

第8章 事件处理

事件处理基础
动作

鼠标事件
AWT事件继承层次

对于图形用户界面的程序来说，事件处理是十分重要的。要想实现用户界面，必须掌握Java事件处理的基本方法。本章将讲解Java AWT事件模型的工作机制，从中可以看到如何捕获用户界面组件和输入设备产生的事件。另外，本章还介绍如何以更加结构化的方式处理动作（actions）事件。

8.1 事件处理基础

任何支持GUI的操作环境都要不断地监视敲击键盘或点击鼠标这样的事件。操作环境将这些事件报告给正在运行的应用程序。如果有事件产生，每个应用程序将决定如何对它们做出响应。在Visual Basic这样的语言中，事件与代码之间有着明确的对应关系。程序员对相关的特定事件编写代码，并将这些代码放置在过程中，通常人们将它们称为事件过程（event procedure）。例如，有一个名为HelpButton的Visual Basic按钮有一个与之关联的HelpButton_Click事件过程。这个过程代码将在点击按钮后执行。每个Visual Basic的GUI组件都响应一个固定的事件集，不可能改变Visual Basic组件响应的事件集。

另一方面，如果使用像原始的C这样的语言进行事件驱动的程序设计，那就需要编写代码来不断地检查事件队列，以便查询操作环境报告的内容（通常这些代码被放置在包含很多switch语句的循环体中）。显然，这种方式编写的程序可读性很差，而且在有些情况下，编码的难度也非常大。它的好处在于响应的事件不受限制，而不像Visual Basic这样的语言，将事件队列对程序员隐藏起来。

Java程序设计环境折中了Visual Basic与原始C的事件处理方式，因此，它既有着强大的功能，又具有一定的复杂性。在AWT所知的事件范围内，完全可以控制事件从事件源（event source）例如，按钮或滚动条，到事件监听器（event listener）的传递过程，并将任何对象指派给事件监听器。不过事实上，应该选择一个能够便于响应事件的对象。这种事件委托模型（event delegation model）与Visual Basic那种预定义监听器模型比较起来更加灵活。

事件源有一些向其注册事件监听器的方法。当某个事件源产生事件时，事件源会向为事件注册的所有事件监听器对象发送一个通告。

像Java这样的面向对象语言，都将事件的相关信息封装在一个事件对象（event object）中。在Java中，所有的事件对象都最终派生于java.util.EventObject类。当然，每个事件类型还有子类，例如，ActionEvent和WindowEvent。

不同的事件源可以产生不同类别的事件。例如，按钮可以发送一个ActionEvent对象，而窗口可以发送WindowEvent对象。

综上所述，下面给出AWT事件处理机制的概要：

- 监听器对象是一个实现了特定监听器接口（listener interface）的类的实例。
- 事件源是一个能够注册监听器对象并发送事件对象的对象。
- 当事件发生时，事件源将事件对象传递给所有注册的监听器。
- 监听器对象将利用事件对象中的信息决定如何对事件做出响应。

图8-1显示了事件处理类和接口之间的关系。

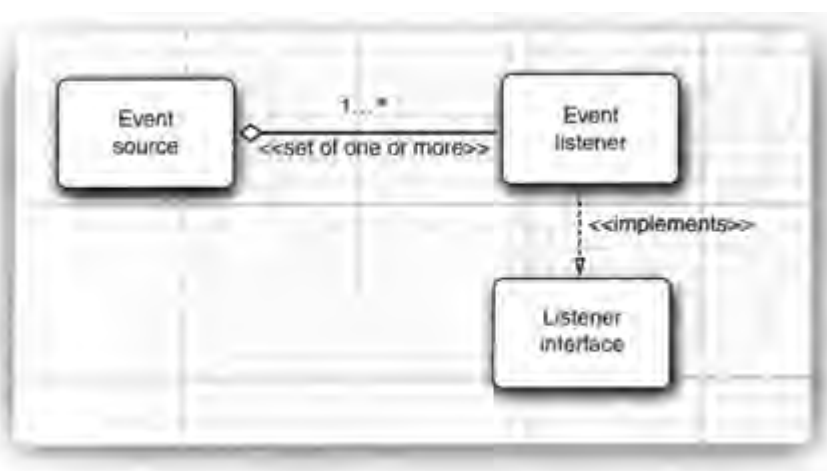


图8-1 事件源和监听器之间的关系

下面是监听器的一个示例：

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

现在，只要按钮产生了一个“动作事件”，listener对象就会得到通告。对于按钮来说，正像我们所想到的，动作事件就是点击按钮。

为了实现ActionListener接口，监听器类必须有一个被称为actionPerformed的方法，该方法接收一个ActionEvent对象参数。

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

只要用户点击按钮，JButton对象就会创建一个ActionEvent对象，然后调用listener.actionPerformed(event) 传递事件对象。可以将多个监听器对象添加到一个像按钮这样的事件源中。这样一来，只要用户点击按钮，按钮就会调用所有监听器的actionPerformed方法。

图8-2显示了事件源、事件监听器和事件对象之间的协作关系。

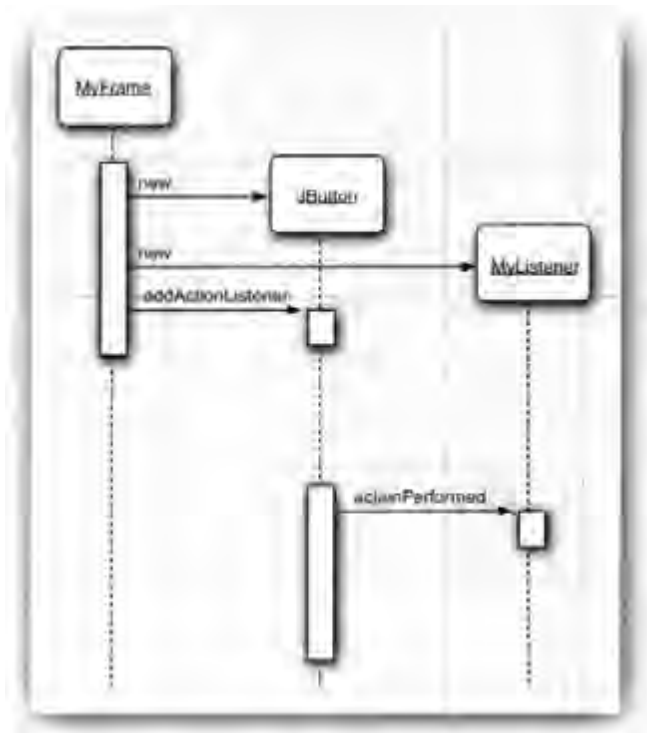


图8-2 事件通告

8.1.1 实例：处理按钮点击事件

为了加深对事件委托模型的理解，下面以一个响应按钮点击事件的简单示例来说明所需要知道的所有细节。在这个示例中，想要在一个面板中放置三个按钮，添加三个监听器对象来作为按钮的动作监听器。

在这个情况下，只要用户点击面板上的任何一个按钮，相关的监听器对象就会接收到一个 Action Event 对象，它表示有个按钮被点击了。在示例程序中，监听器对象将改变面板的背景颜色。

在演示如何监听按钮点击事件之前，首先需要讲解一下如何创建按钮以及如何将它们添加到面板中（有关GUI元素更加详细的内容请参看第9章）。

可以通过在按钮构造器中指定一个标签字符串、一个图标或两项都指定来创建一个按钮。下面是两个示例：

```
JButton yellowButton = new JButton("Yellow");
JButton blueButton = new JButton(new ImageIcon("blue-ball.gif"));
```

将按钮添加到面板中需要调用add方法：

```
JButton yellowButton = new JButton("Yellow");
JButton blueButton = new JButton("Blue");
JButton redButton = new JButton("Red");
```

```
buttonPanel.add(yellowButton);
buttonPanel.add(blueButton);
buttonPanel.add(redButton);
```

图8-3显示了结果。



图8-3 填充按钮的面板

至此，知道了如何将按钮添加到面板上，接下来需要增加让面板监听这些按钮的代码。这需要一个实现了ActionListener接口的类。如前所述，应该包含一个actionPerformed方法，其签名为：

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



注释：在按钮示例中，使用的ActionListener接口并不仅限于按钮点击事件。它可以应用于很多情况：

- 当采用鼠标双击的方式选择了列表框中的一个选项时；
- 当选择一个菜单项时；
- 当在文本域中敲击ENTER键时；
- 对于一个Timer组件来说，当到达指定的时间间隔时。

在本章和下一章中，读者将会看到更加详细的内容。

在各种情况下，使用ActionListener接口的方式都是一样的：actionPerformed方法（ActionListener中的唯一方法）将接收一个ActionEvent类型的对象作为参数。这个事件对象包含了事件发生时的相关信息。

当按钮被点击时，希望将面板的背景颜色设置为指定的颜色。这个颜色存储在监听器类中：

```
class ColorAction implements ActionListener
{
    public ColorAction(Color c)
    {
        backgroundColor = c;
    }

    public void actionPerformed(ActionEvent event)
    {
        // set panel background color
        . . .
    }

    private Color backgroundColor;
}
```

然后，为每种颜色构造一个对象，并将这些对象设置为按钮监听器。

```
ColorAction yellowAction = new ColorAction(Color.YELLOW);
ColorAction blueAction = new ColorAction(Color.BLUE);
ColorAction redAction = new ColorAction(Color.RED);

yellowButton.addActionListener(yellowAction);
blueButton.addActionListener(blueAction);
redButton.addActionListener(redAction);
```

例如，如果一个用户在标有“Yellow”的按钮上点击了一下，yellowAction对象的actionPerformed方法就会被调用。这个对象的backgroundColor实例域被设置为Color.YELLOW，现在就将面板的背景色设置为黄色了。

这里还有一个需要考虑的问题。ColorAction对象不能访问buttonpanel变量。可以采用两种方式解决这个问题。一个是将面板存储在ColorAction对象中，并在ColorAction的构造器中设置

它；另一个是将ColorAction作为ButtonFrame类的内部类，这样一来，它的方法就自动地拥有访问外部面板的权限了（有关内部类的详细介绍请参看第6章）。

这里使用第二种方法。下面说明一下如何将ColorAction类放置在ButtonFrame类内。

```
class ButtonPanel extends JFrame
{
    . . .

    private class ColorAction implements ActionListener
    {
        . . .

        public void actionPerformed(ActionEvent event)
        {
            buttonPanel.setBackground(background-color);
        }

        private Color background-color;
    }
    private JPanel buttonPanel;
}
```

下面仔细地研究一下actionPerformed方法。在ColorAction类中没有buttonPanel 域，但在外部ButtonFrame类中却有。

这种情形经常会遇到。事件监听器对象通常需要执行一些对其他对象可能产生影响的操作。可以策略性地将监听器类放置在需要修改状态的那个类中。

例8-1包含了完整的程序。无论何时点击任何一个按钮，对应的动作监听器就会修改面板的背景颜色。

例8-1 ButtonTest .java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.33 2007-06-12
7.  * @author Cay Horstmann
8.  */
9. public class ButtonTest
10. {
11.     public static void main(String[] args)
12.     {
13.         EventQueue.invokeLater(new Runnable()
14.         {
15.             public void run()
16.             {
17.                 ButtonFrame frame = new ButtonFrame();
18.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.                 frame.setVisible(true);
20.             }
21.         });
22.     }
```

```
23. }
24.
25. /**
26.  * A frame with a button panel
27.  */
28. class ButtonFrame extends JFrame
29. {
30.     public ButtonFrame()
31.     {
32.         setTitle("ButtonTest");
33.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.         // create buttons
36.         JButton yellowButton = new JButton("Yellow");
37.         JButton blueButton = new JButton("Blue");
38.         JButton redButton = new JButton("Red");
39.
40.         buttonPanel = new JPanel();
41.
42.         // add buttons to panel
43.         buttonPanel.add(yellowButton);
44.         buttonPanel.add(blueButton);
45.         buttonPanel.add(redButton);
46.
47.         // add panel to frame
48.         add(buttonPanel);
49.
50.         // create button actions
51.         ColorAction yellowAction = new ColorAction(Color.YELLOW);
52.         ColorAction blueAction = new ColorAction(Color.BLUE);
53.         ColorAction redAction = new ColorAction(Color.RED);
54.
55.         // associate actions with buttons
56.         yellowButton.addActionListener(yellowAction);
57.         blueButton.addActionListener(blueAction);
58.         redButton.addActionListener(redAction);
59.     }
60.
61.     /**
62.     * An action listener that sets the panel's background color.
63.     */
64.     private class ColorAction implements ActionListener
65.     {
66.         public ColorAction(Color c)
67.         {
68.             backgroundColor = c;
69.         }
70.
71.         public void actionPerformed(ActionEvent event)
72.         {
73.             buttonPanel.setBackground(backgroundColor);
74.         }
75.
76.         private Color backgroundColor;
77.     }
78.
79.     private JPanel buttonPanel;
```

```
80.  
81. public static final int DEFAULT_WIDTH = 300;  
82. public static final int DEFAULT_HEIGHT = 200;  
83. }
```

API javax.swing.JButton 1.2

- JButton(String label)
- JButton(Icon icon)
- JButton(String label, Icon icon)

构造一个按钮。标签可以是常规的文本，从Java SE 1.3开始，也可以是HTML。例如，“<html>Ok</html>”。

API java.awt.Container 1.0

- Component add(Component c)
将组件c添加到这个容器中。

API javax.swing.ImageIcon 1.2

- ImageIcon(String filename)
构造一个图标，它的图像存储在一个文件中。

8.1.2 建议使用内部类

有些人不喜欢使用内部类，其原因是觉得类和对象的增殖会使得程序的执行速度变慢。下面讨论一下这个问题。首先，不需要为每个用户界面组件定义一个新类。在前面列举的示例中，三个按钮共享同一个监听器类。当然，每个按钮分别使用不同的监听器对象。但是，这些对象并不大，它们只包含一个颜色值和一个面板的引用。而使用传统的if...else语句的解决方案也需要引用动作监听器存储的上述颜色对象，只不过这是一个局部变量，而不是实例域。

下面是一个说明使用匿名内部类简化代码的示例。如果仔细看一下例8-1的代码，就会注意到每个按钮的处理过程都是一样的：

- 1) 用标签字符串构造按钮。
- 2) 将按钮添加到面板上。
- 3) 用对应的颜色构造一个动作监听器。
- 4) 添加动作监听器。

为了简化这些操作过程，可以设计一个辅助方法：

```
public void makeButton(String name, Color backgroundColor)  
{  
    JButton button = new JButton(name);  
    buttonPanel.add(button);  
    ColorAction action = new ColorAction(backgroundColor);  
    button.addActionListener(action);  
}
```

然后简化调用

```
makeButton("yellow", Color.YELLOW);
makeButton("blue", Color.BLUE);
makeButton("red", Color.RED);
```

接着，进一步进行简化。请注意，ColorAction类只在makeButton方法中使用一次。因此，可以将它设计为一个匿名类：

```
public void makeButton(String name, final Color backgroundColor)
{
    JButton button = new JButton(name);
    buttonPanel.add(button);
    button.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            buttonPanel.setBackground(backgroundColor);
        }
    });
}
```

动作监听器代码现在变得更加简单了。actionPerformed方法仅仅引用参数变量backgroundColor（与内部类中访问的所有局部变量一样，应该将参数声明为final）。

这里不需要显式的构造器。在第6章中已经看到，内部类机制将自动地生成一个构造器，其中存储着所有用在内部类方法中的final局部变量。



提示：匿名内部类看起来可能让人感觉有些困惑。如果训练自己的眼睛能够捕获程序代码中的关键字，那就可以破解它们，例如：

```
button.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        buttonPanel.setBackground(backgroundColor);
    }
});
```

这就是说，按钮动作设置背景颜色。只要事件处理器包含的语句条数不多，就认为这段代码的可读性还是不错的，尤其是在对内部类机制没有什么抵触心理的情况下。



注释：任何实现了ActionListener接口的类对象都可以作为按钮监听器。我们更加倾向于为将要执行的按钮动作创建一个新类和类对象。然而，有些程序员不愿意使用内部类，却选择了不同的策略。他们找出因事件而改变的容器，让这些容器实现ActionListener接口。然后，将容器设置为监听器，如下所示：

```
yellowButton.addActionListener(this);
blueButton.addActionListener(this);
redButton.addActionListener(this);
```

注意，现在这里的三个按钮不再是独立的监听器。它们共享一个监听器对象，即按钮面板。因此，actionPerformed方法必须判断点击了哪个按钮。

```
class ButtonFrame extends JFrame implements ActionListener
```



```

{
    . . .
    public void actionPerformed(ActionEvent event)
    {
        Object source = event.getSource();
        if (source == yellowButton) . . .
        else if (source == blueButton) . . .
        else if (source == redButton) . . .
        else . . .
    }
}

```

可以看到，这样会变得有些零散杂乱，我们不建议这样做。

API java.util.EventObject 1.1

- Object getSource()
返回发生事件的对象引用。

API java.awt.event.ActionEvent 1.1

- String getActionCommand()
返回与这个动作事件相关的命令字符串。如果动作事件来源于按钮，命令字符串就等于按钮标签，除非已经使用setActionCommand方法对字符串进行了修改。

API java.beans.EventHandler 1.4

- static Object create(Class listenerInterface, Object target, String action)
 - static Object create(Class listenerInterface, Object target, String action, String eventProperty)
 - static Object create(Class listenerInterface, Object target, String action, String eventProperty, String listenerMethod)
- 构造实现给定接口的代理类对象。无论是命名方法，还是接口的所有方法都将执行目标对象上的给定动作。
- 动作可以是一个方法名或目标的属性。如果是属性，则执行它的设置方法。例如，动作text将变为调用setText方法。
- 事件属性由一个或多个用逗号分隔的属性名组成。第一个属性从监听器方法的参数中读出，第二个属性由结果对象读出，等等。最后的结果将作为动作的参数。例如，属性source.text将变为调用getSource和getText方法。

8.1.3 创建包含一个方法调用的监听器

在Java SE 1.4中引入了不用内部类来定义简单的事件监听器的机制。例如，假设有一个标签为Load的按钮，它的事件处理只包含下面一个方法调用：

```
frame.loadData();
```

当然，可以使用匿名内部类：

```
loadButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        frame.loadData();
    }
});
```

但是，EventHandler类可以使用下列调用，自动地创建这样一个监听器：

```
EventHandler.create(ActionListener.class, frame, "loadData")
```

当然，仍然需要安装处理器：

```
loadButton.addActionListener(
    EventHandler.create(ActionListener.class, frame, "loadData"));
```

如果事件监听器调用的方法只包含一个从事件处理器继承来的参数，就可以使用另外一种形式的create方法。例如，调用

```
EventHandler.create(ActionListener.class, frame, "loadData", "source.text")
```

等价于

```
new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        frame.loadData(((JTextField) event.getSource()).getText());
    }
}
```

需要将属性source和text的名字转换成方法调用getSource和getText。

8.1.4 实例：改变观感

在默认情况下，Swing程序使用Metal观感，可以采用两种方式改变观感。第一种方式是在Java安装的子目录jre/lib下有一个文件swing.properties。在这个文件中，将属性swing.defaultlaf设置为所希望的观感类名。例如，

```
swing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
```

注意，Metal观感位于javax.swing包中。其他的观感包位于com.sun.java包中，并且不是在每个Java实现中都提供。现在，鉴于版权的原因，Windows和Mac的观感包只与Windows和Mac版本的Java运行时环境一起发布。



提示：由于属性文件中以#字符开始的行被忽略，所以，可以在swing.properties文件中提供几种观感选择，并通过增删#字符来切换选择：

```
#swing.defaultlaf=javax.swing.plaf.metal.MetalLookAndFeel
swing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
#swing.defaultlaf=com.sun.java.swing.plaf.windows.WindowsLookAndFeel
```

采用这种方式开启观感时必须重新启动程序。Swing程序只在启动时读取一次swing.properties文件。

第二种方式是动态地改变观感。这需要调用静态的`UIManager.setLookAndFeel`方法，并提供所想要的观感类名，然后再调用静态方法`SwingUtilities.updateComponentTreeUI`来刷新全部的组件集。这里需要向这个方法提供一个组件，并由此找到其他的所有组件。当`UIManager.setLookAndFeel`方法没有找到所希望的观感或在加载过程中出现错误时，将会抛出异常。与前面一样，建议暂且将异常处理的代码跳过，等到第11章详细地讲述异常时就会理解了。

下面是一个示例，它显示了如何用程序切换至Motif观感：

```
String plaf = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";
try
{
    UIManager.setLookAndFeel(plaf);
    SwingUtilities.updateComponentTreeUI(panel);
}
catch(Exception e) { e.printStackTrace(); }
```

为了列举安装的所有观感实现，可以调用

```
UIManager.LookAndFeelInfo[] infos = UIManager.getInstalledLookAndFeels();
```

然后采用下列方式得到每一种观感的名字和类名

```
String name = infos[i].getName();
String className = infos[i].getClassName();
```

例8-2是一个完整的程序，它演示了如何切换观感（如图8-4所示）的方式。这个程序与例8-1十分相似。我们遵循前一节的建议，使用辅助方法`makeButton`和匿名内部类指定按钮动作，即切换观感。

在这个程序中，还有一点需要注意的地方。在内部动作监听器类的`actionPerformed`方法中，需要将一个外部`PlafFrame`类的`this`引用传递给`updateComponentTreeUI`方法。回想一下第6章所说过的，外部对象的`this`指针必须将外部类名作为前缀：

```
SwingUtilities.updateComponentTreeUI(PlafPanel.this);
```



图8-4 切换观感

例8-2 PlafTest.java

```
1. import java.awt.EventQueue;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.32 2007-06-12
7.  * @author Cay Horstmann
8.  */
9. public class PlafTest
10. {
11.     public static void main(String[] args)
12.     {
```

```
13.     EventQueue.invokeLater(new Runnable()
14.     {
15.         public void run()
16.         {
17.             PlafFrame frame = new PlafFrame();
18.             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.             frame.setVisible(true);
20.         }
21.     });
22. }
23. }
24.
25. /**
26.  * A frame with a button panel for changing look and feel
27.  */
28. class PlafFrame extends JFrame
29. {
30.     public PlafFrame()
31.     {
32.         setTitle("PlafTest");
33.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.         buttonPanel = new JPanel();
36.
37.         UIManager.LookAndFeelInfo[] infos = UIManager.getInstalledLookAndFeels();
38.         for (UIManager.LookAndFeelInfo info : infos)
39.             makeButton(info.getName(), info.getClassName());
40.
41.         add(buttonPanel);
42.     }
43.
44.     /**
45.      * Makes a button to change the pluggable look and feel.
46.      * @param name the button name
47.      * @param plafName the name of the look and feel class
48.      */
49.     void makeButton(String name, final String plafName)
50.     {
51.         // add button to panel
52.
53.         JButton button = new JButton(name);
54.         buttonPanel.add(button);
55.
56.         // set button action
57.
58.         button.addActionListener(new ActionListener()
59.         {
60.             public void actionPerformed(ActionEvent event)
61.             {
62.                 // button action: switch to the new look and feel
63.                 try
64.                 {
65.                     UIManager.setLookAndFeel(plafName);
66.                     SwingUtilities.updateComponentTreeUI(PlafFrame.this);
67.                 }
68.                 catch (Exception e)
69.                 {
```

```

70.         e.printStackTrace();
71.     }
72. }
73. });
74. }
75.
76. private JPanel buttonPanel;
77.
78. public static final int DEFAULT_WIDTH = 300;
79. public static final int DEFAULT_HEIGHT = 200;
80. }

```

javax.swing.UIManager 1.2

- static UIManager.LookAndFeelInfo[] getInstalledLookAndFeels()
获得一个用于描述已安装的观感实现的对象数组。
- static setLookAndFeel(String className)
利用给定的类名设置当前的观感。例如，javax.swing.plaf.metal.MetalLookAndFeel

javax.swing.UIManager.LookAndFeelInfo 1.2

- String getName()
返回观感的显示名称。
- String getClassName()
返回观感实现类的名称。

8.1.5 适配器类

并不是所有的事件处理都像按钮点击那样简单。在正规的程序中，往往希望用户在确认没有丢失所做工作之后再关闭程序。当用户关闭框架时，可能希望弹出一个对话框来警告用户没有保存的工作有可能会丢失，只有在用户确认之后才退出程序。

当程序用户试图关闭一个框架窗口时，JFrame对象就是WindowEvent的事件源。如果希望捕获这个事件，就必须有一个合适的监听器对象，并将它添加到框架的窗口监听器列表中。

```

WindowListener listener = ...;
frame.addWindowListener(listener);

```

窗口监听器必须是实现WindowListener接口的类的一个对象。在WindowListener接口中包含7个方法。当发生窗口事件时，框架将调用这些方法响应7个不同的事件。从它们的名字就可以得知其作用，惟一的例外是在Windows下，通常将iconified称为minimized。下面是完整的WindowListener接口：

```

public interface WindowListener
{
    void windowOpened(WindowEvent e);
    void windowClosing(WindowEvent e);
    void windowClosed(WindowEvent e);
    void windowIconified(WindowEvent e);
    void windowDeiconified(WindowEvent e);
    void windowActivated(WindowEvent e);
}

```

```
void windowDeactivated(WindowEvent e);  
}
```



注释：为了能够查看窗口是否被最大化，需要安装WindowStateListener。有关更加详细的信息请参看后面的API注释。

正像前面曾经说过的那样，在Java中，实现一个接口的任何类都必须实现其中的所有方法；在这里，意味着需要实现7个方法。然而只对名为windowClosing的方法感兴趣。

当然，可以这样定义实现这个接口的类：在windowClosing方法中增加一个对System.exit(0)的调用，其他6个方法不做任何事情：

```
class Terminator implements WindowListener  
{  
    public void windowClosing(WindowEvent e)  
    {  
        if (user agrees)  
            System.exit(0);  
    }  
  
    public void windowOpened(WindowEvent e) {}  
    public void windowClosed(WindowEvent e) {}  
    public void windowIconified(WindowEvent e) {}  
    public void windowDeiconified(WindowEvent e) {}  
    public void windowActivated(WindowEvent e) {}  
    public void windowDeactivated(WindowEvent e) {}  
}
```

书写6个没有任何操作的方法代码显然是一种乏味的工作。鉴于简化的目的，每个含有多个方法的AWT监听器接口都配有一个适配器（adapter）类，这个类实现了接口中的所有方法，但每个方法没有做任何事情。这意味着适配器类自动地满足了Java实现相关监听器接口的技术要求。可以通过扩展适配器类来指定对某些事件的响应动作，而不必实现接口中的每个方法（ActionListener这样的接口只有一个方法，因此没必要提供适配器类）。

下面使用窗口适配器。首先定义一个WindowAdapter类的扩展类，其中包含继承的6个没有做任何事情的方法和覆盖的方法windowClosing：

```
class Terminator extends WindowAdapter  
{  
    public void windowClosing(WindowEvent e)  
    {  
        if (user agrees)  
            System.exit(0);  
    }  
}
```

现在，可以将一个Terminator对象注册为事件监听器：

```
WindowListener listener = new Terminator();  
frame.addWindowListener(listener);
```

只要框架产生了窗口事件，就会通过调用7个方法之中的一个方法将事件传递给listener对象（如图8-5所示），其中6个方法没有做任何事情；windowClosing方法调用System.exit(0)终止应用程序的执行。

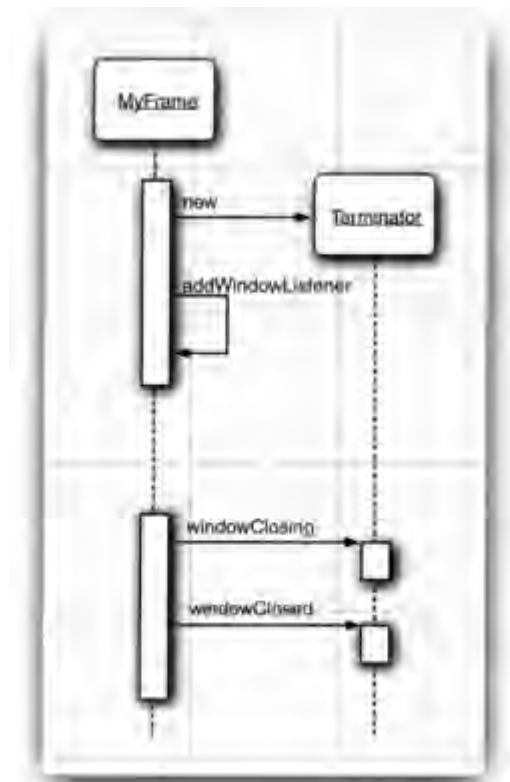


图8-5 窗口监听器

X 警告：如果在扩展适配器类时将方法名拼写错了，编译器不会捕获到这个错误。例如，如果在WindowAdapter类中定义一个windowIsClosing方法，就会得到一个包含8个方法的类，并且windowClosing方法没有做任何事情。

创建一个扩展于WindowAdapter的监听器类是一个很好的改进，但是还可以继续改进。事实上，没有必要为listener对象命名。只需写成：

```
frame.addWindowListener(new Terminator());
```

不要就此止步！我们可以将监听器类定义为框架的匿名内部类。

```
frame.addWindowListener(new
    WindowAdapter()
    {
        public void windowClosing(WindowEvent e)
        {
            if (user agrees)
                System.exit(0);
        }
    });
```

这段代码具有下列作用：

- 定义了一个扩展于WindowAdapter类的无名类。
- 将windowClosing方法添加到匿名类中（与前面一样，这个方法将退出程序）。
- 从WindowAdapter继承6个没有做任何事情的方法。

- 创建这个类的一个对象，这个对象没有名字。
- 将这个对象传递给addWindowListener方法。

这里再次说明一下，匿名内部类的语法需要人们适应一段时间，但得到的是更加简练的代码。

java.awt.event.WindowListener 1.1

- void windowOpened(WindowEvent e)
窗口打开后调用这个方法。
- void windowClosing(WindowEvent e)
在用户发出窗口管理器命令关闭窗口时调用这个方法。需要注意一点，仅当调用hide或dispose方法后窗口才能够关闭。
- void windowClosed(WindowEvent e)
窗口关闭后调用这个方法。
- void windowIconified(WindowEvent e)
窗口图标化后调用这个方法。
- void windowDeiconified(WindowEvent e)
窗口非图标化后调用这个方法。
- void windowActivated(WindowEvent e)
激活窗口后调用这个方法。只有框架或对话框可以被激活。通常，窗口管理器会对活动窗口进行修饰，比如，高亮度标题栏。
- void windowDeactivated(WindowEvent e)
窗口变为未激活状态后调用这个方法。

java.awt.event.WindowStateListener 1.4

- void windowStateChanged(WindowEvent event)
窗口被极大化、图标化或恢复为正常大小时调用这个方法。

java.awt.event.WindowEvent 1.1

- int getNewState() 1.4
 - int getOldState() 1.4
- 返回窗口状态改变事件中窗口的新、旧状态。返回的整型数值是下列数值之一：
- Frame.NORMAL
 - Frame.ICONIFIED
 - Frame.MAXIMIZED_HORIZ
 - Frame.MAXIMIZED_VERT
 - Frame.MAXIMIZED_BOTH

8.2 动作

通常，激活一个命令可以有多种方式。用户可以通过菜单、击键或工具栏上的按钮选择特定的功能。在AWT事件模型中实现这些非常容易：将所有事件连接到同一个监听器上。例如，假设blueAction是一个动作监听器，它的actionPerformed方法可以将背景颜色改变成蓝色。将一个监听器对象加到下面几个事件源上：

- 标记为Blue的工具栏按钮
- 标记为Blue的菜单项
- 击键CTRL+B

然后，无论改变背景颜色的命令是通过哪种方式下达的，是点击按钮、菜单选择，还是按下键盘，其操作动作都是一样的。

Swing包提供了一种非常实用的机制来封装命令，并将它们连接到多个事件源，这就是Action接口。一个动作是一个封装下列内容的对象：

- 命令的说明（一个文本字符串和一个可选图标）；
- 执行命令所需要的参数（例如，在列举的例子中请求改变的颜色）。

Action接口包含下列方法

```
void actionPerformed(ActionEvent event)
void setEnabled(boolean b)
boolean isEnabled()
void putValue(String key, Object value)
Object getValue(String key)
void addPropertyChangeListener(PropertyChangeListener listener)
void removePropertyChangeListener(PropertyChangeListener listener)
```

第一个方法是ActionListener接口中很熟悉的一个：实际上，Action接口扩展于ActionListener接口，因此，可以在任何需要ActionListener对象的地方使用Action对象。

接下来的两个方法允许启用或禁用这个动作，并检查这个动作当前是否启用。当一个连接到菜单或工具栏上的动作被禁用时，这个选项就会变成灰色。

putValue和getValue方法允许存储和检索动作对象中的任意名/值。有两个重要的预定义字符串：Action.NAME和Action.SMALL_ICON，用于将动作的名字和图标存储到一个动作对象中：

```
action.putValue(Action.NAME, "Blue");
action.putValue(Action.SMALL_ICON, new ImageIcon("blue-ball.gif"));
```

表8-1给出了所有预定义的动作表名称。

表8-1 预定义动作表名称

名 称	值
NAME	动作名称，显示在按钮和菜单上
SMALL_ICON	存储小图标的地方；显示在按钮、菜单项或工具栏中
SHORT_DESCRIPTION	图标的简要说明；显示在工具提示中
LONG_DESCRIPTION	图标的详细说明；使用在在线帮助中。没有Swing组件使用这个值
MNEMONIC_KEY	快捷键缩写；显示在菜单项中（请参看第9章）
ACCELERATOR_KEY	存储加速击键的地方；Swing组件不使用这个值
ACTION_COMMAND_KEY	历史遗留；仅在旧版本的registerKeyboardAction方法中使用
DEFAULT	常用的综合属性；Swing组件不使用这个值

如果动作对象添加到菜单或工具栏上，它的名称和图标就会被自动地提取出来，并显示在菜单项或工具栏项中。SHORT_DESCRIPTION值变成了工具提示。

Action接口的最后两个方法能够让其他对象在动作对象的属性发生变化时得到通告，尤其是菜单或工具栏触发的动作。例如，如果增加一个菜单，作为动作对象的属性变更监听器，而这个动作对象随后被禁用，菜单就会被调用，并将动作名称变为灰色。属性变更监听器是一种常用的构造形式，它是JavaBeans组件模型的一部分。有关这方面更加详细的信息请参看卷II。

需要注意，Action是一个接口，而不是一个类。实现这个接口的所有类都必须实现刚才讨论的7个方法。庆幸的是，有一个类实现了这个接口除actionPerformed方法之外的所有方法，它就是AbstractAction。这个类存储了所有名/值对，并管理着属性变更监听器。我们可以直接扩展AbstractAction类，并在扩展类中实现actionPerformed方法。

下面构造一个用于执行改变颜色命令的动作对象。首先存储这个命令的名称、图标和需要的颜色。将颜色存储在AbstractAction类提供的名/值对表中。下面是ColorAction类的代码。构造器设置名/值对，而actionPerformed方法执行改变颜色的动作。

```
public class ColorAction extends AbstractAction
{
    public ColorAction(String name, Icon icon, Color c)
    {
        putValue(Action.NAME, name);
        putValue(Action.SMALL_ICON, icon);
        putValue("color", c);
        putValue(Action.SHORT_DESCRIPTION, "Set panel color to " + name.toLowerCase());
    }

    public void actionPerformed(ActionEvent event)
    {
        Color c = (Color) getValue("color");
        buttonPanel.setBackground(c);
    }
}
```

在测试程序中，创建了这个类的三个对象，如下所示：

```
Action blueAction = new ColorAction("Blue", new ImageIcon("blue-ball.gif"), Color.BLUE);
```

接下来，将这个动作与一个按钮关联起来。由于JButton有一个用Action对象作为参数的构造器，所以实现这项操作很容易。

```
JButton blueButton = new JButton(blueAction);
```

构造器读取动作的名称和图标，为工具提示设置简要说明，将动作设置为监听器。从图8-6中可以看到图标和工具提示。在下一章中将会看到，将这个动作添加到菜单中也非常容易。

最后，想要将这个动作对象添加到击键中，以便让用户敲击键盘命令来执行这项动作。为了将动作与击键关联起来，首先需要生成KeyStroke类对象。这是一个很有用的类，它封装了对键的说明。要想生成一个KeyStroke对象，



图8-6 按钮显示来自动作对象的图标

不要调用构造器，而是调用KeyStroke类中的静态getKeyStroke方法：

```
KeyStroke ctrlBKey = KeyStroke.getKeyStroke("ctrl B");
```

为了能够理解下一个步骤，需要知道`keyboard focus`的概念。用户界面中可以包含许多按钮、菜单、滚动栏以及其他的组件。当用户敲击键盘时，这个动作会被发送给拥有焦点的组件。通常具有焦点的组件可以明显地察觉到（但并不总是这样），例如，在Java观感中，具有焦点的按钮在按钮文本周围有一个细的矩形边框。用户可以使用TAB键在组件之间移动焦点。当按下SPACE键时，就点击了拥有焦点的按钮。还有一些键执行一些其他的动作，例如，按下箭头键可以移动滚动条。

然而，在这里的示例中，并不希望将击键发送给拥有焦点的组件。否则，每个按钮都需要知道如何处理CTRL+Y、CTRL+B 和CTRL+R这些组合键。

这是一个常见的问题，Swing设计者给出了一种很便捷的解决方案。每个JComponent有三个输入映射（input maps），每一个映射的KeyStroke对象都与动作关联。三个输入映射对应着三个不同的条件（请参看表8-2）。

表8-2 输入映射条件

标 志	激 活 动 作
WHEN_FOCUSED	当这个组件拥有键盘焦点时
WHEN_ANCESTOR_OF_FOCUSED_COMPONENT	当这个组件包含了拥有键盘焦点的组件时
WHEN_IN_FOCUSED_WINDOW	当这个组件被包含在一个拥有键盘焦点组件的窗口中时

击键处理将按照下列顺序检查这些映射：

1) 检查具有输入焦点组件的WHEN_FOCUSED映射。如果这个击键存在，将执行对应的动作。如果动作已启用，则停止处理。

2) 从具有输入焦点的组件开始，检查其父组件的WHEN_ANCESTOR_OF_FOCUSED_COMPONENT映射。一旦找到击键对应的映射，就执行对应的动作。如果动作已启用，将停止处理。

3) 查看具有输入焦点的窗口中的所有可视的和启用的组件，这个击键被注册到WHEN_IN_FOCUSED_WINDOW映射中。给这些组件（按照击键注册的顺序）一个执行对应动作的机会。一旦第一个启用的动作被执行，就停止处理。如果一个击键在多个WHEN_IN_FOCUSED_WINDOW映射中出现，这部分处理就可能会出现问題。

可以使用getInputMap方法从组件中得到输入映射。例如：

```
InputMap imap = panel.getInputMap(JComponent.WHEN_FOCUSED);
```

WHEN_FOCUSED条件意味着在当前组件拥有键盘焦点时会查看这个映射。在这里的示例中，并不想使用这个映射。是某个按钮拥有输入焦点，而不是面板。其他的两个映射都能够很好地完成增加颜色改变击键的任务。在示例程序中使用的是WHEN_ANCESTOR_OF_FOCUSED_COMPONENT。

InputMap不能直接地将KeyStroke对象映射到Action对象。而是先映射到任意对象上，然后由ActionMap类实现将对象映射到动作上的第2个映射。这样很容易实现来自不同输入映射的击

键共享一个动作的目的。

因而，每个组件都可以有三个输入映射和一个动作映射。为了将它们组合起来，需要为动作命名。下面是将键与动作关联起来的方式：

```
imap.put(KeyStroke.getKeyStroke("ctrl Y"), "panel.yellow");
ActionMap amap = panel.getActionMap();
amap.put("panel.yellow", yellowAction);
```

习惯上，使用字符串none表示空动作。这样可以轻松地取消一个键动作：

```
imap.put(KeyStroke.getKeyStroke("ctrl C"), "none");
```



警告：JDK文档提倡使用动作名作为动作键。我们并不认为这是一个好建议。在按钮和菜单项上显示的动作名，UI设计者可以随心所欲地进行更改，也可以将其翻译成多种语言。使用这种不牢靠的字符串作为查询键不是一种好的选择。建议将动作名与显示的名字分开。

下面总结一下用同一个动作响应按钮、菜单项或击键的方式：

- 1) 实现一个扩展于AbstractAction类的类。多个相关的动作可以使用同一个类。
- 2) 构造一个动作类的对象。
- 3) 使用动作对象创建按钮或菜单项。构造器将从动作对象中读取标签文本和图标。
- 4) 为了能够通过击键触发动作，必须额外地执行几步操作。首先定位顶层窗口组件，例如，包含所有其他组件的面板。
- 5) 然后，得到顶层组件的WHEN_ANCESTOR_OF_FOCUS_COMPONENT输入映射。为需要的击键创建一个KeyStroke对象。创建一个描述动作字符串这样的动作键对象。将（击键，动作键）对添加到输入映射中。
- 6) 最后，得到顶层组件的动作映射。将（动作键，动作对象）添加到映射中。

例8-3给出了将按钮和击键映射到动作对象的完整程序代码。试试看，点击按钮或按下CTRL+Y、CTRL+B或CTRL+R来改变面板颜色。

例8-3 ActionTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.33 2007-06-12
7.  * @author Cay Horstmann
8.  */
9. public class ActionTest
10. {
11.     public static void main(String[] args)
12.     {
13.         EventQueue.invokeLater(new Runnable()
14.         {
15.             public void run()
16.             {
```

```

17.         ActionFrame frame = new ActionFrame();
18.         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.         frame.setVisible(true);
20.     }
21. });
22. }
23. }
24.
25. /**
26.  * A frame with a panel that demonstrates color change actions.
27.  */
28. class ActionFrame extends JFrame
29. {
30.     public ActionFrame()
31.     {
32.         setTitle("ActionTest");
33.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.         buttonPanel = new JPanel();
36.
37.         // define actions
38.         Action yellowAction = new ColorAction("Yellow", new ImageIcon("yellow-ball.gif"),
39.             Color.YELLOW);
40.         Action blueAction = new ColorAction("Blue", new ImageIcon("blue-ball.gif"), Color.BLUE);
41.         Action redAction = new ColorAction("Red", new ImageIcon("red-ball.gif"), Color.RED);
42.
43.         // add buttons for these actions
44.         buttonPanel.add(new JButton(yellowAction));
45.         buttonPanel.add(new JButton(blueAction));
46.         buttonPanel.add(new JButton(redAction));
47.
48.         // add panel to frame
49.         add(buttonPanel);
50.
51.         // associate the Y, B, and R keys with names
52.         InputMap imap = buttonPanel.getInputMap(JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT);
53.         imap.put(KeyStroke.getKeyStroke("ctrl Y"), "panel.yellow");
54.         imap.put(KeyStroke.getKeyStroke("ctrl B"), "panel.blue");
55.         imap.put(KeyStroke.getKeyStroke("ctrl R"), "panel.red");
56.
57.         // associate the names with actions
58.         ActionMap amap = buttonPanel.getActionMap();
59.         amap.put("panel.yellow", yellowAction);
60.         amap.put("panel.blue", blueAction);
61.         amap.put("panel.red", redAction);
62.     }
63.
64.     public class ColorAction extends AbstractAction
65.     {
66.         /**
67.          * Constructs a color action.
68.          * @param name the name to show on the button
69.          * @param icon the icon to display on the button
70.          * @param c the background color
71.          */
72.         public ColorAction(String name, Icon icon, Color c)
73.         {

```

```
74.         putValue(Action.NAME, name);
75.         putValue(Action.SMALL_ICON, icon);
76.         putValue(Action.SHORT_DESCRIPTION, "Set panel color to " + name.toLowerCase());
77.         putValue("color", c);
78.     }
79.
80.     public void actionPerformed(ActionEvent event)
81.     {
82.         Color c = (Color) getValue("color");
83.         buttonPanel.setBackground(c);
84.     }
85. }
86.
87. private JPanel buttonPanel;
88.
89. public static final int DEFAULT_WIDTH = 300;
90. public static final int DEFAULT_HEIGHT = 200;
91. }
```

API javax.swing.Action 1.2

- boolean isEnabled()
返回这个动作的enabled属性。
- void setEnabled(boolean b)
获得或设置这个动作的enabled属性。
- void putValue(String key, Object value)
将名/值对放置在动作对象内。
参数：key 用动作对象存储性能的名字。它可以是一个字符串，但预定义了几个名字，其含义参看表8-1。
 value 与名字关联的对象。
- Object getValue(String key)
返回被存储的名/值对的值。

API javax.swing.KeyStroke 1.2

- static KeyStroke getKeyStroke(String description)
根据一个便于人们阅读的说明创建一个击键(由空格分隔的字符串序列)。这个说明以0个或多个修饰符 shift control ctrl meta alt altGraph开始，以由typed和单个字符构成的字符串(例如：“typed a”)或者一个可选的事件说明符(pressed默认，或released)紧跟一个键码结束。以VK_前缀开始的键码应该对应一个KeyEvent常量，例如，“INSERT”对应KeyEvent.VK_INSERT。

API javax.swing.JComponent 1.2

- ActionMap getActionMap() 1.3
返回关联动作映射键(可以是任意的对象)和动作对象的映射。
- InputMap getInputMap(int flag) 1.3

获得将击键映射到动作键的输入映射。

参数：flag 触发动作的键盘焦点条件。具体的值请参看表8-2。

8.3 鼠标事件

如果只希望用户能够点击按钮或菜单，那么就不需要显式地处理鼠标事件。鼠标操作将由用户界面中的各种组件内部处理。然而，如果希望用户使用鼠标画图，就需要捕获鼠标移动点击和拖动事件。

在本节中，将展示一个简单的图形编辑器应用程序，它允许用户在画布上（如图8-7所示）放置、移动和擦除方块。

当用户点击鼠标按钮时，将会调用三个监听器方法：鼠标第一次被按下时调用mousePressed；鼠标被释放时调用mouseReleased；最后调用mouseClicked。如果只对最终的点击事件感兴趣，就可以忽略前两个方法。用MouseEvent类对象作为参数，调用getX和getY方法可以获得鼠标被按下时鼠标指针所在的x和y坐标。要想区分单击、双击和三击(!)，需要使用getClickCount方法。



图8-7 鼠标测试程序

有些用户界面设计者喜欢让用户采用鼠标点击与键盘修饰符组合（例如，CONTROL+SHIFT+CLICK）的方式进行操作。我们感觉这并不是一种值得赞许的方式。如果对此持有不同的观点，可以看一看同时检测鼠标按键和键盘修饰符所带来的混乱。

可以采用位掩码来测试已经设置了哪个修饰符。在最初的API中，有两个按钮的掩码与两个键盘修饰符的掩码一样，即

```
BUTTON2_MASK == ALT_MASK
BUTTON3_MASK == META_MASK
```

这样做是为了能够让用户使用仅有一个按钮的鼠标通过按下修饰键来模拟按下其他鼠标键的操作。然而，在Java SE 1.4中，建议使用一种不同的方式。有下列掩码：

```
BUTTON1_DOWN_MASK
BUTTON2_DOWN_MASK
BUTTON3_DOWN_MASK
SHIFT_DOWN_MASK
CTRL_DOWN_MASK
ALT_DOWN_MASK
ALT_GRAPH_DOWN_MASK
META_DOWN_MASK
```

getModifiersEx方法能够准确地报告鼠标事件的鼠标按钮和键盘修饰符。

需要注意，在Windows环境下，使用BUTTON3_DOWN_MASK检测鼠标右键（非主要的）的状态。例如，可以使用下列代码检测鼠标右键是否被按下：

```
if ((event.getModifiersEx() & InputEvent.BUTTON3_DOWN_MASK) != 0)
    ... // code for right click
```

在列举的简单示例中，提供了mousePressed和mouseClicked方法。当鼠标点击在所有小方块的像素之外时，就会绘制一个新的小方块。这个操作是在mousePressed方法中实现的，这样可以让用户的操作立即得到响应，而不必等到释放鼠标按键。如果用户在某个小方块中双击鼠

标，就会将它擦除。由于需要知道点击次数，所以这个操作将在mouseClick方法中实现。















```
public void mousePressed(MouseEvent event)
{
    current = find(event.getPoint());
    if (current == null) // not inside a square
        add(event.getPoint());
}

public void mouseClicked(MouseEvent event)
{
    current = find(event.getPoint());
    if (current != null && event.getClickCount() >= 2)
        remove(current);
}
```

当鼠标在窗口上移动时，窗口将会收到一连串的鼠标移动事件。请注意：有两个独立的接口MouseListener和MouseMotionListener。这样做有利于提高效率。当用户移动鼠标时，只关心鼠标点击（*clicks*）的监听器就不会被多余的鼠标移动（*moves*）所困扰。

这里给出的测试程序将捕获鼠标动作事件，以便在光标位于一个小方块之上时变成另外一种形状（十字）。实现这项操作需要使用Cursor类中的getPredefinedCursor方法。表8-3列出了在Windows环境下，鼠标的形状和方法对应的常量（注意，有若干个光标的形状完全一样，但在其他平台上未必如此）。

表8-3 光标形状样例

图 标	常 量	图 标	常 量
	DEFAULT_CURSOR		NE_RESIZE_CURSOR
	CROSSHAIR_CURSOR		E_RESIZE_CURSOR
	HAND_CURSOR		SE_RESIZE_CURSOR
	MOVE_CURSOR		S_RESIZE_CURSOR
	TEXT_CURSOR		SW_RESIZE_CURSOR
	WAIT_CURSOR		W_RESIZE_CURSOR
	N_RESIZE_CURSOR		NW_RESIZE_CURSOR

下面是示例程序中MouseMotionListener类的mouseMoved方法：

```
public void mouseMoved(MouseEvent event)
{
    if (find(event.getPoint()) == null)
        setCursor(Cursor.getDefaultCursor());
    else
        setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
}
```




注释：可以利用Toolkit类中的createCustomCursor方法自定义光标类型：

```
Toolkit tk = Toolkit.getDefaultToolkit();
Image img = tk.getImage("dynamite.gif");
Cursor dynamiteCursor = tk.createCustomCursor(img, new Point(10, 10), "dynamite stick");
```

createCustomCursor的第一个参数指向光标图像。第二个参数给出了光标的“热点”偏移。第三个参数是一个描述光标的字符串。这个字符串可以用于访问性支持，例如，可以将光标形式读给视力受损或没有在屏幕前面的人。

如果用户在移动鼠标的同时按下鼠标，就会调用mouseMoved而不是调用mouseDragged。在测试应用程序中，用户可以用光标拖动小方块。在程序中，仅仅用拖动的矩形更新当前光标位置。然后，重新绘制画布，以显示新的鼠标位置。

```
public void mouseDragged(MouseEvent event)
{
    if (current != null)
    {
        int x = event.getX();
        int y = event.getY();

        current.setFrame(x - SIDELENGTH / 2, y - SIDELENGTH / 2, SIDELENGTH, SIDELENGTH);
        repaint();
    }
}
```



注释：只有鼠标在一个组件内部停留才会调用mouseMoved方法。然而，即使鼠标拖动到组件外面，mouseDragged方法也会被调用。

还有两个鼠标事件方法：mouseEntered和mouseExited。这两个方法是在鼠标进入或移出组件时被调用。

最后，解释一下如何监听鼠标事件。鼠标点击由mouseClicked过程报告，它是MouseListener接口的一部分。由于大部分应用程序只对鼠标点击感兴趣，而对鼠标移动并不感兴趣，但鼠标移动事件发生的频率又很高，因此将鼠标移动事件与拖动事件定义在一个称为MouseMotionListener的独立接口中。

在示例程序中，对两种鼠标事件类型都感兴趣。这里定义了两个内部类：MouseHandler和MouseMotionHandler。MouseHandler类扩展于MouseAdapter类，这是因为它只定义了5个MouseListener方法中的两个方法。MouseMotionHandler实现了MouseMotionListener接口，并定义了这个接口中的两个方法。例8-4是这个程序的清单：

例8-4 MouseTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.util.*;
4. import java.awt.geom.*;
5. import javax.swing.*;
6.
7. /**
8.  * @version 1.32 2007-06-12
```

```
9.  * @author Cay Horstmann
10. */
11. public class MouseTest
12. {
13.     public static void main(String[] args)
14.     {
15.         EventQueue.invokeLater(new Runnable()
16.         {
17.             public void run()
18.             {
19.                 MouseFrame frame = new MouseFrame();
20.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.                 frame.setVisible(true);
22.             }
23.         });
24.     }
25. }
26.
27. /**
28.  * A frame containing a panel for testing mouse operations
29.  */
30. class MouseFrame extends JFrame
31. {
32.     public MouseFrame()
33.     {
34.         setTitle("MouseTest");
35.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
36.
37.         // add component to frame
38.
39.         MouseComponent component = new MouseComponent();
40.         add(component);
41.     }
42.
43.     public static final int DEFAULT_WIDTH = 300;
44.     public static final int DEFAULT_HEIGHT = 200;
45. }
46.
47. /**
48.  * A component with mouse operations for adding and removing squares.
49.  */
50. class MouseComponent extends JComponent
51. {
52.     public MouseComponent()
53.     {
54.         squares = new ArrayList<Rectangle2D>();
55.         current = null;
56.
57.         addMouseListener(new MouseHandler());
58.         addMouseMotionListener(new MouseMotionHandler());
59.     }
60.
61.     public void paintComponent(Graphics g)
62.     {
63.         Graphics2D g2 = (Graphics2D) g;
64.
65.         // draw all squares
```

```

66.     for (Rectangle2D r : squares)
67.         g2.draw(r);
68.     }
69.
70.     /**
71.      * Finds the first square containing a point.
72.      * @param p a point
73.      * @return the first square that contains p
74.      */
75.     public Rectangle2D find(Point2D p)
76.     {
77.         for (Rectangle2D r : squares)
78.         {
79.             if (r.contains(p)) return r;
80.         }
81.         return null;
82.     }
83.
84.     /**
85.      * Adds a square to the collection.
86.      * @param p the center of the square
87.      */
88.     public void add(Point2D p)
89.     {
90.         double x = p.getX();
91.         double y = p.getY();
92.
93.         current = new Rectangle2D.Double(x - SIDELENGTH / 2, y - SIDELENGTH / 2, SIDELENGTH,
94.             SIDELENGTH);
95.         squares.add(current);
96.         repaint();
97.     }
98.
99.     /**
100.      * Removes a square from the collection.
101.      * @param s the square to remove
102.      */
103.     public void remove(Rectangle2D s)
104.     {
105.         if (s == null) return;
106.         if (s == current) current = null;
107.         squares.remove(s);
108.         repaint();
109.     }
110.
111.     private static final int SIDELENGTH = 10;
112.     private ArrayList<Rectangle2D> squares;
113.     private Rectangle2D current;
114.
115.     // the square containing the mouse cursor
116.
117.     private class MouseHandler extends MouseAdapter
118.     {
119.         public void mousePressed(MouseEvent event)
120.         {
121.             // add a new square if the cursor isn't inside a square
122.             current = find(event.getPoint());

```

```
123.     if (current == null) add(event.getPoint());
124.     }
125.
126.     public void mouseClicked(MouseEvent event)
127.     {
128.         // remove the current square if double clicked
129.         current = find(event.getPoint());
130.         if (current != null && event.getClickCount() >= 2) remove(current);
131.     }
132. }
133.
134. private class MouseMotionHandler implements MouseMotionListener
135. {
136.     public void mouseMoved(MouseEvent event)
137.     {
138.         // set the mouse cursor to cross hairs if it is inside
139.         // a rectangle
140.
141.         if (find(event.getPoint()) == null) setCursor(Cursor.getDefaultCursor());
142.         else setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
143.     }
144.
145.     public void mouseDragged(MouseEvent event)
146.     {
147.         if (current != null)
148.         {
149.             int x = event.getX();
150.             int y = event.getY();
151.
152.             // drag the current rectangle to center it at (x, y)
153.             current.setFrame(x - SIDELENGTH / 2, y - SIDELENGTH / 2, SIDELENGTH, SIDELENGTH);
154.             repaint();
155.         }
156.     }
157. }
158. }
```

java.awt.event.MouseEvent 1.1

- int getX()
返回事件发生时，事件源组件左上角的坐标x（水平）和y（竖直），或点信息。
- int getY()
- Point getPoint()
返回事件发生时，事件源组件左上角的坐标x（水平）和y（竖直），或点信息。
- int getClickCount()
返回与事件关联的鼠标连击次数（“连击”所指定的时间间隔与具体系统有关）。

java.awt.event.InputEvent 1.1

- int getModifiersEx() 1.4
返回事件扩展的或“按下”（down）的修饰符。使用下面的掩码值检测返回值：
BUTTON1_DOWN_MASK

```

BUTTON2_DOWN_MASK
BUTTON3_DOWN_MASK
SHIFT_DOWN_MASK
CTRL_DOWN_MASK
ALT_DOWN_MASK
ALT_GRAPH_DOWN_MASK
META_DOWN_MASK

```

- `static String getModifiersExText(int modifiers)` 1.4
返回用给定标志集描述的扩展或“按下”(down)的修饰符字符串,例如“Shift+Button1”。

API java.awt.Toolkit 1.0

- `public Cursor createCustomCursor(Image image, Point hotSpot, String name)` 1.2
创建一个新的定制光标对象。
参数: image 光标活动时显示的图像
 hotSpot 光标热点(箭头的顶点或十字中心)
 name 光标的描述,用来支持特殊的访问环境

API java.awt.Component 1.0

- `public void setCursor(Cursor cursor)` 1.1
用光标图像设置给定光标。

8.4 AWT事件继承层次

弄清了事件处理的工作过程之后,作为本章的结束,总结一下AWT事件处理的体系架构。

前面已经提到,Java事件处理采用的是面向对象方法,所有的事件都是由java.util包中的EventObject类扩展而来的(公共超类不是Event,它是旧事件模型中的事件类名。尽管现在不赞成使用旧的事件模型,但这些类仍然保留在Java库中)。

EventObject类有一个子类AWTEvent,它是所有AWT事件类的父类。图8-8显示了AWT事件的继承关系图。

有些Swing组件将生成其他事件类型的事件对象;它们都直接扩展于EventObject,而不是AWTEvent。

事件对象封装了事件源与监听器彼此通信的事件信息。在必要的时候,可以对传递给监听器对象的事件对象进行分析。在按钮例子中,是借助getSource和getActionCommand方法实现对象分析的。

对于有些AWT事件类来说,Java程序员并不会实际地使用它们。例如,AWT将会把PaintEvent对象插入事件队列中,但这些对象并没有传递给监听器。Java程序员并不监听绘图事件,如果希望控制重新绘图操作,就需要覆盖paintComponent方法。另外,AWT还可以生成许多只对系统程序员有用的事件,用于提供表义语言的输入系统以及自动检测机器人等等。在此,将不讨论这些事件类型。

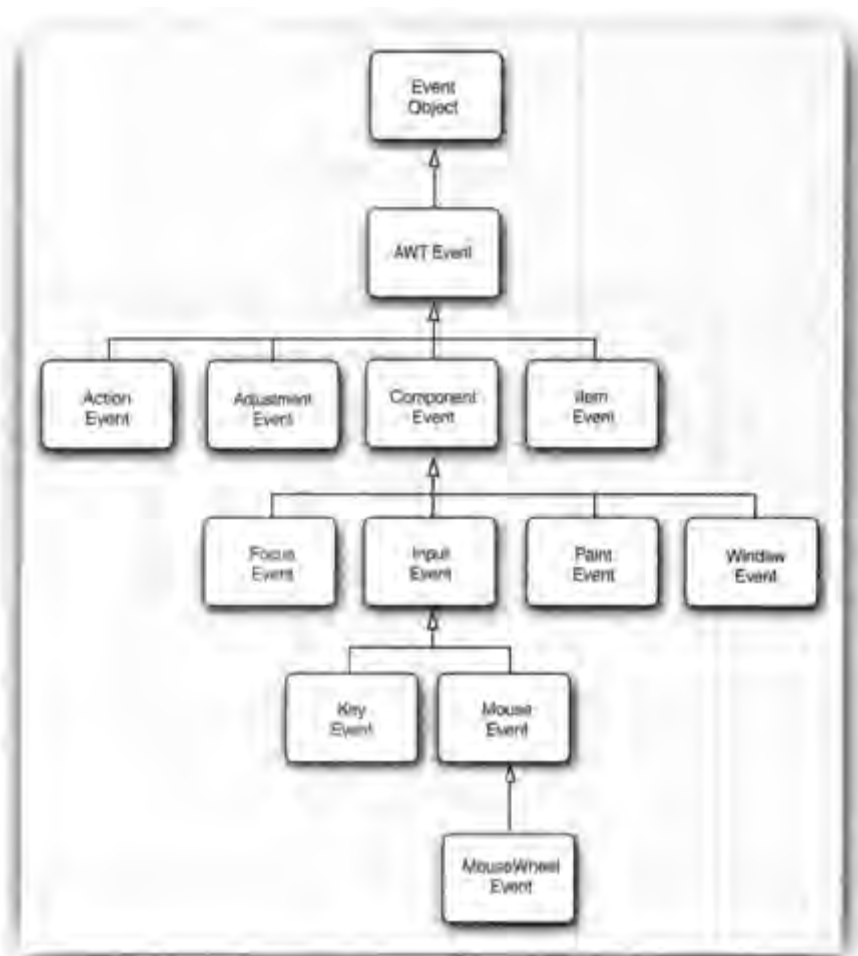


图8-8 AWT事件类的继承关系图

AWT的语义事件和低级事件

AWT将事件分为低级（low-level）事件和语义（semantic）事件。语义事件是表示用户动作的事件，例如，点击按钮；因此，ActionEvent是一种语义事件。低级事件是形成那些事件的事件。在点击按钮时，包含了按下鼠标、连续移动鼠标、抬起鼠标（只有鼠标在按钮区中抬起才引发）事件。或者在用户利用TAB键选择按钮，并利用空格键激活它时，发生的敲击键盘事件。同样，调节滚动条是一种语义事件，但拖动鼠标是低级事件。

下面是java.awt.event包中最常用的语义事件类：

- ActionEvent（对应按钮点击、菜单选择、选择列表项或在文本框中ENTER）；
- AdjustmentEvent（用户调节滚动条）；
- ItemEvent（用户从复选框或列表框中选择一项）。

常用的5个低级事件类是：

- KeyEvent（一个键被按下或释放）；
- MouseEvent（鼠标键被按下、释放、移动或拖动）；
- MouseWheelEvent（鼠标滚轮被转动）；

- FocusEvent (某个组件获得焦点或失去焦点);
- WindowEvent (窗口状态被改变)。

下列接口将监听这些事件。

ActionListener	MouseMotionListener
AdjustmentListener	MouseWheelListener
FocusListener	WindowListener
ItemListener	WindowFocusListener
KeyListener	WindowStateListener
MouseListener	

有几个AWT监听器接口包含多个方法，它们都配有一个适配器类，在这个类中实现了相应接口中的所有方法，但每个方法没有做任何事情。（有些接口只包含一个方法，因此，就没有必要为它们定义适配器类了）下面是常用的适配器类：

FocusAdapter	MouseMotionAdapter
KeyAdapter	WindowAdapter
MouseAdapter	

表8-4显示了最重要的AWT监听器接口、事件和事件源。

表8-4 事件处理总结

接 口	方 法	参数/访问方法	事 件 源
ActionListener	actionPerformed	ActionEvent • getActionCommand • getModifiers	AbstractButton JComboBox JTextField Timer
AdjustmentListener	adjustmentValueChanged	AdjustmentEvent • getAdjustable • getAdjustmentType • getValue	JScrollbar
ItemListener	itemStateChanged	ItemEvent • getItem • getItemSelectable • getStateChange	AbstractButton JComboBox
FocusListener	focusGained focusLost	FocusEvent • isTemporary	Component
KeyListener	keyPressed keyReleased keyTyped	KeyEvent • getKeyChar • getKeyCode • getKeyModifiersText • getKeyText • isActionKey	Component
MouseListener	mousePressed mouseReleased mouseEntered	MouseEvent • getClickCount • getX	Component

(续)

接 口	方 法	参数/访问方法	事 件 源
	mouseExited mouseClicked	• getY • getPoint • translatePoint	
MouseMotionListener	mouseDragged mouseMoved	MouseEvent	Component
MouseWheelListener	mouseWheelMoved	MouseWheelEvent • getWheelRotation • getScrollAmount	Component
WindowListener	windowClosing windowOpened windowIconified windowDeiconified windowClosed windowActivated windowDeactivated	WindowEvent • getWindow	Window
WindowFocusListener	windowGainedFocus windowLostFocus	WindowEvent • getOppositeWindow	Window
WindowStateListener	windowStateChanged	WindowEvent • getOldState • getNewState	Window

javax.swing.event包中包含了许多专门用于Swing组件的附加事件，下一章中将介绍其中的一部分。

AWT事件处理的讨论到此结束。在下一章中，读者将学习Swing提供的更多的常用组件，并详细地介绍它们所产生的事件。