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按键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。这个过程中的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。

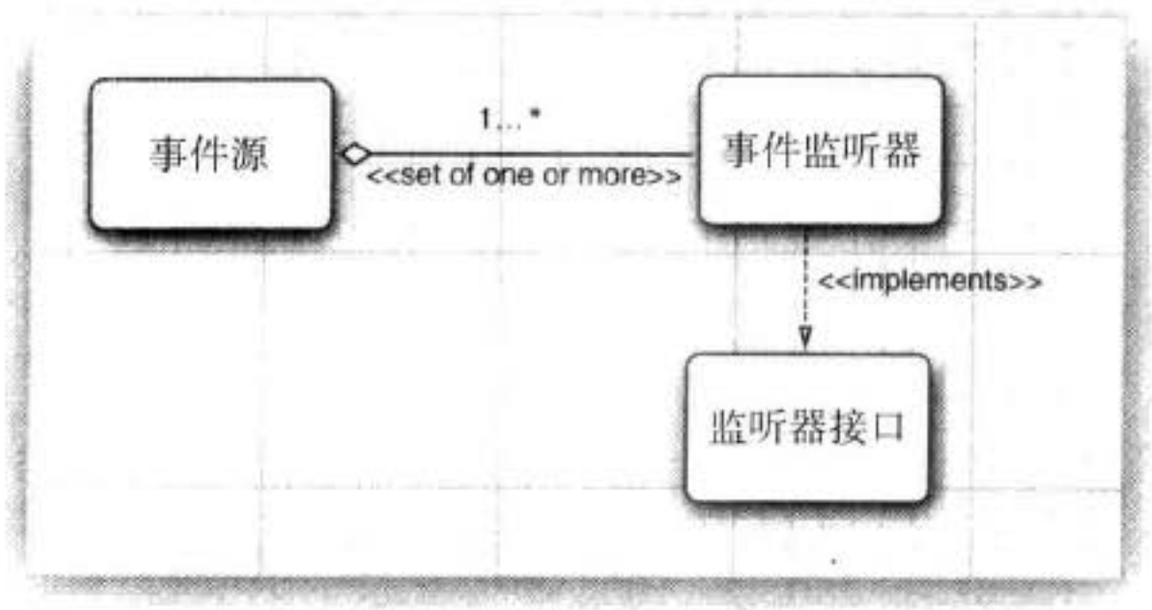


图 8-1 事件源和监听器之间的关系

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.action Performed (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。

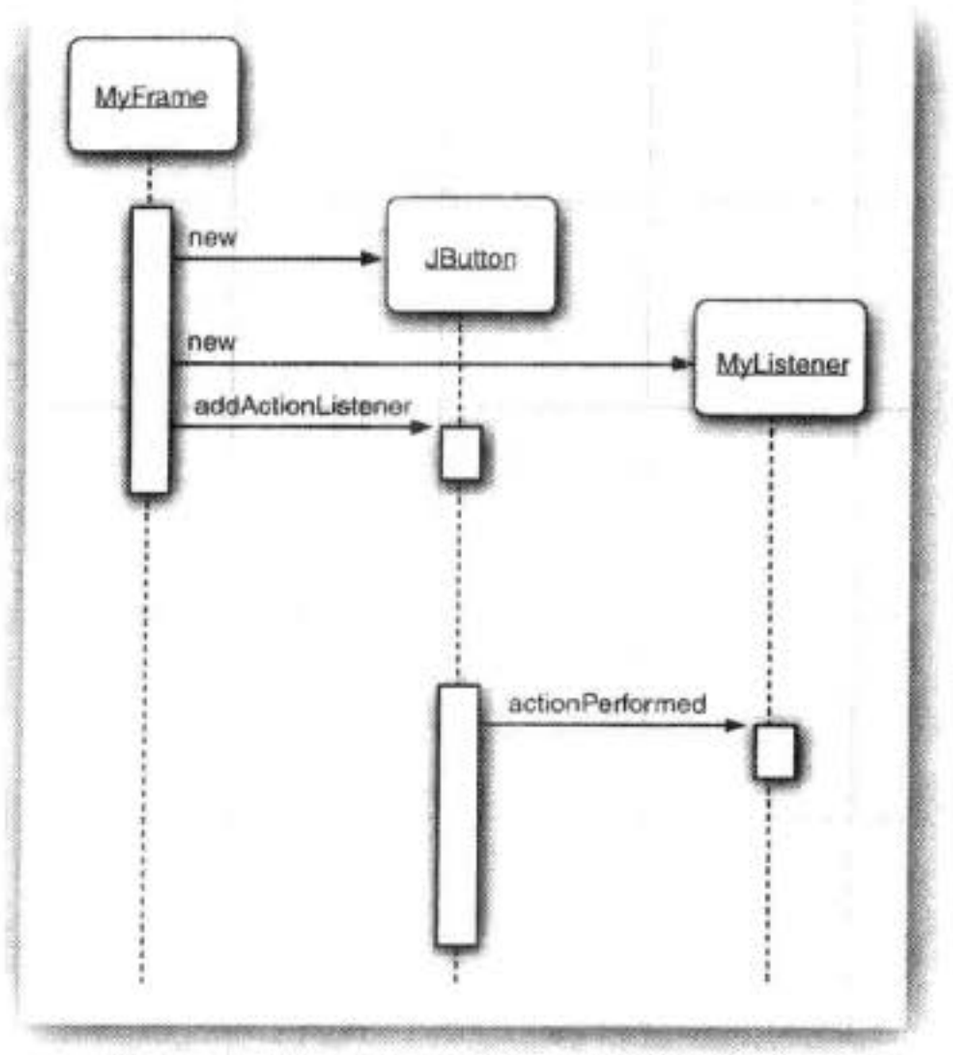


图 8-2 事件通知

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用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

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；
- 当在文本域中按回车键时；
- 对于一个 Timer 组件来说，当达到指定的时间间隔时。

在本章和下一章中，读者将会看到更加详细的内容。

在所有这些情况下，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
{
   private Color backgroundColor;

   public ColorAction(Color c)
   {
      backgroundColor = c;
   }

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

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 ButtonFrame extends JFrame
{
   private JPanel buttonPanel;
   . . .
   private class ColorAction implements ActionListener
   {
      private Color backgroundColor;
      . . .
      public void actionPerformed(ActionEvent event)
      {
         buttonPanel.setBackground(backgroundColor);
      }
   }
}
```

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。

程序清单 8-1 包含了完整的框架类。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。

程序清单 8-1 button/ButtonFrame.java

```
package button;

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

/**
 * A frame with a button panel
 */
public class ButtonFrame extends JFrame
{
   private JPanel buttonPanel;
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;

   public ButtonFrame()
```

```
17    {
18       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
19
20       // create buttons
21       JButton yellowButton = new JButton("Yellow");
22       JButton blueButton = new JButton("Blue");
23       JButton redButton = new JButton("Red");
24
25       buttonPanel = new JPanel();
26
27       // add buttons to panel
28       buttonPanel.add(yellowButton);
29       buttonPanel.add(blueButton);
30       buttonPanel.add(redButton);
31
32       // add panel to frame
33       add(buttonPanel);
34
35       // create button actions
36       ColorAction yellowAction = new ColorAction(Color.YELLOW);
37       ColorAction blueAction = new ColorAction(Color.BLUE);
38       ColorAction redAction = new ColorAction(Color.RED);
39
40       // associate actions with buttons
41       yellowButton.addActionListener(yellowAction);
42       blueButton.addActionListener(blueAction);
43       redButton.addActionListener(redAction);
44    }
45
46    /**
47     * An action listener that sets the panel's background color.
48     */
49    private class ColorAction implements ActionListener
50    {
51       private Color backgroundColor;
52
53       public ColorAction(Color c)
54       {
55          backgroundColor = c;
56       }
57
58       public void actionPerformed(ActionEvent event)
59       {
60          buttonPanel.setBackground(backgroundColor);
61       }
62    }
63 }
```

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。

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。如果仔细看一下程序清单 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。

为了简化这些任务，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，容器将自身设置为监听器，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。

8.1.3 创建包含一个方法调用的监听器

不用内部类也可以指定简单的事件监听器。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。

警告：EventHandler.create 调用中的第二个参数必须属于一个公共（public）类。否则，反射机制将无法定位和调用目标方法。

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一个代理类对象。命名方法或接口的所有方法都将在目标对象上执行给定动作。

这个动作可以是一个方法名或目标的一个属性。如果是一个属性，将执行其设置方法。例如，动作 "text" 将转换为一个 setText 方法调用。

事件属性包括一个或多个用点号分隔的属性名。第一个属性从监听器方法的参数读取，第二个属性由结果对象读取，依此类推。最后的结果将作为动作的参数。例如，属性 "source.text" 会转换为 getSource 和 getText 方法调用。

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 Macintosh 的观感包只与 Windows 和 Macintosh 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();
```

程序清单 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。这个程序与程序清单 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。

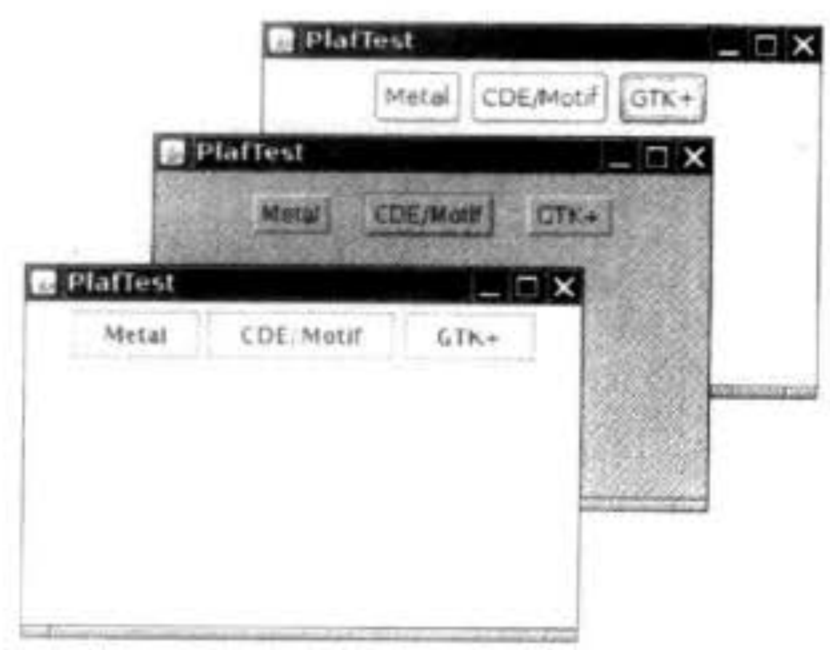


图 8-4 切换观感

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);
```

程序清单 8-2 plaf/PlafFrame.java

```
package plaf;

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

/**
 * A frame with a button panel for changing look-and-feel
 */
public class PlafFrame extends JFrame
{
   private JPanel buttonPanel;
   public PlafFrame()
   {
      buttonPanel = new JPanel();

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

```
      for (UIManager.LookAndFeelInfo info : infos)
         makeButton(info.getName(), info.getClassName());

      add(buttonPanel);
      pack();
   }

   /**
    * Makes a button to change the pluggable look-and-feel.
    * @param name the button name
    * @param plafName the name of the look-and-feel class
    */
   void makeButton(String name, final String plafName)
   {
      // add button to panel

      JButton button = new JButton(name);
      buttonPanel.add(button);

      // set button action

      button.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               // button action: switch to the new look-and-feel
               try
               {
                  UIManager.setLookAndFeel(plafName);
                  SwingUtilities.updateComponentTreeUI(PlafFrame.this);
                  pack();
               }
               catch (Exception e)
               {
                  e.printStackTrace();
               }
            }
         });
   }
}
```

API 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

API 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，唯一的例外是在 Windows 下，通常将 iconified（图标化）称为 minimized（最小化）。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需求。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一个覆盖的方法 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。

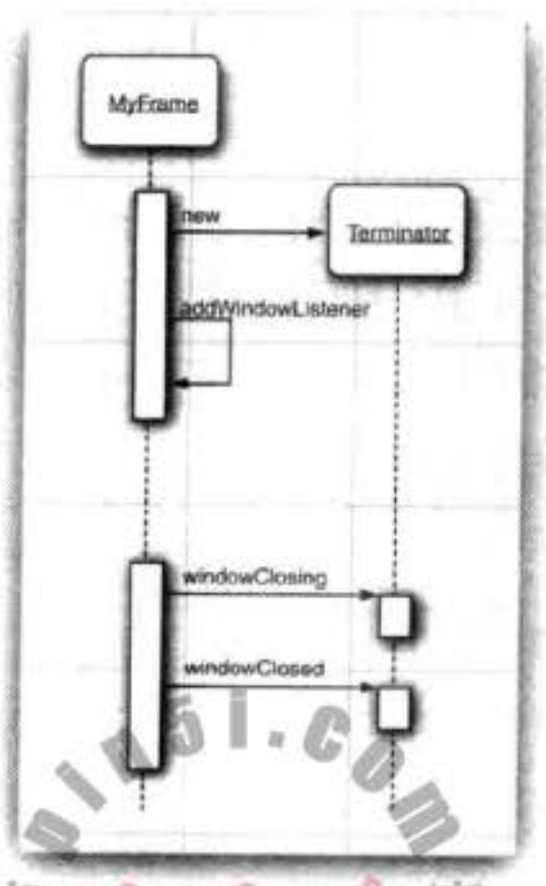


图 8-5 窗口监听器

◆ **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。

API 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。需要注意的是，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。

API java.awt.event.WindowStateListener 1.4

- `void windowStateChanged(WindowEvent event)`

  窗口被最大化、图标化或恢复为正常大小时调用这个方法。

API 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
- 按键 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，还是按键，其处理都是一样的。

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 AsbstractAction 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。

图 8-6 按钮显示来自动作对象的图标

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，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（imput 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

按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。如果这个按键存在，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。一旦找到按键对应的映射，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，这个按键被注册到 WHEN_IN_FOCUSED_WINDOW 映射中。给这些组件（按照按键注册的顺序）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。如果一个按键在多个 WHEN_IN_FOCUSED_WINDOW 映射中出现，这部分处理就可能会出现问题。

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按键的任务。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按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按键动作：

```
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、菜单项或按键的方式：

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）为了能够通过按键触发动作，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。为需要的按键创建一个 KeyStrike 对象。创建一个描述动作字符串这样的动作键对象。将（按键，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。

程序清单 8-3 给出了将按钮和按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。

程序清单 8-3 action/ActionFrame.java

```java
package action;

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

/**
 * A frame with a panel that demonstrates color change actions.
 */
public class ActionFrame extends JFrame
{
   private JPanel buttonPanel;
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;

   public ActionFrame()
   {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      buttonPanel = new JPanel();

      // define actions
      Action yellowAction = new ColorAction("Yellow", new ImageIcon("yellow-ball.gif"),
            Color.YELLOW);
      Action blueAction = new ColorAction("Blue", new ImageIcon("blue-ball.gif"), Color.BLUE);
      Action redAction = new ColorAction("Red", new ImageIcon("red-ball.gif"), Color.RED);

      // add buttons for these actions
      buttonPanel.add(new JButton(yellowAction));
      buttonPanel.add(new JButton(blueAction));
      buttonPanel.add(new JButton(redAction));

      // add panel to frame
      add(buttonPanel);
```

```

      // associate the Y, B, and R keys with names
      InputMap imap = buttonPanel.getInputMap(JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT);
      imap.put(KeyStroke.getKeyStroke("ctrl Y"), "panel.yellow");
      imap.put(KeyStroke.getKeyStroke("ctrl B"), "panel.blue");
      imap.put(KeyStroke.getKeyStroke("ctrl R"), "panel.red");

      // associate the names with actions
      ActionMap amap = buttonPanel.getActionMap();
      amap.put("panel.yellow", yellowAction);
      amap.put("panel.blue", blueAction);
      amap.put("panel.red", redAction);
   }

   public class ColorAction extends AbstractAction
   {
      /**
       * Constructs a color action.
       * @param name the name to show on the button
       * @param icon the icon to display on the button
       * @param c the background color
       */
      public ColorAction(String name, Icon icon, Color c)
      {
         putValue(Action.NAME, name);
         putValue(Action.SMALL_ICON, icon);
         putValue(Action.SHORT_DESCRIPTION, "Set panel color to " + name.toLowerCase());
         putValue("color", c);
      }

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

API javax.swing.Action 1.2

- `boolean isEnalbled()`
- `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按键（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`

  获得将按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。

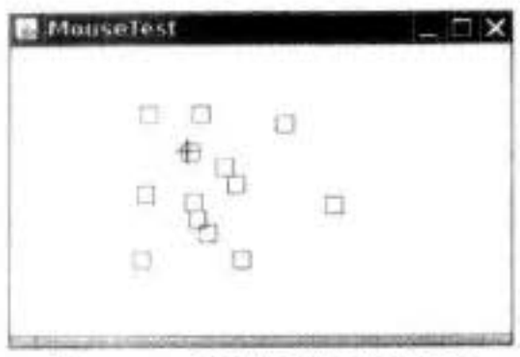


图 8-7　鼠标测试程序

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。

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符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。

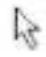
```
public void mousePressed(MouseEvent event)
{
   current = find(event.getPoint());
   if (current == null) // not inside a square
      add(event.getPoint());
}

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： 还可以利用 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。程序清单 8-4 是这个程序的清单。

程序清单 8-4 mouse/MouseFrame.java

```java
package mouse;

import javax.swing.*;

/**
 * A frame containing a panel for testing mouse operations
 */
public class MouseFrame extends JFrame
{
   public MouseFrame()
   {
      add(new MouseComponent());
      pack();
   }
}
```

程序清单 8-5 mouse/MouseComponent.java

```java
package mouse;

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

/**
 * A component with mouse operations for adding and removing squares.
 */
public class MouseComponent extends JComponent
{
   private static final int SIDELENGTH = 10;
   private ArrayList<Rectangle2D> squares;
   private Rectangle2D current;

   public MouseComponent()
   {
```

```
      squares = new ArrayList<>();
      current = null;

      addMouseListener(new MouseHandler());
      addMouseMotionListener(new MouseMotionHandler());
   }

   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;

      // draw all squares
      for (Rectangle2D r : squares)
         g2.draw(r);
   }
   /**
    * Finds the first square containing a point.
    * @param p a point
    * @return the first square that contains p
    */
   public Rectangle2D find(Point2D p)
   {
      for (Rectangle2D r : squares)
      {
         if (r.contains(p)) return r;
      }
      return null;
   }

   /**
    * Adds a square to the collection.
    * @param p the center of the square
    */
   public void add(Point2D p)
   {
      double x = p.getX();
      double y = p.getY();

      current = new Rectangle2D.Double(x - SIDELENGTH / 2, y - SIDELENGTH / 2, SIDELENGTH,
            SIDELENGTH);
      squares.add(current);
      repaint();
   }

   /**
    * Removes a square from the collection.
    * @param s the square to remove
    */
   public void remove(Rectangle2D s)
   {
      if (s == null) return;
      if (s == current) current = null;
      squares.remove(s);
      repaint();
```

```
74      }
75      // the square containing the mouse cursor
76      private class MouseHandler extends MouseAdapter
77      {
78         public void mousePressed(MouseEvent event)
79         {
80            // add a new square if the cursor isn't inside a square
81            current = find(event.getPoint());
82            if (current == null) add(event.getPoint());
83         }
84         public void mouseClicked(MouseEvent event)
85         {
86            // remove the current square if double clicked
87            current = find(event.getPoint());
88            if (current != null && event.getClickCount() >= 2) remove(current);
89         }
90      }
91
92      private class MouseMotionHandler implements MouseMotionListener
93      {
94         public void mouseMoved(MouseEvent event)
95         {
96            // set the mouse cursor to cross hairs if it is inside a rectangle
97
98            if (find(event.getPoint()) == null) setCursor(Cursor.getDefaultCursor());
99            else setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
100        }
101
102        public void mouseDragged(MouseEvent event)
103        {
104           if (current != null)
105           {
106              int x = event.getX();
107              int y = event.getY();
108
109              // drag the current rectangle to center it at (x, y)
110              current.setFrame(x - SIDELENGTH / 2, y - SIDELENGTH / 2, SIDELENGTH, SIDELENGTH);
111              repaint();
112           }
113        }
114     }
115  }
```

API java.awt.event.MouseEvent 1.1

- int getX()
- 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）。

API 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。在此，将不讨论这些特殊的事件类型。

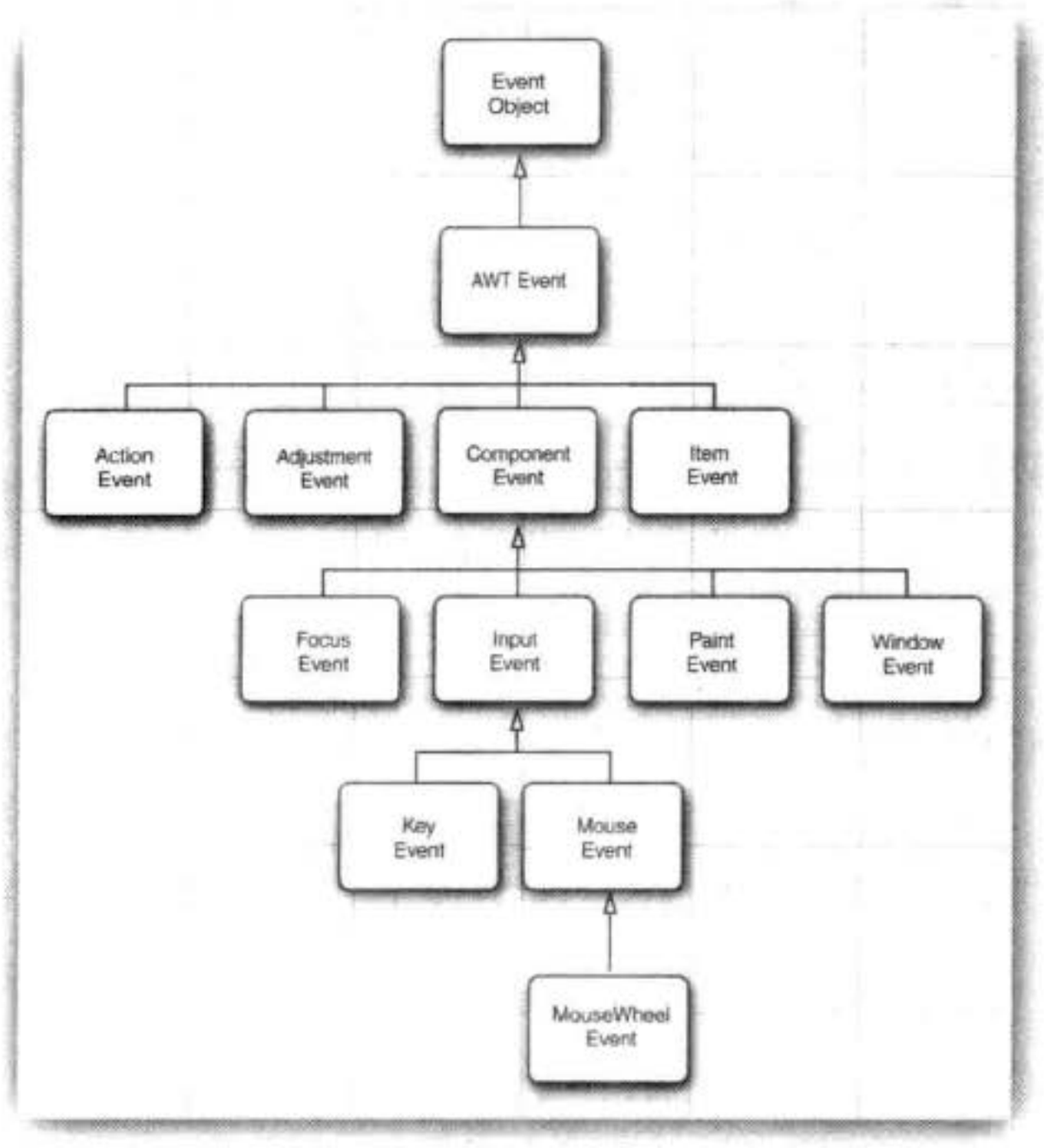


图 8-8 AWT 事件类的继承关系图

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

（续）

接　口	方　法	参数 / 访问方法	事 件 源
MouseListener	mousePressed mouseReleased mouseEntered mouseExited mouseClicked	MouseEvent • getClickCount • getX • getY • getPoint • translatePoint	Component
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。