Under the Java 1.1 event model, the event source is only half of the event delegation equation. The torrents of events that a GUI application may generate don’t do any good without an object that can receive and process the events. These event-receiving objects are called event listeners.

Java provides the framework for an event listener but leaves the implementation of that framework up to the programmer. It does this by implementing the event listeners as interfaces. These interfaces declare methods that will be called when the appropriate type of event is generated from an event source. An event listener class of a given type must implement the methods defined in the event listener, but what code is placed inside the event listener methods is left completely up to the developer. When an event of that type is generated by an event source, the event-dispatching thread will send the event to the appropriate event listener method.

This is a very powerful event handling process and is a central and crucial element of the Java 1.1 event model. It allows Java event handling to be both compact and completely expandable. For example, an ItemEvent can be generated by many different types of components and can represent many different types of user interactions. The methodology is compact because every ItemEvent can be processed by the same ItemEvent listener as long as the listener’s itemStateChanged() method is written to process the desired
ItemEvent. The methodology is expandable because new components that generate ItemEvents can be added to the Java API without altering the structure of the ItemEvent listener.

A Brief Review of Interfaces

Java does not allow multiple inheritance. This means that a class can have only one superclass, which in turn can have only one superclass, and so on. This simplifies many aspects of Java programming, but can lead to problems. For instance, what if classes that have different inheritance hierarchies need to share some common behavior? One way to solve this problem is to place the shared behavior higher up the inheritance hierarchy in a location common to both classes. Another solution is to simply copy the functionality into both classes. Neither of these solutions is desirable, however, as they violate object-oriented programming principles of shared behavior and code reuse.

The acceptable solution to this problem is through the use of interfaces. An interface declares a set of methods and constants. There are no instance variables in an interface, and the methods declared in the interface are stubs (methods without bodies). A class can implement any number of interfaces. The implementation of the methods declared in the interface is left up to the class that implements the interface. In effect, the interface imposes a certain functionality upon a class, but provides the class the flexibility to determine how to implement that functionality. For instance, the ActionListener interface declares one method, actionPerformed(). Every class that implements the ActionListener interface must provide an actionPerformed() method. But what is inside the body of the actionPerformed() method can be different for every class. In one case, the actionPerformed() method might terminate the program. In another, the actionPerformed() method might change the background color, and so on.

Event Listener Interfaces in the J2SE

The EventListener interface, defined in the java.util package, is the parent interface for all event listener interfaces in the Java API. The EventListener interface declares no methods but is used as a marker to indicate that any subinterfaces of it are event listeners. Most listener interfaces are direct
subinterfaces of EventListener, but some are subinterfaces of EventListener subinterfaces.

Most of the event listener interfaces contained in the J2SE are defined in the java.awt.event, javax.swing.event, and java.beans packages. These interfaces will be described briefly here and in more detail in Chapter 8, “Event Listener Interfaces.” Event listener interfaces are also defined in the java.beans.beancontext, java.awt.dnd, javax.sound.sampled, javax.naming.event, and javax.naming.ldap packages. These interfaces are touched on briefly but are not covered in detail.

**EventListener Interface**

The EventListener interface is from the java.util package and is the parent of all listener interfaces. It declares no methods or constants but is used as a marker to indicate that any interface that extends EventListener is an event listener interface.

**Listener Interfaces Contained in the java.awt.event Package**

The listener interfaces defined in the java.awt.event package are all direct subinterfaces of EventListener. They are used to receive and process the events found in the java.awt.event package. The following is a table and a short description of these interfaces. A complete description of these listener interfaces as well as examples on how they are used can be found in Chapter 8.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Extends</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionListener</td>
<td>EventListener</td>
</tr>
<tr>
<td>AdjustmentListener</td>
<td>EventListener</td>
</tr>
<tr>
<td>AWTEventListener</td>
<td>EventListener</td>
</tr>
<tr>
<td>ComponentListener</td>
<td>EventListener</td>
</tr>
<tr>
<td>ContainerListener</td>
<td>EventListener</td>
</tr>
<tr>
<td>FocusListener</td>
<td>EventListener</td>
</tr>
<tr>
<td>HierarchyBoundsListener</td>
<td>EventListener</td>
</tr>
<tr>
<td>HierarchyListener</td>
<td>EventListener</td>
</tr>
<tr>
<td>InputMethodListener</td>
<td>EventListener</td>
</tr>
<tr>
<td>ItemListener</td>
<td>EventListener</td>
</tr>
</tbody>
</table>
The names of these listener interfaces pretty much tell you what it is they do. The ActionListener interface declares a method for processing ActionEvents, The KeyListener interface declares methods for processing KeyEvents, and so on. There are two listener interfaces for dealing with MouseEvents and HierarchyEvents. There is also the AWTEventListener interface that is not tied to any specific AWTEvent. These listeners are described here in more detail.

AWTEventListener declares one method, eventDispatched(), which is used to passively monitor events being dispatched in the AWT. The method is called whenever an event is dispatched in the AWT.

HierarchyBoundsListener is one of two listener interfaces that receive and process HierarchyEvent objects. It declares two methods that are called if an ancestor in a registered component’s hierarchy is moved or resized.

HierarchyListener declares methods for processing non-bounding-type HierarchyEvent objects. These can happen, for instance, if an ancestor is added or removed from the hierarchy or if the hierarchy is made visible or invisible.

MouseListener is one of two interfaces for responding to MouseEvents. The MouseListener interface declares methods for processing non-motion-oriented MouseEvents. These occur whenever the mouse is pressed, released, clicked (pressed and released), or enters or leaves the bounding area of a GUI component.

MouseMotionListener declares methods to process motion-oriented MouseEvent objects. These occur whenever the mouse moves or is dragged within the bounding area of a GUI component.

As a side note on WindowEvent objects, one of the idiosyncrasies of Java is that when the main window of an application is closed, the application does not automatically terminate. Before J2SE 1.3, the best way to ensure that the application would terminate when the main window was closed was to register the window with a WindowListener. The windowClosing() method defined in that interface is implemented to call the System.exit() method. For Swing
containers running under J2SE 1.3, the application can be set to terminate upon window closing by using the `setDefaultCloseOperation()` method.

### Listener Interfaces Contained in the `javax.swing.event` Package

The interfaces contained in the `javax.swing.event` package are used to receive and process the Swing event classes. With one exception, they are all direct subinterfaces of the `EventListener` interface. The following is a table and a brief description of each interface. A complete description of the `javax.swing.event` listener interfaces as well as examples on how to use them can be found in Chapter 8.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Extends</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>AncestorListener</code></td>
<td><code>EventListener</code></td>
</tr>
<tr>
<td><code>CaretListener</code></td>
<td><code>EventListener</code></td>
</tr>
<tr>
<td><code>CellEditorListener</code></td>
<td><code>EventListener</code></td>
</tr>
<tr>
<td><code>ChangeListener</code></td>
<td><code>EventListener</code></td>
</tr>
<tr>
<td><code>DocumentListener</code></td>
<td><code>EventListener</code></td>
</tr>
<tr>
<td><code>HyperlinkListener</code></td>
<td><code>EventListener</code></td>
</tr>
<tr>
<td><code>InternalFrameListener</code></td>
<td><code>EventListener</code></td>
</tr>
<tr>
<td><code>ListDataListener</code></td>
<td><code>EventListener</code></td>
</tr>
<tr>
<td><code>ListSelectionListener</code></td>
<td><code>EventListener</code></td>
</tr>
<tr>
<td><code>MenuDragMouseListener</code></td>
<td><code>EventListener</code></td>
</tr>
<tr>
<td><code>MenuKeyListener</code></td>
<td><code>EventListener</code></td>
</tr>
<tr>
<td><code>MenuListener</code></td>
<td><code>EventListener</code></td>
</tr>
<tr>
<td><code>MouseInputListener</code></td>
<td><code>MouseListener, MouseMotionListener</code></td>
</tr>
<tr>
<td><code>PopupMenuListener</code></td>
<td><code>EventListener</code></td>
</tr>
<tr>
<td><code>TableColumnModelListener</code></td>
<td><code>EventListener</code></td>
</tr>
<tr>
<td><code>TableModelListener</code></td>
<td><code>EventListener</code></td>
</tr>
<tr>
<td><code>TreeExpansionListener</code></td>
<td><code>EventListener</code></td>
</tr>
<tr>
<td><code>TreeModelListener</code></td>
<td><code>EventListener</code></td>
</tr>
</tbody>
</table>
As with the java.awt.event package event listener interfaces, the javax.swing.event interface names generally indicate what the interfaces are intended to do. A DocumentListener declares methods to process DocumentEvents, a PopupMenuListener declares methods to process PopupMenuEvents, and so on. There are some exceptions. The CellEditorListener receives ChangeEvents generated by a CellEditor, for instance.

A CellEditorListener is used to process ChangeEvents generated by a CellEditor when an editing session is canceled or stopped. The HyperlinkListener serves a dual purpose. It is used to receive and process both HyperlinkEvents and HTMLFrameHyperlinkEvent objects. The TableColumnModelListener defines methods for receiving and processing every type of event a TableColumnModel can generate. These include TableColumnModelEvents, ChangeEvents, and ListSelectionEvents.

TreeExpansionEvents can be sent to either a TreeExpansionListener or a TreeWillExpandListener. The TreeExpansionListener is called after a JTree node has been expanded or collapsed. The TreeWillExpandListener is called when a JTree node is about to be expanded or collapsed, before the action has taken place.

### Listener Interfaces Contained in the java.beans Package

The java.beans package provides the event listener interfaces that are used to implement bound and constrained properties. Recall that a bound property is one that is shared between multiple objects. A constrained property is one whose change can be restricted. As with the java.awt.event and javax.swing.event package listener interfaces, a complete description of the listener interfaces from the java.beans package can be found in Chapter 8.

PropertyChangeListener declares one method, propertyChange(), that is called when a PropertyChangeEvent object is fired from an event.
source. The PropertyChangeEvent might represent a change to a bound property or a proposed change to a constrained property.

VetoableChangeListener declares one method, vetoableChange(), that is called when an attempt is made to change a constrained property of an event source.

### Other Listener Interfaces Defined in the J2SE

There are some additional listener interfaces defined in the J2SE beyond those found in the java.awt.event, javax.swing.event, and java.beans packages. These listener interfaces are not described in detail in this book. They are listed in the table below, and a brief description of them is provided in Chapter 8.

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Extends</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeanContextMembershipListener</td>
<td>EventListener</td>
</tr>
<tr>
<td>BeanContextServiceRevokedListener</td>
<td>EventListener</td>
</tr>
<tr>
<td>BeanContextServicesListener</td>
<td>EventListener</td>
</tr>
<tr>
<td>ControllerEventListener</td>
<td>EventListener</td>
</tr>
<tr>
<td>DragGestureListener</td>
<td>EventListener</td>
</tr>
<tr>
<td>DragSourceListener</td>
<td>EventListener</td>
</tr>
<tr>
<td>DropTargetListener</td>
<td>EventListener</td>
</tr>
<tr>
<td>LineListener</td>
<td>EventListener</td>
</tr>
<tr>
<td>MetaEventListener</td>
<td>EventListener</td>
</tr>
<tr>
<td>NamespaceChangeListener</td>
<td>NamingListener</td>
</tr>
<tr>
<td>NamingListener</td>
<td>EventListener</td>
</tr>
<tr>
<td>ObjectChangeListener</td>
<td>NamingListener</td>
</tr>
<tr>
<td>UnsolicitedNotificationListener</td>
<td>NamingListener</td>
</tr>
</tbody>
</table>

### Event Listener Objects

An event listener class is one that implements one or more event listener interfaces. An event listener class provides implementations of the methods declared in the interfaces to suit the specific needs of an application. Any class
can serve as an event listener, and there is more than one way to implement an event listener.

For an event listener to receive events generated from an event source, the event source must add the listener to its listener list. The event source does this by invoking an `addListener()` method. This process is also known as having the event source register the listener. Similarly, if you wish to disconnect the listener from the event source, the event source must remove the listener from its listener list. The event source does this by calling a `removeListener()` method. The process of connecting or disconnecting an event listener and event source is covered in more detail later in this chapter.

Creating an Event Listener

Because any class can serve as an event listener, there are several ways to implement an event listener. Remember that for a class to serve as an event listener class, all the class need do is implement the appropriate event listener interface and provide implementation for the event handling methods defined in the interface. Each of the different ways to create an event listener is more useful in some situations, less useful in others. Four ways to create an event listener are discussed in the following sections.

Method 1: Have a GUI Component Serve as the Event Listener

This is the situation that is most similar to the Java 1.0.2 event model, where event handling was always performed by a Component subclass. The component, oftenentimes the container, implements the listener interface and provides implementation of the methods declared in the interface. This method has its advantages. Because the event handling code is contained within the GUI code, the event handling methods automatically have access to the private data members of the GUI. However, this method becomes awkward if a large number of GUI components are sending events to the same listener because logic must be built in to the event handler method to determine which event generated the event before applying the appropriate event processing.

EXAMPLE

Example 4.1 is very similar to the SimpleExample class described in Chapter 1, “An Introduction to Java Events.” The container class serves as an ActionListener and implements the `actionPerformed()` method.
EXAMPLE 4.1 A GUI COMPONENT SERVES AS THE EVENT LISTENER

```java
import java.awt.*;
import java.awt.event.*;

public class TestListener extends Frame implements ActionListener
{
    private Button b;

    public TestListener()
    {
        /* A Button is created and registers an ActionListener         */
        /* Because the TestListener class itself is serving as the     */
        /* ActionListener, the addActionListener() method is passed    */
        /* the *this* reference.                                       */

        b = new Button("quit");
        b.addActionListener(this);

        setLayout(new FlowLayout());
        add(b);

        setBounds(100, 100, 200, 200);
        setVisible(true);
    }

    /* The TestListener class serves as the ActionListener and must */
    /* provide an implementation of the actionPerformed method.      */

    public void actionPerformed(ActionEvent ae)
    {
        System.exit(0);
    }

    public static void main(String args[])
    {
        TestListener tl = new TestListener();
    }
}
```

In this example, a button is placed on a frame. The TestListener class serves as an ActionListener and implements the `actionPerformed()` method. The button adds the ActionListener to its listener list by invoking the `addActionListener()` method and passing the method the “this” reference. When the button is pressed, an ActionEvent is generated and sent to the `actionPerformed()` method where the program is terminated.

As a general note, there is no limit to the number of listener interfaces a class can implement. If the container is serving as the event listener, it is not uncommon for it to implement multiple event listeners.
Method 2: Implement the Event Listener as a Separate Class

One of the advantages of the Java 1.1 event model is that it allows the event handling to be performed separately from the GUI development. The event listener can be implemented as a separate class. This has the advantage of making the event handling code completely portable and is useful for situations where the event handler does not need access to the private data members of the GUI.

EXAMPLE

In Example 4.2, an ActionListener is implemented as a separate class. The QuitHandler class implements the ActionListener interface and overrides the actionPerformed() method.

EXAMPLE 4.2 EVENT LISTENER IMPLEMENTED AS A SEPARATE CLASS

```java
import java.awt.*;
import java.awt.event.*;

public class TestListener2 extends Frame
{
    private Button b;
    public TestListener2()
    {
        /* A Button is created and registers an ActionListener. */
        /* The addActionListener() method is passed a reference to */
        /* an ActionListener object. */
        b = new Button("quit");
        b.addActionListener(new QuitHandler());

        setLayout(new FlowLayout());
        add(b);
        setBounds(100, 100, 200, 200);
        setVisible(true);
    }

    public static void main(String args[])
    {
        TestListener2 tl = new TestListener2();
    }
}

/* The ActionListener is implemented as a separate class */
class QuitHandler implements ActionListener
public void actionPerformed(ActionEvent ae) {
    System.exit(0);
}

This example is similar to the previous one, except that the event listener is implemented as a separate class. The frame no longer implements the ActionListener interface and no longer provides implementation of the actionPerformed() method. This is done in the QuitHandler class. The button adds the ActionListener to its listener list by invoking the addActionListener() method and passing the method a reference to a QuitHandler object.

Method 3: Implement the Event Listener as an Inner Class

This is a combination of methods 1 and 2. The event handling is performed by a separate class, but this class is written as an inner class. The advantage of this method is that the event handler has access to the private data members of the outer class and allows the event handling to be modularized. Every GUI component could conceivably have its own corresponding inner class to handle the events it generates.

EXAMPLE

In Example 4.3, the TestListener3 class is a frame that contains a button and a textfield. The TestListener3 class also defines an instance variable named count. The button registers an ActionListener that is implemented as an inner class. When the button is pressed, an ActionEvent is generated and sent to the actionPerformed() method defined in the ButtonHandler class. The count is incremented and the textfield is updated to display the current value of the count.

EXAMPLE 4.3 EVENT LISTENER IMPLEMENTED AS AN INNER CLASS

import java.awt.*;
import java.awt.event.*;

public class TestListener3 extends Frame {
    private Button b;
    private TextField textField;
    private int count;
```java
public TestListener3()
{
    /* The count is initialized to be zero */
    count = 0;

    /* A Button is created and placed on a Frame */
    /* The Button registers an ActionListener */

    b = new Button("Add");
b.addActionListener(new ButtonHandler());

    textField = new TextField(20);
textField.setEditable(false);
textField.setText("count is "+count);

    setLayout(new FlowLayout());
    add(b);
    add(textField);

    addWindowListener(new WinAdapter());
    setBounds(100, 100, 200, 200);
    setVisible(true);
}

/* The ActionListener is implemented as an inner class, */
/* so the actionPerformed() method has access to the */
/* count variable. */

class ButtonHandler implements ActionListener
{
    public void actionPerformed(ActionEvent ae)
    {
        ++count;
textField.setText("count is "+count);
    }
}

public static void main(String args[])
{
    TestListener3 tl = new TestListener3();
}

/* This makes sure the application terminates if the window is closed */

class WinAdapter extends WindowAdapter
{
    public void windowClosing(WindowEvent event)
    {
```
When you run this example, a button and a textfield appear on a frame. Press the button and notice how the count is updated in the textfield. This is possible because the `actionPerformed()` method has access to the count instance variable.

Method 4: Implement the Event Listener as an Anonymous Inner Class

The most compact way of implementing an event listener is by using an anonymous inner class. The code for the anonymous inner class is passed as an argument to the `addListener()` method. This is useful for simple event handlers, although it does tend to make the code somewhat more difficult to follow.

**EXAMPLE**

In Example 4.4, a button is placed on a frame. The code to process the events generated by the button is passed to the `addActionListener()` method as an anonymous inner class.

**EXAMPLE 4.4 Event Listener Implemented as an Anonymous Inner Class**

```java
import java.awt.*;
import java.awt.event.*;

public class TestListener4 extends Frame {
    private Button b;

    public TestListener4() {
        /* A Button is created and registers an ActionListener. */
        /* The addActionListener() method is passed a reference to an */
        /* ActionListener object. The ActionListener is implemented */
        /* as an anonymous inner class. The entire anonymous inner */
        /* class is passed as an argument to the addActionListener() */
        /* method. */

        b = new Button("quit");
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                // Count is updated
            }
        });
    }
}
```
System.exit(0);
}
});

setLayout(new FlowLayout());
add(b);

setBounds(100, 100, 200, 200);
setVisible(true);

public static void main(String args[])
{
    TestListener4 tl = new TestListener4();
}

The code inside the curved brackets of the `addActionListener()` syntax defines an anonymous inner class that implements the `ActionListener` interface. It provides an implementation of the `actionPerformed()` method. It is very compact and is well suited to simple operations such as terminating the program when the Quit button is pressed. But the code looks a little squirrelly, and it can take even experienced programmers a while to figure out what is going on here.

### Listener Adapter Classes

An annoying aspect of interfaces is that a class that implements an interface must provide implementations for all methods declared in the interface. Oftentimes, a developer only needs to use some of the methods for a given application. Implementations of the other methods must be included even if they are written as stub methods (methods with an empty body).

To avoid this nuisance, the Java programming language provides event listener adapter classes for many of the listener interfaces that define more than one method. These adapter classes implement the corresponding listener interface and provide stubs for all of the methods declared in the interface. The developer then need only override the methods required for a given application.

### EXAMPLE

As an example, let us look at the problem described previously—that of ensuring that an application terminates when its main window is closed. First, in
Example 4.5, let us look at the code required if a WindowListener is used that directly implements the WindowListener interface.

**EXAMPLE 4.5 WINDOWLISTENER IMPLEMENTING WINDOWLISTENER INTERFACE**

```java
import java.awt.*;
import java.awt.event.*;

public class NoAdapter extends Frame
{
    private Label lbl;

    public NoAdapter()
    {
        lbl = new Label("No Adapters Here");
        setLayout(new FlowLayout());
        add(lbl);
        addWindowListener(new WinListener());
        setBounds(100, 100, 200, 200);
        setVisible(true);
    }

    public static void main(String args[])
    {
        NoAdapter na = new NoAdapter();
    }
}

/* This is a WindowListener that implements the */
/* WindowListener interface directly. Every method defined */
/* in the WindowListener interface must be implemented. */
/* In this case six of the seven methods are implemented */
/* as stub methods. */

class WinListener implements WindowListener
{
    public void windowClosing(WindowEvent we)
    {
        System.exit(0);
    }

    public void windowActivated(WindowEvent we) {}
    public void windowClosed(WindowEvent we) {}
    public void windowDeactivated(WindowEvent we) {}
    public void windowDeiconified(WindowEvent we) {}
    public void windowIconified(WindowEvent we) {}
    public void windowOpened(WindowEvent we) {}
}
The WindowListener is defined as a separate class called WinListener that implements the WindowListener interface. We only really care about the windowClosing() method, but have to provide stubs for the other six methods defined in the interface.

Now, in Example 4.6, let’s look at the code if an adapter class is used.

**Example 4.6 WindowListener Using an Adapter Class**

```java
import java.awt.*;
import java.awt.event.*;

public class WithAdapter extends Frame
{
    private Label lbl;

    public WithAdapter()
    {
        lbl = new Label("Adapters are nice");

        setDefaultCloseOperation(new FlowLayout());
        add(lbl);

        addWindowListener(new WinListener());
        setBounds(100, 100, 200, 200);
        setVisible(true);
    }

    public static void main(String args[])
    {
        WithAdapter na = new WithAdapter();
    }

    /* This WindowListener extends the WindowAdapter class */
    /* The WindowAdapter class is a WindowListener that */
    /* provides stub implementations for all the methods */
    /* from the WindowListener interface. The program */
    /* only needs the windowClosing() method, so that is */
    /* the only method that needs to be implemented in the */
    /* WinListener class. */

    class WinListener extends WindowAdapter
    {
        public void windowClosing(WindowEvent we)
        {
            System.exit(0);
        }
    }
}
```
In this case, the WinListener class is written as a subclass of the WindowAdapter class. The WindowAdapter class implements the WindowListener interface and provides stubs for the seven methods declared in that interface. The WinListener class only needs to override the windowClosing() method. The other six methods are taken care of by the adapter class.

Listener Adapter Classes Contained in the java.awt.event Package

The following is a short description of the event listener adapter classes found in the java.awt.event package. There is an adapter class for every listener interface that declares more than one method, with the exception of the InputMethodListener interface. A complete description of these adapter classes as well as examples of how to use them can be found in Chapter 9, “Listener Adapter Classes.”

<table>
<thead>
<tr>
<th>class</th>
<th>extends</th>
<th>implements</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComponentAdapter</td>
<td>Object</td>
<td>ComponentListener</td>
</tr>
<tr>
<td>ContainerAdapter</td>
<td>Object</td>
<td>ContainerListener</td>
</tr>
<tr>
<td>FocusAdapter</td>
<td>Object</td>
<td>FocusListener</td>
</tr>
<tr>
<td>HierarchyBoundsAdapter</td>
<td>Object</td>
<td>HierarchyBoundsListener</td>
</tr>
<tr>
<td>KeyAdapter</td>
<td>Object</td>
<td>KeyListener</td>
</tr>
<tr>
<td>MouseAdapter</td>
<td>Object</td>
<td>MouseListener</td>
</tr>
<tr>
<td>MouseMotionAdapter</td>
<td>Object</td>
<td>MouseMotionListener</td>
</tr>
<tr>
<td>WindowAdapter</td>
<td>Object</td>
<td>WindowListener</td>
</tr>
</tbody>
</table>

Note that all of these adapter classes are abstract. This is because adapter classes are intended to be subclassed rather than used directly. They all provide stubs for the methods declared in their associated listener interface. The adapter classes are subclasses of Object and have access to the methods defined in the Object class.
Listener Adapter Classes Contained in the javax.swing.event Package

Unlike the java.awt.event package, where there is an adapter class for nearly every event listener that defines more than one method, the javax.swing.event package contains only two adapter classes. A short description of the two classes follows. As before, a complete description of these classes and examples of how to use them can be found in Chapter 9.

```java
public abstract class InternalFrameAdapter extends Object
  implements InternalFrameListener

public abstract class MouseInputAdapter extends Object
  implements MouseInputListener
```

InternalFrameAdapter provides stubs for the seven methods declared in the InternalFrameListener interface.

MouseInputAdapter provides stubs for the methods used to process MouseEvent and is useful if you want to access both motion-oriented and non-motion-oriented MouseEvent processing methods.

User-Defined Event Listener Interfaces

It really is quite simple to create a user-defined listener interface. The user-defined interface will extend either the EventListener interface or one of the EventListener subinterfaces. The user-defined interface then declares whatever methods are desired. An example of how this is done is provided in Chapter 12, “User-Defined Event Classes and Event Listeners.”

Once the user-defined listener interface has been created, a mechanism must be set up to add or remove a user-defined listener object from the listener list maintained by the event source component. This can be done using either the EventListenerList or AWTEventMulticaster classes. A description of these classes and an example of how to use them are provided in Chapter 11, “Event Listener Manager Classes.”
Connecting an Event Source to an Event Listener

The delegation model used by the Java 1.1 event model is not complete without a mechanism for delivering the events from source to listener. The system must know where to send the events. The mechanism that is used is a listener list. Every event-generating object, whether system-defined or user-defined, maintains a list of associated event listeners. When the object generates a certain type of event, the listener list is checked to see if it contains any listeners of that type. If it does, the event is sent to those listeners. This process is performed automatically for system-generated events. For user-generated events, the list-checking code is incorporated into the method used to fire the event.

Adding a listener to the listener list of an event-generating object is quite easy. Java provides methods to add every built-in listener type. The methods to add low-level event listeners are provided in the Component, Container, and JComponent classes. For example, the Component class defines the following methods.

```java
public void addComponentListener(ComponentListener listener)
public void addFocusListener(FocusListener listener)
public void addHierarchyBoundsListener(HierarchyBoundsListener listener)
public void addHierarchyListener(HierarchyListener listener)
public void addInputMethodListener(InputMethodListener listener)
public void addKeyListener(KeyListener listener)
public void addMouseListener(MouseListener listener)
public void addMouseMotionListener(MouseMotionListener listener)
public void addPropertyChangeListener(PropertyChangeListener listener)
```

A reference to the listener to be added is passed to the method as an argument. The process of an event-generating object adding a listener to its listener list is also referred to as having the object register an event listener. Note that it is the event source that invokes these methods. Therefore the proper terminology is that the event source registers an event listener.
Methods to add high-level event listeners to a listener list are generally defined in the classes of the objects that generate the event. For example, a JEditorPane object can generate HyperlinkEvents. The JEditorPane class defines the `addHyperlinkListener()` method to add a HyperlinkListener to the JEditorPane object’s listener list.

There is no practical limit to the number of event listeners that can be added to an object’s listener list. There are limits to the type of event listeners that can be added, and this varies from object to object, depending on the types of events the object can generate. For instance, a JButton can add a ComponentListener, aMouseListener, and 10 ActionListeners, but it cannot add a DocumentListener because a JButton does not generate DocumentEvents and does not have access to an `addDocumentListener()` method. An event will be sent to every listener of that type contained in the listener list. In the previous example, if the JButton generated an ActionEvent, it would be sent to all 10 ActionListeners in the JButton object’s listener list.

See Example 4.7 in the next section, “Disconnecting an Event Source from an Event Listener.”

### Disconnecting an Event Source from an Event Listener

Just as there is a mechanism for adding event listeners to an object’s listener list, there is a way to remove listeners as well. The process is identical. Java provides `remove()` methods for every type of event listener. The event-generating object invokes the method and passes it a reference to the listener it wants to remove from its listener list. The low-level event listener removal methods are contained in the Component, Container, and JComponent classes. For instance, the Component class defines the following methods.

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public void removeComponentListener(…)</code></td>
</tr>
<tr>
<td><code>public void removeFocusListener(…)</code></td>
</tr>
<tr>
<td><code>public void removeHierarchyBoundsListener(…)</code></td>
</tr>
<tr>
<td><code>public void removeHierarchyListener(…)</code></td>
</tr>
<tr>
<td><code>public void removeInputMethodListener(…)</code></td>
</tr>
</tbody>
</table>
public void removeKeyListener(KeyListener listener)
public void removeMouseListener(MouseListener listener)
public void removeMouseMotionListener(MouseMotionListener listener)
public void removePropertyChangeListener(PropertyChangeListener listener)

A listener that is removed from an object’s listener list will no longer be notified when the object generates an event of that type. The process of removing a listener from a listener list is also referred to as disconnecting, or deregistering, the listener. As with the methods to add listeners, it is the event source that invokes the remove() methods.

Methods to remove high-level event listeners from a listener list are generally defined in the classes of the objects that generate the event. For example, a JMenu object can generate MenuEvents. The JMenu class defines the removeMenuListener() method to remove a MenuListener from the JMenu object’s listener list.

EXAMPLE

Example 4.7 demonstrates how components add and remove listeners from their listener lists. A large yellow JPanel is placed in a JFrame. A JButton and a non-editable JTextField are placed at the bottom of the JFrame. The JPanel initially adds a MouseMotionListener to its listener list using the addMouseMotionListener() method. Whenever the mouse is moved or dragged inside the bounding area of the JPanel, a motion-oriented MouseEvent is generated and sent to the MouseMotionListener. If the mouse is moved within the bounding area of the JPanel, the mouseMoved() method is called and the current position of the mouse is printed inside the JTextField.

The JButton adds an ActionListener to its listener list using the addActionListener() method. When the JButton is pressed, an ActionEvent is generated and sent to the actionPerformed() method. Inside the actionPerformed() method, the JPanel removes the MouseMotionListener from its listener list using the removeMouseMotionListener() method. Motion-oriented MouseEvents generated by the JPanel are no longer sent to the MouseMotionListener.

EXAMPLE 4.7 HOW COMPONENTS ADD AND REMOVE LISTENERS

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class AddRemoveListener extends JFrame
{
    private JPanel panel;
    private JTextField jtf;
    private JButton button;
    private MMListener mml;

    public AddRemoveListener()
    {
        /* A MouseMotionListener object is created */
        mml = new MMListener();

        /* A JTextField, JPanel, and JButton are created and placed */
        /* on a JFrame. The JButton adds an ActionListener to its */
        /* listener list. The JPanel adds a MouseMotionListener to */
        /* its listener list. */
        jtf = new JTextField(30);
        jtf.setEditable(false);

        panel = new JPanel();
        panel.setBackground(Color.yellow);
        panel.addMouseMotionListener(mml);

        button = new JButton("Remove");
        button.setBorder(BorderFactory.createRaisedBevelBorder());
        button.setFont(new Font("Serif", Font.PLAIN, 14));
        button.addActionListener(new ButtonListener());

        JPanel p = new JPanel();
        p.add(jtf);
        p.add(button);
        getContentPane().add(panel, BorderLayout.CENTER);
        getContentPane().add(p, BorderLayout.SOUTH);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setBounds(100, 100, 600, 250);
        setVisible(true);
    }

    /* When the "remove" button is pressed, an ActionEvent is */
    /* generated and sent to the actionPerformed() method. */
    /* The JPanel removes the MouseMotionListener from its */
    /* listener list. MouseEvents generated inside the JPanel */
    /* are no longer sent to the MouseMotionListener */

    class ButtonListener implements ActionListener
    {
        public void actionPerformed(ActionEvent ae)
        {
            panel.removeMouseMotionListener(mml);
            jtf.setText("*");
        }
    }
}
/* The MouseMotionListener is called whenever a motion-oriented mouse event is generated. This MouseMotionListener extends an adapter class. Only the mouseMoved() method is overridden. */
/* When a motion-oriented MouseEvent is generated, the current mouse position is printed in the textfield. */

class MMListener extends MouseMotionAdapter
{
    public void mouseMoved(MouseEvent me)
    {
        jtf.setText("Mouse at position "+me.getX()+","+me.getY());
    }
}

public static void main(String args[])
{
    AddRemoveListener arl = new AddRemoveListener();
}

Run this application and move the mouse around inside the yellow area of the frame. You will see the current position of the mouse updated in the textfield. Press the button. The textfield is cleared. Now if you move the mouse around inside the yellow area of the frame, nothing appears inside the textfield. The JPanel is still generating MouseEvent objects, but there is nothing to listen to them and process them.

The getListeners() Method

Java provides the getListeners() method to return an array containing every listener of the specified type contained in the listener list of the invoking object. The getListeners() method is defined in many classes, including the Component class. The method syntax is given below.

```
public EventListener[] getListeners(Class listenerType)
throws ClassCastException
```

If there are no registered listeners of the specified type, an empty array is returned. The method throws a ClassCastException if the argument passed to the method is not a recognized subclass of java.util.EventListener.
The return array can be cast to whatever type may be desired. For example, if you wanted to return an array of all ActionListeners in the listener list of an object named `blah`, you would use the syntax

```java
ActionListener[] listeners = (ActionListener[])(blah.getListeners(ActionListener.class));
```

**Event Listener Manager Classes**

The J2SE provides two classes that are used to manage event listener lists and for dispatching an event to the appropriate listener when one is generated. The `AWTEventMulticaster` class is used for managing AWT event listeners. It maintains a separate list for each listener type. The `EventListenerList` class can be used for any event type and maintains a single list for all listeners. Both the `AWTEventMulticaster` and `EventListenerList` classes can be used to create components that fire user-defined events.

**AWTEventMulticaster Class**

The `AWTEventMulticaster` class, defined in the java.awt package, is used to manage registered event listeners and to dispatch events that are subclasses of the `AWTEvent` class. What the `AWTEventMulticaster` does is maintain a linked list of event listeners. It can do this for every listener type defined in the java.awt package. Every time an event source updates its listener list, a new `AWTEventMulticaster` object is created containing the updated list. The `AWTEventMulticaster` object also has access to the methods defined in the event listeners. Whenever an event is generated, the `AWTEventMulticaster` sends the event to the appropriate method for every listener in the linked list.

This operation usually goes on in the background and is of no concern to the programmer. However, the `AWTEventMulticaster` class can be used to create new types of components. It can also be used to extend the event capabilities of existing GUI components. For instance, it is possible to define a new `JLabel` class such that this new `JLabel` object will generate an `ItemEvent` when the mouse is pressed over its bounding area. In other words, it is possible to define a `JLabel` object that can be selected. This is really more of an advanced topic and is covered in detail, including an example, in Chapter 11.
EventListenerList Class

Like the AWTEventMulticaster class, the EventListenerList class, defined in the javax.swing.event package, facilitates the addition and removal of event listeners from a listener list. Unlike the AWTEventMulticaster class, an EventListenerList can handle any type of event. An EventListenerList maintains a single list containing all of the listeners registered to a given object. The list is really a series of class description and listener object pairs. The Class objects let you determine the type of each listener.

The JComponent class defines an EventListenerList field named listenerList. This means that any subclass of JComponent has access to an EventListenerList that can be used to register event listeners to that component.

A detailed discussion of the workings and use of the EventListenerList class is an advanced topic that can be found in Chapter 11.