Chapter 11. Handling Mouse and Keyboard Events

Topics in This Chapter

- The general strategy for event handling
- Using a separate object to handle events
- Implementing an interface to handle events
- Using inner classes to handle events
- Understanding the standard event listeners
- Handling events with the `process`*`Xxx`*`Event` methods
- A spelling-correcting textfield
- An interactive whiteboard

Certain user actions are classified as *events.* These include such actions as clicking a mouse button, typing a key, or moving a window. The interesting thing about event handling in Java is that you never actually check to see if events have occurred. Instead, you simply tell the system "If an event of the following type occurs in this window, tell such and such an object about it." Then, the system watches for the relevant events, notifying your listener object when it does. Handling events involves three basic steps:

1. **Determine what type of listener is of interest.** There are 11 standard AWT listener types, listed in Table 11.1 of Section 11.5 (The Standard Event Listeners). Browse through these options and choose the one that corresponds to the action that you want to monitor. For example, `KeyListener` corresponds to keystroke actions on the keyboard, `MouseListener` corresponds to mouse button actions, `FocusListener` corresponds to a GUI component getting or losing the keyboard focus, and so forth.

2. **Define a class of that type.** One way to accomplish this task is to directly implement the listener interface (`MouseListener`, `KeyListener`, `FocusListener`, etc.). You need to provide behavior for the methods that correspond to the specific subcategory of action that interests you. For example, if you are interested in handling mouse clicks, you need to use a `MouseListener`, but you still have to decide if you care about when the button is first pressed, when it is released, or both. Each action corresponds to a different method in the `MouseListener` interface, as described in Section 11.5. As you will see shortly, if the interface includes more than one method, then you are also provided with an adapter class that provides a no-op implementation of each of the methods. This approach

lets you override the method(s) of interest without bothering to implement the other methods. Adapters are described in detail in Sections11.1, 11.3, and 11.4.

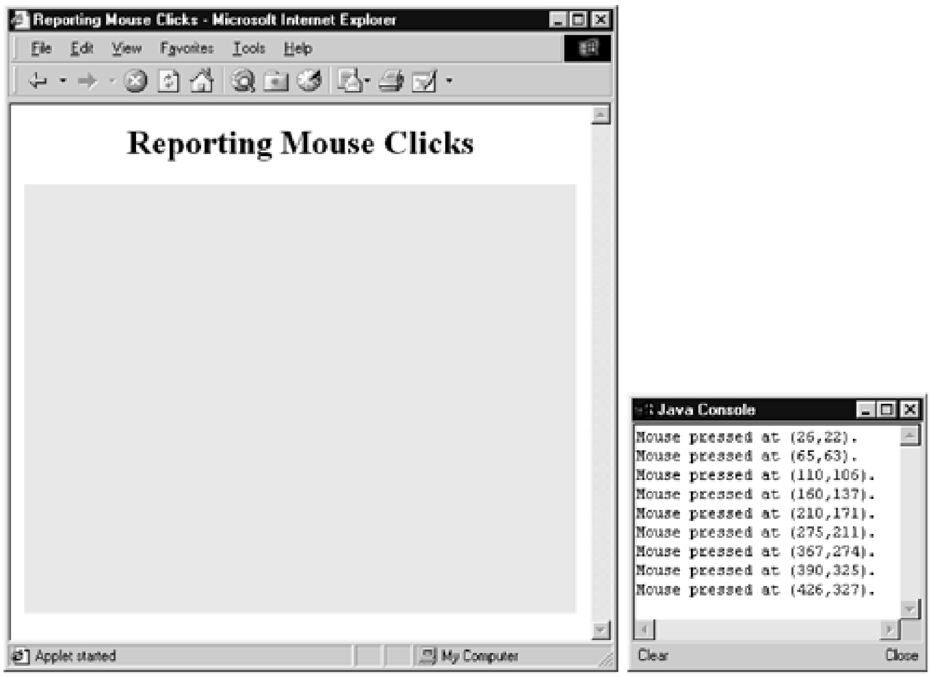
3. **Register an object of your listener class with the window.** Each listener of type `*Something*Listener` has a corresponding method called `add*Something*Listener` that is used to register it. For example, if you want mouse button events in window `w` to be handled by an instance of class `MyMouseListener`, you would use `w.addMouseListener(new MyMouseListener());`.

Within this general framework, there are four major variations in how you define the event handler class. You can define completely separate objects to be event listeners, designate an existing object that implements the listener interface, or use inner classes in two different ways. Sections 11.1 through 11.4 explain each of the four approaches.

11.1 Handling Events with a Separate Listener

Suppose you want to create an applet that, when the user clicks in it, prints the click location in the Java console. (Recall that you can bring up the Java console by means of the Communicator or Communicator/Tools menu in Netscape and by means of the View menu in Internet Explorer.) To accomplish this task, you would first look at the available listener types in Section 11.5 and determine that `MouseListener` is the listener type that corresponds to mouse button actions and that `MouseAdapter` is an implementation with empty versions of the `MouseListener` methods. Section 11.5 would also tell you that `mousePressed` is the method that is triggered when the button is first pressed, that the method takes an argument of type `MouseEvent`, and that the `MouseEvent` class has methods `getX` and `getY` that return the click location relative to the top-left corner of the window. So, you could define a listener like the one shown in Listing 11.1 and attach it to an applet as shown in Listing 11.2. Figure 11-1 shows the results. The tiny HTML file associated with the applet is available in the book's source code archive at http://www.corewebprogramming.com/.

Figure 11-1. Results of the `ClickReporter` applet after a user opens the Java console and clicks several times while moving the mouse from the top-left corner toward the bottom right corner of the applet.


Reporting Mouse Clicks - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Reporting Mouse Clicks
Applet started
My Computer
Java Console
Clear
Close

```
Mouse pressed at (26,22).
Mouse pressed at (65,63).
Mouse pressed at (110,106).
Mouse pressed at (160,137).
Mouse pressed at (210,171).
Mouse pressed at (275,211).
Mouse pressed at (367,274).
Mouse pressed at (390,325).
Mouse pressed at (426,327).
```

Listing 11.1 `ClickListener.java`

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

/** The listener used by ClickReporter. */

public class ClickListener extends MouseAdapter {
  public void mousePressed(MouseEvent event) {
    System.out.println("Mouse pressed at (" +
                       event.getX() + "," +
                       event.getY() + ").");
  }
}
```

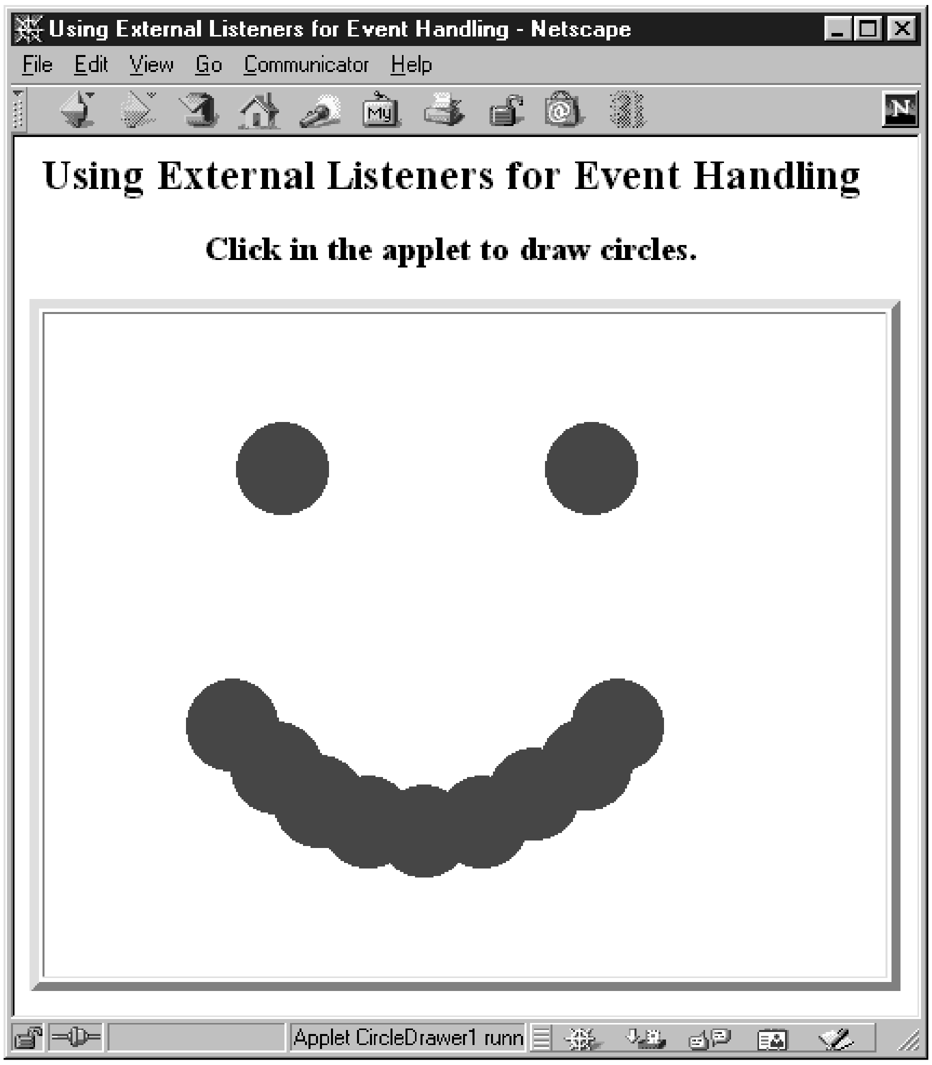
Listing 11.2 `ClickReporter.java`

```
import java.applet.Applet;
import java.awt.*;

/** Prints a message saying where the user clicks.
 *  Uses an external listener.
 */

public class ClickReporter extends Applet {
  public void init() {
    setBackground(Color.yellow);
    addMouseListener(new ClickListener());
  }
}
```

Drawing Circles

Now, instead of simply printing out the location of each click, suppose that you want to draw a circle wherever the user clicks the mouse. Obtaining a `Graphics` object in an applet when you are not in `paint` is easy: you just call `getGraphics`. The problem is that the event handler is not part of the `Applet` class and thus has no `getGraphics` method. The solution is to use the `getSource` method (defined for all event types) to obtain a reference to the window from which the event originated. This reference can be cast to type `Applet`, and then the `getGraphics` method can be called on it. Listings 11.3 and 11.4 give an example, with results shown in Figure 11-2.

Figure 11-2. An external listener can call public methods in the window it is associated with by using the reference obtained by `getSource`.


Using External Listeners for Event Handling - Netscape
File Edit View Go Communicator Help
Using External Listeners for Event Handling
Click in the applet to draw circles.
Applet CircleDrawer1 runn

Core Approach

Use `event.getSource()` to obtain a reference to the window where `event` originated.

Listing 11.3 `CircleListener.java`

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

/** The listener used by CircleDrawer1. Note call
 *  to getSource to obtain reference to the applet.
 */

public class CircleListener extends MouseAdapter {
  private int radius = 25;

  public void mousePressed(MouseEvent event) {
    Applet app = (Applet)event.getSource();
    Graphics g = app.getGraphics();
    g.fillOval(event.getX()-radius,
               event.getY()-radius,
               2*radius,
               2*radius);
  }
}
```

Listing 11.4 `CircleDrawer.java`

```
import java.applet.Applet;
import java.awt.*;

/** Draw circles centered where the user clicks.
 *  Uses an external listener.
 */

public class CircleDrawer1 extends Applet {
  public void init() {
    setForeground(Color.blue);
    addMouseListener(new CircleListener());
  }
}
```

11.2 Handling Events by Implementing a Listener Interface

In the previous example, the listener object was completely separate from the applet that used it. Although in some cases such a separation promotes object reuse, in this case the separation made it more difficult for the event listener to call the applet's methods. First of all, the event handler had to use the somewhat awkward approach of calling `getSource` and performing a typecast, just to obtain a reference to the applet. Second, this reference only enabled the listener to call public methods, not private or protected ones.

An alternative to this approach is for the applet to register itself as the mouse event listener. In order for this approach to comply with Java's strict typing rules, however, the applet has to declare that it implements the `MouseListener` interface. Implementing an interface can be viewed as a promise to the compiler that you will have certain methods. In this case, implementing the `MouseListener` interface is a promise that you will have the `mouseEntered`, `mouseExited`, `mousePressed`, `mouseReleased`, and `mouseClicked` methods, as described in Section 11.5 (The Standard Event Listeners).

Listing 11.5 shows an applet that follows this approach to obtain the same effect as in Figure 11-2. Notice that the `mousePressed` method is simplified, since it can call `getGraphics` directly because the `mousePressed` method is already part of the `Applet` class. However, the applet has to supply definitions for all five methods of `MouseListener` interface, even though it only cares about one of them.

Listing 11.5 `CircleDrawer2.java`

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

/** Draw circles centered where the user clicks.
```

```
 *  Attaches itself as a listener.
 */

public class CircleDrawer2 extends Applet
                           implements MouseListener {
  private int radius = 25;

  public void init() {
    setForeground(Color.blue);
    addMouseListener(this);
  }

  // Remaining methods are from the MouseListener interface.

  public void mouseEntered(MouseEvent event) {}
  public void mouseExited(MouseEvent event) {}
  public void mouseReleased(MouseEvent event) {}
  public void mouseClicked(MouseEvent event) {}

  public void mousePressed(MouseEvent event) {
    Graphics g = getGraphics();
    g.fillOval(event.getX()-radius,
               event.getY()-radius,
               2*radius,
               2*radius);
  }
}
```

11.3 Handling Events with Named Inner Classes

In the first circle-drawing applet, the separate event listener had to perform some minor contortions to call methods in the `Applet` class. That's because of the strict object-oriented nature of the Java programming language. In the second circle-drawing applet, the applet handled its own events, so needed nothing extra to call methods in the `Applet` class. However, to pass typing rules, that applet had to declare that it implemented the `MouseListener` interface, resulting in the necessity of implementing four methods in which the applet had no interest. *Inner classes* (or *nested classes*) provide the best of both worlds. Inner classes are simply classes whose definitions appear inside another class definition. They have full access to the code of the enclosing object, including private methods and fields. Here is an example:

```
public class Outer {
  private class Inner { ... }

  private Inner test = new Inner();
  ...
}
```

`Inner` can implement an interface or extend an arbitrary superclass, just like a regular class can. If you use the `this` reference within `Inner`, it refers to the inner class, but a reference to the outer class can be obtained by `Outer.this`. When compiled, each inner class generates a separate class file, even if the class is anonymous (see Section 11.4). In many cases, the file for the inner class will be a variation of the name of the file for the outer class (e.g., `Outer$Inner.class`). You usually don't care about the class file name, but if you move the class files to a location different from that of the source files, it is easy to overlook the files for the inner classes.

Core Warning

If you move the `.class` *file of a class containing inner classes, don't forget to also move the* `.class` *files for the inner classes.*

Listing 11.6 shows a variation of the `CircleDrawer` class that uses named inner classes. The inner class calls `getGraphics` directly without having to first call `event.getSource` to obtain a reference to the applet. The inner class extends `MouseAdapter` instead of implementing `MouseListener`, obviating the need for the empty implementations of `mouseEntered`, `mouseExited`, `mouseReleased`, and `mouseClicked` that appeared in the previous example (Section 11.2).

Listing 11.6 `CircleDrawer3.java`

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

/** Draw circles centered where the user clicks.
 *  Uses named inner classes.
 */

public class CircleDrawer3 extends Applet {
  public void init() {
    setForeground(Color.blue);
    addMouseListener(new CircleListener());
  }

  private class CircleListener extends MouseAdapter {
    private int radius = 25;

    public void mousePressed(MouseEvent event) {
      Graphics g = getGraphics();
      g.fillOval(event.getX()-radius,
                 event.getY()-radius,
                 2*radius,
                 2*radius);
    }
  }
}
```

11.4 Handling Events with Anonymous Inner Classes

In the previous example we defined a class within another class. Although the inner class definition appeared in an unusual location (inside another class), it still followed the same basic syntax rules as normal classes. In contrast, *anonymous* inner classes let you define and create a class within an expression. You never even name the class being defined. To define anonymous classes, you make what looks like a call to the constructor of the parent class, but you include the definition of the subclass right after the parentheses, as in the following two examples:

```
Color someColor = pickColor();
add(new Panel() {
      public Color origColor = someColor;
      public void init() {
        setBackground(origColor);
      }
    });

KeyAdapter myAdapter =
  new KeyAdapter() {
    public void keyPressed(KeyEvent event) { ... }
  };
addKeyListener(myAdapter);
```

Listing 11.7 uses anonymous inner classes to implement the circle-drawing applet. Although this version is slightly shorter than the one with named inner classes, it is not clear that it is any simpler. Throughout the book, we will mostly use named inner classes when performing event handling that calls for inner classes.

Listing 11.7 `CircleDrawer4.java`

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

/** Draw circles centered where the user clicks.
 *  Uses anonymous inner classes.
 */

public class CircleDrawer4 extends Applet {
  public void init() {
    setForeground(Color.blue);
    addMouseListener
      (new MouseAdapter() {
         private int radius = 25;

         public void mousePressed(MouseEvent event) {
           Graphics g = getGraphics();
           g.fillOval(event.getX()-radius,
                      event.getY()-radius,
                      2*radius,
                      2*radius);
         }
      });
  }
}
```

11.5 The Standard Event Listeners

Table 11.1 summarizes the 11 AWT event listeners. Each is described in more detail later in this section. The method to register a listener has the same name as the listener, prefixed by "add". For example, use `addMouseListener` to attach a `MouseListener`, `addComponentListener` to attach a `ComponentListener`, etc. For each `add`*`Xxx`*`Listener` there is a corresponding `remove`*`Xxx`*`Listener` method. In addition to the 11 AWT listener types, Swing defines some of its own. Many of these are discussed in the chapters on Swing. Note that all listener classes that contain more than one method have a corresponding adapter class that contains empty implementations of all the methods. Since you always implement at least one of the event handling methods (what good is an event "handler" that doesn't do anything?), there is no benefit to having an adapter for a class that contains only a single method.

Table 11.1. Event listeners

Listener	***Adapter Class (If Any)***	***Registration Method***
`ActionListener`		`addActionListener`
`AdjustmentListener`		`addAdjustmentListener`
`ComponentListener`	`ComponentAdapter`	`addComponentListener`
`ContainerListener`	`ContainerAdapter`	`addContainerListener`
`FocusListener`	`FocusAdapter`	`addFocusListener`
`ItemListener`		`addItemListener`
`KeyListener`	`KeyAdapter`	`addKeyListener`
`MouseListener`	`MouseAdapter`	`addMouseListener`
`MouseMotionListener`	`MouseMotionAdapter`	`addMouseMotionListener`
`TextListener`		`addTextListener`
`WindowListener`	`WindowAdapter`	`addWindowListener`

The listeners define methods that you can override to handle certain types of situations. Each of those methods takes a single argument that is a subclass of `AWTEvent`. `AWTEvent` contains four important methods: `consume` (delete the event), `isConsumed` (a `boolean` designating whether the event has already been consumed by another listener on the same source), `getID` (an `int` representing the event type), and `getSource` (the `Object` that the event came from).

ActionListener This interface defines a single method:

```
public void actionPerformed(ActionEvent event)
```

Because there is only a single method, no adapter class is provided. This listener applies to buttons, list items, menu items, and textfields only, and `actionPerformed` is invoked when the user clicks on the button, presses ENTER in the textfield, and so forth. In addition to the standard methods of `AWTEvent`, `ActionEvent` has two additional methods: `getActionCommand` (return the `command` field of the source as a `String`) and `getModifiers` (return an `int` with SHIFT, CONTROL and other modifier flags set). The `getActionCommand` method returns the "command string" of the source—for buttons, this string defaults to the button label but can be defined separately to simplify processing of buttons whose labels change depending on the language in use. However, in most cases the most important thing is the `getSource` method, which specifies the particular component that generated the event.

AdjustmentListener This interface defines a single method:

```
public void adjustmentValueChanged(AdjustmentEvent event)
```

Because there is only a single method, no adapter class is provided. This listener applies only to scrollbars, and `adjustmentValueChanged` is invoked when the scrollbar is moved. `AdjustmentEvent` has several new methods in addition to the standard `AWTEvent` methods. In particular, `getAdjustable` returns the source scrollbar; `getAdjustmentType` returns one of the static constants `UNIT_DECREMENT`, `UNIT_INCREMENT`, `BLOCK_DECREMENT`, `BLOCK_INCREMENT`, or `TRACK`; and `getValue` returns the current setting.

ComponentListener This interface defines four methods:

```
public void componentResized(ComponentEvent event)
public void componentMoved(ComponentEvent event)
public void componentShown(ComponentEvent event)
public void componentHidden(ComponentEvent event)
```

The Java API provides a class called `ComponentAdapter` that has empty implementations of each of these four methods. Implementing this class allows you to override one method without bothering to implement the others. These methods are called when a component is resized, moved, made visible, or hidden, respectively. The `getComponent` method of `ComponentEvent` returns the originating `Component`. Using it is easier than calling `getSource` and casting the result to a `Component`.

ContainerListener This interface defines two methods:

```
public void componentAdded(ContainerEvent event)
public void componentRemoved(ContainerEvent event)
```

The `ContainerAdapter` class provides empty implementations of both of these methods. A `ContainerListener` is invoked when components are added to or removed from a `Container`. The `ContainerEvent` class defines `getContainer` and `getChild` for accessing the window and its associated component.

FocusListener This interface defines two methods:

```
public void focusGained(FocusEvent event)
public void focusLost(FocusEvent event)
```

`FocusAdapter` provides empty versions of these methods. The `FocusEvent` class has a boolean `isTemporary` method that determines whether the focus change was temporary or permanent.

ItemListener This interface defines a single method:

```
public void itemStateChanged(ItemEvent event)
```

No adapter is provided. This listener applies to `Checkbox`, `CheckboxMenuItem`, `Choice`, and `List` and is invoked when an item is changed. The `ItemEvent` class defines three methods: `getItemSelectable` (the source object), `getItem` (the item selected), and `getStateChange` (an `int` that is either

ItemEvent.SELECTED or ItemEvent.DESELECTED).

KeyListener This interface defines three methods:

```
public void keyPressed(KeyEvent event)
public void keyReleased(KeyEvent event)
public void keyTyped(KeyEvent event)
```

The KeyAdapter class provides empty versions of these methods so that you can override one or more without implementing all three. This listener is invoked when a key is typed while the component has the input focus. Ordinary windows such as Panel or Canvas have to explicitly request the focus (by requestFocus) to get key events. The keyPressed and keyReleased methods are used to catch lower-level actions, where SHIFT, CONTROL, and so forth, are sent separately from the key they modify. Use consume in keyPressed if you want to prevent a component from seeing the keystroke. This lets you restrict textfields to certain formats, for instance. If you are only interested in printable characters, override keyTyped instead.

The KeyEvent class defines a variety of methods and constants. Two methods of particular importance are getKeyChar and setKeyChar. The first returns the character typed, and the second can be used to replace the character with a different one. This behavior lets you do things like map tabs and newlines to spaces, convert lowercase to uppercase, and so forth. There are also getModifiers and setModifiers methods that let you retrieve and/or replace the modifier keys, and a boolean isActionKey method that differentiates function and arrow keys from normal keys. You can also use one of four related methods inherited from InputEvent: isAltDown, isControlDown, isMetaDown, and isShiftDown. Rather than acting on characters with getKeyChar, you can retrieve an integer with getKeyCode, then pass that to getKeyText to find the associated string. This is useful in internationalized code. There is a corresponding setKeyCode as well. Finally, you can obtain the Component receiving the event through getComponent, and the time the key was pressed through getWhen.

MouseListener This interface defines five methods:

```
public void mouseEntered(MouseEvent event)
public void mouseExited(MouseEvent event)
public void mousePressed(MouseEvent event)
public void mouseReleased(MouseEvent event)
public void mouseClicked(MouseEvent event)
```

The mouseEntered and mouseExited methods correspond to the event of the mouse entering or exiting the window or component that is using the listener. Note that mouseExited will be called when the mouse enters a component that is on top of the current one, not just when mouse leaves the outside boundaries of the window. For example, suppose you put a button in an applet and move the mouse from outside the applet onto the button and then back outside the applet. The applet will first receive a mouseEntered event (when the mouse pointer enters the boundaries of the applet), then a mouseExited event (when the mouse moves over top of the button), then a second mouseEntered event (when the mouse leaves the button and is back over the main part of the applet), and then a second mouseExited event (when the mouse leaves the boundaries of the applet). Note that the *x* and *y* locations reported by mouseExited can be outside the applet boundaries; note, too, that Netscape often gives nonsensical values for the *x* and *y* positions of `mouseExited`.

The `mousePressed` method corresponds to the event of the mouse button first being pressed, `mouseReleased` corresponds to the mouse button release, and `mouseClicked` corresponds to mouse button release after it was pressed in the current location. So, for example, if you press and release the mouse without moving it, `mousePressed`, `mouseReleased`, and `mouseClicked` all get triggered (in that order). If, however, you press the mouse, drag it, and then release it, `mousePressed` and `mouseReleased` still get triggered, but `mouseClicked` doesn't.

You can use the `MouseAdapter` class if you want to override some but not all of these methods. If you consume the event in `mousePressed`, the associated graphical component will not see the mouse click. You can call `getModifiers` to determine which button was clicked. Java runs on systems that use a 1-button mouse (MacOS), a 2-button mouse (Windows), or a 3-button mouse (Unix), so Java doesn't have separate events for selections with the primary mouse button, the secondary button, and so on. However, on a multiple-button system, `event.getModifiers()` will be equal to `MouseEvent.Button2_MASK` for a middle click and equal to `MouseEvent.Button3_MASK` for a right (secondary button) click. Note that the right button corresponds to `Button3_MASK` even on a 2-button mouse. To allow for multiple modifiers, you generally just check that the flag is set. So, rather than checking if

```
(event.getModifiers() == event.ALT_MASK)
```

you would check

```
((event.getModifiers() & event.ALT_MASK) != 0)
```

The `getClickCount` method lets you differentiate single clicks from multiclicks. The determination of what constitutes a double click versus what should be interpreted as two consecutive single clicks is made by the operating system (often based on user settings), not by Java. Note that for a double click, `mousePressed` will be invoked *twice,* first with `event.getClickCount` equal to 1, then with `event.getClickCount` equal to 2. This is a standard procedure so that the system doesn't have to wait for the multiclick time to expire before reporting the first click, but it catches many first-time users off guard.The `getX`, `getY`, and `getPoint` methods determine the location of the click. The `isPopupTrigger` method is used to determine whether the user pressed the platform-specific key that requests a `PopupMenu`. Like `KeyEvent`, `MouseEvent` inherits from `InputEvent`, getting `isAltDown`, `isControlDown`, `isMetaDown`, `isShiftDown`, `getComponent`, and `getWhen` methods.

MouseMotionListener This interface defines two methods:

```
public void mouseMoved(MouseEvent event)
public void mouseDragged(MouseEvent event)
```

`MouseMotionAdapter` provides empty versions of these two methods. A `MouseMotionListener` is invoked when the mouse is moved. The `MouseEvent` class is described in the previous `MouseListener` section.

TextListener This interface defines a single method:

```
public void textValueChanged(TextEvent event)
```

There is no associated adapter. This listener applies only to `TextArea`, `TextField`, and any custom subclasses of `TextComponent`. The `textValueChanged` method is invoked when text changes, regardless of whether this is from user action or from the program (e.g., using `setText` or `append`).

WindowListener This final interface defines seven methods:

```
public void windowOpened(WindowEvent event)
public void windowClosing(WindowEvent event)
public void windowClosed(WindowEvent event)
public void windowIconified(WindowEvent event)
public void windowDeiconified(WindowEvent event)
public void windowActivated(WindowEvent event)
public void windowDeactivated(WindowEvent event)
```

If you are only concerned with one or two of these methods, you can extend a `WindowAdapter` class, which comes with empty versions of all seven methods. The `windowOpened` method is called when a window is first opened; `windowClosing` is called when the user tries to quit the window; `windowClosed` is called when the window actually is closed; `windowIconified` and `windowDeiconified` are called when the window is minimized and restored; and `windowActivated` and `windowDeactivated` are called when the window is brought to the front and either buried, minimized, or otherwise deactivated. The `getWindow` method of `WindowEvent` returns the `Window` being acted upon.

11.6 Behind the Scenes: Low-Level Event Processing

When a component receives an event, that event gets passed to a method called `process`*`Xxx`*`Event`, where *`Xxx`* is `Mouse` (i.e., the method is called `processMouseEvent`), `Key`, `Focus`, `Action`, and so forth. For a summary, see Table 11.2 at the end of this section. The `process`*`Xxx`*`Event` method passes the event on to each of the attached listeners. In most cases, you just want to add listeners and don't care about the behind-the-scenes method. Occasionally, however, you may want to use these methods directly. For example, if you want to perform the same action for each of the possible mouse events, it is shorter to use `processMouseEvent` than to override all five methods of the `MouseListener` interface. If you override these methods, however, you almost always want to call `super.process`*`Xxx`*`Event` from within the body of `process`*`Xxx`*`Event` because `process`*`Xxx`*`Event` calls any attached listeners. If you forget to make the call to `super.process`*`Xxx`*`Event` and later come back and add some listeners to your component, the listeners will be ignored. You usually call the superclass version at the end of your method, but if you want the listeners to be activated before your code, you can call `super.process`*`Xxx`*`Event` first.

Core Approach

If you override `processMouseEvent`, call `super.processMouseEvent` from within the method. Similarly, if you override any other `processWhateverEvent` method, call `super.processWhateverEvent` from inside it.

For efficiency, the system does not send events to the `processXxxEvent` methods unless you ask it to. The way to instruct it to send events is to call `enableEvents`, using a mask in the `AWTEvent` class corresponding to the type of event. For instance, to enable the reporting of mouse button events to `processMouseEvent`, use

```
enableEvents(AWTEvent.MOUSE_EVENT_MASK);
```

Similarly, for key events use

```
enableEvents(AWTEvent.KEY_EVENT_MASK);
```

and for focus events use

```
enableEvents(AWTEvent.FOCUS_EVENT_MASK);
```

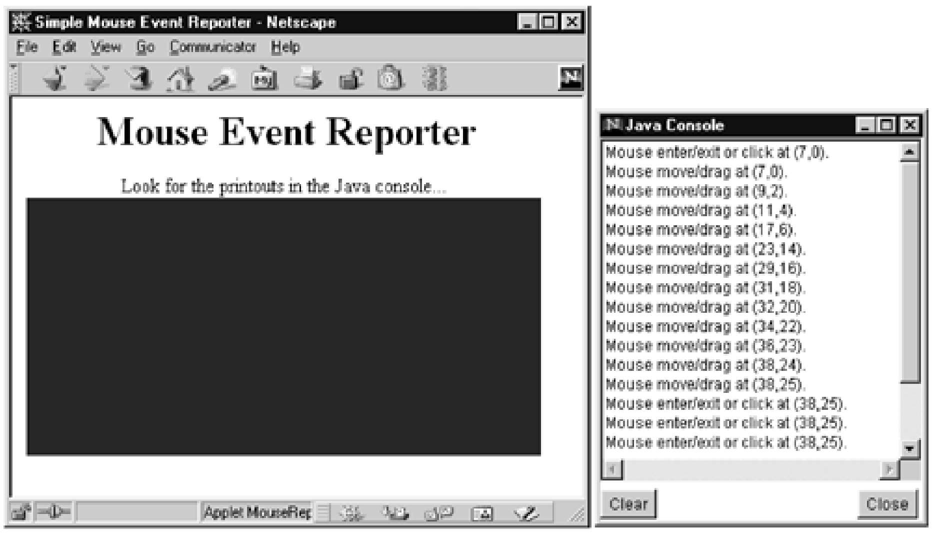
To enable multiple events within a single call, use the bitwise OR of the masks. For example:

```
enableEvents(AWTEvent.MOUSE_EVENT_MASK |
             AWTEvent.KEY_EVENT_MASK |
             AWTEvent.FOCUS_EVENT_MASK);
```

In general, the `processXxxEvent` methods take an argument of type `XxxEvent`. For instance, `processMouseEvent` gets a `MouseEvent`, `processKeyEvent` gets a `KeyEvent`, `processFocusEvent` gets a `FocusEvent`, `processItemEvent` gets an `ItemEvent`, `processActionEvent` gets an `ActionEvent`, and so forth. There is one exception however: `processMouseMotionEvent` does not have its own event type but instead shares `MouseEvent` with `processMouseEvent`. Also, there is a `disableEvents` method corresponding to `enableEvents`; it is used to undo inherited settings. Finally, note that `AWTEvent` is in the `java.awt` package, and `MouseEvent` and the other specific event types are in `java.awt.event`. So, you need to import both packages.

For example, Listing 11.8 shows an applet that reports mouse enter, mouse exit, and mouse press/release/click events from `processMouseEvent` and reports mouse move and drag events from `processMouseMotionEvent`. Figure 11-3 shows a typical result.

Figure 11-3. Result of `MouseReporter` applet after mouse enters the top-left corner of the applet and the button is pressed once.


Simple Mouse Event Reporter - Netscape
File Edit View Go Communicator Help
Mouse Event Reporter
Look for the printouts in the Java console...
Applet MouseRep
Java Console
Clear
Close

```
Mouse enter/exit or click at (7,0).
Mouse move/drag at (7,0).
Mouse move/drag at (9,2).
Mouse move/drag at (11,4).
Mouse move/drag at (17,6).
Mouse move/drag at (23,14).
Mouse move/drag at (29,16).
Mouse move/drag at (31,18).
Mouse move/drag at (32,20).
Mouse move/drag at (34,22).
Mouse move/drag at (36,23).
Mouse move/drag at (38,24).
Mouse move/drag at (38,25).
Mouse enter/exit or click at (38,25).
Mouse enter/exit or click at (38,25).
Mouse enter/exit or click at (38,25).
```

Listing 11.8 **`MouseReporter.java`**

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

/** Prints nondetailed reports of mouse events.
 *  Uses the low-level processXxxEvent methods instead
 *  of the usual event listeners.
 */

public class MouseReporter extends Applet {
  public void init() {
    setBackground(Color.blue); // So you can see applet in page
    enableEvents(AWTEvent.MOUSE_EVENT_MASK |
                 AWTEvent.MOUSE_MOTION_EVENT_MASK);
  }
  public void processMouseEvent(MouseEvent event) {
    System.out.println("Mouse enter/exit or click at (" +
                       event.getX() + "," +
                       event.getY() + ").");
    // In case there are MouseListeners attached:
    super.processMouseEvent(event);
  }

  public void processMouseMotionEvent(MouseEvent event) {
    System.out.println("Mouse move/drag at (" +
                       event.getX() + "," +
                       event.getY() + ").");
    // In case there are MouseMotionListeners attached:
    super.processMouseMotionEvent(event);
  }
}
```

Table 11.2 summarizes the low-level event-processing methods.

Table 11.2. Low-level event-processing methods

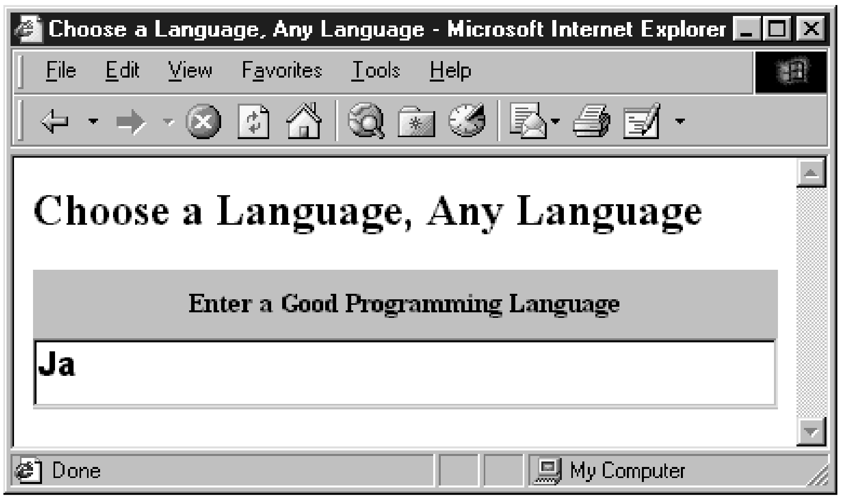
Low-Level Event Method	***Corresponding Event Type***	***Mask for enableEvents (Static var in AWTEvent)***
`processActionEvent`	`ActionEvent`	`ACTION_EVENT_MASK`
`processAdjustmentEvent`	`AdjustmentEvent`	`ADJUSTMENT_EVENT_MASK`
`processComponentEvent`	`ComponentEvent`	`COMPONENT_EVENT_MASK`
`processContainerEvent`	`ContainerEvent`	`CONTAINER_EVENT_MASK`
`processFocusEvent`	`FocusEvent`	`FOCUS_EVENT_MASK`
`processItemEvent`	`ItemEvent`	`ITEM_EVENT_MASK`
`processKeyEvent`	`KeyEvent`	`KEY_EVENT_MASK`
`processMouseEvent`	`MouseEvent`	`MOUSE_EVENT_MASK`
`processMouseMotionEvent`	`MouseEvent`	`MOUSE_MOTION_EVENT_MASK`
`processTextEvent`	`TextEvent`	`TEXT_EVENT_MASK`

processWindowEvent	WindowEvent	WINDOW_EVENT_MASK

11.7 A Spelling-Correcting Textfield

Listing 11.9 shows a spelling-correcting textfield that lets the user enter the name of a good programming language. The textfield monitors three types of events: key events (when a printable character is typed while the textfield has the keyboard focus), focus events (when the textfield obtains the keyboard focus), and action events (when ENTER is pressed while the textfield has the focus). When it detects a printable character (through the `keyTyped` method of the `KeyAdapter` class), it compares the string entered so far to a dictionary of good programming languages and replaces the string entered with the most closely matching substring from its dictionary. When it detects ENTER (through the `actionPerformed` method of the `ActionListener` interface), the text is filled in with the complete name of a good programming language whose name most closely matches the text entered so far. Finally, when the textfield detects that it has just received the keyboard focus (through the `focusGained` method of the `FocusAdapter` class), it repeatedly but briefly flashes a hint. Listing 11.10 presents an applet that uses this textfield, and Figure 11-4 shows a typical result.

Figure 11-4. `JavaTextField` after user entered "C#" (an obvious typo).


Choose a Language, Any Language - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Choose a Language, Any Language
Enter a Good Programming Language
Ja
Done
My Computer

Listing 11.9 `LanguageField.java`

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

/** A spelling-correcting TextField for entering
 *  a language name.
 */

public class LanguageField extends TextField {
  private String[] substrings =
    { "", "J", "Ja", "Jav", "Java" };

  public LanguageField() {
    addKeyListener(new SpellingCorrector());
    addActionListener(new WordCompleter());
    addFocusListener(new SubliminalAdvertiser());
  }

  // Put caret at end of field.

  private void setCaret() {
    setCaretPosition(5);
  }
```

```
// Listener to monitor/correct spelling as user types.
```

```
private class SpellingCorrector extends KeyAdapter {
  public void keyTyped(KeyEvent event) {
    setLanguage();
    setCaret();
  }

  // Enter partial name of good programming language that
  // most closely matches what they've typed so far.

  private void setLanguage() {
    int length = getText().length();
    if (length <= 4) {
      setText(substrings[length]);
    } else {
      setText("Java");
    }
    setCaret();
  }
}
// Listener to replace current partial name with
// most closely matching name of good language.

private class WordCompleter implements ActionListener {

  // When they press RETURN, fill in the right answer.

  public void actionPerformed(ActionEvent event) {
    setText("Java");
    setCaret();
  }
}

// Listener to give the user a hint.

private class SubliminalAdvertiser extends FocusAdapter {
  public void focusGained(FocusEvent event) {
    String text = getText();
    for(int i=0; i<10; i++) {
      setText("Hint: Java");
      setText(text);
    }
    }
  }
}
```

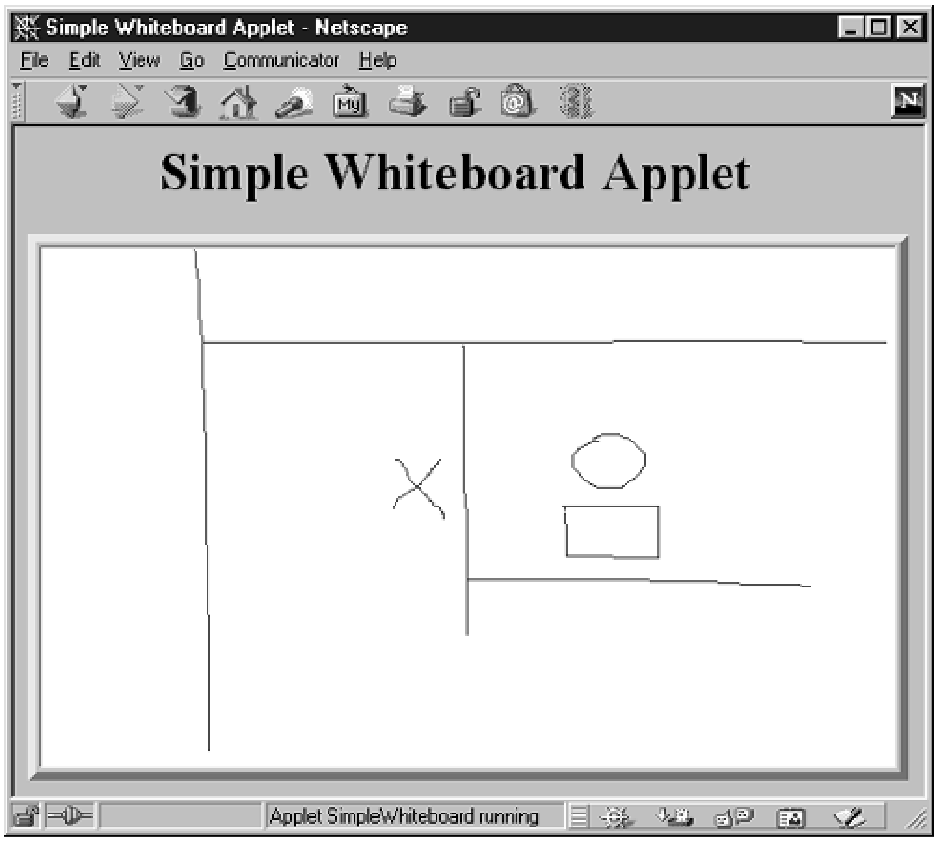
Listing 11.10 `JavaTextField.java`

```
import java.applet.Applet;
import java.awt.*;

/** Lets the user enter the name of <B>any</B>
 *  good programming language. Or does it?
 */

public class JavaTextField extends Applet {
  public void init() {
    setFont(new Font("Serif", Font.BOLD, 14));
    setLayout(new GridLayout(2, 1));
    add(new Label("Enter a Good Programming Language",
                  Label.CENTER));
    LanguageField langField = new LanguageField();
    Font langFont = new Font("SansSerif", Font.BOLD, 18);
    langField.setFont(langFont);
    add(langField);
  }
}
```

11.8 A Whiteboard Class

Listing 11.11 shows an applet that lets the user create freehand drawings. The idea is to record the initial *x* and *y* position when the user presses the mouse and then to draw line segments from the previous position to the current position as the user drags the mouse. For example, Figure 11-5 shows a map to the Kossiakoff Center at the Johns Hopkins University Applied Physics Lab, and indicates where to park once there.

Figure 11-5. A simple whiteboard; drawing is performed when the mouse is dragged.


Simple Whiteboard Applet - Netscape
File Edit View Go Communicator Help
Simple Whiteboard Applet
Applet SimpleWhiteboard running

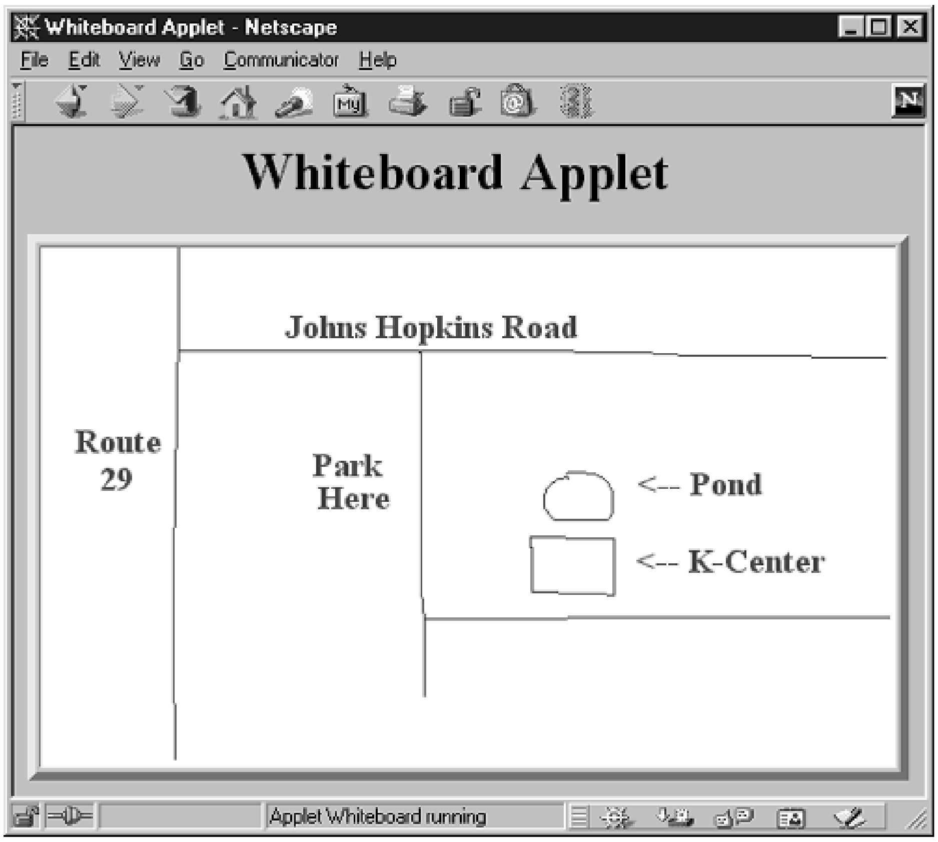
Listing 11.11 `SimpleWhiteboard.java`

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

/** An applet that lets you perform freehand drawing. */

public class SimpleWhiteboard extends Applet {
  protected int lastX=0, lastY=0;

  public void init() {
    setBackground(Color.white);
    setForeground(Color.blue);
    addMouseListener(new PositionRecorder());
    addMouseMotionListener(new LineDrawer());
  }
  protected void record(int x, int y) {
    lastX = x;
    lastY = y;
  }

  // Record position that mouse entered window or
  // where user pressed mouse button.

  private class PositionRecorder extends MouseAdapter {
    public void mouseEntered(MouseEvent event) {
      requestFocus(); // Plan ahead for typing
      record(event.getX(), event.getY());
    }

    public void mousePressed(MouseEvent event) {
      record(event.getX(), event.getY());
    }
  }

  // As user drags mouse, connect subsequent positions
  // with short line segments.

  private class LineDrawer extends MouseMotionAdapter {
    public void mouseDragged(MouseEvent event) {
      int x = event.getX();
      int y = event.getY();
      Graphics g = getGraphics();
      g.drawLine(lastX, lastY, x, y);
      record(x, y);
    }
  }
}
```

A Better Whiteboard

Now, although Figure 11-5 gave an absolutely clear map, it must be admitted that some people have trouble interpreting true art. So, we next add the ability to annotate the drawings with typed text (see Listing 11.12). Whenever a key is typed, the `keyTyped` method converts the key (an `int` in this case) to a `String`, draws the `String` at the current location, then shifts the current location to the right, depending on the width of the `String` in the current font. Figure 11-6 shows the result.

Figure 11-6. A better whiteboard; text is drawn when keys are typed.


Whiteboard Applet - Netscape
File Edit View Go Communicator Help
Whiteboard Applet
Johns Hopkins Road
Route 29
Park Here
<-- Pond
<-- K-Center
Applet Whiteboard running

Listing 11.12 `Whiteboard.java`

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


/** A better whiteboard that lets you enter
 *  text in addition to freehand drawing.
 */
public class Whiteboard extends SimpleWhiteboard {
  protected FontMetrics fm;

  public void init() {
    super.init();
    Font font = new Font("Serif", Font.BOLD, 20);
    setFont(font);
    fm = getFontMetrics(font);
    addKeyListener(new CharDrawer());
  }

  private class CharDrawer extends KeyAdapter {
    // When user types a printable character,
    // draw it and shift position rightwards.

    public void keyTyped(KeyEvent event) {
      String s = String.valueOf(event.getKeyChar());
      getGraphics().drawString(s, lastX, lastY);
      record(lastX + fm.stringWidth(s), lastY);
    }
  }
}
```

11.9 Summary

The whiteboard presented in Section 11.8 is not too bad for such a small amount of code, but it is lacking in several ways. First, there are no options for choosing colors or fonts or for specifying the type of drawing operation (freehand, straight lines, circles, rectangles, and so forth). In Chapter 13 (AWT Components), we cover combo boxes, pull-down menus, and other GUI widgets needed to create such an interface. Second, because drawing is done directly to the window, it is transient. If another window is moved over the browser and then removed, the drawing is lost. In Chapter 16 (Concurrent Programming with Java Threads), we discuss several techniques for overcoming this problem, the most general of which is *double buffering,* a technique for drawing into off-screen images. Finally, for whiteboards to be most useful they should be shareable. We discuss methods for making a server that would permit this in Chapter 17 (Network Programming).

CONTENTS