920, 970, 995, 1004, 1011, 1016
java.beans package, 1112, 1114–1116
 classes, table of, 1114–1115
 interfaces, table of, 1114
java.desktop module, 427, 783, 894, 1030, 1114
java.instrument module, 535
java.io package, 315, 316–318, 324, 330, 679–680, 686, 727, 750
 automatically imported by JShell, 1163
 classes, list of, 680. *See also* Streams, byte; Streams, character
 interfaces, list of, 680
java.io.Externalizable interface, 721, 1113
java.io.IOException. *See* IOException
java.io.Serializable interface, 1113. *See also* Serializable interface

java.lang package, 207, 240, 296, 305, 318, 324, 330, 366, 369, 449, 427, 477–536, 686
 classes and interfaces, list of, 477
 implicit importation of the, 207, 477
 subpackages, 534–536

java.lang.annotation package, 295, 305, 312, 534, 535

java.lang.annotation.RententionPolicy enumeration, 295, 535

java.lang.image package, 901

java.lang.instrument package, 534, 535

java.lang.invoke package, 534, 535

java.lang.management package, 534, 535

java.lang.module, 529, 534, 535

java.lang.ref package, 535

java.lang.reflect package, 296, 301, 535–536, 995, 1004
 classes, table of, 1004

java.management module, 535

java.net package, 759, 772, 777
 classes and interfaces, list of, 760–761

java.net.http, 19, 759, 777–782

java.nio package, 316, 679, 727, 728

java.nio.channels package, 727, 730, 732

java.nio.channels.spi package, 727

java.nio.charset package, 727, 731

java.nio.charset.spi package, 727

java.nio.file package, 683, 727, 732

java.nio.file.attribute package, 727, 732, 736

java.nio.file.spi package, 727, 732

java.nio.file.WatchService, 758

java.prefs module, 676

java.rmi module, 1008

java.rmi package, 995, 1008

java.text package, 995, 1011

java.time package, 628, 995, 1015, 1016, 1021

java.time.Duration, 947, 1021

java.time.format package, 1015, 1017, 1018

java.time.temporal.ChronoUnits, 947

java.util package, 434, 537–538, 602, 619, 783, 786, 975, 990
automatically imported by JShell, 1163
classes, list of top-level, 537–538
interfaces defined by, list of, 538
subpackages, 673–677

java.util.concurrent package, 673, 920–921, 946, 952

java.util.concurrent.atomic package, 673, 920, 921, 950, 951

java.util.concurrent.locks package, 673, 920, 921, 948, 949, 950

java.util.function package, 18, 417, 533, 566, 584, 600, 619, 673, 976, 977, 982, 989
functional interfaces defined by, table of, 674–676

java.util.jar package, 673

java.util.List class. *See* List class (java.util)

java.util.logging package, 673, 676

java.util.prefs package, 673, 676

java.util.regex package, 673, 677, 995

java.util.spi package, 673, 677

java.util.stream package, 18, 673, 677, 970, 986, 994

java.util.zip package, 673, 677

java.xml module, 427

JAVA_HOME environmental variable, 444

javac (Java compiler), 27–28, 201, 308, 372, 1156, 1158, 1165, 1166
and modules, 419, 420, 424–425, 446
and multi-module compilation mode, 429

javadoc, 19, 1147–1153
tags, 1147–1153

JavaFX, 19, 315, 801

javap, 383

javax.imageio package, 894, 918

javax.servlet package, 1125, 1126–1130, 1132
interfaces and classes, list of core, 1126–1127

javax.servlet.http package, 1126, 1132–1137
interfaces and classes, list of some, 1132

javax.swing package, 1028, 1030, 1032, 1033, 1043, 1066
classes, list of, 1028–1029

javax.swing.event package, 1035, 1046, 1061, 1066

javax.swing.table package, [1069](#)
javax.swing.tree package, [1066](#)
JButton class, [1029](#), [1037](#), [1043](#), [1047–1049](#), [1074](#), [1077](#), [1095](#)
JCheckBox class, [1043](#), [1047](#), [1050](#), [1052–1053](#), [1095](#)
JCheckBoxMenuItem class, [1074](#), [1085](#), [1086–1087](#), [1095](#)
JComboBox class, [1043](#), [1063–1065](#)
JComponent class, [1028](#), [1029](#), [1038](#), [1039](#), [1043](#), [1047](#), [1075](#), [1085](#)
JDialog class, [1029](#), [1105](#)
JFormattedTextField class, [1105](#)
JFrame class, [1029](#), [1030–1031](#), [1032](#), [1033–1034](#), [1042](#)
 adding a menu bar to a, [1076](#), [1078](#), [1081](#)
 content pane layout manager, [1033](#), [1036–1037](#)
JIT (Just-In-Time) compiler, [10](#), [14](#)
JLabel class, [1029](#), [1030](#), [1031](#), [1033](#), [1035](#), [1038](#), [1043–1045](#), [1081](#)
JLayeredPane class, [1029](#)
jlink tool, [11](#), [18](#), [444–445](#)
JList class, [1043](#), [1060–1063](#)
JMenu class, [1074](#), [1075](#), [1076–1077](#), [1078](#)
 mnemonic, [1082](#)
JMenuBar class, [1074](#), [1075–1076](#), [1078](#)
JMenuItem class, [1074](#), [1075](#), [1076](#), [1077](#), [1078](#), [1085](#), [1086](#)
 accelerator key, [1083](#), [1084](#)
 action to create a, using an, [1095](#)
 and action events, [1077](#), [1078](#), [1081](#), [1082](#)
 mnemonic, [1082](#), [1083–1084](#)
JMOD files, [18](#), [444](#), [445](#)
join(), [250](#), [258–260](#), [466–467](#), [522](#), [947](#), [953](#), [963](#), [964](#), [967](#)
 JOptionPane class, [1105](#)
Joy, Bill, [6](#)
 JPanel class, [1029](#), [1030](#), [1040](#), [1042](#), [1056](#), [1058](#)
JPEG image file format, [893](#)
 JPopupMenu class, [1074](#), [1087–1090](#), [1095](#)
 and mouse events, [1088](#), [1089–1090](#)
 JButton class, [1043](#), [1047](#), [1050](#), [1053–1055](#), [1095](#)
 JRadioButtonMenuItem class, [1074](#), [1086–1087](#), [1095](#)
 JRootPane class, [1029](#)

JScrollBar class, [1029](#)
JSeparator class, [1074](#), [1076](#)
JScrollPane class, [1043](#), [1058–1060](#), [1061](#), [1066](#), [1067](#), [1069](#)
jshell, [1155](#)
JShell, [19](#), [1155–1164](#)
 commands, [1158](#), [1163–1164](#)
 and state information, [1155](#), [1157](#)
JSpinner class, [1105](#)
JTabbedPane class, [1043](#), [1056–1058](#)
JTable class, [1043](#), [1069–1071](#)
JTextComponent class, [1045](#)
JTextField class, [1043](#), [1045–1047](#)
JToggleButton class, [1043](#), [1047](#), [1049–1051](#), [1053](#)
JToggleButton.ToggleButtonModel class, [1050](#)
JToolbar class, [1074](#), [1091–1093](#)
 adding an action to a, [1095](#)
JTree class, [1043](#), [1065–1068](#)
Jump statements, [87](#), [109–115](#)
Just In Time (JIT) compiler, [10](#), [14](#)
JVM (Java Virtual Machine), [11](#), [14](#), [15](#), [20](#), [28](#), [29](#), [497](#), [520](#), [535](#)
JWindow class, [1029](#)

K

Kernighan, Brian, 4
Key codes, virtual, 792, 806
KeyAdapter class, 809, 810
Keyboard events, handling, 804–808
KeyEvent class, 787, 789, 790, 792, 1082, 1083
KeyListener interface, 797, 798–799, 804–808, 809
keyPressed(), 798–799, 804, 806
keyReleased(), 798–799, 804
keys(), 609, 610, 611
keySet(), 571, 573, 612, 670, 772
KeyStroke class, 1083
keyTyped(), 798–799, 804, 806
Keywords
 context-sensitive, 38, 420
 restricted, 420
 table of Java, 37

L

Label
 AWT standard control, 844, 845–846
 Swing, 1031, 1033, 1043–1045
 used with break statement, 111–113
 used with continue statement, 114
Label class, AWT, 817, 845
Lambda expression(s), 18, 389–406, 416–417, 566
 as arguments, passing, 399–401
 block, 390, 395–397
 body, 390, 395, 396
 and comparators, 586–587, 590
 definition of, 389
 and exceptions, 392, 402–403
 and generics, 397
 to handle action events, 850–851, 1037–1038, 1054, 1075

parameters, 390–391, 392, 393–395, 403
and the stream API, 969
target type, 390, 391, 392, 397, 399, 401, 403
and var, 19
and variable capture, 403–404

Lambda arrow operator (\rightarrow), 18, 67, 85, 390

last(), 546, 876

lastElement(), 604, 605

lastIndexOf(), 460, 461–462, 474, 475, 543, 544, 604, 605

lastKey(), 575

Late binding, 194

Layered pane, 1029–1030

Layout managers, AWT, 819, 843, 846, 868–883
default, 869

LayoutManager interface, 869

Lazy behavior (stream API stream), 972

length instance variable of arrays, 155–157

length(), 161, 452, 469, 532, 622

Lexer (lexical analyzer), 619

Libraries, class, 27, 38

Library, compact profiles of the API, 344

Lindholm, Tim, 6

LineNumberInputStream deprecated class, 680

LineNumberReader class, 318

lines(), 715, 733, 973

LinkedBlockingDeque class, 947

LinkedBlockingQueue class, 947

LinkedHashMap class, 578, 581–582

LinkedHashSet class, 551, 557–558

LinkedList class, 551, 555–556
example program using the, 569–570
from a stream API stream, obtaining a, 989

LinkedTransferQueue class, 947

List
controls, 844, 858–861
event source, as an, 787, 791, 796, 859

pop-up, 855

List class (`java.awt`), 817, 858, 859

List interface (`java.util`), 540, 543–545, 551, 555, 556, 563, 564, 597, 603, 604
from a stream API stream, obtaining an instance of the, 986–988
methods, table of, 544

List, Swing, 1060–1063

`list()`, 614, 733
and directories, 681, 683–685

`list()`, `ThreadGroup`, 524

`listFiles()`, 685–686

ListIterator interface, 540, 561, 563–565, 566
methods, table of, 563

`listIterator()`, 544, 563

ListModel interface, 1061

ListResourceBundle class, 670–672

ListSelectionEvent class, 1061, 1062, 1069

ListSelectionListener interface, 1061, 1062

ListSelectionModel interface, 1061, 1069

Literals, 36, 45–48
class, 298
regular expression, 997
string, 48, 452

`load()`, 434–435, 439–440, 499, 507, 614, 616–618

LocalDate class, 1016, 1017, 1020, 1021

LocalDateTime class, 1016, 1017, 1020, 1021

Locale class, 466, 635–636, 1012

Locale Data Markup Language (LDML), 636

Locale.Builder class, 636

Locale.IsoCountryCode, 636

LocalTime class, 1016, 1017, 1020, 1021

Lock interface, 921, 948, 949, 950
methods, table of, 948

`lock()`, 921, 948, 949

Locks, 948–950
reentrant, 949, 950

Log, program, 509

`log()`
 math method, 516
 servlet method, 1128, 1130

Logical operators
 bitwise, 73–75
 Boolean, 81–83

`long`, 39, 40, 41–42, 45
 and automatic type conversion, 52
 and automatic type promotion, 54
 literal, 46

`Long` class, 287, 288, 477, 483, 490, 491
 methods, table of, 488–490

`LongAccumulator` class, 951

`LongAdder` class, 951

`LongBuffer` class, 729

`longs()`, 638

`LongStream` interface, 972, 973

`longValue()`, 288, 478, 480, 482, 484, 485, 486, 489

Look and feels, 1026–1027

`lookup()`, 1009

Loop(s), 87
 Boolean object to control, using a, 293
 break statement and, 109–111
 continue statement and, 113–114
 do-while, 96–99
 for. *See* for loop
 infinite, 102–103, 110
 nested, 108–109, 110, 111–112
 with polling, event, 248, 265
 while, 85–96

Low surrogate char, 495

M

`main` (default name of main thread), 252
`main()`, 29–30, 117, 118, 120, 150, 153

and the java application launcher, 29
and command-line arguments, 29, 162–163
and the source-file launch feature, 1165, 1166
and Swing programs, 1034
`major()`, 503
`MalformedURLException`, 767
`Map` interface, 571–574, 575, 578, 582, 609, 610, 612
 methods, table of, 572–574
`map()`, 625, 626, 731, 742, 743, 746, 780
 and stream API streams, 971, 982–986
`Map(s)`, 539, 570–582
 classes, 578–582
 collection-view of a, obtaining a, 539, 570, 571
 flat, 986
 interfaces, 571–577
 and stream API streams, 982–986
 submaps of, 574–575
`Map.Entry` interface, 571, 575, 579
 methods, table of non-static, 577
`MapMode.PRIVATE`, 742, 743
`MapMode.READ_ONLY`, 742, 743
`MapMode.READ_WRITE`, 742, 743, 746
`MappedByteBuffer` class, 729, 743
`mapToDouble()`, 971, 985
`mapToInt()`, 971, 985–986
`mapToLong()`, 971, 985
`mark()`, 689, 690, 695, 699, 702, 709, 715, 729
`markSupported()`, 689, 690, 699, 702, 709, 715
`Matcher` class, 995, 996–997, 998, 999, 1002, 1003
`matcher()`, 996
`matches()`, 467, 996, 1000, 1003
`Math` class, 49, 139, 515–519
 exponential functions, methods for, 516
 miscellaneous methods, table of, 518–519
 rounding methods, table of, 517–518
 and static import example, 339–341

trigonometric functions, methods, for, 515

max(), 410–412, 480, 482, 486, 489, 517, 971, 976, 977
algorithm defined by Collections, 593, 596

MAX_EXPONENT, 479

MAX_PRIORITY, 260–261, 520

MAX_RADIX, 492

MAX_VALUE, 479, 490, 492

MediaTracker class, 817

Member, class, 23, 118
access and inheritance, 173–174
access, table of, 203
controlling access to a, 149–152, 202–206
static, 153–154

Member interface, 536, 1004

Memory
allocation using new, 56, 57, 121–122
deallocation, 133
leaks, 324, 329, 687
management, in Java, 14, 133
and the Runtime class, 497–498, 500

MemoryImageSource class, 899–901, 904

Menu bars and AWT menus, 843, 883–888
action command string of, 884
and events, 884

Menu, Choice control as a form of, 855

Menu class, AWT, 817, 883, 884, 888

Menu item as an event source, AWT, 787, 791, 796, 884

Menu(s), Swing, 1073–1105
accelerator keys, 1073, 1082, 1083–1084, 1097
action command string, 1074–1075, 1082
action to manage multiple components of a, using an, 1073, 1093–1098
and check boxes, 1085, 1086–1087
classes, interaction of core, 1074–1075
demonstration program, 1099–1105
events, 1074–1075, 1077, 1078, 1081–1082, 1085, 1086, 1088, 1089–1090
and images, 1084–1085

item as an event source, 786, 791, 796
main, creating a, 1078–1082
menu bar, 1073, 1075–1076, 1078, 1081, 1089, 1091
mnemonics, 1073, 1077, 1082, 1083–1084, 1097–1098
popup, 1073, 1074, 1087–1090
and radio buttons, 1085, 1086–1087
and submenus, 1074, 1076, 1077, 1078
and toolbars, 1073, 1074, 1091–1093
and tooltips, 1085, 1091, 1092, 1093
MenuBar class, AWT, 817, 883, 884
MenuDragMouseEvent, 1075
MenuEvent, 1075
MenuItem class, AWT, 817, 883–884
MenuKeyEvent, 1075
MenuListener, 1075
Metadata, 289, 294. *See also* Annotation(s)
Metal look and feel, 1027
Method class, 297, 300, 301, 536, 1004, 1006, 1116
Method reference(s), 389, 404–412
 and the Collections Framework, 410–412
 and generics, 409–412
 to instance methods, 405–409
 to static methods, 404–405
 to a superclass version of a method, 409
method(), 779
Method(s), 23, 118, 123–129
 abstract. *See* Abstract method(s)
 and annotations, 295, 314
 and autoboxing/unboxing, 290–291
 bridge, 382–384
 calling, 125, 126
 collector, 986
 default interface, 18, 19, 209, 219–223, 389, 391
 dispatch, dynamic, 188–189
 and the dot (.) operator, 119, 125
 factory, 275–276, 761

final, [155](#), [194](#)
general form, [123](#)–[124](#)
generic, [346](#), [358](#), [364](#)–[367](#), [386](#)
getter, [1110](#)–[1111](#)
hidden, using super to access, [180](#)–[181](#), [186](#)
inlining, [194](#)
interface. *See* Interface methods
and Java Bean introspection, [1112](#)
and JShell, [1159](#)–[1160](#)
lookup, dynamic, [211](#)
native, [336](#)
overloading, [137](#)–[142](#), [166](#)–[168](#), [187](#), [384](#)–[385](#)
overriding. *See* Overriding, method
and parameters, [124](#), [127](#)–[129](#)
passing an object to, [145](#)–[146](#)
recursive, [147](#)–[149](#)
reference. *See* Method reference(s)
resolution, dynamic, [208](#), [211](#), [212](#), [216](#)
returning an object from, [146](#)–[147](#)
returning a value from, [124](#), [126](#)–[127](#)
scope defined by, [50](#)–[52](#)
setter, [1110](#)–[1111](#)
static, [153](#)–[154](#), [223](#)–[224](#), [339](#)–[341](#), [404](#)–[405](#)
subclasser responsibility, [192](#)
synchronized, [250](#), [261](#)–[263](#), [921](#), [968](#)
synthetic, [1156](#)
type inference and, [366](#), [380](#)–[381](#)
varargs. *See* Varargs
variable-arity, [163](#)
MethodDescriptor class, [1112](#), [1115](#), [1116](#)
MethodHandle class, [535](#)
methodModifiers(), [1007](#)
MethodType class, [535](#)
Microsoft, [8](#)
MIME (Multipurpose Internet Mail Extensions) type, [1121](#), [1125](#)
min(), [480](#), [482](#), [486](#), [489](#), [517](#), [972](#), [975](#), [976](#), [977](#)

algorithm defined by Collections, 593, 596
minimumLayoutSize(), 869
minor(), 503
MIN_EXPONENT, 479
MIN_NORMAL, 479
MIN_PRIORITY, 260–261, 520
MIN_RADIX, 492
MIN_VALUE, 479, 490, 492
mismatch(), 601
mkdir(), 686
mkdirs(), 686
Model-Delegate component architecture, 1028
Model-View-Controller (MVC) component architecture, 1027–1028
Modifier class, 1004, 1006, 1007
“is” methods, table of, 1006
Module class, 528–529
module keyword, 37, 420
module statement, 420, 426, 443
Module(s), 18–19, 419–446
and access control, 150, 419, 420, 426, 428–429, 443
automatic, 446
and compact profiles, 344
compiling and running code based on, 424–425
creating code based on, 420–424
declaration, 419–420
definition of the term, 419
description, 420
graph, 442–443
Java API library packages and, 18, 419, 427
layers, 446
and legacy code, 427–428
and multi-module compilation mode, 429, 433
naming conventions, 421
open, 443
and packages, 201
path, 201, 424, 425, 428, 444

platform, 427, 445
and services and service providers, example application demonstrating, 435–442
unnamed, 427

module-info.java file, 420, 424, 425, 426
and javadoc, 1148

ModuleDescriptor class, 529, 535

ModuleFinder interface, 535

ModuleLayer class, 529

ModuleLayer.Controller class, 529

ModuleReader interface, 535

ModuleReference class, 535

Modulus operator (%), 67, 69

Monitor, 250, 261, 264, 265, 271

Mouse events, handling, 801–804

MouseAdapter class, 808, 809, 813, 1088, 1089–1090, 1117

mouseClicked(), 799, 808, 1088

mouseDragged(), 799, 808, 898

mouseEntered(), 799, 1088

MouseEvent class, 787, 789, 790, 793–794, 1088

mouseExited(), 799, 1088

MouseListener interface, 797, 799, 801–804, 809, 810, 1088 1089–1090

MouseMotionAdapter class, 808, 809

MouseMotionListener interface, 785, 797, 799, 801–804, 808, 809, 810

mouseMoved(), 799, 808, 898

mousePressed(), 799, 811, 1088, 1090, 1117

mouseReleased(), 799, 1088, 1090

MouseWheelEvent class, 787, 794–795

MouseWheelListener interface, 797, 799, 801, 809, 810

mouseWheelMoved(), 799

Multicore systems, 17–18, 248, 276, 389, 919, 920, 952, 956, 959, 979

Multitasking, 247

 preemptive, 249

 process-based versus thread-based, 247

Multithreaded programming, 8, 14, 247–276, 520

 and context switching. *See* Context switching

effectively using, 276
and multicore versus single-core systems, 248
and spurious wakeup, 266
and `StringBuilder` class, 475
and synchronization. *See* Synchronization
and threads. *See* Thread(s)
versus the concurrency utilities, traditional, 919, 968
and parallel programming, 952
versus single-threaded system, 248
`MutableComboBoxModel`, 1064
`MutableTreeNode` interface, 1066
MVC (Model-View-Controller) component architecture, 1027–1028

N

Namespace collisions
 between instance variables and local variables, 133
 packages and, 199, 207, 341
 and static import, 341
Naming class, 1008, 1009
NaN, 479, 482
`nanoTime()`, 507, 508, 960
@Native built-in annotation, 305
native modifier, 336
Natural ordering, 532, 582
`naturalOrder()`, 583
Naughton, Patrick, 6
NavigableMap interface, 571, 575, 580
 methods, table of, 576–577
NavigableSet interface, 540, 547–548, 558, 559
 methods, table of, 547
Negative numbers in Java, representation of, 72–73
NEGATIVE_INFINITY, 479
NegativeArraySizeException, 240, 598
Nest, 515
 host, 515

.NET Framework, 8
NetBeans, 1122, 1123
Networking, 759–782
 basics, 759–760
 bidirectional communication and, 777, 782
 classes and interfaces, list of, 760–761
new, 56, 57, 121–122, 129, 131, 133, 147, 192, 236, 237
 constructor reference and, 412, 416
 and enumerations, 278, 281
 and type inference, 380–381
NEW, 274
New I/O. *See* NIO
newBuilder(), 778, 779
newByteChannel(), 731, 734, 739, 741, 742, 743, 744, 746
newCachedThreadPool(), 941
newCondition(), 948, 949
newDirectoryStream(), 734, 752, 754–755
newFileSystem(), 738
newFixedThreadPool(), 941–942
newHttpClient(), 778
newInputStream(), 735, 748–749
Newline, inserting a, 649
newOutputStream(), 735, 748, 749–750
newScheduledThreadPool(), 941, 942
next(), 562, 563, 566, 660, 876, 990, 991
nextBoolean(), 636, 637, 660
nextBytes(), 636, 637
nextDouble(), 218, 636, 637, 660, 662, 665
nextElement(), 603, 620, 703
nextFloat(), 636, 637, 660
nextGaussian(), 636–638
nextInt(), 636, 637, 660, 665
nextLong(), 636, 637, 660
nextToken(), 620
nextX() Scanner methods, 658, 660–661, 662, 665
 table of, 660

NIO, 679, 727–758
 buffers, 728–730
 channels, 728, 730–731
 charsets, 732
 and directories, 752–758
 packages, list of, 727
 for path and file system operations, using, 750–758
 selectors, 732
 and the stream API, 735
 for stream-based I/O, using, 738, 748–750

NIO and channel-based I/O
 copying a file using, 747–748
 reading a file using, 739–743
 writing to a file using, 743–747

NIO.2, 17, 727, 738, 750

noneMatch(), 994

NORM_PRIORITY, 261, 520

NoSuchElementException, 546, 548, 550, 574, 603, 625, 661, 667

NoSuchMethodException, 241, 297

NOT operator
 bitwise unary (~), 72, 73, 74–75
 Boolean logical unary (!), 81, 82

NotDirectoryException, 752

notepad, 501, 505

notify(), 197, 265–266, 268–269, 272–274, 510, 919, 949, 968

notifyAll(), 197, 265–266, 510

NotSerializableException, 725

now(), 1016–1017

null, 38, 131
 alternative to using, 624

Null statement, 96

NullPointerException, 237, 240, 541, 543, 546, 548, 550, 561, 571, 574, 575, 598, 610, 668, 703
 using Optional to prevent a, 624, 626

nullsFirst(), 583–584

nullsLast(), 583–584

Number class, 288, 355, 477, 478
NumberFormatException, 240, 288
Numbers
 formatting, 645–646, 649–655
 pseudorandom, 636

O

Oak, 6
Object class, 196–197, 265, 346, 348, 382, 509–510
 as a data type, problems with using the, 350–352
Object class methods
 and functional interfaces, 390
 table of, 195, 510
Object reference variables
 and abstract classes, 192–193, 194
 and argument passing, 144, 145–146
 assigning, 122–123
 declaring, 121
 and cloning, 510
 and dynamic method dispatch, 188–191
 to superclass reference variable, assigning subclass, 176, 180, 188
Object-oriented programming (OOP), 5, 6, 21–27, 117
 model in Java, 13
Object(s), 23, 117, 122
 creating/declaring, 119, 121–122
 exact copy (clone) of, 509–510
 initialization with a constructor, 129–132
 to a method, passing, 145–146
 monitor, implicit, 250, 264
 as parameters, 142–144
 returning, 146–147
 serialization of. *See* Serialization
 type at run time, determining, 333–335
Object.notify(). *See* notify()
Object.wait(). *See* wait()

ObjectInput interface, [723](#)
 methods defined by, table of, [723](#)

ObjectInputFilter, [724](#)

ObjectInputFilter.Config, [724](#)

ObjectInputFilter.FilterInfo, [724](#)

ObjectInputFilter.Status, [724](#)

ObjectInputStream class, [317](#), [723](#)–724
 methods defined by, table of, [724](#)

ObjectOutput interface, [721](#)–722
 methods defined by, table of, [721](#)

ObjectOutputStream class, [317](#), [722](#)–723
 methods defined by, table of, [722](#)

Objects class, [672](#)

Observable class, [619](#)

Observer interface, [619](#)

Octals, [45](#)
 as character values, [47](#)
 and string literals, [48](#)

of(), [543](#), [544](#), [545](#)–546, [561](#), [562](#), [571](#), [573](#), [574](#), [625](#), [626](#), [732](#), [736](#), [739](#), [741](#)–[742](#), [947](#), [994](#)

offer(), [548](#), [560](#)

offerFirst(), [549](#), [550](#), [555](#)

offerLast(), [549](#), [550](#), [555](#)

offFile(), [780](#)

offsetByCodePoints(), [467](#), [474](#)

ofInputStream(), [780](#), [781](#)

ofLines(), [782](#)

ofLocalizedDate(), [1017](#), [1018](#)

ofLocalizedDateTime(), [1017](#)

ofLocalizedTime(), [1017](#)

ofNullable(), [625](#), [626](#)

ofPattern(), [1018](#)–1020
 pattern letters, [1019](#)

ofString(), [780](#), [782](#)

onAdvance(), [937](#)–938, [940](#)

open

keyword, [37](#), [420](#)
modifier, [443](#)
`open()`, [731](#)
`openConnection()`, [768](#), [769](#)–[770](#)
OpenOption interface, [735](#)
opens
 keyword, [37](#), [420](#)
 statement, [443](#)
Operator(s)
 arithmetic, [67](#)–[72](#)
 assignment. *See* Assignment operator(s)
 bitwise, [72](#)–[80](#)
 Boolean logical, [81](#)–[83](#)
 conditional-and, [83](#)
 conditional-or, [83](#)
 diamond (`<>`), [380](#)–[381](#)
 parentheses and, [45](#), [85](#)
 precedence, table of, [85](#)
 relational, [32](#), [44](#), [45](#), [80](#)–[81](#)
 ternary if-then-else (`?:`), [82](#), [84](#)
Optional class, [624](#)–[626](#), [975](#), [976](#), [977](#)
 methods, table of, [624](#)–[625](#)
optional(), [503](#)
OptionalDouble class, [624](#), [626](#)
OptionalInt class, [624](#), [626](#)
OptionalLong class, [624](#), [626](#)
OR operator
 bitwise (`|`), [72](#), [73](#), [74](#)–[75](#)
 bitwise exclusive (`^`), [72](#), [73](#), [74](#)–[75](#)
 Boolean logical (`|`), [81](#), [82](#)
 Boolean logical exclusive (`^`), [81](#), [82](#)
OR operator, short-circuit (`||`) Boolean logical, [81](#), [83](#)
`or()`, [625](#), [626](#)
Oracle, [17](#), [1122](#)
Ordinal value, enumeration constant's, [283](#), [284](#)
`ordinal()`, [283](#), [284](#)–[285](#), [531](#)

orElse(), 625, 626
orElseThrow(), 625, 975
out output stream, 30, 38, 318, 319, 322, 323, 501, 505. *See also* System.out
out(), 643, 645
OutputStream class, 316, 317, 322, 688, 689, 692, 698, 700, 704, 706, 718, 722, 726, 749, 1130
 methods, table of, 691
OutputStreamWriter class, 318
Overloading methods, 137–142, 166–168, 187, 384–385
@Override, built-in annotation, 305, 307
Overriding, method, 185–191
 and abstract classes, 191–194
 and bridge methods, 382–384
 and dynamic method dispatch, 188–191
 final to prevent, using, 194
 in a generic class, 379–380
 and run-time polymorphism, 188, 189, 191

P

Package(s), 150, 199–208, 225
 access to classes contained in, 202–206, 207
 built-in standard Java classes and, 206, 207
 the default, 200, 206
 defining, 200
 finding, 200–201
 importing, 206–208
 and JShell, 1162–1163
 and modules, 201
 Swing, 1030
Package class, 301, 527–528
 methods, table of, 527–528
package statement, 200, 206
Paint mode, setting the, 832–833
paint(), 801, 804, 821, 822, 824, 898, 1038–1039, 1118
 lightweight AWT components and overriding, 892

Paintable area, computing, 1039–1040
`paintBorder()`, 1038–1039
`paintChildren()`, 1038–1039
`paintComponent()`, 1038–1039, 1042
Painting in Swing, 1038–1042
Panel class, 817, 818, 819, 875
Panes, Swing container, 1029–1030. *See also* Content pane
Parallel processing, 18, 389, 566, 568
 of a stream API stream, 969, 972, 973, 979–981, 988, 991, 993
Parallel programming. *See* Programming, parallel
`parallel()`, 970, 979–980
`parallelPrefix()`, 600
`parallelSetAll()`, 600
`parallelSort()`, 599–600
`parallelStream()`, 541, 543, 973, 979, 980
Parameter(s), 29, 124, 127–129
 and constructors, 131–132
 final, 155
 and lambda expressions, 390–391, 392, 393–395, 403
 names and instance variable names, 133
 objects as, 142–144
 and overloaded constructors, 140–141
 and overloaded methods, 137, 187
 and the scope of a method, 50
 servlet, reading, 1130–1132
 type. *See* Type parameter(s)
 variable-length (varargs), 165, 545
Parameterized types, 346, 348
`parameterModifiers()`, 1007
`parse()`, 503, 1020
`parseByte()`, 484, 490
`parseDouble()`, 482
`parseFloat()`, 480
`parseInt()`, 487, 490–491
`parseLong()`, 489, 490
`parseShort()`, 485, 490

Parsing, definition of, 619
Pascal, 4
Passwords, reading, 720
patch(), 502–503
Path interface, 681, 683, 732–733, 739, 750, 751, 752–753, 756
 converting a File object into an instance of the, 683, 732, 750
 instance for stream-based I/O, using a, 748–750
 methods, table of a sampling of, 733
 obtaining an instance of the, 735–736, 738–739, 741, 742, 743, 744, 746
Path.of(), 736, 739, 741
Paths class, 735–736, 739
Paths.get(), 735–736, 738–739
Pattern class, 995, 996, 999, 1002, 1003
Pattern matching, regular expressions, 997–1003
PatternSyntaxException, 997
Payne, Jonathan, 6
peek(), 548, 607, 608
peekFirst(), 549, 555
peekLast(), 549, 555
Peers, native, 892, 1025–1026
Period class, 1021
Persistence (Java Beans), 1113
Phaser class, 920, 922, 934–941
 compatibility with fork/join, 968
PI (Math constant), 515
PIPE, 505
Pipeline for actions on stream API streams, 18, 389, 972, 984–985
PipedInputStream class, 317
PipedOutputStream class, 317
PipedReader class, 318
PipedWriter class, 318
PixelGrabber class, 901–904
Pluggable look and feel (PLAF), 1026–1027, 1028
Plug-in and pluggable application architecture, 434, 436
PNG file format, 894
Point class, 793, 794, 817

poll(), 548, 560
pollFirst(), 547, 549, 555
Polling, 248, 265
pollLast(), 547, 549, 555
Polygon class, 817, 826
Polymorphism, 5, 22, 25–27, 195
 and dynamic method dispatch, 188–191, 192
 and interfaces, 208, 211, 216
 and overloaded methods, 137, 139, 140
pop(), 549, 550, 607, 608
Popup trigger, 1088, 1090
PopupMenu class, 817, 888
PopupMenuEvent, 1075
Port, 759, 760, 767
Portable Operating System Interface (POSIX), 737
Portability problem, 6–7, 8, 9–10, 12, 14, 20
 and data types, 40
 and thread context switching, 249
POSITIVE_INFINITY, 479
PosixFileAttributes class, 736, 737, 752
PosixFileAttributeView interface, 738
Possessive behavior (regular expression pattern matching), 1001
postVisitDirectory(), 756
pow(), 339–341, 516
pre(), 503
Predicate<T> predefined functional interface, 417, 676, 976
preferredLayoutSize(), 869
previous(), 563, 876
preVisitDirectory(), 756
PrimitiveIterator interface, 539, 990
PrimitiveIterator.OfDouble interface, 539, 990
PrimitiveIterator.OfInt interface, 990
PrimitiveIterator.OfLong interface, 990
print(), 31, 38, 322, 323, 454, 704, 718, 1130
printf()
 function, C/C++, 642, 704

method, Java, 163, 657, 704–706, 718, 719
println(), 30, 31, 38, 197, 322, 323, 454–455, 645, 704, 718, 1130
 and Boolean output, 45
 and String objects, 65
PrintStream class, 317, 319, 322, 657, 703–706
PrintWriter class, 318, 322–323, 657, 717–718, 1125
PriorityBlockingQueue class, 947
PriorityQueue class, 551, 559–560
private access modifier, 29, 150–152, 202–203
 and inheritance, 173–174
Process class, 495–496, 501, 503
 methods, table of, 498
Process, definition of, 247, 495
Process-based versus thread-based multitasking, 247
ProcessBuilder class, 495, 503–505
 methods, table of, 503–504
ProcessBuilder.Redirect class, 505
ProcessBuilder.Redirect.Type enumeration, 505
ProcessHandle class, 495–496
ProcessHandle.Info, 495
Program log, 509
Programming
 multithreaded. *See* Multithreaded programming
 object-oriented. *See* Object-oriented programming
 process-oriented, 21, 22, 26
 structured, 4, 5
Programming, parallel, 17–18, 249, 566, 920, 921, 952, 979
 and the level of parallelism, 955
 and specifying the level of parallelism, 955, 959–962, 968
Project Coin, 17
Properties class, 538, 602, 613–618
 methods, table of, 614
Properties, environment, 509
Property, Java Bean, 1110, 1118
 boolean, 1111
 bound and constrained, 1113, 1116

design patterns for, 1110–1111, 1113
PropertyChangeEvent, 1113
PropertyChangeListener interface, 1113, 1114
PropertyDescriptor class, 1112, 1115, 1116, 1118
PropertyPermission class, 672
PropertyResourceBundle class, 670
PropertyVetoException, 1113
protected access modifier, 150, 202–203
Protocols, overview of networking, 759–760
provides
 keyword, 37, 420
 statement, 435, 542
Pseudorandom numbers, 636
public access modifier, 29, 150–152, 202–203
Push buttons, AWT, 844, 847–851
 action command string of, 847, 849, 850, 1047–1058
Push buttons, Swing, 1035–1038, 1047–1049
 action command string of, 1047–1048
push(), 550, 607, 608
Pushback, 700–702
PushbackInputStream, 317, 698, 700–702
PushbackReader class, 318, 716–717
put(), 571, 573, 578, 580, 582, 609, 610, 611
 and buffers, 729, 730, 745
putAll(), 573, 582
PutField inner class, 722–723
putValue(), 1094–1095, 1096

Q

Query string, 1138
Queue, double-ended, 549–550
Queue interface, 540, 548, 549, 555, 559, 560
 methods, table of, 548
quietlyInvoke(), 967
quietlyJoin(), 967

R

Race condition, 263
Radio buttons, 853
 Swing, 1053–1055
 and Swing menus, 1086–1087
Radix, 483
radix(), 667
Random class, 218, 636–638
 methods, table of core, 637
RandomAccess interface, 540, 570
RandomAccessFile class, 708, 730, 731
range(), 561, 562
Raw types, 370–372
READ, 505
read(), 99, 317, 318, 319–320, 324–325, 534, 690, 699, 701, 709, 717, 723,
 724, 731, 740, 741, 748, 894–895
Readable interface, 534, 657, 663, 709
ReadableByteChannel interface, 657
readAttributes(), 735, 737, 751–752
readBoolean(), 707, 724
readDouble(), 707, 724
Reader class, 317–318, 319, 688, 689, 708–709, 710, 712, 721
 methods defined by, table of, 709
readExternal(), 721
readInt(), 707, 724
readLine(), 320–322, 490, 719, 1130
readObject(), 723, 724, 725
readPassword(), 719–720
readString(), 735
ReadWriteLock interface, 950
Real numbers, 42
rebind(), 1008–1009
receive(), 774
Recursion, 147–149
 and the Fork/Join Framework divide-and-conquer strategy, 956

RecursiveAction class, 921, 952, 954, 956, 957, 958, 962
RecursiveTask class, 921, 952, 954, 956
 example program using, 962–965
Redirect class, 505
reduce(), 972, 977–979, 980–981, 982–983
Reduction operations, 977–979
 mutable, 989
ReentrantLock, 949
ReentrantReadWriteLock, 950
Reflection, 296, 443, 535–536, 995, 1004–1007
 and annotations, 296–301
 and nests, 515
regionMatches(), 457–458
register(), 934
Regular expressions, 468, 619, 657, 665, 677, 995–1003
 definition of the term, 995
 syntax, 997
 wildcards and quantifiers, 995, 997, 999–1001
reinitialize(), 966
Relational operators, 32, 44, 45, 80–81
Relative index, 655–656
Release cadence, 12, 502
release(), 922–925
Reluctant behavior (regular expression pattern matching), 1000–1001
Remote interface, 1008
Remote method invocation (RMI), 15, 720, 995, 1008–1011
RemoteException, 1008
remove(), 541, 542, 544, 548, 556, 561, 562, 563, 573, 609, 610, 611, 844,
 1034, 1075, 1077
removeActionListener(), 1037
removeAll(), 541, 542, 844
removeAttribute(), 1135, 1142
removeEldestEntry(), 582
removeElement(), 604, 605
removeElementAt(), 604, 605
removeFirst(), 550, 555

`removeIf()`, 541, 542
`removeKeyListener()`, 785
`removeLast()`, 550, 555
`removeTListener()`, 1112
`removeTypeListener()`, 785
`renameTo()`, 682
`repaint()`, 801, 804, 809, 810, 811, 822–823, 1039, 1118
`@Repeatable` annotation, 305, 312, 313
REPL (read-evaluate-print loop) execution, 1155
`replace()`, 445, 473, 573
`replaceAll()`, 468, 543, 544, 573, 997, 1002
`replaceRange()`, 867
ReplicateScaleFilter class, 904
requires keyword, 37, 420
requires statement, 420, 426, 427
 using static with a, 443, 444
 using transitive with a, 430–434
`reset()`, 667, 689, 690, 695, 697, 699–700, 702, 709, 715, 729
`resolve()`, 732, 733
Resource bundles, 667–672
 ResourceBundle class, 667–670
 methods, table of, 668–670
 and modules, 670
 ResourceBundle.Control class, 668
`resume()`, 15, 272, 520, 522
`retainAll()`, 541, 542
`@Retention` built-in annotation, 296, 305
RetentionPolicy enumeration, 296, 535
return statement, 114–115, 124
 in a lambda expression, 396
`reverse()`
 algorithm defined by Collections, 593
 StringBuffer method, 472
`reversed()`, 583, 586
`reverseOrder()`
 collection algorithm, 593, 595–596

Comparator method, 583
rewind(), 729, 741, 744, 745
RGB (red-green-blue) color model, 829
 default, 899
RGBImageFilter class, 904, 906–907
 example program demonstrating the, 907–918
RGBtoHSB(), 830
Richards, Martin, 4
Ritchie, Dennis, 4, 5
rmi protocol, 1009
RMI (Remote Method Invocation), 15, 720, 995, 1008–1011, 1122
rmic compiler, 1010
rmiregistry (RMI registry), 1010, 1011
Run-time
 system, Java, 10. *See also* Java Virtual Machine (JVM)
 type information, 15, 333, 376–378, 514
run(), 250, 253, 254, 390, 520, 522, 639, 966, 1034
 overriding, 255, 256, 639
 using a flag variable with, 272–274
RUNNABLE, 274
Runnable interface, 250, 252, 390, 520, 638, 639, 919, 965, 966, 1034
 implementing the, 253–254, 256
Runtime class, 495, 496–501, 962
 executing other programs and, 501–501
 memory management and, 497–498, 500
 methods, table of some, 499
RUNTIME retention policy, 295–296, 297, 300
Runtime.exec(), 495, 499, 501
Runtime.Version class, 502–503
Runtime.version(), 499, 502
RuntimeException class, 228, 229, 237, 240, 244
RuntimePermission class, 529

S

@SafeVarargs built-in annotation, 305, 307

SAM (Single Abstract Method) type, 390
Sandbox, 10
`save()`, 613
Scanner, 619
Scanner class, 657–667
 closing an instance of the, 663
 constructors, 657, 658
 delimiters, 657, 665–666
 demonstration programs, 661–665
 `hasNextX()` methods, table of, 659
 how to use, 657–658, 660–661
 methods, miscellaneous, 666–667
 `nextX()` methods, table of, 660
`schedule()`, 639, 640
ScheduledExecutorService interface, 941
ScheduledThreadPoolExecutor class, 921, 941
Scientific notation, 46, 645–646
Scopes in Java, 49–52
 nested, 50–51, 158
Scroll bars, 844, 861–864, 1058, 1059
 as an event source, 788, 796, 862
ScrollPane, 1058–1060
Scrollbar class, 817, 861
ScrollPane class, 817
`search()`, 607, 608
Security manager, 324, 505, 529, 687, 704, 1011, 1122
Security problem, 8, 9, 10, 12, 20, 720
 and servlets, 1122
`security()`, 503
SecurityException, 240, 324, 497, 505, 687, 704, 752, 756
SecurityManager class, 529
`seek()`, 708
SeekableByteChannel interface, 731, 739, 742, 744
`select()`, 856, 859, 864–865, 867
Selection statements, 87–95
Selectors, 731, 732

Semaphore, 919, 920, 922–927, 968
 and setting initial synchronization state, 927
Semaphore class, 920, 922
send(), 774, 777, 778, 779, 781, 782
Separable Model architecture, 1028
Separators, 37
SequenceInputStream class, 317, 702–703
sequential(), 970, 981
Serializable interface, 721, 725, 967, 1113
Serialization, 720–726
 example program, 725–726
 and Java Beans, 1113
 and security, 720
 and static variables, 721
 and transient variables, 721, 725
serialVersionUID constant, 726
Server process, 759
ServerSocket class, 730, 763, 773
service(), 1122, 1125, 1127, 1131, 1137
ServiceLoader class, 434, 632
Services and service providers, 434–442
 definition of the terms, 434
 module-based example application demonstrating, 435–442
 and modules, 435
Servlet interface, 1126, 1127, 1130
 methods, table of, 1127
Servlet(s), 12, 1121–1143
 advantages of, 1122
 API, 1126
 development options, 1122–1124
 example program for a simple, 1124–1126
 life cycle of, 1122
 parameters, reading, 1130–1132
 and portability, 12
 and security, 1122
 and session tracking, 1142–1143

using Tomcat to develop, [1122](#), [1123–1126](#)
ServletConfig interface, [1126](#), [1128](#), [1130](#)
ServletContext interface, [1126](#), [1128](#)
 methods, table of various, [1128](#)
ServletException class, [1127](#), [1130](#)
ServletInputStream class, [1127](#), [1130](#)
ServletOutputStream class, [1127](#), [1130](#)
ServletRequest interface, [1125](#), [1126](#), [1128](#), [1130](#)
 methods, table of various, [1129](#)
ServletResponse interface, [1125](#), [1126](#), [1128](#)
 methods, table of various, [1129](#)
Session tracking, HTTP, [1142](#), [1143](#)
Set interface, [540](#), [545–546](#), [556](#), [561](#), [571](#), [575](#)
 from a stream API stream, obtaining an instance of the, [986–988](#)
Set-view, obtaining, [579](#), [612](#)
set(), [543](#), [544](#), [556](#), [563](#), [622](#), [629](#), [951](#)
setAccelerator(), [1083](#)
setActionCommand(), [850](#), [884](#), [1047–1048](#), [1054](#)
setAlignment(), [845](#)
setAll(), [600](#)
setAttribute(), [1128](#), [1135](#), [1142](#)
setBackground(), [821](#), [829](#)
setBlockIncrement(), [862](#)
setBorder(), [1042](#)
setBounds(), [819](#), [869](#)
setCharAt(), [470](#)
setColor(), [830](#)
setConstraints(), [878](#)
setContent-Type(), [1125](#), [1129](#)
setDefault(), [634](#), [635](#)
setDefaultCloseOperation(), [1032–1033](#)
setDisabledIcon(), [1047](#), [1085](#)
setDisplayMnemonicIndex(), [1082](#)
setEchoChar(), [865](#)
setEditable(), [865](#), [867](#)
setEnabled(), [883](#), [1077](#), [1095](#)

`setFont()`, 836
`setForeground()`, 821, 829
`setForkJoinTaskTag()`, 967
`setHorizontalTextPosition()`, 1085
`setIcon()`, 1044, 1084–1085
`setJMenuBar()`, 1076, 1078
`setLabel()`, 847, 851, 883–884
`setLastModified()`, 683
`setLayout()`, 846, 869, 1034
`setLength()`, 470, 708
`setLocation()`, 819
`setMaxAge()`, 1136, 1142
`setMnemonic()`, 1082
`setN()` setter method design pattern, 1110, 1111
`setName()`, 251, 252, 522
`setObjectInputFiller()`, 724
`setPaintMode()`, 832
`setPreferredSize()`, 819, 862
`setPressedIcon()`, 1047
`setPriority()`, 260–261, 522
`setReadOnly()`, 683
`setRolloverIcon()`, 1047
`setSelectedCheckbox()`, 854
`setSelectedIcon()`, 1047
`setSelectionMode()`, 1061
`setSize()`, 605, 819, 820, 824, 1032
`setState()`, 851, 884
`setText()`, 845, 864, 867, 1044, 1047
`setTitle()`, 820
`setToolTipText()`, 1085
`setUnitIncrement()`, 862
`setValue()`, 577, 861–862, 1136
`setValues()`, 861
`setVisible()`, 820, 823, 824, 1034
`setXORMode()`, 832–833
Sheridan, Mike, 6

Shift operators, bitwise, 72, 75–79
Short class, 287, 288, 477, 483, 490, 491
 methods defined by, table of, 485
short data type, 39, 40, 41, 45
 and automatic type conversion, 52
 and automatic type promotion, 54, 75
ShortBuffer class, 729
shortValue(), 288, 478, 480, 482, 484, 485, 487, 489
show(), 876, 1088, 1089, 1090
shuffle() algorithm defined by Collections, 593, 594, 596
shutdown(), 941, 943, 956, 967–968
shutdownNow(), 967–968
Sign extension, 75, 77–78, 78–79
signal(), 949
SimpleBeanInfo class, 1113, 1118, 1119
SimpleDateFormat class, 636, 1013–1015, 1019
 formatting string symbols, table of, 1014
SimpleFileVisitor class, 757, 758
SimpleTimeZone class, 634–635
sin(), 42, 515
SingleSelectionModel, 1056
SIZE, 479, 490
size(), 541, 542, 556, 574, 605, 609, 610, 611, 622, 735, 737, 743
skip(), 667, 691–692, 699, 709, 723
SKIP_SIBLINGS, 756
SKIP_SUBTREE, 756, 757
sleep(), 250, 251–252, 257, 522, 947
slice(), 729
Slider box, 861
Snippet
 code, 1155, 1157, 1158, 1160, 1163
 ID, 1158
Socket class, 730, 731, 763–766, 773
Socket(s)
 datagram, 773–774
 overview, 759–760

and servlets, 1122
TCP/IP client, 763–766
TCP/IP server, 763, 773
SocketAddress class, 774
SocketChannel class, 731
SocketException, 774
sort(), 543, 544, 599
 algorithm defined by Collections, 594
sorted(), 972, 976
SortedMap interface, 571, 574–575
 methods, table of, 575
SortedSet interface, 540, 546, 547
 methods, table of, 546
Source code file, naming a, 27
Source file(s), 27–28
 annotations and, 294
 compiling and running a program directly from a, 28, 1165–1166
 naming a, 27
SOURCE retention policy, 295–296
split(), 468, 1002–1003
Spliterator, 533, 539, 543, 566–568
 and arrays, 600
 characteristics, 568
Spliterator interface, 533, 539, 540, 561, 566–568
 methods declared by, table of, 567
 and streamAPI streams, 990, 991–993
spliterator(), 533, 541, 543, 600, 970
Spliterator.OfDouble interface, 568
Spliterator.OfInt interface, 568
Spliterator.OfLong interface, 568
Spliterator.OfPrimitive interface, 568
sqrt(), 42, 49, 339–341, 516
Stack
 definition of, 25, 134
 frame, 530
 trace, 229–230, 236, 530

walking, 531
ways to implement a, 213
Stack class, 538, 551, 602, 607–609
 methods, table of, 608
StackTraceElement class, 242, 530
 methods, table of, 530
StackWalker class, 531
StackWalker.StackFrame interface, 531
StampedLock interface, 950
StandardCopyOption values, partial list of, 747
StandardOpenOption class, 735, 748, 749
 enumeration, table of values for the, 736
StandardOpenOption.CREATE, 736, 744, 749
StandardOpenOption.READ, 736, 746, 748
StandardOpenOption.TRUNCATE_EXISTING, 736, 749
StandardOpenOption.WRITE, 736, 744, 749
Standard Template Library (STL), 539
start(), 250, 253, 254, 255, 495, 505, 522, 996–997, 999
startsWith(), 458, 733
State enumeration, 274
Statements, 30
 null, 96
Statements, control, 31, 44
 iteration, 87, 95–106
 jump, 87, 106–114
 selection, 87–95
static, 29, 153–154, 157, 339
 member restrictions, 386
 used in a requires statement, 444
Static import, 16, 339–341
statusCode(), 779
stop(), 15, 272, 520, 522
store(), 613, 614, 616–618
str(), 295
Stream API, 18, 969–994
 and collections, 618, 969, 970, 973, 975, 988

interfaces, 970–973
and lambda expressions, 969
and NIO, 735
Stream interface, 543, 600, 715, 733, 971–972, 993, 994
methods, table of some, 971–972
and primitive types, 972, 973
Stream, intermediate operations on a stream API, 972
to create a pipeline of actions, 972, 984–985
lazy behavior of, 972
stateless versus stateful, 972
Stream, stream API
and accumulator operations, 977, 978, 980–981, 989
collection from a, obtaining a, 986–989
and combiner operations, 980–981, 983, 989
definition of the term, 969–970
iterators and a, 990–993
mapping a, 982–986
obtaining a, 973
operations on a, terminal versus intermediate, 972
ordered versus unordered, 981
parallel processing of a, 969, 972, 973, 979–981, 988, 991, 993
parallel, using a, 979–981, 993
reduction operations, 977–979
stream(), 542, 543, 600, 622, 625, 973, 975, 980, 985
Stream(s), byte, 316–317, 319, 323, 689–708
 buffered, 698–702
 classes in java.io, table of, 317
 filtered, 698
Stream(s), character, 316, 317–318, 319, 323, 689, 708–718
 classes in java.io, table of, 318
Stream(s), I/O
 benefits, 726
 buffered, 698–702, 714–717
 classes, top-level, 688–689
 closing, 687–688
 concatenating input to, 702–703

definition of the term, 316, 709
filtered, 698, 726
flushing, 686
and NIO, 738, 748–750
predefined, 318–319
reactive, 921
variables, predefined, 318–319
versus stream API streams, 970

strictfp, 335–336

StrictMath class, 519

String class, 29, 65, 160–162, 449, 468, 532, 657, 996
constructors, 450–452
methods, table of some, 467–468

String(s)

- arrays of, 65, 162
- changing case of characters in, 465–466, 492–494
- comparison, 161, 456–460
- concatenating, 31, 160–161, 453–454, 463, 466–467, 471
- constants, 65, 160
- converting data into a, 454–455, 465
- creating, 160, 450–452
- extracting characters from, 455–456
- formatted, creating a, 643–645
- formatting a, 645, 651
- immutability of, 160, 449, 462, 468
- joining, 466–467
- length of a, obtaining the, 161, 452
- literals, 48, 452
- modifiable, creating and working with, 468–475
- modifying, 462–464
- numbers to and from, converting, 490–492
- as objects, 48, 65, 160, 449
- parsing a formatted input, 619–621
- reading, 320–322
- searching, 460–462

StringBuffer class, 160, 449, 451, 462, 468–475, 532

methods, table of some, 474
StringBufferInputStream deprecated class, 680
StringBuilder class, 160, 449, 451, 462, 475, 532, 642, 643
 and synchronization, 475
StringJoiner class, 672
StringReader class, 318
 StringTokenizer class, 619–621
 methods, table of, 620
stringWidth(), 840, 841
StringWriter class, 318
strip(), 464
stripLeading(), 465
stripTrailing(), 465
Stroustrup, Bjarne, 6
Stubs (RMI), 1010
Subclass, 24, 171–174, 189
 generic, 372–376
subList(), 543, 544
subMap(), 574, 575, 577
submit(), 944
subSequence(), 468, 474, 532
subSet(), 546, 547, 559
substring(), 462–463, 474
Sun Microsystems, 6, 15, 16, 17
super, 153, 177
 and bounded wildcard arguments, 364
 and interface default methods, 223
 and method references, 409
 and methods or instance variables, 180–181, 186
super(), 344
 and superclass constructors, 177–180, 184, 256
Superclass, 24, 171–174, 189, 199
 abstract, 191–194
 generic, 372–374
Supplemental character, definition of, 494
Supplier<T> predefined functional interface, 417, 676, 989

@SuppressWarnings built-in annotation, 305, 307
suspend(), 15, 272, 520, 522
Swing, 15, 315, 800–801, 843, 1025–1071
 applet, 1030
 application, example of a simple, 1030–1035
 and the AWT, 815, 843, 1025–1026
 component classes, list of, 1028–1029
 components. *See* Components, Swing
 containers. *See* Container(s), Swing
 event handling, 1034–1038
 history of, 1025–1026
 menus. *See* Menu(s), Swing
 and MVC architecture, 1027–1028
 packages, list of, 1030
 and painting, 1038–1042
 threading issues, 1034, 1038
Swing: A Beginner’s Guide (Schildt), 1025
SwingConstants interface, 1044
SwingUtilities class, 1034
switch statement, 90–95, 109, 111
 and auto-unboxing, 292
 nested, 94
 using enumeration constants to control a, 91, 278–280
 using a String to control a, 16, 91, 93–94
 versus the if statement, 94–95
Synchronization, 14, 249–250, 261–265
 and atomic operations, 950–951
 and collections, 550, 595, 602
 and deadlock, 270–271
 and interprocess communication, 265–270
 objects, using, 922–941
 race condition and, 263
 and StringBuilder class, 475
 via synchronized block, 263–265, 595
 via synchronized method, 250, 261–262
 versus concurrency utilities, traditional, 919, 968

synchronized modifier, 261, 919, 948, 968
 used with a method, 261–263
 used with an object, 263–265
synchronizedList(), 594, 595
synchronizedSet(), 594, 595
Synchronizers, 919, 920–921
SynchronousQueue class, 947
System class, 30, 38, 318, 505–509
 methods, table of, 506–507
System.console(), 506, 719
System.err standard error stream, 318, 319, 505
System.exit(), 506, 801, 804, 821, 823, 1082
System.getLogger(), 506, 509
System.getProperties(), 506, 613
System.getProperty(), 506, 509
System.in standard input stream, 318, 319, 501, 505, 657, 718
System.in.read(), 99
System.Logger interface, 509
System.LoggerFactory class, 509
System.nanoTime(), 507, 508, 960
System.out standard output stream, 30, 38, 318, 319, 322, 323, 501, 505, 657,
 703, 705, 718
 and static import, 341

T

Tabbed panes, 1056–1058
Table, Swing, 1069–1071
TableColumnModel, 1069
TableModel, 1069
TableModelEvent class, 1069
tailMap(), 574, 575, 577
tailSet(), 546, 547
@Target built-in annotation, 305, 306, 308
TCP/IP, 14, 760
 client sockets, 763–766

disadvantages of, 773

server sockets, 763, 773

See also Transmission Control Protocol (TCP)

TemporalAccessor, 647

TERMINATE, 756, 757

TERMINATED, 274

Ternary if-then-else operator (?:), 82, 84

test(), 417, 674, 675, 676, 976–977

Text

area, 795, 799, 867–868

components as an event source, 795, 796

editing controls, 844, 864–868

formatting using java.text classes, 995, 1011–1016

output using font metrics, managing, 839–842

Text fields, 795

AWT, 864–866

Swing, 1045–1047

TextArea class, AWT, 818, 867–868

TextComponent class, 818, 864, 867

TextEvent class, 787, 795, 867

TextField class, AWT, 818, 864–865

TextListener interface, 797, 799

textValueChanged(), 799

thenComparing(), 584, 589–590

thenComparingDouble(), 584

thenComparingInt(), 584

thenComparingLong(), 584

this, 132–133, 153

and lambda expressions, 403

and type annotations, 307, 308, 311

this(), 341–344

Thompson, Ken, 4

Thread class, 15, 250, 251, 256, 272, 274, 520–522, 638, 919, 952

constructors, 253, 256, 520

extending, 255–256

methods, table of, 520–522

Thread pool, 919, 941–942, 953, 954–956, 959, 962, 967–968
common, 953, 954, 955, 956, 959, 962, 968

Thread(s)
communication among, 265–270
creating, 252–257
creating and starting a, 275–276
daemon, 639, 956, 965
and deadlock, 270–271, 522, 1034
definition of the term, 247
executors to manage, using, 921, 941–943
group, 252, 523–527
local variables, 527
main, 250–252, 254, 256, 257, 258, 1034
messaging, 250
priorities, 249, 260–261, 520
race condition of, 263
resuming, 272–274, 524–527
return a value, that, 943–946
and spurious wakeup, 266
states of, possible, 249, 274–275
stopping, 272
suspending, 250, 252, 272–274, 524–527
and Swing, event dispatching, 1034, 1038
synchronization. *See* Synchronization

Thread.UncaughtExceptionHandler interface, 534

ThreadGroup class, 520, 523–527, 534
methods, table of, 523–524

ThreadLocal class, 527

ThreadPoolExecutor class, 921, 941

throw, 227, 236–237, 246

Throwable class, 228, 229, 232, 236, 237, 241, 244, 332, 387, 529, 530
methods defined by, table of, 242
obtaining an object of the, 236–237

throws, 227, 237–238, 240

Thumb, 861

time Calendar class instance variable, 628

Time and date

 formatting, 647–648, 995, 1011–1015, 1017–1020
 java.util classes that deal with, 626–636
 strings, parsing, 1020–1021

Time and Date API, 628, 995, 1011, 1015–1021
 packages, list of, 1015

 timedJoin(), 947

 timedWait(), 947

 TIMED_WAITING, 274

 Timer class, 638–640
 methods, table of, 640

 TimerTask class, 638–640
 methods, table of, 639

 Timestamp, event, 787

 TimeUnit enumeration, 921, 928, 944, 946–947

 TimeZone class, 633–634
 methods defined by, table of some, 633–634

to
 keyword, 37, 420
 clause, 429, 443

to(), 505

toAbsolutePath(), 733, 750

toArray(), 542, 553–554, 972

toBinaryString(), 487, 489, 490, 491–492

toCharArray(), 456

toChronoUnit(), 947

toDays(), 947

toDegrees(), 519

ToDoubleFunction functional interface, 676, 985

toFile(), 732

Toggle button, Swing, 1049–1051

toHexString(), 481, 482, 487, 490, 491–492

toHours(), 947

Tokens, 619, 657, 665

tokens(), 667

toLanguageTag(), 636

`toList()`, 986, 988
`toLocalDate()`, 1017
`toLocalTime()`, 1017
`toLowerCase()`, 465–466, 493, 495
Tomcat, 1122, 1123–1126
`toMicros()`, 947
`toMillis()`, 947
`toMinutes()`, 947
`toNanos()`, 947
`toOctalString()`, 487, 490, 491–492
Toolbars, 1073, 1074, 1091–1093
 undocked, 1091
Tooltips, 1085, 1091, 1092, 1093
TooManyListenersException, 1112
`toPath()`, 683, 732, 750
`toRadians()`, 519
`toSeconds()`, 947
`toSet()`, 986, 988
`toString()`, 197, 232, 241, 242, 288, 295, 301, 323, 454, 465, 481, 482, 484, 485, 487, 490, 491, 497, 510, 514, 522, 524, 528, 529, 530, 531, 532, 553, 600, 605, 611, 622, 623, 625, 627, 641, 643, 645, 704, 718, 732, 733, 763, 786, 834, 840, 967, 1004, 1066
 advantages to overriding, 197, 454–455
`totalCPUDuration()`, 495
`totalMemory()`, 498, 499, 500
`toUpperCase()`, 465–466, 493
`toZonedDateTime()`, 632
transient modifier, 333, 1113
transitive keyword, 37, 420, 430–434
`translatePoint()`, 793
Transmission Control Protocol (TCP)
 definition of, 759–760
 and stream-based I/O, 760
 See also TCP/IP
TreeExpansionEvent class, 1066
TreeExpansionListener interface, 1066

TreeMap class, 578, 580–581, 582, 618
example using a comparator, 587–590

TreeModel, 1066

TreeModelEvent class, 1066

TreeModelListener interface, 1066

TreeNode interface, 1066

TreePath class, 1066

Trees, Swing, 1065–1068

TreeSelectionEvent class, 1066

TreeSelectionListener interface, 1066, 1067

TreeSelectionModel, 1066

TreeSet class, 551, 557, 558–559, 582, 618
example using a comparator for sorting an instance of the, 585–587

trim(), 464

trimToSize(), 474, 553, 605

true, 38, 44, 45, 47, 81, 82, 83

TRUE, 495

True and false in Java, 47, 81

Truncation, 53–54

try block(s), 227, 228, 230–236, 237–239, 246
nested, 234–236

try-with-resources statement, 17, 228, 245, 324, 329–332, 534, 656, 663, 686, 687–688, 694, 730, 732, 739, 753, 764, 765–766, 774, 970
advantages to using, 329, 331, 688
and local variable type inference, 331

tryAdvance(), 566–568, 991

tryLock(), 921, 948–949

trySplit(), 992–993

tryUnfork(), 967

Two's complement, 72–73

TYPE, 479, 490, 492, 495

Type argument(s), 348–349, 350, 354
and bounded types, 355–357
and generic class hierarchies, 372
and generic interface implementation, 370
and type inference, 366, 380–381

Type conversion
 automatic, 39, 52, 138–139
 narrowing, 52
 widening, 52

Type enumeration, 505

Type inference, local variable, 19, 62–65
 advantages of using, 63
 in a for loop, 107–108
 and generics, 381
 and inheritance, 195–196
 and lambda expressions, 394
 with reference types, 168–170
 and streamlining code, 19, 63, 168–169, 381
 and the try-with-resources statement, 331

Type interface, 536

Type parameter(s)
 and bounded types, 354–357, 369–370
 cannot create an instance of a, 385
 and class hierarchies, 373–376
 and erasure, 349, 382
 and primitive types, 350
 and static members, 386
 and type safety, 350
 used with a class, 348, 353–355 357
 used with a method, 348, 364–367
 and wildcard arguments, 357–366

Type safety
 and collections, 595
 and generic methods, 367
 and generics, 345, 346, 349, 350–352
 and raw types, 370–372
 and wildcard arguments, 357–360, 361

type(), 505

Type(s), 31
 annotations, 18, 307–312
 bounded, 355–357

casting, 52–54
checking, 13, 17, 39, 349, 350, 371, 387
class as a data, 117, 119, 121, 122, 123, 134
inference, 366, 380–381, 391, 394, 397, 403
inference, local variable. *See* Type inference, local variable
intersection, 357
non-reifiable, 307
parameterized, 346, 348
promotion, 41, 54–55, 75–77
raw, 370–372
SAM (Single Abstract Method), 390
simple, 39
Types, primitive (simple), 39–40, 122, 144, 286, 350, 478
autoboxing/unboxing and, 289–292, 294, 554
and collections, 554
iterators for, 539
to a string representation, converting, 453, 454, 465
to or from a sequence of bytes, converting, 706–708
wrappers for, 286–289, 294, 350, 478–495
Typesafe view of a collection, obtaining a dynamically, 595

U

UDP protocol, 760, 773
UI delegate, 1028
ulp(), 516, 518
UnaryOperator<T> functional interface, 417, 676
UnavailableException, 1127, 1130
Unboxing, 289
UncaughtExceptionHandler interface, 534
 method uncaughtException(), 534
Unchecked warnings and raw types, 372
UnicastRemoteObject, 1008
Unicode, 43, 44, 47, 316, 317, 451, 452, 456, 494, 708
 code points, table of some Character methods providing support for, 496
 support for 34-bit, 494–495

Unicode Technical Standard (UTS) [37](#), [636](#)
Uniform Resource Identifier (URI), [772](#)
Uniform Resource Locator (URL). *See* URL (Uniform Resource Locator)
UNIX, [5](#), [759](#)
UnknownHostException, [762](#)
unlock(), [921](#), [948](#), [949](#)
unmodifiable... collections methods, [594](#)–[595](#)
unordered(), [970](#), [981](#)
Unreachable code, [115](#), [233](#)
unread(), [701](#), [716](#)–[717](#)
UnsupportedOperationException, [222](#), [240](#), [540](#), [543](#), [561](#), [571](#), [595](#), [737](#)
update(), [502](#)–[503](#), [822](#), [823](#), [824](#)
URI (Uniform Resource Identifier), [772](#)
URI class, [772](#)
URL (Uniform Resource Locator), [766](#)–[767](#), [772](#), [1121](#), [1122](#)
 query string, [1138](#)
 specification format, [766](#)–[767](#)
URL class, [766](#)–[768](#), [769](#), [770](#)
URLConnection class, [768](#)–[771](#), [772](#)
 methods, list of some, [768](#)–[769](#)
useDelimiter(), [665](#)–[666](#)
User Datagram Protocol (UDP), [760](#), [773](#)
useRadix(), [667](#)
uses
 keyword, [37](#), [420](#)
 statement, [435](#), [441](#), [442](#)
UTS [37](#), [636](#)
UUID class, [672](#)

V

val(), [295](#)
value (annotation member name), [304](#), [305](#)
valueChanged(), [1061](#), [1062](#), [1066](#), [1067](#)
valueOf(), [280](#)–[281](#), [287](#), [288](#), [454](#), [465](#), [479](#), [481](#), [482](#), [483](#), [484](#), [485](#), [488](#),
 [490](#), [492](#), [495](#), [497](#), [531](#), [622](#)

values(), 280–281, 571, 574
van Hoff, Arthur, 6
var, context-sensitive reserved type name, 19, 38, 63–65, 107, 169, 195, 331, 381, 394
 restrictions on using, 64–65, 348
Varargs, 16, 163–168
 and ambiguity, 167–168
 methods, overloading, 166–167
 and Java’s printf(), 163
 parameter, 165, 545
Variable(s), 48–52
 capture, 403–404
 declaration, 31, 33, 48–49, 50–52, 62–65
 definition of the term, 30, 48
 dynamic initialization of, 49
 effectively final, 403–404
 enumeration, 278
 final, 155, 277
 instance. *See* Instance variables
 interface, 209, 216–218
 interface reference, 210–212, 216
 and JShell, 1156–1157
 loop control, 99, 100, 101, 102, 103
 member, 23
 object reference. *See* Object reference variables
 scope and lifetime of, 49–52
 temporary, 1162
 type inference, local. *See* Type inference, local variable
 volatile, 333
Vector class, 538, 551, 570, 602, 603–607
 legacy methods, table of, 604–605
Version information, obtaining, 502–503
version(), 499, 503
VetoableChangeListener interface, 1113, 1114
Viewport, scroll pane, 1058, 1059
visitFile(), 756, 758

void, 29, 123
Void class, 495
volatile modifier, 333

W

wait(), 197, 265–266, 268–269, 272–274, 510, 919, 947, 949, 968
waitFor(), 498, 501
WAITING, 274
WALL_TIME, 635
walk(), 733
walkFileTree(), 756–758
Warth, Chris, 6
Watchable interface, 732
WeakHashMap class, 578
Web. *See* World Wide Web
Web browser, 766, 1121, 1122, 1125, 1126
executing applet in, 8, 11
Web server and servlets, 1122
Web Start. *See* Java Web Start
WebSocket
 class, 782
 protocol, 777
weightx constraint field, 879, 880
weighty constraint field, 879, 880
while loop, 95–96
 and break, 110
 and continue, 113
Whitespace, 36, 88
 characters, 619
 from a string, removing, 464–465
with keyword, 37, 420, 435
whois, 760, 764–766
Wildcard arguments, 357–364, 378
 bounded, 360–364
 used in creating an array, 387
Window AWT-based

Window, Window-based

 class hierarchy, [818–819](#)
 close event, [801, 804, 809, 821](#)
 as an event source, [795, 796](#)
 frame. *See* Frame window
 and graphics, [824](#)
 paint mode, setting the, [832–833](#)
 top-level, [819](#)
 Window class, [796, 818, 819, 821, 888](#)
 Window, Swing JFrame, [1032](#)
 windowActivated(), [800](#)
 WindowAdapter class, [808, 809](#)
 windowClosed(), [800](#)
 windowClosing(), [800, 804, 808, 821, 823](#)
 WindowConstants interface, [1033](#)
 windowDeactivated(), [800](#)
 windowDeiconified(), [800](#)
 WindowEvent class, [787, 789, 795–796](#)
 WindowFocusListener interface, [797, 799, 809](#)
 windowGainedFocus(), [799](#)
 windowIconified(), [800](#)
 WindowListener interface, [797, 800, 804, 808, 809, 821, 823](#)
 windowLostFocus(), [799](#)
 windowOpened(), [800](#)
 WindowStateListener interface, [809](#)
 Work stealing, [955, 967](#)
 World Wide Web (WWW), [6, 7, 13, 20, 767](#)
 wrap(), [729](#)
 Wrappers, primitive type, [286–289, 294, 350, 478–495](#)
 WRITE, [505](#)
 write(), [317, 318, 322, 328–329, 691, 710, 721, 722, 731, 744, 745, 749](#)
 writeBoolean(), [706, 722](#)
 writeDouble(), [706, 722](#)
 writeExternal(), [721](#)
 writeInt(), [706, 722](#)
 writeObject(), [721, 722, 725](#)

Writer class, 317–318, 688, 689, 708, 709–710, 711, 721
methods defined by, table of, 710
`writeString()`, 735
`writeTo()`, 698

X

XOR (exclusive OR) operator (^)
bitwise, 72, 73, 74–75
Boolean logical, 81, 82
XOR paint mode, 832–833

Y

Yellin, Frank, 6

Z

Zero crossing, 73
ZIP file format, 677

Beta Test Oracle Software

**Get a first look at our newest products
and features—and help perfect them.**

You must meet the following criteria:

- ✓ Licensed Oracle customer or
Oracle PartnerNetwork member
- ✓ Oracle software expert
- ✓ Early adopter of Oracle products

Apply at: oracle.com/goto/beta

ORACLE®

Join the World's Largest Developer Community



Download the latest software, tools,
and developer templates



Get exclusive access to hands-on
trainings and workshops



Grow your network with the Developer
Champion and Oracle ACE Programs



Publish your technical articles—and
get paid to share your expertise

ORACLE DEVELOPER COMMUNITY developer.oracle.com

Membership Is Free | Follow Us on Social:

@OracleDevs facebook.com/OracleDevs

ORACLE®

Certification **MATTERS**

72% Experienced a Greater Demand for Their Skills¹

67% Said Certification was a Key Factor in Recent Raise¹

64% Received Positive Impact on Professional Image²

Oracle University

Differentiate Yourself to Attract Employers

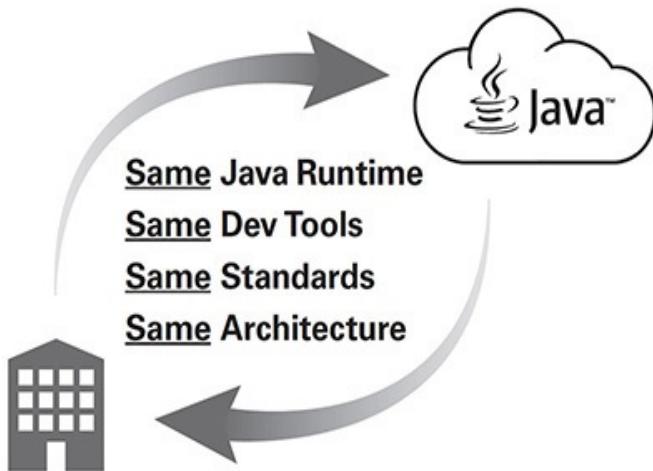
ORACLE®

certification.oracle.com

Source: 1—Certification Magazine, Annual Salary Survey, January 2018 | 2—Pearson VUE, Value of IT Certification Survey, 2017.
Copyright © 2018, Oracle and/or its affiliates. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates.

Push a Button

Move Your Java Apps to the Oracle Cloud



... or Back to Your Data Center

ORACLE®

cloud.oracle.com/java
or call 1.800.ORACLE.1

Oracle Learning Library

**Created by Oracle Experts
FREE for Oracle Users**

- ✓ Vast array of learning aids
- ✓ Intuitive & powerful search
- ✓ Share content, events & saved searches
- ✓ Personalize your learning dashboard
- ✓ Find & register for training events

ORACLE®

oracle.com/oll

Don't Miss Out

ENGAGE WITH YOUR PEERS ON
THE PLATFORM OF YOUR CHOICE



ORACLE
MAGAZINE

PROFIT

Java
magazine

Subscribers:
320,000+

Audience:
IT Managers, DBAs,
Programmers, and
Developers

Subscribers:
66,000+

Audience:
Top Executives and
Line of Business
Managers

Subscribers:
254,000+

Audience:
Developers,
Programmers,
and Architects

ORACLE®



For more information or to sign up for a FREE subscription:
Scan the QR code to visit Oracle Publishing online.

Copyright © 2018, Oracle and/or its affiliates. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates.