

CHAPTER

6

- Timers on page 249
- Event Listener Lists on page 258
- Swing Utilities on page 262
- Swing Constants on page 271
- BorderLayout and the Box Class on page 272
 - BorderLayout on page 273
 - The Box Class on page 276
- Progress Monitoring on page 281
 - ProgressMonitor on page 281
 - ProgressMonitorInputStream on page 288
- Undo/Redo on page 292
 - UndoableEditSupport on page 300
 - Compound Edits on page 303
 - UndoManager on page 308
- Parting Shots on page 315

Utilities



Swing includes a number of utilities that are discussed in this chapter. Some of the utilities, such as timers and the `static` methods provided by the `SwingUtilities` class, are used internally by Swing, whereas others, such as progress monitors and progress monitor streams, are not. All of the utilities discussed in this chapter are available for use by developers using Swing.

Timers

Timers are useful in many situations when some action must be performed periodically or at a specific time. For example, animations can use timers for updating the next frame of the animation and for controlling the rate at which animated objects move and change their appearance. See the first volume of *Graphic Java* for a complete animation system that uses timers in exactly the manner described above.¹ In fact, Swing itself uses timers for autoscrolling and tooltips.

1. The timers used in the first volume of *Graphic Java* are not Swing timers.



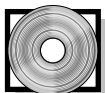
Swing provides a `Timer` class whose instances can have one or more action listeners associated with them. When a timer “rings”—meaning, it fires an action event—each `ActionListener` associated with the timer has its `actionPerformed` method invoked.

Timers can ring once, or they can ring repeatedly at a periodic interval. Each timer can have two delays, specified in milliseconds, associated with it. An *initial delay* specifies the amount of time that elapses before the timer rings for the first time. A *periodic delay* specifies the amount of time that elapses between rings for a repeating timer. By default, all timers repeat.

The application listed in Example 6-1 creates three timers. The `oneSecondTimer` repeatedly rings at one-second intervals and has one `ActionListener`—the application—associated with it. When the `oneSecondTimer` rings, it invokes the application’s `actionPerformed` method, which prints the number of seconds that have elapsed.

The application creates a second timer that rings repeatedly at two-second intervals and has an initial delay of five seconds. The action listener associated with the second timer is an instance of `TimerWithDelayListener`, which prints a message that the timer with delay is ringing.

A third timer does not repeat and has an initial delay of 10 seconds. The `ActionListener` associated with the third timer is an instance of `OneTimeListener`, which prints a notification that the “one time” timer is ringing. Note that the timer examples that follow do not display a window and must be killed with a CTRL-C.



Example 6-1 Using Swing Timers

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

public class Test implements ActionListener {
    private int seconds=1;

    public Test() {
        Timer oneSecondTimer = new Timer(1000, this);
        Timer timerWithInitialDelay = new Timer(2000,
            new TimerWithDelayListener());
        Timer oneTimeTimer = new Timer(10000,
            new OneTimeListener());

        timerWithInitialDelay.setInitialDelay(5000);
        oneTimeTimer.setRepeats(false);

        oneSecondTimer.start();
    }
}
```



```
        timerWithInitialDelay.start();
        oneTimeTimer.start();
    }
    public void actionPerformed(ActionEvent e) {
        if(seconds == 0)
            System.out.println("Time:  " + seconds + " second");
        else
            System.out.println("Time:  " + seconds + " seconds");

        seconds++;
    }
    public static void main(String args[]) {
        new Test();
        while(true);
    }
}
class TimerWithDelayListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Timer with Delay Ringing");
    }
}
class OneTimeListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("One Time Timer Ringing");
    }
}
}
```

All three timers are instantiated with the only constructor that the `Timer` class provides. The constructor is passed two arguments; the first specifies the delay associated with the timer, and the second specifies a listener that is to be notified when the timer rings.

After the three timers are constructed, the timer with an initial delay has its initial delay specified by the `Timer.setInitialDelay` method. Because timers repeat by default, the “one time” timer invokes `Timer.setRepeats(false)` to specify that the timer does not repeat. Subsequently, all three timers are started with the `Timer.start` method.

The output of the application listed in Example 6-1 is as follows:

```
Time:  1 seconds
Time:  2 seconds
Time:  3 seconds
Time:  4 seconds
Timer with Delay Ringing
Time:  5 seconds
Time:  6 seconds
Timer with Delay Ringing
```



```
Time: 7 seconds
Time: 8 seconds
Timer with Delay Ringing
Time: 9 seconds
One Time Timer Ringing
Time: 10 seconds
Timer with Delay Ringing
Time: 11 seconds
Time: 12 seconds
...
```

The Timer Class

A summary of the `Timer` class methods is provided in Class Summary 6-1.

Class Summary 6-1 Timer

Constructors

```
public Timer(int delay, ActionListener)
```

The `Timer` class provides only one constructor. Timers must have a delay and at least one action listener, both of which must be supplied at construction time.

The integer value passed to the constructor specifies the delay associated with the timer. If the timer repeats, the delay specified in the call to the constructor represents both the initial delay and the delay between subsequent rings of the timer. If the timer does not repeat, the delay specified in the call to the constructor represents only the initial delay. If the initial delay is specified for a nonrepeating timer after construction with the `setInitialDelay` method or if the periodic display is specified for a repeating timer with the `setDelay` method, the delay passed to the constructor is superseded.

For example, the application listed in Example 6-2 constructs a timer with a delay of one second. After construction, the initial delay of the timer is set to 10 seconds, and `setRepeats(false)` is called so that the timer does not repeat. Since the



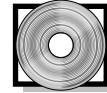
initial delay is specified after construction and the timer does not repeat, the delay of one second specified when the timer is constructed is irrelevant—the timer will ring once ten seconds after it is started and will not ring again.

Example 6-2 Overriding the Delay Specified at the Time of Construction

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

public class Test implements ActionListener {
    public Test() {
        Timer oneSecondTimer = new Timer(1000, this);

        oneSecondTimer.setInitialDelay(10000);
        oneSecondTimer.setRepeats(false);
        oneSecondTimer.start();
    }
    public void actionPerformed(ActionEvent e) {
        System.out.println("ring ...");
    }
    public static void main(String args[]) {
        new Test();
        while(true);
    }
}
```



Methods

Log Timers

```
public static boolean getLogTimers()
public static void setLogTimers(boolean)
```

Timers can be logged, meaning that a message is printed to the `System.out` stream whenever a timer rings. Logging can only be set for all timers at once by invoking the static `setLogTimers` method; there is no option to turn logging on for an individual timer.

By default, log messages print the string "Timer ringing: " followed by the return value of the timer's `toString` method. For example, the application listed in Example 6-3 invokes `Timer.setLogTimers()` to turn logging on.



Example 6-3 Timer Logging

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

public class Test implements ActionListener {
    public Test() {
        Timer.setLogTimers(true);

        Timer oneSecondTimer = new MyTimer(1000, this);
        oneSecondTimer.start();
    }
    public void actionPerformed(ActionEvent e) {
        System.out.println("ring ...");
    }
    public static void main(String args[]) {
        new Test();
        while(true);
    }
}

class MyTimer extends Timer {
    public MyTimer(int delay, ActionListener listener) {
        super(delay, listener);
    }
    public String toString() {
        return "MyTimer";
    }
}
```

The `Timer` class does not implement a `toString` method, so by default, `Timer` logging prints the value returned by `Object.toString`, which represents the location of the timer in memory. If timer logging is used, it makes sense to extend the timer class and implement a meaningful `toString` method, as is the case for the application listed in Example 6-3.

The output of the application listed in Example 6-3 is as follows:

```
Timer ringing: MyTimer
ring ...
Timer ringing: MyTimer
ring ...
...
```

Start / Stop / Restart

```
public void start()
public void stop()
public void restart()
```



Timers can be started, stopped, and restarted. If a timer is stopped and then restarted, the timer will go through its entire initial delay before ringing for the first time after being restarted.

```
public void addActionListener(ActionListener)
public void removeActionListener(ActionListener)
protected void fireActionPerformed(ActionEvent)
```

More than one action listener can be associated with a single timer, and action listeners can be removed from the timer's list of listeners after the timer is constructed.

The `fireActionPerformed` method invokes `actionPerformed` for each listener registered with the timer. For example, the application listed in Example 6-4 associates three action listeners with a single timer.

Example 6-4 Multiple Action Listeners Associated with a Single Timer

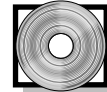
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test implements ActionListener {
    private int seconds=1;

    public Test() {
        Timer oneSecondTimer = new Timer(1000, this);

        oneSecondTimer.addActionListener(new SecondListener());
        oneSecondTimer.addActionListener(new ThirdListener());
        oneSecondTimer.start();
    }
    public void actionPerformed(ActionEvent e) {
        if(seconds == 0)
            System.out.println("Time: " + seconds + " second");
        else
            System.out.println("Time: " + seconds + " seconds");
        seconds++;
    }
    public static void main(String args[]) {
        new Test();
        while(true);
    }
}

class SecondListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
```





```
        System.out.println("Second Listener");
    }
}
class ThirdListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Third Listener");
    }
}
```

The output of the application listed in Example 6-4 is as follows:

```
Third Listener
Second Listener
Time: 1 second
Third Listener
Second Listener
Time: 2 seconds
Third Listener
Second Listener
Time: 3 seconds
```

Notice that the `fireActionPerformed` method is protected and therefore can be overridden by extensions that wish to alter the order in which listeners are notified.

The source of the `ActionEvent` passed to the action listener's `actionPerformed` method is the timer that is ringing.

Delays

```
public int getDelay()
public int getInitialDelay()

public void setDelay(int milliseconds)
public void setInitialDelay(int milliseconds)
```

Both the initial delay and the repeat delay associated with a timer are settable after construction. Additionally, the `Timer` class provides methods for obtaining the delay associated with a timer.



Coalesce / Repeating / Running

```
public void setCoalesce(boolean)
public void setRepeats(boolean)

public boolean isCoalesce()
public boolean isRepeats()
public boolean isRunning()
```

The `Timer` class provides methods for finding out whether a timer repeats or is currently running. A method is also provided to set whether the timer repeats.

By default, timers coalesce events. If an applet or application is busy and cannot keep up with events generated by a timer, the timer will collapse pending events into a single notification. For example, the application listed in Example 6-5 explicitly turns coalescing off and therefore will receive 10 notifications, one right after the other, after it is done sleeping. If the call to `setCoalesce(false)` is commented out, the application receives only one event after it awakes.

Example 6-5 Coalescing Timer Events

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

public class Test implements ActionListener {
    private boolean firstRing = true;
    private int ring = 1;

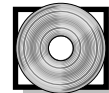
    public Test() {
        Timer.setLogTimers(true);

        Timer oneSecondTimer = new Timer(1000, this);

        // comment out the following line for coalescing
        oneSecondTimer.setCoalesce(false);

        System.out.println("Timer is coalescing: " +
            oneSecondTimer.isCoalesce());

        oneSecondTimer.start();
    }
    public void actionPerformed(ActionEvent e) {
        System.out.println("ring #" + ring++);
    }
}
```





```
        if(firstRing) {
            // simulate a time consuming operation by sleeping
            // for 10 seconds ...
            try {
                Thread.currentThread().sleep(10000);
            }
            catch(InterruptedException ex) {
                ex.printStackTrace();
            }
            firstRing = false;
        }
    }
    public static void main(String args[]) {
        new Test();
        while(true);
    }
}
```

Event Listener Lists

The `swing.event` package includes an `EventListenerList` class that can be used to maintain a list of event listeners of different types. In effect, Swing's `EventListenerList` replaces the AWT's `AWTEventMulticaster` class; instances of `EventListenerList` are used by all Swing components that fire events. Instances of `EventListenerList` can be used for event listeners of any type, whereas the `AWTEventMulticaster` is limited to the set of event listener types supported by the AWT.

Developers are encouraged to use the `EventListenerList` class for maintaining lists of event listeners for custom components (or any other class) that fire events. Using the `EventListenerList` class entails the following three steps: (where XXX represents the type of event)

1. Implement an `addXXXListener` method that adds listeners to the list.
2. Implement a `removeXXXListener` method that removes listeners from the list.
3. Implement a `fireXXXPerformed` method that iterates over the list and fires events to listeners.

Swing's timers use an instance of `EventListenerList` to maintain a list of `ActionListeners`. The code listed below is from the `swing.Timer` class:



```
// From swing.Timer ...

public void addActionListener(ActionListener listener) {
    listenerList.add(ActionListener.class, listener);
}
public void removeActionListener(ActionListener listener) {
    listenerList.remove(ActionListener.class, listener);
}
protected void fireActionPerformed(ActionEvent e) {
    // Guaranteed to return a non-null array
    Object[] listeners = listenerList.getListenerList();

    // Process the listeners last to first, notifying
    // those that are interested in this event
    for (int i = listeners.length-2; i>=0; i-=2) {
        if (listeners[i]==ActionListener.class) {
            ((ActionListener)listeners[i+1]).actionPerformed(e);
        }
    }
}
```

The `addActionListener` and `removeActionListener` methods for the `Timer` class simply delegate to the `EventListenerList` `add` and `remove` methods, which is typical. Both the `add` and `remove` methods of `EventListenerList` class take two arguments: the first argument is the class of the listener, and the second argument is a reference to the listener to be added or removed.

The `EventListenerList` class maintains an `Object` array that contains an alternating sequence of `[Class, EventListener]`, enabling it to maintain listeners of different types on the same list. That is the reason that the `Timer` `fireActionPerformed` method iterates over the list of listeners by advancing two slots in the array every time through the `for` loop.

Notice that neither `Timer.addActionListener` nor `Timer.removeActionListener` is synchronized. Because the `EventListenerList` class is thread safe, its `add` and `remove` methods are synchronized, and therefore the `Timer` methods do not need to be synchronized.

As is almost always the case, the `fireActionPerformed` is protected. Firing action events is an implementation detail of the `Timer` class; other objects should not be allowed to invoke a timer's `fireActionPerformed` method, and therefore the method is not `public`. The method is protected so that extensions of the `Timer` class can modify the manner in which action events are fired. Also, as with all Swing components that fire events, the list of listeners is traversed from the last listener added to the list to the first.



There's nothing about the `EventListenerList` class that forces the list of listeners to be traversed from back to front. In fact, it is a simple matter to extend the `Timer` class and override the `fireActionPerformed` method so that traversal of the list is from front to back:

```
// An extension of Timer that reverses the traversal of the
// listener list when action events are fired.

class ReverseTimer extends Timer {
    public ReverseTimer(int delay, ActionListener listener) {
        super(delay, listener);
    }
    protected void fireActionPerformed(ActionEvent e) {
        Object[] list = listenerList.getListenerList();

        // Process the listeners first to last ...

        for (int i = 0; i <= list.length-2; i+=2) {
            if (list[i]==ActionListener.class) {
                ((ActionListener)list[i+1]).actionPerformed(e);
            }
        }
    }
}
```

The EventListenerList Class

Class Summary 6-2 lists the public methods implemented by the `EventListenerList` class.

Class Summary 6-2 `swing.EventListenerList`

Extends: `java.lang.Object`
Implements: `java.io.Serializable`



Constructors

```
public EventListenerList()
```

The no-argument constructor is compiler generated and therefore does nothing.

Methods

Add/Remove Listeners

```
public synchronized void add(Class, EventListener)  
public synchronized void remove(Class, EventListener)
```

The `add` and `remove` methods are both passed the class of the listener and the listener to be added or removed. It is up to users of the `EventListenerList` class to ensure that the class passed to the methods matches the class of the listener.

Listener Count

```
public int getListenerCount()  
public int getListenerCount(Class)
```

`getListenerCount()` returns the number of listeners that are currently on the list. `getListenerCount(Class)` returns the number of listeners of type `Class` that are currently on the list.

Listener List / String Representation

```
public Object[] getListenerList()  
public String toString()
```



`getListenerList()` returns the actual `Object` array that holds the classes of the listeners and the listeners themselves. The array returned by `getListenerList()` is guaranteed to be non-null; if no listeners are currently on the list, the length of the array will be 0. For efficiency, the array returned is not a copy and therefore *should not be modified by the caller in any manner whatsoever*.

`EventListenerList.toString()` prints the number of listeners currently on the list, in addition to the type of each listener and its string representation.

Swing Utilities

The `swing` package includes a `SwingUtilities` class that contains more than 30 static utility methods. The methods are grouped into the following functional areas:

- *Computational methods* that compute intersections, unions, and differences of rectangles in addition to the width of a string with specific font metrics
- *Conversion methods* that convert events, points, and rectangles from one component's coordinate system to another
- *Accessibility methods* that return accessibility information for a given component
- *Retrieval methods* that retrieve objects associated with a given component, such as focus owners, ancestors, and the root pane in which a component resides
- *Methods for executing code from a thread* other than the event dispatch thread
- *Boolean methods* that can be used to determine relationships, for example, if one component is an ancestor of another, if the current thread is the event dispatch thread, and which mouse button has been pressed given a particular mouse event.
- *Layout, painting, and UI updating methods* for a given component

The methods provided by the `SwingUtilities` class are heavily used throughout Swing. For example, `SwingUtilities.layoutCompoundLabel` is used by both `JButton` and `JLabel` for laying out the text and icon associated with a button or label, respectively. The methods are also useful for developers using Swing and are therefore given `public` access.



The public methods implemented by the `SwingUtilities` class are listed in Class Summary 6-3.

Class Summary 6-3 SwingUtilities

Extends: `java.lang.Object`

Implements: `SwingConstants`

Constructors

```
public SwingUtilities()
```

The no-argument constructor is compiler generated. Because all of the methods implemented by the `SwingUtilities` class are static, there should never be any reason to instantiate an instance of `SwingUtilities`. In fact, it probably would have been a good idea to implement a private no-argument constructor so that instances of `SwingUtilities` could not be instantiated.

Methods

Computational Methods

```
public static Rectangle[] computeDifference(Rectangle, Rectangle)
public static Rectangle computeIntersection(int x, int y, int w, int h, Rectangle)
public static Rectangle computeUnion(int x, int y, int w, int h, Rectangle)
public static final boolean isRectangleContainingRectangle(Rectangle, Rectangle)
public static int computeStringWidth(FontMetrics, String)
```

The methods listed above compute the union, intersection, and difference between two rectangles, in addition to determining whether one rectangle completely contains another.



The applet shown in Figure 6-1 illustrates the use of the `computeDifference`, `computeIntersection`, and `computeUnion` methods.

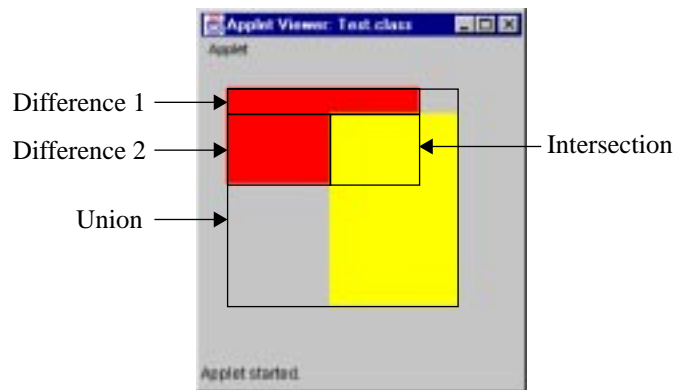


Figure 6-1 Computing Differences, Intersection, and Union of Two Rectangles

The `computeIntersection` and `computeUnion` methods both override the values of the rectangle they are passed so that a new `Rectangle` does not have to be instantiated. When the methods return, the rectangle passed to the method represents the intersection or union of the two rectangular regions.

The `computeDifference` method returns an array of rectangles representing the regions within the first rectangle that do not overlap with the second rectangle.

The applet shown in Figure 6-1 is listed in Example 6-6.

Example 6-6 Computing Differences, Intersection, and Union of Two Rectangles



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

public class Test extends JApplet {
    Rectangle r1 = new Rectangle(20,20,150,75);
    Rectangle r2 = new Rectangle(100,40,100,150);
    Rectangle destination;

    public Test() {
        destination = new Rectangle(r2);

        // print out the intersection of r1 and r2 ...
    }
}
```



```

// computeUnion stores the union of r1 and
// destination in destination

System.out.println("Intersection: " +
    SwingUtilities.computeIntersection(r1.x,r1.y,
        r1.width,r1.height,destination));
System.out.println();

// print out the union of r1 and r2 ...
// computeUnion stores the union of r1 and
// destination in destination

System.out.println("Union: " +
    SwingUtilities.computeUnion(r1.x,r1.y,
        r1.width,r1.height,destination));
System.out.println();

// print out the difference of r1 and r2 ...
// computeDifference does not stuff return values
// in an input argument

Rectangle[] difference =
    SwingUtilities.computeDifference(r1, r2);

System.out.println("Difference:");

for(int i=0; i < difference.length; ++i) {
    System.out.println(difference[i]);
}
}
public void paint(Graphics g) {
    g.setColor(Color.red);
    g.fillRect(r1.x, r1.y, r1.width, r1.height);

    g.setColor(Color.yellow);
    g.fillRect(r2.x, r2.y, r2.width, r2.height);
}
}

```

The output of the applet shown in Figure 6-1 is listed below.

```

Intersection: java.awt.Rectangle[x=100,y=40,width=70,height=55]

Union: java.awt.Rectangle[x=20,y=20,width=150,height=75]

Difference:
java.awt.Rectangle[x=20,y=20,width=150,height=20]
java.awt.Rectangle[x=20,y=40,width=80,height=55]

```



Conversion Methods

```
public static MouseEvent convertMouseEvent(Component, MouseEvent, Component)
public static Point convertPoint(Component, int x, int y, Component)
public static Point convertPoint(Component, Point, Component)
public static Rectangle convertRectangle(Component, Rectangle, Component)
```

```
public static void convertPointFromScreen(Point, Component)
public static void convertPointToScreen(Point, Component)
```

The methods listed above convert either a point, rectangle, or mouse event from one component's coordinate system to another's. The first group of methods listed above are all passed references to two components—a source and a destination. The source represents the component from which the location or event originated, and the destination is the component to which the location or event is translated.

The last two methods provide a convenient way to translate from a component's coordinate systems to screen coordinates, and vice versa.

The applet shown in Figure 6-2 contains three nested panels, all of which paint their backgrounds with a specific color and display mouse coordinates relative to their coordinate systems.

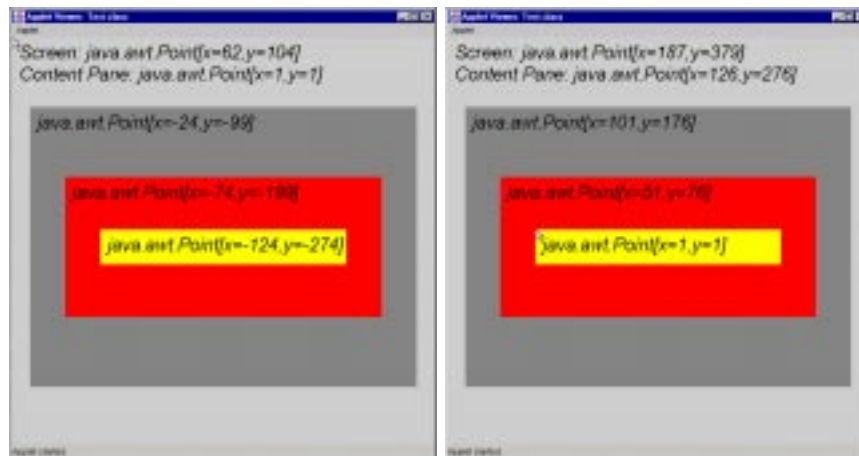


Figure 6-2 Converting Coordinates



A mouse motion listener added to the applet's content pane translates the location of mouse moved events from the content pane's coordinate system to that of the other panels and the screen. The string displayed in each panel is updated, and the applet repaints.

```

...
contentPane.addMouseListener(
    new MouseMotionAdapter() {
    public void mouseMoved(MouseEvent e) {
        Point pt = e.getPoint();

        outer.setString(SwingUtilities.convertPoint(
            contentPane, pt, outer).toString());

        inner.setString(SwingUtilities.convertPoint(
            contentPane, pt, inner).toString());

        innermost.setString(SwingUtilities.convertPoint(
            contentPane, pt, innermost).toString());

        SwingUtilities.convertPointToScreen(
            pt, contentPane);

        lastScreenPt = pt;
        repaint();
    }
});
...

```

The applet shown in Figure 6-2 is listed in Example 6-7.

Example 6-7 Converting Mouse Coordinates

```

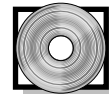
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    private Point lastScreenPt = null;
    private final Container contentPane = getContentPane();
    private PanelWithString
        outer = new PanelWithString(Color.orange),
        inner = new PanelWithString(Color.red),
        innermost = new PanelWithString(Color.yellow);

    public Test() {
        Font font = new Font("Times-Roman", Font.ITALIC, 26);

        contentPane.setLayout(new OverlayLayout(contentPane));
    }
}

```





```
contentPane.add(innermost);
contentPane.add(inner);
contentPane.add(outer);

innermost.setMaximumSize(new Dimension(350,50));
inner.setMaximumSize(new Dimension(450,200));
outer.setMaximumSize(new Dimension(550,400));

setFont(font);
innermost.setFont(font);
inner.setFont(font);
outer.setFont(font);

contentPane.addMouseMotionListener(
    new MouseMotionAdapter() {
    public void mouseMoved(MouseEvent e) {
        Point pt = e.getPoint();

        outer.setString(SwingUtilities.convertPoint(
            contentPane, pt, outer).toString());

        inner.setString(SwingUtilities.convertPoint(
            contentPane, pt, inner).toString());

        innermost.setString(SwingUtilities.convertPoint(
            contentPane, pt, innermost).toString());

        SwingUtilities.convertPointToScreen(
            pt, contentPane);

        lastScreenPt = pt;
        repaint();
    }
});
}
public void paint(Graphics g) {
    super.paint(g);

    if(lastScreenPt != null) {
        String s = new String("Screen: " + lastScreenPt);

        g.setColor(getForeground());
        g.drawString(s,10,g.getFontMetrics().getHeight());

        SwingUtilities.convertPointFromScreen(lastScreenPt,
            contentPane);

        s = "Content Pane: " + lastScreenPt;

        g.drawString(s,10,g.getFontMetrics().getHeight()*2);
    }
    else {
        g.setColor(getForeground());
        g.drawString("MOVE THE MOUSE IN HERE",10,
            g.getFontMetrics().getHeight());
    }
}
```



```
    }  
  }  
}  
class PanelWithString extends JPanel {  
    String s;  
    Color color;  
  
    public PanelWithString(Color color) {  
        this.color = color;  
    }  
    public void setString(String s) {  
        this.s = s;  
    }  
    public void paintComponent(Graphics g) {  
        super.paintComponent(g);  
  
        Dimension size = getSize();  
  
        g.setColor(color);  
        g.fillRect(0,0,size.width,size.height);  
  
        if(s != null) {  
            g.setColor(getForeground());  
            g.drawString(s,10,g.getFontMetrics().getHeight());  
        }  
    }  
}
```

Accessibility Methods

```
public static Accessible getAccessibleAt(Component, Point)  
public static Accessible getAccessibleChild(Component, int)  
public static int getAccessibleChildrenCount(Component)  
public static int getAccessibleIndexInParent(Component)  
public static AccessibleStateSet getAccessibleStateSet(Component)
```

The methods listed above are used to obtain accessibility information from a given component.

Focus Owners / Ancestors / Root Pane / Window

```
public static Component findFocusOwner(Component)  
public static Container getAncestorNamed(String, Component)  
public static Container getAncestorOfClass(Class, Component)  
public static Component getDeepestComponentAt(Component, int, int)
```



```
public static Rectangle getLocalBounds(Component)
public static JRootPane getRoot(Component)
public static JRootPane getRootPane(Component)
public static Window windowForComponent(Component)
```

The methods listed above return information associated with a particular component. Focus owners, named ancestors, root panes, windows, etc., can all be obtained, given a component.

Invoking Runnable Objects

```
public static void invokeAndWait(Runnable)
    throws InterruptedException, InvocationTargetException;
public static void invokeLater(Runnable)
```

The `invokeAndWait` and `invokeLater` methods are used to execute code from a thread other than the event dispatch thread. Both methods are discussed in “Swing and Threads” on page 57 and therefore are not covered here.

Descendants / Event Dispatch Thread

```
public static boolean isDescendingFrom(Component allegedDescendent,
    Component allegedAncestor)
public static boolean isEventDispatchThread()
```

The two methods listed above can be used to determine whether a component is a descendant of another and whether the current thread is the event dispatched thread.

Mouse Buttons

```
public static boolean isLeftMouseButton(MouseEvent)
public static boolean isMiddleMouseButton(MouseEvent)
public static boolean isRightMouseButton(MouseEvent)
```



With the AWT, determining which mouse button was pressed, given a mouse event, involved examining the modifiers of the event and ANDing them with a bit mask, a process that was nonintuitive and difficult to remember. Swing remedies this shortcoming of the AWT by providing three convenience methods for determining which mouse button was pressed for a given mouse event.

Compound Labels / Painting / Updating Component Tree UI

```
public static String layoutCompoundLabel(FontMetrics, String, Icon, int, int, int, int,  
                                           Rectangle, Rectangle, Rectangle, int)  
static String layoutCompoundLabel(JComponent, FontMetrics, String, Icon,  
                                     int, int, int, int, Rectangle, Rectangle, Rectangle, int)  
  
public static void paintComponent(Graphics, Component, Container, int, int, int, int)  
public static void paintComponent(Graphics, Component, Container, Rectangle)  
  
public static void updateComponentTreeUI(Component)
```

The `layoutCompoundLabel` methods are used by Swing buttons and labels to lay out their text and icon. The `paintComponent` methods are used to paint a component in an arbitrary graphics, and the `updateComponentTreeUI` method is used to update the UI delegates associated with a component's descendants.

Swing Constants

Swing provides three interfaces in the `swing` package that define constants. The `SwingConstants` class defines positional constants, in addition to `ScrollPaneConstants` and `WindowConstants` classes that define constants for their respective components.

Interface Summary 6-1 lists the constants defined by the `SwingConstants` interface.



Interface Summary 6-1 SwingConstants

Constants

```
public static final int BOTTOM
public static final int CENTER
public static final int EAST
public static final int HORIZONTAL
public static final int LEFT
public static final int NORTH
public static final int NORTH EAST
public static final int NORTH WEST
public static final int RIGHT
public static final int SOUTH
public static final int SOUTH EAST
public static final int SOUTH WEST
public static final int TOP
public static final int VERTICAL
public static final int WEST
```

A number of Swing classes implement the `SwingConstants` interface: `JCheckBoxMenuItem`, `JLabel`, `JProgressBar`, `JSlider`, and `JTextField`. Implementing the `SwingConstants` interface allows the constants defined by the `SwingConstants` interface to be accessed as though they were declared in the implementing class.

BoxLayout and the Box Class

Swing provides a layout manager—`BoxLayout`—for laying out components horizontally or vertically, and a container—`Box`—that utilizes a `BoxLayout`.



BoxLayout

`BoxLayout` lays out components in a horizontal or vertical line. An axis—either `BoxLayout.X_AXIS` or `BoxLayout.Y_AXIS`—is specified at the time of construction and determines the orientation of the components.

Components laid out along the x axis are laid out horizontally, and components laid out along the y axis are laid out vertically. Components are laid out top to bottom or left to right in the order in which they are added to their container.

`BoxLayout` sizes components laid out horizontally or vertically at their preferred widths or preferred heights, respectively. For a horizontal layout, if all of components are not the same height, `BoxLayout` attempts to set the height of each component to the height of the tallest component in the container. If a component's height cannot be set to the height of the tallest component, the component is placed vertically according to its vertical alignment. For vertical layouts, the component's width is adjusted in the same manner as component height for horizontal layout. This section provides a simple example of the use of the `BoxLayout` layout manager. See “`JToolBar`” on page 560 for an example that is somewhat more complicated and takes component alignment into account.

Figure 6-3 shows an applet that contains two containers, each of which contains buttons. Both containers have a `BoxLayout` as their layout manager; the container on the left lays out components along the `X_AXIS`, and the container on the right lays out components along the `Y_AXIS`.

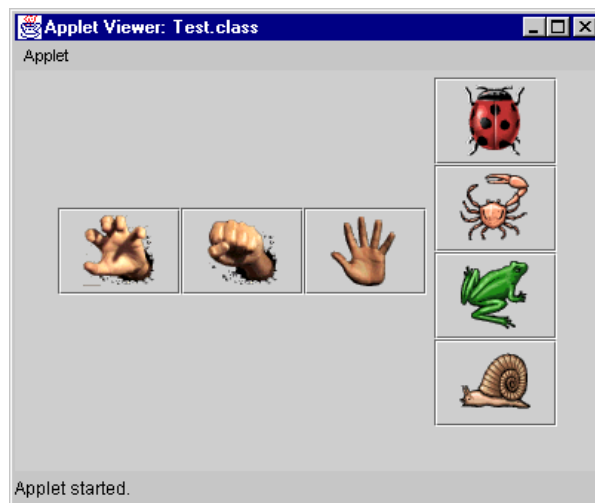


Figure 6-3 Two Containers Using `BoxLayout`



The applet shown in Figure 6-3 implements an extension of the `JPanel` class that is fitted with an instance of `BoxLayout` for its layout manager:

```
class ContainerWithBoxLayout extends JPanel {
    public ContainerWithBoxLayout(int orientation) {
        setLayout(new BoxLayout(this, orientation));
    }
}
```

The applet instantiates two instances of `ContainerWithBoxLayout`: one with an orientation of `BoxLayout.X_AXIS`, and the other with an orientation of `BoxLayout.Y_AXIS`:

```
public class Test extends JApplet {
    public Test() {
        ...
        ContainerWithBoxLayout yaxis =
            new ContainerWithBoxLayout(BoxLayout.Y_AXIS);

        ContainerWithBoxLayout xaxis =
            new ContainerWithBoxLayout(BoxLayout.X_AXIS);
    }
}
```

Buttons are subsequently added to the two containers, and the containers are added to the applet's content pane.

```
...
xaxis.add(new JButton(new ImageIcon("reach.gif")));
xaxis.add(new JButton(new ImageIcon("punch.gif")));
xaxis.add(new JButton(new ImageIcon("open_hand.gif")));

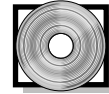
yaxis.add(new JButton(new ImageIcon("ladybug.gif")));
yaxis.add(new JButton(new ImageIcon("crab.gif")));
yaxis.add(new JButton(new ImageIcon("frog.gif")));
yaxis.add(new JButton(new ImageIcon("snail.gif")));

contentPane.setLayout(new FlowLayout());
contentPane.add(xaxis);
contentPane.add(yaxis);
}
}
```



The applet is listed in its entirety in Example 6-8.

Example 6-8 Using BorderLayout



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

public class Test extends JApplet {
    public Test() {
        Container contentPane = getContentPane();
        ContainerWithBoxLayout yaxis =
            new ContainerWithBoxLayout(BoxLayout.Y_AXIS);

        ContainerWithBoxLayout xaxis =
            new ContainerWithBoxLayout(BoxLayout.X_AXIS);

        contentPane.setLayout(new FlowLayout());

        xaxis.add(new JButton(new ImageIcon("reach.gif")));
        xaxis.add(new JButton(new ImageIcon("punch.gif")));
        xaxis.add(new JButton(new ImageIcon("open_hand.gif")));

        yaxis.add(new JButton(new ImageIcon("ladybug.gif")));
        yaxis.add(new JButton(new ImageIcon("crab.gif")));
        yaxis.add(new JButton(new ImageIcon("frog.gif")));
        yaxis.add(new JButton(new ImageIcon("snail.gif")));

        contentPane.add(xaxis);
        contentPane.add(yaxis);
    }
}

class ContainerWithBoxLayout extends JPanel {
    public ContainerWithBoxLayout(int orientation) {
        setLayout(new BorderLayout(this, orientation));
    }
}
```

`BoxLayout` is a simple layout manager. Unlike most of the AWT layout managers, there is no provision to set the gaps between components, or the gap between components and the edge of the container.



The Box Class

The best thing about `BoxLayout` is the `Box` class, an extension of `java.awt.Container` that lays out components with an instance of `BoxLayout`. The `Box` class doesn't really make using `BoxLayout` any easier; in fact, `Box` does little more than the `ContainerWithBoxLayout` class listed in Example 6-8 as far as reducing the complexities of dealing with `BoxLayout`. Where the `Box` class really shines is in the static methods it provides that return one of three substances: *glue*, *struts*, and *rigid areas*.

Glue components can be thought of as gooey substances that expand to fill the region defined by their neighboring components. Glue is somewhat of a misnomer because the "glue" never sets, and its use never results in the immobility of components that it borders. Perhaps a better analogy would be "slime balls," a gooey gel that is available in most toy stores.

Struts are fixed in a dimension of your choice, and they fill the other dimension. For example, the `BoxLayout.createVerticalStrut` method is passed a height, and the component—a.k.a. strut—maintains the height it was created with and expands its width as wide as it can get away with.

Rigid areas are constructed with a `Dimension` and, as you might guess, do not deviate from it. For example, if a rigid area is created with a dimension of `(100, 100)`, the size of the rigid area will always be `(100, 100)`.

Glue, struts, and rigid areas are useful in a number of different contexts, for example, designing input forms. Figure 6-4 shows before and after photos of a panel with some labels and combo boxes. The before picture (top) is straight `GridBagLayout`, whereas the after picture (bottom) uses horizontal and vertical struts to space things in a more aesthetically pleasing manner.

Example 6-9 lists a portion of the code for the panel shown in Figure 6-4. The panel uses an instance of `GridBagLayout` as its layout manager, in conjunction with horizontal and vertical struts.

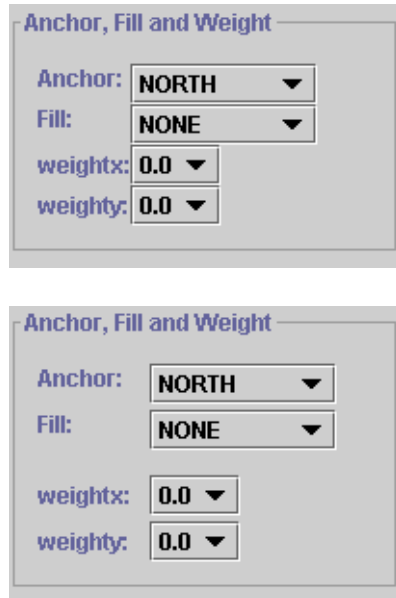


Figure 6-4 Using Struts: before (top) and after (bottom)

Example 6-9 Using Horizontal and Vertical Struts

```
class AnchorFillWeightPanel extends JPanel {
    ...
    public AnchorFillWeightPanel() {
        ...
        GridBagLayout gbl = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();

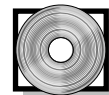
        setLayout(gbl);

        gbc.anchor = GridBagConstraints.NORTHWEST;
        add(anchorLabel, gbc);
        add(Box.createHorizontalStrut(10), gbc);

        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbc.weightx = 1.0;
        add(anchorCombo, gbc);

        gbc.weightx = 0;
        add(Box.createVerticalStrut(3), gbc);

        gbc.gridwidth = 1;
    }
}
```





```
add(fillLabel, gbc);
add(Box.createHorizontalStrut(10), gbc);

gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.weightx = 1.0;
add(fillCombo, gbc);

gbc.weightx = 0;
add(Box.createVerticalStrut(13), gbc);

gbc.gridwidth = 1;
gbc.anchor = GridBagConstraints.WEST;
add(weightxLabel, gbc);
add(Box.createHorizontalStrut(10), gbc);

gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.weightx = 1.0;
add(weightxCombo, gbc);

gbc.weightx = 0;
add(Box.createVerticalStrut(3), gbc);

gbc.gridwidth = 1;
add(weightyLabel, gbc);
add(Box.createHorizontalStrut(10), gbc);

gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.weightx = 1.0;
add(weightyCombo, gbc);
...
}
...
}
```

Although `GridBagLayout` can be somewhat intimidating, it is actually easy to use for forms such as the one depicted in Figure 6-4. A simple pattern of rotating the grid width between `GridBagConstraints.REMAINDER` and 1 puts things in their respective columns. See “Using `GridBagLayout` to Lay Out Text Fields” on page 1468 for a more complicated example of using `GridBagLayout` to layout forms.

The `Box` class is summarized in Class Summary 6-4.



Class Summary 6-4 Box

Extends: `java.awt.Container`

Implements: `Javax.accessibility.Accessible`

Constructors

`public Box(int axis)`

All boxes are constructed with an `integer` value representing the axis upon which components will be laid out. Valid values are:

- `BoxLayout.X_AXIS`
- `BoxLayout.Y_AXIS`

Methods

Boxes and Box Substances

`public static Box createHorizontalBox()`

`public static Box createVerticalBox()`

`public static Component createGlue()`

`public static Component createHorizontalGlue()`

`public static Component createVerticalGlue()`

`public static Component createHorizontalStrut(int width)`

`public static Component createVerticalStrut(int height)`

`public static Component createRigidArea(Dimension)`



Boxes can be created with the `createHorizontalBox` and `createVerticalBox` convenience methods, which are equivalent to the following two lines of code, respectively:

```
Box hb = new Box(Box.X_AXIS);  
Box vb = new Box(Box.Y_AXIS);
```

Glue, struts, and rigid areas are implemented by controlling their minimum, preferred, and maximum sizes. Table 6-1 shows the size requirements for each type of substance that can be created with `Box` static methods.

Table 6-1 Min/Pref/Max Sizes for Box Substances

Substance	Minimum Size	Preferred Size	Maximum Size
Glue	0, 0	0, 0	Short.MAX_VALUE, Short.MAX_VALUE
Horizontal Glue	0, 0	0, 0	Short.MAX_VALUE, 0
Vertical Glue	0, 0	0, 0	0, Short.MAX_VALUE
Horizontal Strut	width, 0	width, 0	width, Short.MAX_VALUE
Vertical Strut	0, height	0, height	Short.MAX_VALUE, height
Rigid Area	dimension ¹	dimension	dimension

1. Rigid areas are constructed with three dimensions.

It should be stressed that the substances returned by static `Box` methods may not behave like glue, struts, and rigid areas if they are added to a container that does not have a `BoxLayout` as its layout manager. Other layout managers are free to ignore the size requests for minimum, preferred, and maximum sizes.

Layout Manager / Accessible Context

```
public void setLayout(LayoutManager)  
public AccessibleContext getAccessibleContext()
```

Each box uses an instance of `BoxLayout` as its layout manager, which it accomplishes by overriding the `setLayout` method.



Like all Swing components, the `Box` class implements `getAccessibleContext()`, which returns an instance of `AccessibleContext` for accessibility purposes.

Progress Monitoring

Operations that take a long time to complete should provide some visual indication of the percentage of the task that is completed. Typically, such visual indications involve a progress bar that is usually displayed in a dialog that allows the task to be canceled.

Swing includes a progress bar component—`JProgressBar`—that is used to communicate the percentage of a task that has been completed. In fact, “Progress Bars, Sliders, and Separators” on page 579 discusses an applet that monitors a time-consuming task with a progress bar.

Swing also provides two classes for monitoring progress that create and display a progress bar in a dialog: `ProgressMonitor` and `ProgressMonitorInputStream`. The former provides methods for setting progress, which updates the progress bar created by the monitor. The latter is an extension of `java.io.FilterInputStream` that is used like any other input stream, except that it displays a dialog containing a progress bar if reading the stream is time consuming.

ProgressMonitor

The `ProgressMonitor` class is used to monitor the progress of a time-consuming task by showing a progress dialog. Progress monitors decide whether an operation takes long enough to justify a progress dialog, with the help of two properties: `millisToDecideToPopup` and `millisToPopup`, which by default are 500 and 2000 milliseconds (0.5 and 2 seconds), respectively. Both properties are settable.

After `millisToDecideToPopup` milliseconds have expired since the creation of a progress monitor, the progress monitor calculates the total time required to complete the operation, depending on the percentage of the task that has been completed thus far. If the total time required is greater than `millisToPopup`, a progress dialog is displayed.



Using a progress monitor involves the following steps:

1. Instantiate an instance of `ProgressMonitor`.
2. Invoke `ProgressMonitor.setProgress(int)` periodically (and, optionally, `ProgressMonitor.setNote(String)`).
3. Invoke `ProgressMonitor.close()` when the operation is complete.

The application shown in Figure 6-5 contains a button that initiates reading of a file. The application uses a progress monitor to show the progress being made while the file is read.

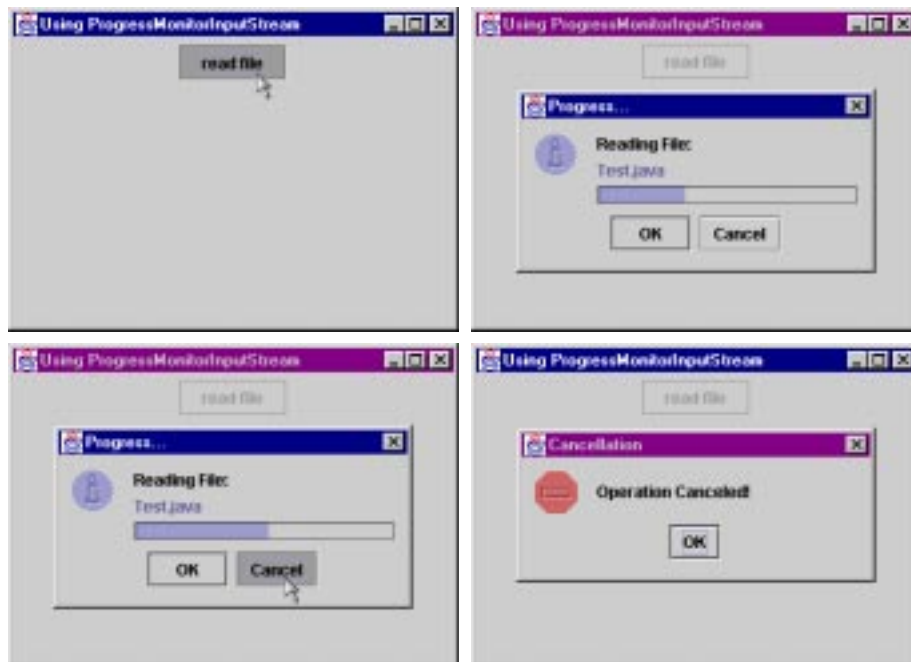


Figure 6-5 Using a ProgressMonitor



The upper-left picture shows the button being activated, thus initiating the reading of the file. The upper-right picture shows the progress monitor's dialog displayed as the file is being read. The lower-left picture shows the dialog (and the task) being canceled, handled by showing a message dialog.

The application adds an action listener to the button that creates a buffered input stream and a progress monitor. The listener subsequently creates an instance of `ReadThread`, which reads the file and updates the progress monitor.

Progress monitors are created with a parent component that is used to parent the monitor's progress dialog. The `ProgressMonitor` constructor is also passed a message that is displayed in the dialog, along with a note. The note can be updated while the dialog is displayed; for example, a progress monitor could be created with a "Reading Files" message, and the monitor's note could be updated for the particular file being read. Progress monitors are also created with minimum and maximum values representing the task at hand.

The application shown in Figure 6-5 creates a `ProgressMonitor` specifying the application's content pane as the parent component, with "Reading File" as the message and the name of the file as the note. The minimum value is specified as 0, and the maximum value is set to the number of bytes in the file.

```
public class Test extends JFrame {
    private JButton readButton = new JButton("read file");
    private BufferedInputStream in;
    private ProgressMonitor pm;
    private String fileName = "Test.java";

    public Test() {
        ...
        readButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                try {
                    in = new BufferedInputStream(
                        new FileInputStream(fileName));

                    pm = new ProgressMonitor(contentPane,
                        "Reading File:",
                        fileName,
                        0, in.available());
                }
                catch(FileNotFoundException fnfx) {
                    fnfx.printStackTrace();
                }
                catch(IOException iox) {
                    iox.printStackTrace();
                }
            }
        });
    }
}
```



```
        ReadThread t = new ReadThread();
        t.start();
    };
}
...
```

The run method of the ReadThread class sets the button's enabled state to false. As long as the progress monitor has not been canceled, bytes are read from the file and the progress monitor is updated by invoking `ProgressMonitor.setProgress()`. If the progress monitor is canceled, a message dialog is displayed. Note that the thread sleeps for 25 milliseconds after every read and prints the character read to the standard out stream to slow the reading of the file.

After the file has been read or the progress monitor has been canceled, the progress monitor's dialog is closed by invoking `ProgressMonitor.close()` and the application's button is enabled.

```
...
class ReadThread extends Thread {
    public void run() {
        try {
            readButton.setEnabled(false);

            while(!pm.isCanceled() && (i = in.read()) != -1) {
                try {
                    Thread.currentThread().sleep(25);
                }
                catch(InterruptedException ex) {
                    ex.printStackTrace();
                }
                System.out.print((char)i);

                SwingUtilities.invokeLater(new Runnable(){
                    public void run() {
                        pm.setProgress(++cnt);
                    }
                });
            }
        }
        if(pm.isCanceled()) {
            JOptionPane.showMessageDialog(
                Test.this,
                "Operation Canceled!",
                "Cancellation",
                JOptionPane.ERROR_MESSAGE);
        }
    }
}
```



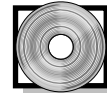
```

        pm.close();
    }
    catch(IOException ex) {
        ex.printStackTrace();
    }
    readButton.setEnabled(true);
}

```

The application shown in Figure 6-5 is listed in its entirety in Example 6-10.

Example 6-10 Using a ProgressMonitor



```

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

public class Test extends JFrame {
    private JButton readButton = new JButton("read file");
    private BufferedInputStream in;
    private ProgressMonitor pm;
    private String fileName = "Test.java";

    public Test() {
        final Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(readButton);

        readButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                try {
                    in = new BufferedInputStream(
                        new FileInputStream(fileName));

                    pm = new ProgressMonitor(contentPane,
                        "Reading File:",
                        fileName,
                        0, in.available());
                }
                catch(FileNotFoundException fnfx) {
                    fnfx.printStackTrace();
                }
                catch(IOException iox) {
                    iox.printStackTrace();
                }
            }
        });

        ReadThread t = new ReadThread();
        t.start();
    }
}

```



```
    });
}
class ReadThread extends Thread {
    int i, cnt=0;
    String s;

    public void run() {
        try {
            readButton.setEnabled(false);

            while(!pm.isCanceled() && (i = in.read()) != -1) {
                try {
                    Thread.currentThread().sleep(25);
                }
                catch(InterruptedException ex) {
                    ex.printStackTrace();
                }
                System.out.print((char)i);

                SwingUtilities.invokeLater(new Runnable(){
                    public void run() {
                        pm.setProgress(++cnt);
                    }
                });
            }
            if(pm.isCanceled())
                JOptionPane.showMessageDialog(
                    Test.this,
                    "Operation Canceled!",
                    "Cancellation",
                    JOptionPane.ERROR_MESSAGE);
            pm.close();
        }
        catch(IOException ex) {
            ex.printStackTrace();
        }
        readButton.setEnabled(true);
    }
}
public static void main(String args[]) {
    GJApp.launch(new Test(),
        "Using Progress Monitors", 300, 300, 450, 300);
}
}
```

The ProgressMonitor class is summarized in Class Summary 6-5.



Class Summary 6-5 ProgressMonitor

Extends: java.lang.Object

Constructors

```
public ProgressMonitor(Component parentComponent, Object message, String note,  
                          int minimum, int maximum)
```

As noted previously, instances of `ProgressMonitor` are created with a parent component, message, note, and minimum and maximum values.

Methods

```
public void close()
```

```
public int getMaximum()
```

```
public int getMillisToDecideToPopup()
```

```
public int getMillisToPopup()
```

```
public int getMinimum()
```

```
public String getNote()
```

```
public boolean isCanceled()
```

```
public void setMaximum(int)
```

```
public void setMillisToDecideToPopup(int)
```

```
public void setMillisToPopup(int)
```

```
public void setMinimum(int)
```

```
public void setNote(String)
```

```
public void setProgress(int)
```



`ProgressMonitor` provides a `close` method that closes the monitor's progress dialog. If a progress monitor has its `setProgress` method invoked with a value that is greater than the monitor's maximum value, the `close` method is called by the monitor itself.

The `ProgressMonitor` class also provides getter and setter methods for the minimum, maximum, note, `millisToPopup`, and `millisToDecideToPopup` properties.

ProgressMonitorInputStream

The `ProgressMonitorInputStream`, an extension of `java.io.FilterInputStream`, creates a progress monitor to monitor the reading of a stream. Instances of `ProgressMonitorInputStream` are used like any other input stream.

This section discusses an application that is functionally identical to the application shown in Figure 6-5 on page 282, except that a `ProgressMonitorInputStream` is used to read from a file. Because of the similarity, the application discussed in this section is not shown.

Like the application shown in Figure 6-5 on page 282, the application contains a button that initiates reading of a file. The application adds an action listener to its button that creates an instance of `ProgressMonitorInputStream` in addition to an instance of `ReadThread` that reads the file.

```
public class Test extends JFrame {
    private ProgressMonitorInputStream in;
    private JButton readButton = new JButton("read file");

    public Test() {
        ...
        readButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                try {
                    in = new ProgressMonitorInputStream(
                        contentPane,
                        "Reading " + fileName,
                        new FileInputStream(fileName));
                }
                catch(FileNotFoundException ex) {
                    ex.printStackTrace();
                }
            }
        });
    }
}
```



```

        ReadThread t = new ReadThread();
        readButton.setEnabled(false);
        t.start();
    }
}
...

```

The `ReadThread` method simply reads the file with a 10 milliseconds sleep between reads to slow down the reading of the file. If the progress dialog shown by the progress monitor input stream is canceled, the stream will throw an `IOException`, which is handled by the application by displaying a message dialog.

```

...
class ReadThread extends Thread {
    public void run() {
        int i;

        try {
            while((i = in.read()) != -1) {
                System.out.print((char)i);
                try {
                    Thread.currentThread().sleep(10);
                }
                catch(Exception ex) {
                    ex.printStackTrace();
                }
            }
            in.close();
        }
        catch(IOException ex) {
            JOptionPane.showMessageDialog(
                Test.this,
                "Operation Canceled!",
                "Cancellation",
                JOptionPane.ERROR_MESSAGE);
        }
        readButton.setEnabled(true);
    }
}
...
}

```

The application discussed above is listed in its entirety in Example 6-11.



Example 6-11 Using ProgressMonitorInputStream

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

public class Test extends JFrame {
    private ProgressMonitorInputStream in;
    private JButton readButton = new JButton("read file");

    public Test() {
        final Container contentPane = getContentPane();
        final String fileName = "Test.java";

        contentPane.setLayout(new FlowLayout());
        contentPane.add(readButton);

        readButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                try {
                    in = new ProgressMonitorInputStream(
                        contentPane,
                        "Reading " + fileName,
                        new FileInputStream(fileName));
                }
                catch(FileNotFoundException ex) {
                    ex.printStackTrace();
                }

                ReadThread t = new ReadThread();
                readButton.setEnabled(false);
                t.start();
            }
        });
    }

    class ReadThread extends Thread {
        public void run() {
            int i;

            try {
                while((i = in.read()) != -1) {
                    System.out.print((char)i);
                    try {
                        Thread.currentThread().sleep(10);
                    }
                    catch(Exception ex) {
                        ex.printStackTrace();
                    }
                }
            }
            in.close();
        }
    }
}
```



```
        catch(IOException ex) {
            JOptionPane.showMessageDialog(
                Test.this,
                "Operation Canceled!",
                "Cancellation",
                JOptionPane.ERROR_MESSAGE);
        }
        readButton.setEnabled(true);
    }
}
public static void main(String args[]) {
    GJApp.launch(new Test(),
        "Using ProgressMonitorInputStream",
        300, 300, 450, 300);
}
}
```

The `ProgressMonitorInputStream` is summarized in Class Summary 6-6.

Class Summary 6-6 ProgressMonitorInputStream

Extends: `java.io.FilterInputStream`

Constructors

public ProgressMonitorInputStream(Component parentComponent, Object message, InputStream)

Instances of `ProgressMonitorInputStream` are created with a parent component, a message, and an input stream. Like the message specified for instances of `ProgressMonitor`, the message passed to the `ProgressMonitorInputStream` constructor is an `Object` reference. The message specified for a `ProgressMonitorInputStream` is handled in the same manner as messages for option panes; see “`JOptionPane`” on page 815 for more information concerning option panes and messages.



Methods

public ProgressMonitor getProgressMonitor()

public void close() throws IOException

public int read() throws IOException

public int read(byte[]) throws IOException

public int read(byte[], int, int) throws IOException

public synchronized void reset() throws IOException

public long skip(long) throws IOException

The `ProgressMonitorInputStream` provides methods for obtaining a reference to the stream's progress monitor. The rest of the methods implemented by the `ProgressMonitorInputStream` class are overridden from the `java.io.FilterInputStream` class to manipulate the stream's progress monitor.

Undo/Redo

Swing provides support for undoing and redoing operations with classes and interfaces defined in the `javax.swing.undo` package, which represents a general undo/redo facility. A class diagram of the `javax.swing.undo` package is shown in Figure 6-6.

Undoable operations (a.k.a. edits) are represented by the `UndoableEdit` interface. The `javax.swing.undo` package provides four classes that implement the `UndoableEdit` interface: `AbstractUndoableEdit` (which, despite its name is not an abstract class), `CompoundEdit`, `UndoManager`, and `StateEdit`. An `UndoableEditSupport` class is also provided to assist with notifying listeners of undoable edits.

The `UndoableEdit` interface is summarized in Interface Summary 6-2.



Figure 6-6 The javax.swing.undo Package

Interface Summary 6-2 UndoableEdit

Undo / Redo

```
public abstract boolean canRedo()
public abstract boolean canUndo()
```



```
public abstract void redo() throws CannotRedoException
public abstract void undo() throws CannotUndoException

public abstract void die()
```

Undoable edits must be able to report whether they can undo or redo an operation by implementing the `canRedo` and `canUndo` methods defined by the `UndoableEdit` interface.

The `undo` and `redo` methods perform the undo and redo operations, respectively. Notice that both the `undo` and `redo` methods may throw exceptions if they are called when an operation cannot be undone or redone, respectively.

The `die` method is called for edits that can no longer be undone or redone. Undoable edits typically use the `die` method to release resources associated with an operation.

Presentation Names

```
public abstract String getPresentationName()
public abstract String getRedoPresentationName()
public abstract String getUndoPresentationName()
```

Every edit must be able to report a presentation name, an undo presentation name, and a redo presentation name. An edit's presentation name should provide a localized human readable name that represents the edit. The undo presentation name represents a description of the undoable form of the edit, and the redo presentation name represents a description of the redoable form of the edit. All three names are typically presented to users in some fashion; for example, as the text displayed in a menu item.

Coalescing Edits

```
public abstract boolean addEdit(UndoableEdit)
public abstract boolean replaceEdit(UndoableEdit)
```



Undoable edits can be coalesced by having one edit absorb another. The `addEdit` method is passed an edit that is absorbed by the edit on whose behalf the method is invoked. The `replaceEdit` method is the inverse of the `addEdit` method and is called when the edit on whose behalf the method is invoked is absorbed by the edit passed to the method.

Significance

```
public abstract boolean isSignificant()
```

Edits can be specified as significant or insignificant. Insignificant edits are typically side effects of a significant edit; for example, when text is selected and subsequently deleted, the selection of the text would most likely be an insignificant edit.

The significance of an edit is typically used to determine which edits to present to users and is also used by the `UndoManager`, as outlined in “UndoManager” on page 308.

A Simple Undo/Redo Example

The applet shown in Figure 6-7 represents a simple example of undoing and redoing an operation. The applet provides a menu containing a menu item that allows the background color of a panel contained in the applet to be modified. The menu also provides a menu item that allows the background color change to be undone and redone.

The applet implements an extension of the `AbstractUndoableEdit` class—`BackgroundColorEdit`—and an instance of `BackgroundColorEdit` is used to undo and redo the background color change. The applet also keeps track of the previous (old) background color.

```
public class Test extends JApplet {
    private JPanel colorPanel = new JPanel();
    private BackgroundColorEdit edit = new BackgroundColorEdit();
    private Color oldColor;
    ...
}
```

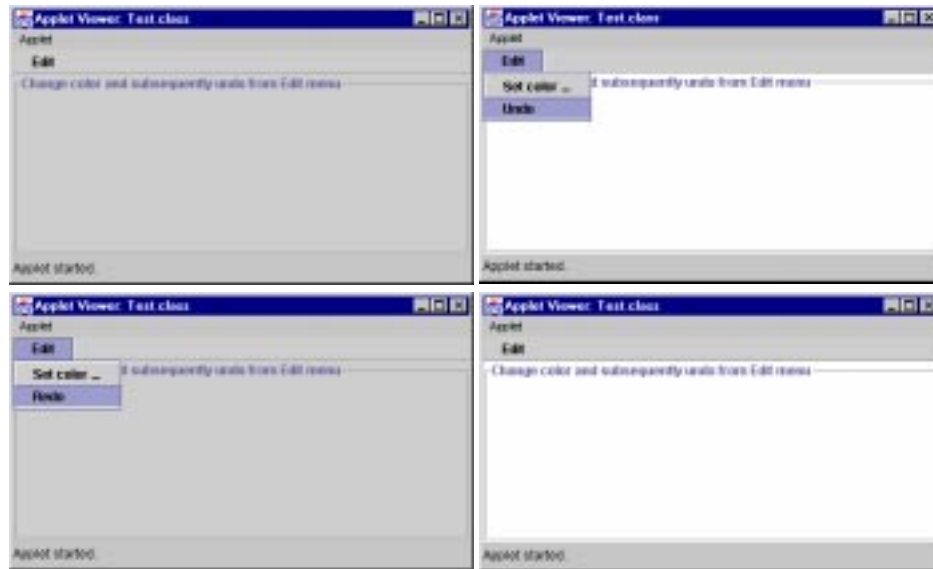


Figure 6-7 Simple Undo/Redo Example

The applet also implements two inner class extensions of the `AbstractAction` class for setting the background color and undoing/redoing the operation: `SetColorAction` and `UndoAction`, respectively. See “Actions” on page 235 for more information concerning actions in general and the `AbstractAction` class. An instance of `SetColorAction` is associated with the “Set color ...” menu item, and an instance of `UndoAction` is associated with the Undo/Redo menu item.

The `SetColorAction.actionPerformed` method—which is invoked when the Set color ... menu item is activated—displays a color chooser for selection of the new background color. If a color was selected from the color chooser, the current background color is saved and the panel’s background color is set.

```
...
class SetColorAction extends AbstractAction {
    public SetColorAction() {
        super("Set color ...");
    }
    public void actionPerformed(ActionEvent e) {
        Color color = JColorChooser.showDialog(
            Test.this, // parent component
```



```

        "Pick A Color", // dialog title
        null); // initial color

    if(color != null) {
        oldColor = colorPanel.getBackground();
        colorPanel.setBackground(color);
    }
}
...

```

The `UndoAction.actionPerformed` method—which is invoked when the Undo/Redo menu item is activated—obtains the text from the menu item (with the `Action.getValue` method). If the item's text is the same as the edit's undo presentation name, the `undo` method is called for the edit; otherwise, the `redo` method is invoked. The name of the action is subsequently updated, and the update is reflected by a change to the text displayed in the menu item.

```

...
class UndoAction extends AbstractAction {
    public UndoAction() {
        putValue(Action.NAME, edit.getUndoPresentationName());
    }
    public void actionPerformed(ActionEvent e) {
        String name = (String)getValue(Action.NAME);
        boolean isUndo = name.equals(
            edit.getUndoPresentationName());

        if(isUndo) {
            edit.undo();
            putValue(Action.NAME,
                edit.getRedoPresentationName());
        }
        else {
            edit.redo();
            putValue(Action.NAME,
                edit.getUndoPresentationName());
        }
    }
}
...

```

The `BackgroundColorEdit` class extends `AbstractUndoableEdit`. Both the `undo` and `redo` methods simply restore the previous background color. Notice that both of the methods invoke their superclass' method of the same name to ensure that the action's state is kept in sync.



```
...
class BackgroundColorEdit extends AbstractUndoableEdit {
    public void undo() throws CannotUndoException {
        super.undo();
        toggleColor();
    }
    public void redo() throws CannotRedoException {
        super.redo();
        toggleColor();
    }
    public String getUndoPresentationName() {
        return "Undo";
    }
    public String getRedoPresentationName() {
        return "Redo";
    }
    private void toggleColor() {
        Color color = colorPanel.getBackground();
        colorPanel.setBackground(oldColor);
        oldColor = color;
    }
}
}
```

The applet shown in Figure 6-7 is listed in its entirety in Example 6-12.



Example 6-12 A Simple Undo/Redo Example

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

public class Test extends JApplet {
    private JPanel colorPanel = new JPanel();
    private BackgroundColorEdit undo = new BackgroundColorEdit();
    private Color oldColor;

    public void init() {
        colorPanel.setBorder(
            BorderFactory.createTitledBorder(
                "Change color and subsequently undo " +
                "from the Edit menu"));

        makeMenuBar();
        getContentPane().add(colorPanel, BorderLayout.CENTER);
    }
    private void makeMenuBar() {
        JMenuBar menuBar = new JMenuBar();
        JMenu editMenu = new JMenu("Edit");
    }
}
```



```

editMenu.add(new SetColorAction());
editMenu.add(new UndoAction());

menuBar.add(editMenu);
setJMenuBar(menuBar);
}
class SetColorAction extends AbstractAction {
    public SetColorAction() {
        super("Set color ...");
    }
    public void actionPerformed(ActionEvent e) {
        Color color = JColorChooser.showDialog(
            Test.this, // parent component
            "Pick A Color", // dialog title
            null); // initial color

        if(color != null) {
            oldColor = colorPanel.getBackground();
            colorPanel.setBackground(color);
        }
    }
}
class UndoAction extends AbstractAction {
    public UndoAction() {
        putValue(Action.NAME, undo.getUndoPresentationName());
    }
    public void actionPerformed(ActionEvent e) {
        String name = (String)getValue(Action.NAME);
        boolean isUndo = name.equals(
            undo.getUndoPresentationName());

        if(isUndo) {
            undo.undo();
            putValue(Action.NAME,
                undo.getRedoPresentationName());
        }
        else {
            undo.redo();
            putValue(Action.NAME,
                undo.getUndoPresentationName());
        }
    }
}
class BackgroundColorEdit extends AbstractUndoableEdit {
    public void undo() throws CannotUndoException {
        super.undo();
        toggleColor();
    }
    public void redo() throws CannotRedoException {
        super.redo();
        toggleColor();
    }
    public String getUndoPresentationName() {

```



```
        return "Undo";
    }
    public String getRedoPresentationName() {
        return "Redo";
    }
    private void toggleColor() {
        Color color = colorPanel.getBackground();
        colorPanel.setBackground(oldColor);
        oldColor = color;
    }
}
}
```

UndoableEditSupport

The example shown in Figure 6-7 on page 296 is a simple illustration of implementing undo/redo; however, it is not very realistic. Typically, support for undoing/redoing operations is built into a component. When an undoable edit is performed on a component, the component fires an undoable edit to registered undoable edit listeners. For example, the applet shown in Figure 6-7 on page 296 and listed in Example 6-12 is rewritten in Example 6-13 so that undoable edits are created and fired to listeners by the `ColorPanel` class.

Example 6-13 Using UndoableEditSupport



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

public class Test extends JApplet {
    private ColorPanel colorPanel = new ColorPanel();
    private UndoAction undoAction = new UndoAction();

    public void init() {
        colorPanel.setBorder(
            BorderFactory.createTitledBorder(
                "Change color and subsequently undo " +
                "from the Edit menu"));

        makeMenuBar();
        colorPanel.addUndoableEditListener(undoAction);
        getContentPane().add(colorPanel, BorderLayout.CENTER);
    }
    private void makeMenuBar() {
        JMenuBar menuBar = new JMenuBar();
        JMenu editMenu = new JMenu("Edit");
```



```

editMenu.add(new SetColorAction());
editMenu.add(undoAction);

menuBar.add(editMenu);
setJMenuBar(menuBar);
}
class UndoAction extends AbstractAction
    implements UndoableEditListener {
    UndoableEdit lastEdit;

    public UndoAction() {
        putValue(Action.NAME, "Undo");
        setEnabled(false);
    }
    public void actionPerformed(ActionEvent e) {
        String name = (String)getValue(Action.NAME);
        boolean isUndo = name.equals(
            lastEdit.getUndoPresentationName());
        if(isUndo) {
            lastEdit.undo();
            putValue(Action.NAME,
                lastEdit.getRedoPresentationName());
        }
        else {
            lastEdit.redo();
            putValue(Action.NAME,
                lastEdit.getUndoPresentationName());
        }
    }
    public void undoableEditHappened(UndoableEditEvent e) {
        lastEdit = e.getEdit();

        putValue(Action.NAME,
            lastEdit.getUndoPresentationName());

        if(lastEdit.canUndo())
            setEnabled(true);
    }
}
class SetColorAction extends AbstractAction {
    public SetColorAction() {
        super("Set color ...");
    }
    public void actionPerformed(ActionEvent e) {
        Color color = JColorChooser.showDialog(
            Test.this, // parent component
            "Pick A Color", // dialog title
            null); // initial color

        if(color != null) {
            colorPanel.setBackground(color);
        }
    }
}

```



```
    }  
  }  
  class ColorPanel extends JPanel {  
    UndoableEditSupport support;  
    BackgroundColorEdit edit = new BackgroundColorEdit();  
    Color oldColor;  
  
    public void addUndoableEditListener(  
        UndoableEditListener l) {  
      support.addUndoableEditListener(l);  
    }  
    public void removeUndoableEditListener(  
        UndoableEditListener l) {  
      support.removeUndoableEditListener(l);  
    }  
    public void setBackground(Color color) {  
      oldColor = getBackground();  
      super.setBackground(color);  
  
      if(support == null)  
        support = new UndoableEditSupport();  
  
      support.postEdit(edit);  
    }  
  }  
  class BackgroundColorEdit extends AbstractUndoableEdit {  
    public void undo() throws CannotUndoException {  
      super.undo();  
      toggleColor();  
    }  
    public void redo() throws CannotRedoException {  
      super.redo();  
      toggleColor();  
    }  
    public String getUndoPresentationName() {  
      return "Undo Background Color Change";  
    }  
    public String getRedoPresentationName() {  
      return "Redo Background Color Change";  
    }  
    private void toggleColor() {  
      Color color = getBackground();  
      setBackground(oldColor);  
      oldColor = color;  
    }  
  }  
}
```

The applet listed in Example 6-13 implements the `BackgroundColorEdit` class as an inner class of the `ColorPanel` class, and the `ColorPanel` class fires undoable edit events when its background color is modified.



The `UndoAction` class implements the `UndoableEditListener` interface by implementing the `undoableEditHappened` method, which retains a reference to the edit and sets the enabled status of the action depending upon whether the edit can be undone.

The `ColorPanel` class fires undoable edit events with the help of the `UndoableEditSupport` class, which provides methods for adding and removing undoable edit listeners and for firing events to the listeners.

Compound Edits

It is often the case that multiple undoable edits must be stored and undone all at once. The `CompoundEdit` class, which extends `AbstractUndoableEdit`, provides that capability.

Here's how compound edits work: A compound edit is instantiated, and undoable edits are added to the compound edit. While edits are being added to a compound edit, the compound edit is in an *in progress* state and cannot be undone until `CompoundEdit.end()` is invoked. Once `CompoundEdit.end()` is called, a call to `CompoundEdit.undo()` undoes all of the edits, from the last edit to the first, that were added to the compound edit while it was in progress.

The applet shown in Figure 6-8 illustrates the use of compound edits. The applet contains an extension of `JList` that provides an `undoableAdd` method that adds an object to the list and subsequently fires an undoable edit to the list's undoable edit listeners. The applet also maintains a compound edit that is used to undo additions to the list.

The top-left picture shows the applet after six items have been added to the list by six activations of the Add Item button. The top-right picture shows the applet after the End button has been activated, causing `CompoundEdit.end()` to be invoked for the compound edit maintained by the applet. The bottom-left picture shows the applet after the Undo button has been activated, thus invoking `CompoundEdit.undo()`, which causes the additions to be undone. The bottom-right picture shows the applet after `CompoundEdit.redo()` has been invoked by activating the Redo button.

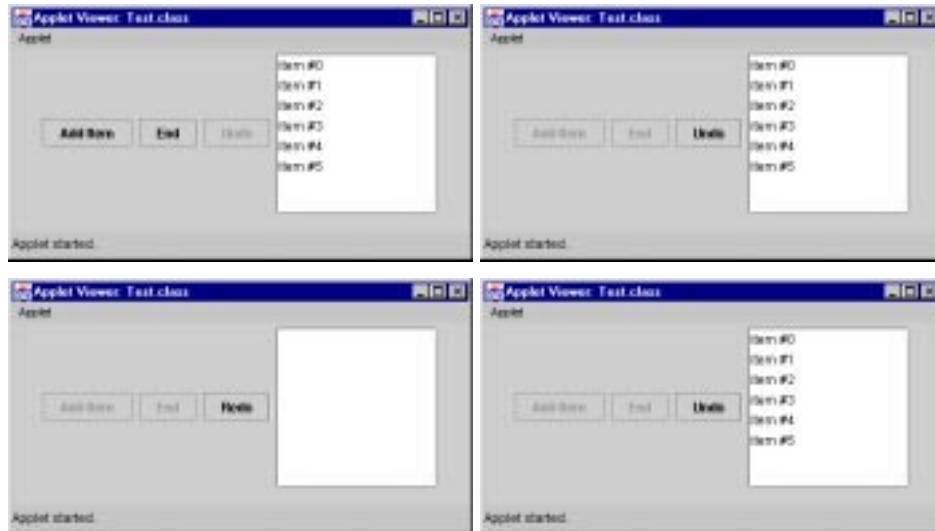


Figure 6-8 Using Compound Edits

The list contained in the applet shown in Figure 6-8 is an instance of `UndoableList`, which supports undoing additions to the list. The `UndoableList` class provides an `undoableAdd` method that adds an object to the list and fires an instance of `UndoableList.AddItemEdit`. Listener registration and event firing is performed with the aid of an instance of `UndoableEditSupport`.

The `AddItemEdit` class extends `AbstractUndoableEdit` and implements `undo()` by removing the last item added to the list. `AddItemEdit.redo()` adds the item that was removed by the `undo` method.

```
class UndoableList extends JList {
    UndoableEditSupport support = new UndoableEditSupport();
    DefaultListModel model;

    public UndoableList() {
        setModel(model = new DefaultListModel());
    }
    public void addUndoableEditListener(UndoableEditListener l) {
        support.addUndoableEditListener(l);
    }
    public void removeUndoableEditListener(
        UndoableEditListener l) {
        support.removeUndoableEditListener(l);
    }
}
```



```

    }
    public void undoableAdd(Object s) {
        model.addElement(s);
        support.postEdit(new AddItemEdit());
    }
    class AddItemEdit extends AbstractUndoableEdit {
        Object lastItemAdded;

        public void undo() throws CannotUndoException {
            super.undo();
            lastItemAdded = model.getElementAt(model.getSize()-1);
            model.removeElement(lastItemAdded);
        }
        public void redo() throws CannotRedoException {
            super.redo();
            model.addElement(lastItemAdded);
        }
    }
}

```

The applet creates an instance of `UndoableList`, wrapped in a scrollpane, that is added to the applet's content pane. The applet also creates an instance of `UndoAction` and an instance of `CompoundEdit`, in addition to the Add, End, and Undo buttons.

An action listener added to the End button invokes `CompoundEdit.end()`, and an action listener added to the Add button adds a string to the list. Both listeners invoke the applet's `updateButtonsEnabledState`, which sets the enabled state for the three buttons depending on the state of the compound edit.

```

public class Test extends JApplet {
    private UndoableList list = new UndoableList();
    private JScrollPane scrollPane = new JScrollPane(list);

    private JButton addButton = new JButton("Add Item"),
        endButton = new JButton("End"),
        undoButton = new JButton("Undo");

    private UndoAction undoAction = new UndoAction();
    private CompoundEdit compoundEdit = new CompoundEdit();
    private int cnt=0;

    public void init() {
        // add buttons and scrollpane to content pane ...

        endButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                compoundEdit.end();
                updateButtonsEnabledState();
            }
        });
    }
}

```



```
    }  
  });  
  addButton.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
      list.undoableAdd("item #" + cnt++);  
      updateButtonsEnabledState();  
    }  
  });  
  undoButton.addActionListener(undoAction);  
  
  endButton.setEnabled(false);  
  undoButton.setEnabled(false);  
}  
private void updateButtonsEnabledState() {  
  boolean inProgress = compoundEdit.isInProgress();  
  
  endButton.setEnabled(inProgress);  
  addButton.setEnabled(inProgress);  
  
  if(undoButton.getText().equals("Undo"))  
    undoButton.setEnabled(compoundEdit.canUndo());  
  else  
    undoButton.setEnabled(compoundEdit.canRedo());  
}
```

The applet shown in Figure 6-8 is listed in its entirety in Example 6-14.

Example 6-14 Using Compound Edits



```
import javax.swing.*;  
import javax.swing.event.*;  
import javax.swing.undo.*;  
import java.awt.*;  
import java.awt.event.*;  
  
public class Test extends JApplet {  
  private UndoableList list = new UndoableList();  
  private JScrollPane scrollPane = new JScrollPane(list);  
  
  private JButton addButton = new JButton("Add Item"),  
    endButton = new JButton("End"),  
    undoButton = new JButton("Undo");  
  
  private UndoAction undoAction = new UndoAction();  
  private CompoundEdit compoundEdit = new CompoundEdit();  
  private int cnt=0;  
  
  public void init() {  
    Container contentPane = getContentPane();
```



```

contentPane.setLayout(new FlowLayout());
contentPane.add(addButton);
contentPane.add(endButton);
contentPane.add(undoButton);
contentPane.add(scrollPane);

scrollPane.setPreferredSize(new Dimension(150,150));
list.addUndoableEditListener(undoAction);

endButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        compoundEdit.end();
        updateButtonsEnabledState();
    }
});
addButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        list.undoableAdd("item #" + cnt++);
        updateButtonsEnabledState();
    }
});
undoButton.addActionListener(undoAction);

endButton.setEnabled(false);
undoButton.setEnabled(false);
}
private void updateButtonsEnabledState() {
    boolean inProgress = compoundEdit.isInProgress();

    endButton.setEnabled(inProgress);
    addButton.setEnabled(inProgress);

    if(undoButton.getText().equals("Undo"))
        undoButton.setEnabled(compoundEdit.canUndo());
    else
        undoButton.setEnabled(compoundEdit.canRedo());
}
class UndoAction extends AbstractAction
    implements UndoableEditListener {

    public UndoAction() {
        putValue(Action.NAME, "Undo");
    }

    public void actionPerformed(ActionEvent e) {
        String name = undoButton.getText();
        boolean isUndo = name.equals("Undo");

        if(isUndo) compoundEdit.undo();
        else    compoundEdit.redo();

        undoButton.setText(isUndo ? "Redo" : "Undo");
    }
}

```



```
    }
    public void undoableEditHappened(UndoableEditEvent e) {
        UndoableEdit edit = e.getEdit();
        compoundEdit.addEdit(edit);
        endButton.setEnabled(true);
    }
}
}
class UndoableList extends JList {
    UndoableEditSupport support = new UndoableEditSupport();
    DefaultListModel model;

    public UndoableList() {
        setModel(model = new DefaultListModel());
    }
    public void addUndoableEditListener(UndoableEditListener l) {
        support.addUndoableEditListener(l);
    }
    public void removeUndoableEditListener(
        UndoableEditListener l) {
        support.removeUndoableEditListener(l);
    }
    public void undoableAdd(Object s) {
        model.addElement(s);
        support.postEdit(new AddItemEdit());
    }
    class AddItemEdit extends AbstractUndoableEdit {
        Object lastItemAdded;

        public void undo() throws CannotUndoException {
            super.undo();
            lastItemAdded = model.getElementAt(model.getSize()-1);
            model.removeElement(lastItemAdded);
        }
        public void redo() throws CannotRedoException {
            super.redo();
            model.addElement(lastItemAdded);
        }
    }
}
```

UndoManager

The `javax.swing.undo` provides an `UndoManager` class that extends `CompoundEdit`. `UndoManager` differs from its superclass in two ways. First, the `UndoManager` class implements the `UndoableEditListener` interface by adding edits it is passed to itself. This approach allows the undo manager to manage undoable events from more than one source.



Second, the undo manager can undo while it is in progress, unlike the `CompoundEdit` class, which can only undo after `CompoundEdit.end()` is invoked. After `end()` is called for an instance of `UndoManager`, the undo manager effectively transforms edits into a compound edit. This feature is useful for situations where minor edits need to be available for undoing until they are committed but afterward should be treated as a single edit.

The applet shown in Figure 6-9 is identical to the applet listed in Example 6-14, except that the compound edit is replaced with an instance of `UndoManager`. The applet shown in Figure 6-9 is not listed but is available on the CD in the back of the book.



Figure 6-9 Using the Undo Manager



The top-left picture in Figure 6-9, like the top-left picture in Figure 6-8 on page 304, shows the applet after six items have been added to the list. Notice that the Undo button is enabled in the top-left picture in Figure 6-9, as opposed to the top-left picture in Figure 6-8, because `undo` can be invoked for instances of `UndoManager` before the manager's `end` method is invoked. The top-right picture in Figure 6-9 shows the applet after the Undo button has been activated, which undoes the addition of item #5.

The middle left picture in Figure 6-9 shows the applet after the Redo button has subsequently been activated, which undoes the deletion of item #5. The middle-right picture shows the applet after the End button has been activated.

The bottom-left picture shows the applet after the Undo button has subsequently been activated. Because the `end` method has been invoked for the undo manager, the manager now behaves like a compound edit, meaning that a call to `UndoManager.undo` calls `undo` on each of the edits added to the manager. Finally, the bottom-right picture shows the applet after the Redo button has been activated, which restores the items originally added to the list.

State Edits

State edits toggle between two arbitrary states associated with a state editable object that is associated with a state edit. State editable objects implement the `StateEditable` interface, which defines two methods: `void storeState(Hashtable)` and `void restoreState(Hashtable)`.

State edits are represented by the `StateEdit` class, which must be constructed with an object that implements the `StateEditable` interface. Instances of `StateEdit` invoke `storeState()` for the `StateEditable` object passed to the `StateEdit` constructor when the state edit is constructed and when `StateEdit.end()` is invoked. Thus, the pre- and post-states for the state editable object are defined. Subsequent calls to `undo` and `redo` for the state edit result in calls to `StateEditable.restoreState()`, which is passed an appropriate hash table.

The applet shown in Figure 6-10 illustrates the use of state edits. The applet contains three buttons for starting, ending, and undoing or redoing an edit and a panel with four text fields. The text fields are contained in a panel that extends `JPanel` and implements the `StateEditable` interface.

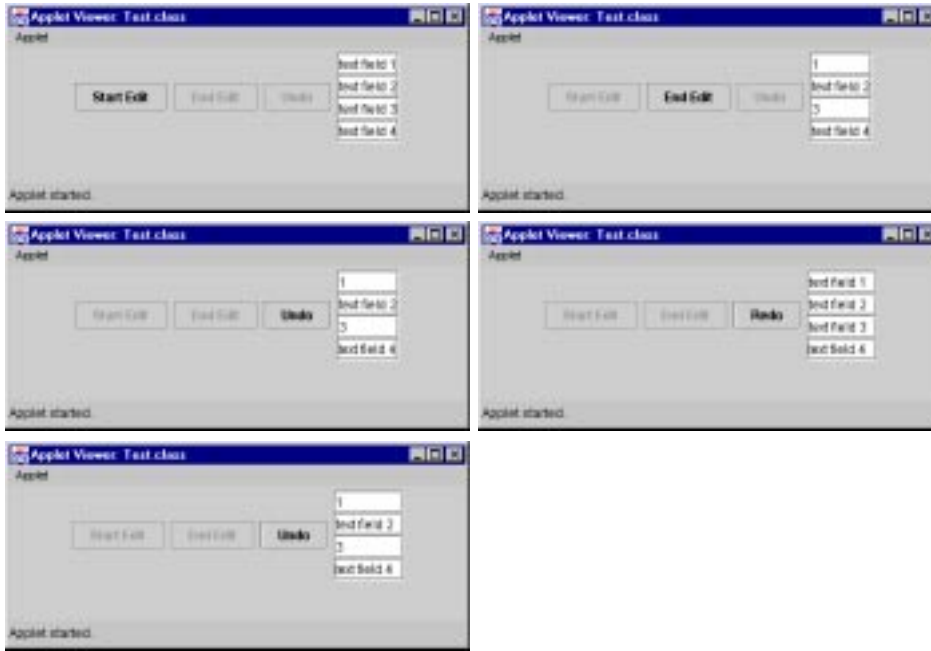


Figure 6-10 Using State Edits

The top-left picture shows the applet as it appears initially. The top-right picture shows the applet after the Start Edit button has been activated and the contents of two of the text fields have been changed. The middle-left picture shows the applet after the End Edit button has been activated, and the middle-right picture shows the applet after the Undo button has been activated. Finally, the bottom picture shows the applet after the Redo button has been activated.

The panel contained in the applet is an instance of `TextFieldPanel`, which contains four text fields that are laid out by an instance of `BoxLayout`. See “`BoxLayout` and the `Box` Class” on page 272 for more information concerning the `BoxLayout` layout manager.

The `TextFieldPanel.storeState` method puts a reference to each text field in the hash table it is passed as key/value pairs. The `TextFieldPanel.restoreState` method iterates over the key/value pairs in the hash table it is passed and sets the text for each of the text fields stored in the hash table as keys. It should be noted that hash tables passed to `StateEditable.restoreState()` contain only entries that have changed.



```
class TextFieldPanel extends JPanel implements StateEditable {
    JTextField[] fields = new JTextField[] {
        new JTextField("text field 1"),
        new JTextField("text field 2"),
        new JTextField("text field 3"),
        new JTextField("text field 4"),
    };

    public TextFieldPanel() {
        setLayout(new BorderLayout(this, BorderLayout.Y_AXIS));

        for(int i=0; i < fields.length; ++i)
            add(fields[i]);
    }
    public void storeState(Hashtable hashtable) {
        for(int i=0; i < fields.length; ++i)
            hashtable.put(fields[i], fields[i].getText());
    }
    public void restoreState(Hashtable hashtable) {
        Enumeration keys = hashtable.keys();

        while(keys.hasMoreElements()) {
            JTextField field = (JTextField)keys.nextElement();
            field.setText((String)hashtable.get(field));
        }
    }
}
```

The applet creates an instance of `TextFieldPanel` and the three buttons, all of which are subsequently added to the applet's content pane. Each of the buttons is fitted with an action listener.

The action listener associated with the Start button creates an instance of `StateEdit`, passing a reference to the `TextFieldPanel`. The listener also sets the enabled state of the End button to `true` and disables the Start button.

The action listener associated with the End button invokes `end()` for the state edit and enables the Undo button and disables the End button.

The action listener associated with the Undo button invokes either `undo()` or `redo()` for the state edit and updates the Undo button's text.

```
public class Test extends JApplet {
    private TextFieldPanel panel = new TextFieldPanel();
    private StateEdit stateEdit;

    private JButton startButton = new JButton("Start Edit"),
        endButton = new JButton("End Edit"),
        undoButton = new JButton("Undo");
```



```

public void init() {
    // add buttons and panel to content pane ...

    endButton.setEnabled(false);
    undoButton.setEnabled(false);

    startButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            stateEdit = new StateEdit(panel);
            endButton.setEnabled(true);
            startButton.setEnabled(false);
        }
    });
    endButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            stateEdit.end();
            undoButton.setEnabled(true);
            endButton.setEnabled(false);
        }
    });
    undoButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            String name = undoButton.getText();
            boolean isUndo = name.equals("Undo");

            if(isUndo) stateEdit.undo();
            else      stateEdit.redo();

            undoButton.setText(isUndo ? "Redo" : "Undo");
        }
    });
}
}

```

The applet shown in Figure 6-10 is listed in its entirety in Example 6-15.

Example 6-15 Using State Edits

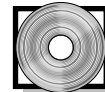
```

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

public class Test extends JApplet {
    private TextFieldPanel panel = new TextFieldPanel();
    private StateEdit stateEdit;

    private JButton startButton = new JButton("Start Edit"),
        endButton = new JButton("End Edit"),

```





```
        undoButton = new JButton("Undo");

public void init() {
    Container contentPane = getContentPane();

    contentPane.setLayout(new FlowLayout());
    contentPane.add(startButton);
    contentPane.add(endButton);
    contentPane.add(undoButton);
    contentPane.add(panel);

    endButton.setEnabled(false);
    undoButton.setEnabled(false);

    startButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            stateEdit = new StateEdit(panel);
            endButton.setEnabled(true);
            startButton.setEnabled(false);
        }
    });
    endButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            stateEdit.end();
            undoButton.setEnabled(true);
            endButton.setEnabled(false);
        }
    });
    undoButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            String name = undoButton.getText();
            boolean isUndo = name.equals("Undo");

            if(isUndo) stateEdit.undo();
            else stateEdit.redo();

            undoButton.setText(isUndo ? "Redo" : "Undo");
        }
    });
}

class TextFieldPanel extends JPanel implements StateEditable {
    JTextField[] fields = new JTextField[] {
        new JTextField("text field 1"),
        new JTextField("text field 2"),
        new JTextField("text field 3"),
        new JTextField("text field 4"),
    };

    public TextFieldPanel() {
        setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));

        for(int i=0; i < fields.length; ++i)
```



```
        add(fields[i]);
    }
    public void storeState(Hashtable hashtable) {
        for(int i=0; i < fields.length; ++i)
            hashtable.put(fields[i], fields[i].getText());
    }
    public void restoreState(Hashtable hashtable) {
        Enumeration keys = hashtable.keys();

        while(keys.hasMoreElements()) {
            JTextField field = (JTextField)keys.nextElement();
            field.setText((String)hashtable.get(field));
        }
    }
}
```

Parting Shots

Swing provides a number of utilities that are used internally within Swing, but can also be used by developers. The `Box` and `BoxLayout` classes provide the ability to layout components in a horizontal or vertical line—something that the AWT lacked. The `SwingUtilities` class provides a wide range of methods encompassing functionality that had been written in some form or another many times over by developers in the past.

Progress monitors and undo/redo support are fundamental features of most user interfaces and are relatively simple to use.