

程序员书库

初学者的入门宝典，程序员的百科全书



CD-ROM

10小时多媒体视频讲解

本书特色

- ※ 起点低，即使没有任何编程经验，也能通过本书掌握Java
- ※ 避免大段理论讲解，而是通过大量实例进行讲解，有很强的实践性
- ※ 对代码进行了详细注释，阅读起来非常容易，没有任何障碍
- ※ 通过现实中的事物类比Java中的概念，使读者可以很容易理解
- ※ 重点讲解Java语言的基础知识和应用，并对一些设计模式也有所介绍
- ※ 全书提供190个实例和2个综合案例，非常实用

Java

从入门到精通

高宏静 等编著



化学工业出版社

第 12 章 事件处理

上一章主要讨论了组件的使用以及容器的布局，这些使 Java 图形用户界面的程序设计变得灵活方便，但是与图形界面程序密切相关的还有一个比较重要的内容——事件处理机制。本章主要讲述事件处理模型的概念，介绍事件处理类、事件监听器、时间适配器等内容。

12.1 事件处理模型

以下 3 类与事件处理机制相关。

- ❑ **Event**（事件对象）：用户界面操作以类的形式描述，例如鼠标操作对应的事件类 `MouseEvent`，界面动作对应的事件类 `ActionEvent`。
- ❑ **Event Source**（事件源）：产生事件的场所，通常指组件，例如按钮 `Checkbox`。
- ❑ **Event handler**（事件处理器）：接收事件类并进行相应的处理对象。

例如，在窗口中有一个按钮，当用户用鼠标单击这个按钮时，会产生 `ActionEvent` 类的一个对象。该按钮就是所谓的事件源，该对象就是鼠标操作所对应的事件，然后事件监听器接受触发的事件，并进行相应处理。

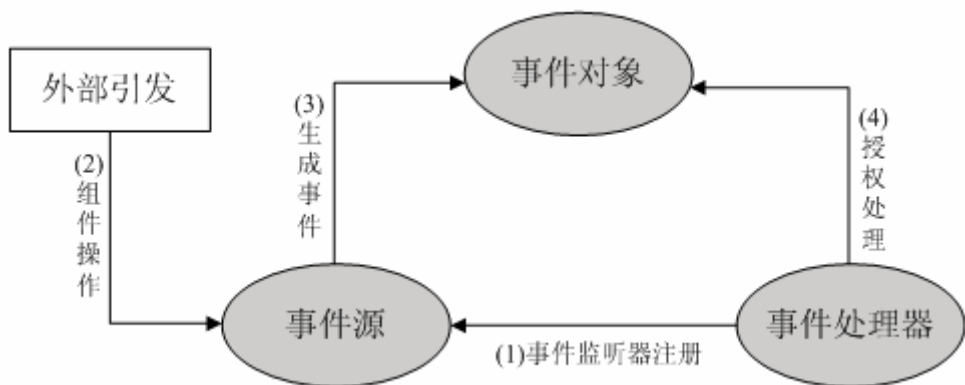


图 12-1 事件处理机制

同一个事件源可能会产生一个或者多个事件，Java 语言采用授权处理机制（**Delegation Model**）将事件源可能产生的事件分发给不同的事件处理器。例如 `Panel` 对象可能发生鼠标事件和键盘事件，它可以授权处理鼠标事件的事件处理器来处理鼠标事件，同时也可以授权处理键盘事件的事件处理器处理键盘事件。事件处理器会一直监听所有的事件，直到有与之相匹配的事件，就马上进行相应的处理，因此事件处理器也称为事件监听器。

授权处理机制可以将事件委托给外部的处理对象进行处理，这就实现了事件源与事件处理器（监听器）的分离。通常事件处理器是一个事件类，该类必须实现处理该类型事件的接

口，并实现某些接口方法。例如程序 12.1 是一个演示事件处理模型的例子，类 `ButtonHandler` 实现了 `ActionListener` 接口，该接口可以处理的事件是 `ActionEvent`。

```
// 文件：程序 12.1      EventManagerDemon.java      描述：事件处理模型演示
//导入需要使用的包和类
import java.awt.*;
import java.awt.event.*;
public class EventManagerDemon {
    public static void main(String[] args) {
        final Frame f = new Frame("Test");           //声明，并初始化窗口对象 f
        Button b = new Button("Press Me!");           //声明，并初始化按钮对象 b
        //注册监听器进行授权，该方法的参数是事件处理者对象
        b.addActionListener(new ButtonHandler());
        f.setLayout(new FlowLayout());                //为窗口设置布局管理器 FlowLayout
        f.add(b);                                     //在窗口中添加按钮 b
        f.setSize(200,100);                           //设置窗口大小
        f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent evt) {
                f.setVisible(false);                    //设置窗口 f 不可见
                f.dispose();                            //释放窗口及其子组件的屏幕资源
                System.exit(0);                          //退出程序
            }
        });
        f.setVisible(true);                            //显示窗口
    }
}

//ButtonHandler 实现接口 ActionListener 才能做事件 ActionEvent 的处理者
class ButtonHandler implements ActionListener {
    public void actionPerformed(ActionEvent e)
        //ActionEvent 事件对象作为参数
    {
        System.out.println("时间发生，已经捕获到");
        //本接口必须实现的方法 actionPerformed
    }
}
```

编写完程序后，使用 `javac` 命令编译该文件产生 `class` 文件，然后使用 `java` 命令运行该 `class` 文件，运行结果如图 12-2，按下“Press Me!”按钮，运行结果如图 12-3 所示。

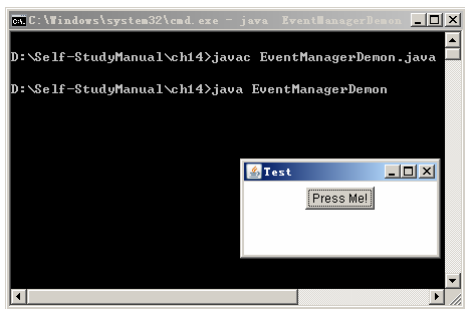


图 12-2 EventManagerDemon.java 运行结果一

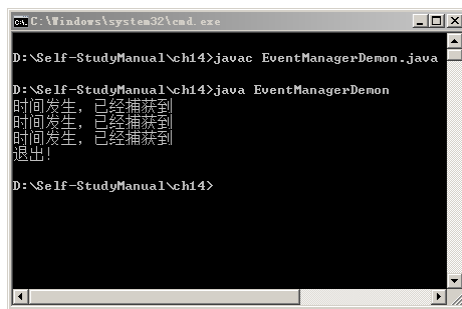


图 12-3 EventManagerDemon.java 运行结果二

程序 12.1 中，为窗口添加了 `WindowListener` 监听器和 `ActionListener` 监听器。监听器监听所有的事件，并当遇到与之匹配的事件，就调用响应的方法进行处理。每一个监听器接口

都有实现的方法，如 `ActionListener` 必须实现 `actionPerformed` 方法。Java 中授权处理机制具有以下特征。

- ❑ 在程序中如果想接受并处理事件 `*Event`，必须定义与之相应的事件处理类，该类必须实现与事件相对应接口 `*Listener`。
- ❑ 定义事件处理类之后，必须将事件处理对象注册到事件源上，使用方法 `add*Listener(*Listener)` 注册监听器。

12.2 事件类

每一个事件类都与一个事件类接口相对应，由事件引起的动作都存放在接口需要实现的方法中。本节主要讲述 Java 中比较常用的事件类接口，动作事件类、调整事件类、焦点事件类、项目事件类、按钮事件类、鼠标事件类以及窗口事件类。

12.2.1 事件类分类

所有与 AWT 相关的事件类都是 `java.awt.AWTEvent` 的派生类，`AWTEvent` 也是 `java.util.EventObject` 类的派生类。事件类的派生关系如下。

```
java.lang.Object
+--java.util.EventObject
    +--java.awt.AWTEvent
        +--java.awt.event.*Event
```

总体来说，AWT 事件有低级事件和高级事件两大类。低级事件是指源于组件或容器的事件，当组件或容器发生事件时（单击左键、右键、拖动以及窗口大小的改变等），将触发事件。高级事件是语义事件，此类事件与特定的具体事件不一定相对应，但是会产生特定的事件对象，如按钮被按下触发 `ActionEvent` 事件、滚动条移动滑块触发 `AdjustmentEvent` 事件、或选中项目列表某项时触发 `ItemEvent` 事件。

低级事件包括以下几种。

- ❑ 组件事件（`ComponentEvent`）。
- ❑ 容器事件（`ContainerEvent`）。
- ❑ 窗口事件（`WindowEvent`）。
- ❑ 焦点事件（`FocusEvent`）。
- ❑ 键盘事件（`KeyEvent`）。
- ❑ 鼠标事件（`MouseEvent`）。

高级事件（语义事件）包括以下几种。

- ❑ 动作事件（`ActionEvent`）。
- ❑ 调整事件（`AdjustmentEvent`）。
- ❑ 项目事件（`ItemEvent`）。
- ❑ 文本事件（`TextEvent`）。

本节介绍了几个经常使用的事件，其他的组件也十分类似，如果遇到什么问题，读者可以查询 API 等相关文档。

12.2.2 动作事件类

动作事件类 (ActionEvent) 指发生组件定义的语义事件，用户在操作 Button、CheckBox、TextField 等组件的时候将出现动作事件，例如单击 Button、TextField，按下回车键等。使用动作事件时需给组件增加一个事件监听器（事件处理器）ActionListener。ActionListener 只有惟一的 actionPerformed() 方法。它的一般格式如下。

```
Public void actionPerformed(ActionEvent e){
    //按钮被操作发生
}
```

ActionEvent 类的方法有以下几种。

- ❑ getActionCommand(): 返回命令字符串。
- ❑ getModifiers(): 取得按下的修饰符键。
- ❑ getWhen(): 取得事件发生的时间。
- ❑ paramString(): 生成事件状态的字符串。

假设存在按钮组件对象 button，动作事件使用如下。

```
button.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        //按钮被操作 dosomething
    }
})
```

程序 12.1 中定义了事件监听类 ButtonHandler，该类实现了 ActionListener 接口，并且在按钮对象 b 中通过 addActionListener(new ButtonHandler()) 注册时间监听器。

12.2.3 调整事件类

调整事件是滑动滚动条的滚动块时发生的事件。ScrollBar 组件在滑动滚动块时就会触发调整事件，为了监听调整事件，必须给 ScrollBar 组件对象添加一个调整事件监听器 AdjustmentListener。该监听器只有一个方法，其一般格式如下。

```
public void adjustmentValueChanged(AdjustmentEvent e)
{
    //调整滚动条发生
}
```

AdjustmentEvent 类的方法有以下几种。

- ❑ getAdjustable(): 取得事件源，返回 Adjustable 接口对象。
- ❑ getAdjustmentType(): 取得调整类型。
- ❑ getValue(): 取得源中的值。

- ❑ `getValueIsAdjusting()`: 判断源是否被移动。
- ❑ `paramString()`: 生成事件状态的字符串。

假设现有滚动条组件对象 `scrollbar`，调整事件使用如下。

```
scrollbar.addAdjustmentListener(new AdjustmentListener()
{
    public void adjustmentValueChanged(AdjustmentEvent e)
    {
        //调整滚动条 do someting
    }
});
```

下面是一个调整事件类的例子。

```
// 文件: 程序 12.2      AdjustmentEventDemo.java      描述: AdjustmentEvent 演示
//导入需要使用的包和类
import java.awt.*;
import java.awt.event.AdjustmentEvent;
import java.awt.event.AdjustmentListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
public class AdjustmentEventDemo extends Frame {
    //声明 AdjustmentEventDemo 构造方法
    public AdjustmentEventDemo ()
    {
        super();                //调用父类构造方法
        init();                  //调用 init 方法
    }
    public static void main(String args[ ])
    {
        new AdjustmentEventDemo(); //实例化 AdjustmentEventDemo 对象
    }
    Scrollbar slider;            //声明滚动条域 slider
    TextField value;             //声明 TextField 域 value
    Label label;                 //声明 Label 域 label

    public void init( ) {
        setLayout(new GridLayout(1, 3)); //设置窗口的布局管理器为 GridLayout
        slider = new Scrollbar(Scrollbar.HORIZONTAL,0,1,0,100); //初始化滚动条对象 slider
        //为滚动条添加 AdjustmentListener 监听器
        slider.addAdjustmentListener(new AdjustmentEventHandler());
        value = new TextField("0",5); //初始化文本域
        value.setEditable(false); //设置文本域不可编辑
        label = new Label("0~100"); //初始化标签对象 label
        label.setBackground(Color.cyan); //设置标签的背景色
        add(label); //将标签添加到窗口中
        add(slider); //将滚动条添加到窗口中
        add(value); //将文本域添加到窗口中
        addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent evt) {
                setVisible(false); //设置窗口 f 不可见
                dispose(); //释放窗口及其子组件的屏幕资源
                System.exit(0); //退出程序
            }
        });
    }
};
```

```

        setSize(300, 50);           //设置窗口大小
        setVisible(true);          //显示窗口
    }
    class AdjustmentEventHandler implements AdjustmentListener
    {
        //实现 adjustmentValueChanged 方法
        public void adjustmentValueChanged(AdjustmentEvent eve) {
            value.setText(Integer.toString(((Scrollbar)eve.getSource()).getValue())); //设置 value
            的值
        }
    }
}

```

编写完程序后,使用 javac 命令编译该文件产生 class 文件,然后使用 java 命令运行该 class 文件,运行结果如图 12-4 所示。

程序 12.2 中创建了一个滚动条对象 slider、一个标签对象 label、一个文本区对象 value。init()用于初始化窗口的组件,设置窗口的布局管理器为



图 12-4 AdjustmentEventDemo.java 运行结果

GridLayout, 并为 slider 添加了 AdjustmentListener 监听器,设置 label 的值为“0~100”,最后将 3 个组件添加窗口。Slider 添加的监听器的参数为 AdjustmentEventHandler 类对象,该类派生于 AdjustmentListener,并实现了 adjustmentValueChanged 方法。

12.2.4 焦点事件类

焦点事件类(FocusEvent)是指用户程序界面的组件失去焦点(即焦点从一个对象转移到另外一个对象)时,就会发生焦点事件。得到焦点事件的组件处于激活状态,例如界面包含文本框组件和单行文本输入区两个组件,如果文本框内部闪烁着光标,则表明该文本框拥有焦点。当鼠标单击单行文本框输入区时,这个时候就会触发焦点事件。使用焦点事件必须给组件增加一个 FocusListener 接口的事件处理器,该接口包含以下两个方法。

- ❑ void focusGained(FocusEvent e): 当获得焦点时发生。
 - ❑ void focusLost(FocusEvent e): 当失去焦点时发生。
- FocusEvent 类的方法有以下几种。
- ❑ getOppositeComponent(): 返回焦点转换到的下一个组件。
 - ❑ isTemporary(): 判断焦点的转换是暂时的还是永久的。
 - ❑ paramString(): 生成事件状态的字符串,用 toString()方法进行。

假设现有一个文本组件对象 textfield,焦点事件的使用示例如下:

```

textfield.addFocusListener(new FocusListener()
{
    public void focusGained(FocusEvent e)
    {
        //获得焦点 do something
    }
    public void focusLost(FocusEvent e)
    {
        //失去焦点 do something
    }
}

```

```
});
```

下面是一个焦点事件的例子。

```
// 文件: 程序 12.3    FocusEventDemo.java    描述: FocusEvent 演示
//导入需要使用的包和类
import java.awt.*;
import java.awt.event.FocusEvent;
import java.awt.event.FocusListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
public class FocusEventDemo extends Frame {
    //声明, 事项 FocusEventDemo 构造方法
    public FocusEventDemo ()
    {
        super();                //调用父类的构造方法
        init();                  //调用 init 方法
    }
    public static void main(String args[ ])
    {
        new FocusEventDemo();    //实例化 FocusEventDemo 对象
    }
    TextArea textarea;          //声明 TextArea 域
    TextField textfield;         //声明 TextField 域
    public void init() {
        setLayout(new GridLayout(2, 1)); //设置窗口的布局管理器为 GridLayout
        textarea = new TextArea();        //初始化 TextArea 对象 textarea
        textarea.addFocusListener(new FocusListener() //为 textarea 添加 FocusListener 监听器
        {
            public void focusGained(FocusEvent eve) {
                textarea.setText("textarea: 获得焦点"); //设置 textarea 的文本内容
            }
            public void focusLost(FocusEvent eve) {
                textarea.setText("textarea: 失去焦点"); //设置 textarea 的文本内容
            }
        }
    );
        textfield = new TextField();        //初始化 TextField 对象 textfield
        textfield.addFocusListener(new FocusListener() //为 textfield 添加 FocusListener 监听器
        {
            public void focusGained(FocusEvent eve) {
                textfield.setText("textfield: 获得焦点"); //设置 textfield 的文本内容
            }
            public void focusLost(FocusEvent eve) {
                textfield.setText("textfield: 失去焦点"); //设置 textfield 的文本内容
            }
        }
    );
        add(textarea); //在窗口中添加 textarea
        add(textfield); //在窗口中添加 textfield
        addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent evt) {
                setVisible(false); //设置窗口 f 不可见
                dispose();         //释放窗口及其子组件的屏幕资源
                System.exit(0);     //退出程序
            }
        });
    }
}
```



```

    });
    setSize(300, 200);
    setVisible(true);
}

```

//设置窗口的大小
//显示窗口

编写完程序后,使用 `javac` 命令编译该文件产生 `class` 文件,然后使用 `java` 命令运行该 `class` 文件,运行结果如图 12-5 所示。

程序 12.3 中有两个组件对象 `textfield` 和 `textarea`,当程序运行后,焦点位于容器中添加的第一个组件 `textarea`,但是当用鼠标单击 `textfield` 之后,可以看到 `textarea` 失去焦点, `textfield` 获得焦点,获得焦点的组件处于激活状态。



图 12-5 FocusEventDemo.java 运行结果

12.2.5 项目事件类

项目事件类 (`ItemEvent`) 是指某一个项目被选定、取消的语义事件。选择 `CheckBox`、`ComboBox`、`List` 等组件的时候将产生项目事件。使用项目事件必须给组件添加一个实现 `ItemListener` 接口的事件处理器,该接口的方法如下。

```
void itemStateChanged(ItemEvent e)
```

项目事件类的方法有以下几种。

- ❑ `getItem()`: 返回取得影响的项目对象。
- ❑ `getItemSelectable()`: 返回事件源 `ItemSelectable` 对象。
- ❑ `getStateChange()`: 返回状态的改变类型,包括 `SELECTED` 和 `DESELECTED` 两种。
- ❑ `paramString()`: 生成事件状态的字符串。

假设现有一个下拉列表框组件对象 `jComboBox1`,项目事件的使用示例如下。

```

jComboBox1.addItemListener(new ItemListener()
{
    public void itemStateChanged(ItemEvent e)
    {
        //项目发生改变 do something
    }
});

```

下面是一个项目事件的例子。

```

// 文件: 程序 12.4    ItemEventDemo.java    描述: ItemEvent 演示
//导入需要使用的包和类
import java.awt.*;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
public class ItemEventDemo extends Frame implements ItemListener{
    //声明,实现 ItemEventDemo 构造方法
    public ItemEventDemo()

```

```

{
    super();                //调用父类构造方法
    init();                 //调用 init 方法
}
public static void main(String args[ ])
{
    new ItemEventDemo();    //实例化 ItemEventDemo 对象
}

List list; //声明滚动文本项列表类型域
Checkbox checkbox1,checkbox2,checkbox3; //声明复选框类型域
Choice colorChooser; //声明弹出式选择菜单域
TextArea textarea; //声明文本区域
public void init( ) {
    setLayout(new GridLayout(4, 1)); //设置窗口布局管理器为 GridLayout
    textarea = new TextArea(); //初始化文本区域 textarea
    add(textarea); //将文本区对象添加到窗口中
    list = new List(4,false); //初始化滚动列表, 显示 4 行, 允许多选
    list.add("语文"); //在滚动项列表末尾添加“语文”项
    list.add("数学"); //在滚动项列表末尾添加“数学”项
    list.add("英语"); //在滚动项列表末尾添加“英语”项
    list.add("物理"); //在滚动项列表末尾添加“物理”项
    list.add("化学"); //在滚动项列表末尾添加“化学”项
    list.add("历史"); //在滚动项列表末尾添加“历史”项
    list.add("地理"); //在滚动项列表末尾添加“地理”项
    add(list); //在窗口中添加滚动列表
    list.addItemListener(this); //为 list 添加监听器 ItemListener
    Panel panel = new Panel(); //创建, 并初始化面板对象 panel
    CheckboxGroup cbg = new CheckboxGroup(); //创建, 并初始化一个复选框组 cbg
    checkbox1 = new Checkbox("one", cbg, true); //初始化复选框对象 checkbox1
    checkbox1.addItemListener(this); //为复选框 checkbox1 添加项目监听器 ItemListener
    panel.add(checkbox1); //在面板中添加复选框 checkbox1
    checkbox2 = new Checkbox("two", cbg, false); //初始化复选框对象 checkbox2
    checkbox2.addItemListener(this); //为复选框 checkbox1 添加项目监听器 ItemListener
    panel.add(checkbox2); //在面板中添加复选框 checkbox1
    checkbox3 = new Checkbox("three", cbg, true); //初始化复选框对象 checkbox3
    checkbox3.addItemListener(this); //为复选框 checkbox1 添加项目监听器 ItemListener
    panel.add(checkbox3); //在面板中添加复选框 checkbox1
    add(panel); //将面板添加到窗口
    colorChooser=new Choice(); //初始化菜单 colorChooser
    colorChooser.add("Green"); //将“Green”项添加到 colorChooser 菜单
    colorChooser.add("Red"); //将“Red”项添加到 colorChooser 菜单
    colorChooser.add("Blue"); //将“Blue”项添加到 colorChooser 菜单
    colorChooser.addItemListener(this); //为菜单 colorChooser 添加项目监听器 ItemListener
    add(colorChooser); //将菜单 colorChooser 添加到窗口
    addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent evt) {
            setVisible(false); //设置窗口 f 不可见
            dispose(); //释放窗口及其子组件的屏幕资源
            System.exit(0); //退出程序
        }
    });
    setSize(200, 300); //设置窗口的大小
}

```

```

        setVisible(true);                //显示窗口
    }
    public void itemStateChanged(ItemEvent eve) {
        if (eve.getSource() == list)
        {
            textarea.setText(list.getSelectedItem());        //设置文本区 textarea 的内容
        }
        if (eve.getSource() == checkbox1)
        {
            textarea.setText(checkbox1.getLabel());            //设置文本区 textarea 的内容
        }
        if (eve.getSource() == checkbox2)
        {
            textarea.setText(checkbox2.getLabel()); /          /设置文本区 textarea 的内容
        }
        if (eve.getSource() == checkbox3)
        {
            textarea.setText(checkbox3.getLabel());            //设置文本区 textarea 的内容
        }
        if (eve.getSource() == colorChooser)
        {
            textarea.setText(colorChooser.getSelectedItem()); //设置文本区 textarea 的内容
        }
    }
}

```

编写完程序后，使用 javac 命令编译该文件产生 class 文件，然后使用 java 命令运行该 class 文件，运行结果如图 12-6 所示。

程序 12.4 中共有 3 个组件，分别为 List、Checkbox、Choice 类的对象。当选这 3 个对象中的项目时，都会产生 ItemEvent 对象，通过授权处理机制来完成事件处理。

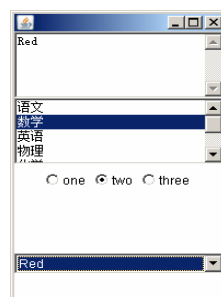


图 12-6 ItemEventDemo.java 运行结果

12.2.6 键盘事件类

键盘事件类（KeyEvent）是容器内的任意组件获得焦点时，组件发生键击事件，当按下释放或键入某一个键时，组件对象将产生该事件。使用键盘事件必须给组件添加一个 KeyListener 接口的事件处理器，该接口包含以下 3 个方法。

- ❑ void keyPressed(KeyEvent e): 按下按键时发生。
- ❑ void keyReleased(KeyEvent e): 松开按键时发生。
- ❑ void keyTyped(KeyEvent e): 敲击键盘，发生在按键按下后，按键放开前。

键盘事件类的方法有以下几种。

- ❑ getKeyChar(): 返回在键盘上按下的字符。
- ❑ getKeyCode(): 返回在键盘上按下的字符码。
- ❑ getKeyLocation(): 返回键位置。
- ❑ getKeyModifiersText(): 返回描述修饰符的文本字符串。
- ❑ getKeyText(): 返回键码编程描述键的文本。

- `isActionKey()`: 判断键是否是操作键。
- `setKeyChar()`: 改变键字符为指定的字符。
- `setModifiers(int modifiers)`: 改变键修饰符为指定的键修饰符。
- `paramString()`: 生成事件状态的字符串。

假设有一个文本框组件对象 `textfield`，键盘事件的使用示例如下。

```
textfield.addKeyListener(new KeyListener()
{
    public void keyPressed(KeyEvent e)
    {
        //按下按键时 dosomething
    }
    public void keyReleased(KeyEvent e)
    {
        //放开按键时 dosomething
    }
    public void keyTyped(KeyEvent e)
    {
        //敲击键盘，发生在按键按下后，按键放开前 dosomething
    }
});
```

下面是一个键盘事件的例子。

```
// 文件: 程序 12.5    KeyEventDemo.java    描述: KeyEvent 演示
//导入需要使用的包和类
import java.awt.*;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
public class KeyEventDemo extends Frame implements KeyListener{
    //声明，并实现 KeyEventDemo 构造方法
    public KeyEventDemo()
    {
        super();                //调用父类构造方法
        init();                  //调用 init 方法
    }
    public static void main(String args[ ])
    {
        new KeyEventDemo();      //实例化 KeyEventDemo 对象
    }

    Button button; //声明按钮类型域 button
    TextArea textarea,textarea1; //声明文本区类型域 textarea,textarea1
    public void init() {
        setLayout(new GridLayout(3, 1)); //设置窗口布局管理器 GridLayout
        textarea = new TextArea();        //初始化文本区 textarea 域
        textarea1 = new TextArea();       //初始化文本区 textarea1 域
        add(textarea);                    //将文本区添加到窗口中
        button = new Button("请您单击我，然后单击键盘键"); //初始化按钮域 button
        add(button);                      //将按钮添加到窗口中
        button.addKeyListener(this);      //为按钮 button 添加键盘监听器
    }
}
```

```

add(textarea1);           //将文本区 textarea1 添加窗口
addWindowListener(new WindowAdapter(){
    public void windowClosing(WindowEvent evt) {
        setVisible(false);    //设置窗口 f 不可见
        dispose();           //释放窗口及其子组件的屏幕资源
        System.exit(0);       //退出程序
    }
});
setSize(200, 300);        //设置窗口大小
setVisible(true);         //显现窗口
}

public void keyPressed(KeyEvent eve) {
    textarea.setText("按下键盘");    //设置文本区 textarea 内容
}
public void keyReleased(KeyEvent eve) {
    textarea.setText("松开键盘");    //设置文本区 textarea 内容
}
public void keyTyped(KeyEvent eve) {
    textarea1.setText(String.valueOf(eve.getKeyChar()));    //设置文本区 textarea1 内容
}
}

```

编写完程序后,使用 javac 命令编译该文件产生 class 文件,然后使用 java 命令运行该 class 文件,运行结果如图 12-7 所示。

从程序 12.5 中可以看到,当用户用鼠标点击上面图所示的按钮时,就触发了键盘监听事件,用户按下键盘上任何一个键时,按钮上方的文本输入区将显示“按下键盘”,该功能对应的方法为 `keyPressed`;同时在按钮的下方的文本区域显示被按下的键盘上面所标示的字符,该功能对应的方法为 `keyTyped`;当用户抬起按下键盘上的手时,上边的文本输入区由原来的“按下键盘”变成了“松开键盘”,该功能对应的方法为 `keyReleased`,就这样的一按一放完成了整个事件的操作。通过上面的小例子,是不是可以清楚的了解到键盘事件的用处了。



图 12-7 KeyEventDemo.java 运行结果

12.2.7 鼠标事件类

鼠标事件类 (`MouseEvent`) 指组件中发生的鼠标动作事件,例如按下鼠标、释放鼠标、单击鼠标、鼠标光标进入或离开组件的几何图形、移动鼠标、拖动鼠标。当鼠标移动到某个区域或鼠标单击某个组件时就会触发鼠标事件。使用鼠标事件必须给组件添加一个 `MouseListener` 接口的事件处理器,该接口包含以下 5 个方法。

- ❑ `void mouseClicked(MouseEvent e)`: 当鼠标在该区域单击时发生。
- ❑ `void mouseEntered(MouseEvent e)`: 当鼠标进入该区域时发生。
- ❑ `void mouseExited(MouseEvent e)`: 当鼠标离开该区域时发生。
- ❑ `void mousePressed(MouseEvent e)`: 当鼠标在该区域按下时发生。
- ❑ `void mouseReleased(MouseEvent e)`: 当鼠标在该区域放开时发生。

鼠标事件类的方法有以下几种。

- ❑ `getButton()`: 返回鼠标键状态改变指示。
- ❑ `getClickCount()`: 返回鼠标键单击的次数。
- ❑ `getMouseModifiersText()`: 返回指定修饰符文本字符串。
- ❑ `getPoint()`: 返回事件源中位置对象。
- ❑ `getX()`: 返回鼠标在指定区域内相对位置的横坐标。
- ❑ `getY()`: 返回鼠标在指定区域内相对位置的纵坐标。
- ❑ `paramString()`: 生成事件状态的字符串。

假设有一个面板组件对象 `panel`，鼠标事件的使用示例如下。

```
panel.addMouseListener(new MouseListener()
{
    public void mouseClicked(MouseEvent e)
    {
        //当鼠标在该区域单击时发生 dosomething
    }
    public void mouseEntered(MouseEvent e)
    {
        //当鼠标进入该区域时发生 dosomething
    }
    public void mouseExited(MouseEvent e)
    {
        //当鼠标离开该区域时发生 dosomething
    }
    public void mousePressed(MouseEvent e)
    {
        //当鼠标在该区域按下时发生 dosomething
    }
    public void mouseReleased(MouseEvent e)
    {
        //当鼠标在该区域放开时发生 dosomething
    }
});
```

下面是一个鼠标事件的例子。

```
// 文件: 程序 12.6      MouseEventDemo.java      描述: MouseEvent 演示
//导入需要使用的包和类
import java.awt.*;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
public class MouseEventDemo extends Frame{
    //声明 MouseEventDemo 的构造方法
    public MouseEventDemo()
    {
        super();           //调用父类的构造方法
        init();             //调用 init 方法
    }
    public static void main(String args[ ])
    {
        new MouseEventDemo(); //创建 MouseEventDemo 实例
    }
}
```

```

}
Panel panel; //声明面板类型域 panel
TextField textfield1,textfield2; //声明单行文本区类型的域 textfield1,textfield2
public void init() {
    setLayout(new GridLayout(3, 1)); //设置布局管理器 GridLayout
    textfield1 = new TextField(20); //初始化单行文本区 textfield1
    textfield2 = new TextField(); //初始化单行文本区 textfield2
    add(textfield1); //在窗口中添加单行文本区 textfield1
    add(textfield2); //在窗口中添加单行文本区 textfield2
    panel = new Panel(); //初始化面板 panel
    panel.setBackground(Color.cyan); //设置面板的背景色
    add(panel); //在窗口中添加面板 panel
    panel.addMouseListener(new MouseListener()
    {
        public void mouseClicked(MouseEvent eve) {
            textfield2.setText("X="+eve.getX()+";Y="+eve.getY());//设置 textfield2 的内容
        }
        public void mouseEntered(MouseEvent eve) {
            textfield1.setText("鼠标进入面板区域"); //设置 textfield1 的内容
        }
        public void mouseExited(MouseEvent eve) {
            textfield1.setText("鼠标离开面板区域"); //设置 textfield1 的内容
        }
        public void mousePressed(MouseEvent eve) {
            textfield1.setText("鼠标被按下"); //设置 textfield1 的内容
        }
        public void mouseReleased(MouseEvent eve) {
            textfield1.setText("鼠标松开"); //设置 textfield1 的内容
        }
    });
    addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent evt) {
            setVisible(false); //设置窗口 f 不可见
            dispose(); //释放窗口及其子组件的屏幕资源
            System.exit(0); //退出程序
        }
    });
    setSize(200,200); //设置窗口的大小
    setVisible(true); //显示窗口
}
}

```

编写完程序后,使用 javac 命令编译该文件产生 class 文件,然后使用 java 命令运行该 class 文件,运行结果如图 12-8 所示。

程序 12.6 中,为面板 panel 添加监听器 MouseListener,该接口必须实现 mouseClicked、mouseEntered、mouseExited、mousePressed、mouseReleased 方法。

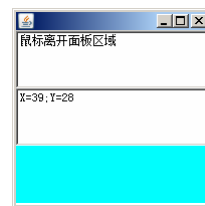


图 12-8 MouseEventDemo.java 运行结果

12.2.8 窗口事件类

窗口事件（WindowEvent）指窗口状态改变的事件，例如当窗口 Window 对象的打开、关闭、激活、停用或者焦点转移到窗口内，以及焦点移除而生成的事件，一般发生在 Window、Frame、Dialog 等类的对象上。使用窗口事件必须为组件添加一个实现 WindowListener 接口的事件处理器，该接口包含以下 7 种方法。

- ❑ void windowActivated(WindowEvent e): 窗口被激活时发生。
- ❑ void windowClosed(WindowEvent e): 窗口关闭之后发生。
- ❑ void windowClosing(WindowEvent e): 窗口关闭过程中发生。
- ❑ void windowDeactivated(WindowEvent e): 窗口不再处于激活状态时发生。
- ❑ void windowDeiconified(WindowEvent e): 窗口大小从最小到正常时发生。
- ❑ void windowIconified(WindowEvent e): 窗口从正常到最小时发生。
- ❑ void windowOpened(WindowEvent e): 窗口第一次被打开时发生。

窗口事件类的方法有以下几种。

- ❑ getState(): 返回窗口改变之后的新状态。
- ❑ getOldState(): 返回窗口改变之后的旧状态。
- ❑ getOppositeWindow(): 返回事件设计的辅助窗口。
- ❑ getWindow(): 返回事件源。
- ❑ paramString(): 生成事件状态的字符串。

假设有一个 frame 框组件对象，窗口事件的使用示例如下。

```
frame.addWindowListener(new WindowListener()
{
    public void windowActivated(WindowEvent e)
    {
        //窗口被激活时发生 dosomething
    }
    public void windowClosed(WindowEvent e)
    {
        //窗口关闭之后发生 dosomething
    }
    public void windowClosing(WindowEvent e)
    {
        //窗口关闭过程中发生 dosomething
    }
    public void windowDeactivated(WindowEvent e)
    {
        //窗口不再处于激活状态时发生 dosomething
    }
    public void windowDeiconified(WindowEvent e)
    {
        //窗口大小从最小到正常时发生 dosomething
    }
    public void windowIconified(WindowEvent e)
    {
        //窗口从正常到最小时发生 dosomething
    }
}
```



```

public void windowOpened(WindowEvent e)
{
    //窗口第一次被打开时发生 dosomething
}
});

```

下面是一个窗口事件类的例子。

```

// 文件: 程序 12.7      WindowEventDemo.java      描述: WindowEvent 演示
//导入需要使用的包和类
import java.awt.*;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
public class WindowEventDemo extends Frame{
    //声明 WindowEventDemo 类的构造方法
    public WindowEventDemo()
    {
        super();                //调用父类的构造方法
        init();                  //调用 init 方法
    }
    public static void main(String args[ ])
    {
        new WindowEventDemo();    //创建 WindowEventDemo 类型实例
    }
    public void init( ) {
        addWindowListener(new WindowListener(){
            public void windowClosing(WindowEvent eve) {
                str = "windowClosing";    //给字符串赋值
                System.out.println(str);  //输出字符串
                setVisible(false);        //设置窗口 f 不可见
                dispose();                //释放窗口及其子组件的屏幕资源
                System.exit(0);           //退出程序
            }
            public void windowActivated(WindowEvent eve) {
                str = "windowActivated"; //给字符串赋值
                System.out.println(str); //输出字符串
                repaint();               //重绘
            }
            public void windowClosed(WindowEvent eve) {
                str = "windowClosed";    //给字符串赋值
                System.out.println(str); //输出字符串
                repaint();               //重绘
            }
            public void windowDeactivated(WindowEvent eve) {
                str = "windowDeactivated"; //给字符串赋值
                System.out.println(str); //输出字符串
                repaint();               //重绘
            }
            public void windowDeiconified(WindowEvent eve) {
                str = "windowDeiconified"; //给字符串赋值
                System.out.println(str); //输出字符串
                repaint();               //重绘
            }
            public void windowIconified(WindowEvent eve) {
                str = "windowIconified"; //给字符串赋值
            }
        });
    }
}

```

```

        System.out.println(str);    //输出字符串
        repaint();                //重绘
    }
    public void windowOpened(WindowEvent eve) {
        str = "windowOpened";    //给字符串赋值
        System.out.println(str);    //输出字符串
        repaint();                //重绘
    }
}
});
setSize(200,200);                //设置窗口大小
setVisible(true);                //显示窗口
}
public void paint(Graphics g) {
    g.drawString(str, 30, 100);    //在窗口中绘制字符串
}
String str = null;                //创建字符串类型域
}

```

编写完程序后,使用 `javac` 命令编译该文件产生 `class` 文件,然后使用 `java` 命令运行该 `class` 文件,运行结果如图 12-9 所示。

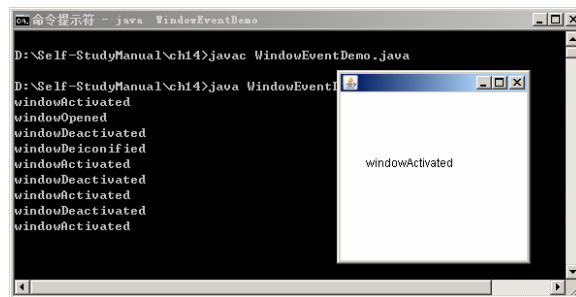


图 12-9 WindowEventDemo.java 运行结果

程序 12.7 中,为窗口添加了 `WindowListener` 监听器,该监听器必须实现 `windowClosing`、`windowActivated`、`windowClosed`、`windowDeactivated`、`windowDeiconified`、`windowIconified`、`windowOpened` 方法,但是这些方法可以为空,表示该事件的动作被激发时不做任何处理。

12.3 事件监听器

事件处理是图形用户界面程序设计中最基本的也是最重要的一项工作。当用户设备、外部输入发生事件时,可以使用事件监听器处理这些事件。事件监听器是为了处理特定事件而编写类的方法。

12.3.1 事件监听器接口

上一节曾提到每种事件类都有对应的事件监听器，它是事件监听器类的接口。例如键盘事件必须实现接口 `EventListener`，重写接口的方法如下。

```
public interface KeyListener extends EventListener
{
    public void keyPressed(KeyEvent ev);
    public void keyReleased(KeyEvent ev);
    public void keyTyped(KeyEvent ev);
}
```

在 `KeyListener` 接口中有 3 个方法：按下键盘的某一个键时，调用 `keyPressed()` 方法；松开键时，调用 `keyReleased()` 方法；键被按下到松开这段时间，调用 `keyTyped()` 方法。表 12-1 列出了所有 AWT 事件类和其相对应的监听器接口以及实现的方法（包括 10 类事件与之相对应的 11 个接口）。

表 12-1 事件监听器接口

事件类别	描述信息	接口名	方法
ActionEvent	激活组件	ActionListener	actionPerformed(ActionEvent)
ItemEvent	选择了某些项目	ItemListener	itemStateChanged(ItemEvent)
MouseEvent	鼠标移动	MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
	鼠标单击等	MouseListener	mousePressed(MouseEvent) mouseReleased(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mouseClicked(MouseEvent)
KeyEvent	键盘输入	KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
FocusEvent	组件收到或失去焦点	FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)
AdjustmentEvent	移动了滚动条等组件	AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
ComponentEvent	对象移动缩放显示隐藏等	ComponentListener	componentMoved(ComponentEvent) componentHidden(ComponentEvent) componentResized(ComponentEvent) componentShown(ComponentEvent)
WindowEvent	窗口收到窗口级事件	WindowListener	windowClosing(WindowEvent) windowOpened(WindowEvent) windowIconified(WindowEvent)

			windowDeiconified(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent)
ContainerEvent	容器中增加删除了组件	ContainerListener	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
TextEvent	文本字段或文本区发生改变	TextListener	textValueChanged(TextEvent)

AWT 的组件不仅可以注册监听器，还可以注销监听器，方法如下。

```
public void add<ListenerType> (<ListenerType>listener);    //注册监听器
public void remove<ListenerType> (<ListenerType>listener); //注销监听器
```

为 Button 对象添加监听器、取消监听器的方法如下。

```
public void addItemListener(ItemListener listener)
```

添加项目事件的侦听器，以接收来自此复选框被激发的项目事件。如果参数 listener 为 null，不抛出异常，也不执行任何操作。

```
public void removeItemListener(ItemListener listener)
```

移除项目侦听器，就不会再发送此复选框的项目事件。同样如果参数 listener 为 null，不抛出异常，也不执行任何操作。

12.3.2 事件监听器应用

下面是一个事件监听器模型应用的例子。ListenerDemo 类可以捕获鼠标动作、鼠标、窗口事件。当用户在窗口内部单击、移动、获得/失去焦点时，将会在组件显示不同的状态信息。

```
// 文件：程序 12.8    ListenerDemo.java    描述：事件监听器模型演示
//导入需要使用的包和类
import java.awt.*;
import java.awt.event.*;
public class ListenerDemo implements MouseMotionListener,MouseListener,WindowListener {
    public static void main(String[] args) {
        new ListenerDemo();           //创建 ListenerDemo 实例
    }
    public ListenerDemo()
    {
        f = new Frame("请单击，或拖动鼠标"); //实例化窗口对象 f
        tf1 = new TextField(20);             //初始化单行文本域 tf1
        tf2 = new TextField(20);             //初始化单行文本域 tf2
        f.add("South", tf1);                 //在窗口中添加 tf1 为 South 位置
        f.add("North", tf2);                 //在窗口中添加 tf1 为 North 位置
        f.addMouseListener(this);           //注册监听器 MouseListener
        f.addMouseMotionListener(this);       //注册监听器 MouseMotionListener
        f.addWindowListener(this);           //注册监听器 WindowListener

        f.setSize(400, 300);                 //设置窗口大小
        f.setVisible(true);                  //显示窗口
    }
}
```

```

//对其不感兴趣的方法可以方法体为空
public void mouseDragged(MouseEvent e) {
    String s = "鼠标拖动";           //为 s 赋值
    tf1.setText(s);                   //设置 tf1 文本
}
public void mouseMoved(MouseEvent e) {
    String s = "鼠标移动";           //为 s 赋值
    tf1.setText(s);                   //设置 tf1 文本内容
    tf2.setText("鼠标坐标为: X="+e.getX()+"Y = "+e.getY()); //设置 tf2 文本内容
}
public void mouseClicked(MouseEvent e) {
}
public void mouseEntered(MouseEvent e) {
    tf1.setText("鼠标进入");          //设置 tf1 文本内容
}
public void mouseExited(MouseEvent e) {
    String s = "鼠标离开";           //为 s 赋值
    tf1.setText(s);                   //设置 tf1 文本内容
}
public void mousePressed(MouseEvent e) { //声明 mousePressed 空方法
}
public void mouseReleased(MouseEvent e) { //声明 mouseReleased 空方法
}
public void windowActivated(WindowEvent e) { //声明 windowActivated 空方法
}
public void windowClosed(WindowEvent e) { //声明 windowClosed 空方法
}
//为了使窗口能正常关闭，程序正常退出，需要实现 windowClosing 方法
public void windowClosing(WindowEvent e) {
    System.exit(1); //程序退出
}
public void windowDeactivated(WindowEvent e) { //声明 windowDeactivated 空方法
}
public void windowDeiconified(WindowEvent e) { //声明 windowDeiconified 空方法
}
public void windowIconified(WindowEvent e) { //声明 windowIconified 空方法
}
public void windowOpened(WindowEvent e){ //声明 windowOpened 空方法
}
private Frame f; //声明窗口类型域 f
private TextField tf1,tf2; //声明单行文本区类型域 tf1,tf2
}

```

编写完程序后，使用 javac 命令编译该文件产生 class 文件，然后使用 java 命令运行该 class 文件，运行结果如图 12-10 所示。

程序 12.8 中，类实现了 MouseMotionListener、MouseListener、WindowListener 3 个接口，这 3 个接口的抽象方法必须实现，如果其中一些不需要做任何处理，写成空方法即可。

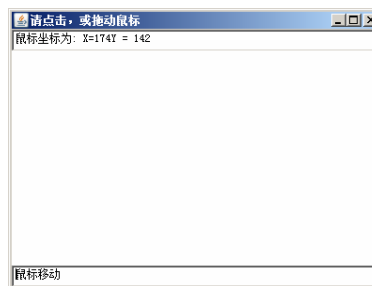


图 12-10 ListenerDemo.java 运行结果

12.3.3 事件监听器特点

一个监听器接口的类可以实现一个或多个接口，并且接口之间用逗号隔开。

```
implements MouseMotionListener, MouseListener, WindowListener;
```

同样，同一个组件可以监听一个事件源的多个事件。

```
f.addMouseMotionListener(this);    //添加鼠标动作监听器
f.addMouseListener(this);          //添加鼠标监听器
f.addWindowListener(this);          //添加窗口监听器
```

上面就是在对象 `f` 上接收多个事件，并被同一个监听器接收和处理。

事件处理者和事件源可以处在同一个类中。例如程序 12.8 中事件源是窗口对象 `f`，事件处理者是类 `ThreeListener`，其中 `f` 是类 `ThreeListener` 的成员变量。通过事件对象可以获得一些信息，例如获得组件中鼠标的位置坐标。

```
public void mouseMoved(MouseEvent e) {
    String s = "鼠标移动";                //声明，并初始化字符串 s
    tf1.setText(s);                        //设置 tf1 的值
    tf2.setText("鼠标坐标为: X="+e.getX()+"Y = "+e.getY()); //设置 tf2 的值
}
```

这里再重新讨论一下接口的内容。Java 语言只支持单继承，为了实现具有多继承的功能，Java 用接口来实现相同的功能。接口机制比直接实现多重继承更具简单、灵活、强大。在 AWT 中经常实现多个监听接口，每一个接口中已定义的方法必须实现，如果针对某事件不做出任何处理，就不需要实现该方法，即方法体为空，但是所有接口的所有方法必须写在类内。

12.4 事件适配器

Java 除了提供实现监听器接口的方法处理事件，还提供了另外一种简单的实现监听器的手段——事件适配器（`EventAdapter`）。程序员可以通过继承事件所对应的适配器类，重写感兴趣的方法。通过事件适配类可以缩短程序代码，但是 Java 只能实现单一的继承，当程序需要捕获多种事件时，就无法使用事件适配器的方法了。`java.awt.event` 包中定义的事件适配器类包括以下几种。

- ☐ `ComponentAdapter`（组件适配器）。
- ☐ `ContainerAdapter`（容器适配器）。
- ☐ `FocusAdapter`（焦点适配器）。
- ☐ `KeyAdapter`（键盘适配器）。
- ☐ `MouseAdapter`（鼠标适配器）。
- ☐ `MouseMotionAdapter`（鼠标运动适配器）。
- ☐ `WindowAdapter`（窗口适配器）。

下面是一个事件适配器使用的例子。

```

// 文件: 程序 12.9   AdapterDemo.java       描述: 事件适配器使用
//导入需要使用的包和类
import java.awt.*;
import java.awt.event.*;
public class AdapterDemo {
    public static void main(String[ ] args) {
        new AdapterDemo();                      //创建 AdapterDemo 实例
    }
    //声明 AdapterDemo 构造方法
    public AdapterDemo()
{
    f = new Frame("请单击, 或拖动鼠标");        //初始化窗口 f

    panel = new Panel();                        //初始化面板 panel
    f.add("Center",panel);                      //在窗口 Center 位置添加 panel
    //注册监听器 MouseEvent
    panel.addMouseListener(new MouseAdapter(){
        public void mousePressed(MouseEvent e){
            start = e.getPoint();                //获取鼠标事件坐标
            System.out.println(start);           //输出坐标
        }
        public void mouseReleased(MouseEvent e) {
            end = e.getPoint();                  //获取鼠标事件坐标
            System.out.println(end);             //输出坐标
            Graphics g = panel.getGraphics();    //获取面板图形上下文
            panel.paint(g);                     //面板图形绘制
            g.drawLine(start.x, start.y, end.x, end.y); //在面板中画线
        }
    });
    //为窗口 f 添加窗口监听器
    f.addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e) { //覆盖方法 windowClosing
            System.exit(1);                       //程序退出
        }
    });
    f.setSize(400, 300);                        //设置窗口大小
    f.setVisible(true);                        //显示窗口
}
private Frame f;                              //声明 Frame 类型域 f
private Point start,end;                      //声明 Point 类型域 start,end
private Panel panel;                          //声明 Panel 类型域 panel
}

```

编写完程序后，使用 `javac` 命令编译该文件产生 `class` 文件，然后使用 `java` 命令运行该 `class` 文件，运行结果如图 12-11 所示。

程序 12.9 中，面板（`panel`）中添加监听器 `MouseAdapter`，并重载了 `mousePressed` 和 `mouseReleased` 方法。在 `mousePressed` 方法中，获取鼠标的坐标并保存；在 `mouseReleased` 中，再次获取鼠标的坐标，然后根据前后获取的坐标画一条线。

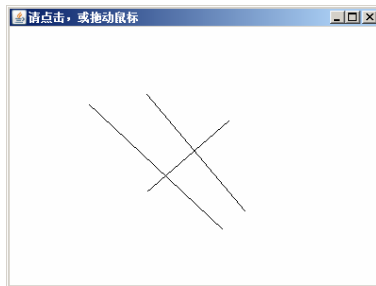


图 12-11 AdapterDemo.java 运行结果

12.5 匿名内部类应用

前面已经讲述过内部类和匿名类的概念，但是在前面的应用程序中基本没有涉及到这些内容。为组件添加事件监听器时，经常会将事件类以内部类和匿名类的方式使用。本节主要讨论内部类、匿名类在用户图形界面开发中的事件处理中的应用。

12.5.1 内部类

内部类（`Inner Class`）是被定义在一个类内部的类，内部类的特点如下。

- 内部类的对象具有外部类变量和方法的访问权限，包括私有成员。
- 实现事件监听器的功能时，采用内部类、匿名类实现相当容易。

所以，内部类在 AWT 的事件处理机制中应用比较广泛。下面是一个内部类使用的实例。

```
// 文件：程序 12.10 ListenerInnerClass.java      描述：内部类使用
//导入需要使用的包和类
import java.awt.*;
import java.awt.event.*;
public class ListenerInnerClass{
    public static void main(String args[ ] ) {
        ListenerInnerClass li = new ListenerInnerClass();//创建，并初始化 ListenerInnerClass 对象 li
        li.show();                                         //调用 show 方法
    }
    //声明 ListenerInnerClass 类的构造方法
    public ListenerInnerClass(){
        f=new Frame("监听器内部类,单击、移动鼠标");      //初始化窗口对象 f
        tf=new TextField(30);                             //初始化文本区对象 tf
    }
    public void show(){
        f.add(tf,"North");                                //将 tf 添加到窗口的 North 位置
        f.addMouseMotionListener(new MyMouseMotionListener()); //为窗口添加 MouseMotionListener 监听器
        f.setSize(300,240);                               //设置窗口大小
        //为窗口 f 添加监听器——匿名内部类
        f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent evt) { //覆盖 windowClosing 方法
```



```

        f.setVisible(false);           //设置窗口 f 不可见
        f.dispose();                   //释放窗口及其子组件的屏幕资源
        System.out.println("退出！");
        System.exit(0);                 //退出程序
    }
    });
    f.setVisible(true);                 //显示窗口
}
//成员内部类
class MyMouseMotionListener extends MouseMotionAdapter{ //内部类开始
    public void mouseMoved(MouseEvent e) {
        tf.setText("鼠标坐标: x="+e.getX()+"Y="+e.getY()); //设置 tf 的显示文本内容
    }
}
private Frame f;                       //声明 Frame 类型域 f
private TextField tf;                   //声明 TextField 类型域 tf
}

```

编写完程序后,使用 javac 命令编译该文件产生 class 文件,然后使用 java 命令运行该 class 文件,运行结果如图 12-12 所示。

程序 12.10 中,在 ListenerInnerClass 类中实现了内部类 MyMouseMotionListener,该类继承了 MouseMotionAdapter 类,并重载了 mouseMoved 方法,获取鼠标的坐标,显示在当行文本区中。窗口添加监听器通过 MyMouseMotionListener 实例完成。

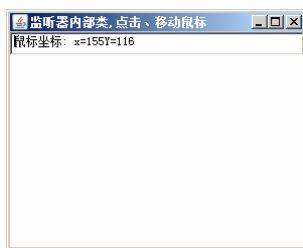


图 12-12 ListenerDemo.java 运行结果

12.5.2 匿名类

在 AWT 事件处理中,不仅可以使成员内部类 (Member Inner Class), 还可以使用匿名内部类 (Anonymous Inner Class)。当声明成员内部类之后,只创建一个对象,并且新的类继承了事件适配器或者是实现了事件监听接口,就可以使用匿名类。下面是一个匿名类在事件处理应用的例子。

```

// 文件: 程序 12.11 ListenerAnonymousClass.java      描述: 匿名类演示
//导入需要使用的包和类
import java.awt.*;
import java.awt.event.*;
public class ListenerAnonymousClass extends MouseMotionAdapter{
    public static void main(String args[ ]) {
        //创建,并初始化 ListenerAnonymousClass 对象 la
        ListenerAnonymousClass la = new ListenerAnonymousClass();
        la.show();                          //执行 show 方法
    }
}

```

```

    }
    public ListenerAnonymousClass(){
        f=new Frame("监听器内部类,单击、移动鼠标");//初始化窗口 f
        tf=new TextField(20);                      //初始化单行文本区 tf
    }
    public void show(){
        f.add(tf,"North");                          //将 tf 添加于窗口的 North 位置
        f.setSize(300,240);                        //设置窗口大小
        //匿名内部类
        f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent evt) {
                f.setVisible(false);                //设置窗口 f 不可见
                f.dispose();                        //释放窗口及其子组件的屏幕资源
                System.out.println("退出！");
                System.exit(0);                      //退出程序
            }
        });
        f.addMouseMotionListener(new MouseMotionAdapter(){ //内部匿名类开始
            public void mouseMoved(MouseEvent e) {
                tf.setText("鼠标坐标: x="+e.getX()+"Y="+e.getY()); //设置 tf 的文本内容
            }
        });
        f.setVisible(true);                        //显示窗口
    }
    private Frame f;                              //声明 Frame 类型域 f
    private TextField tf;                          //声明 TextField 类型域 tf
}

```

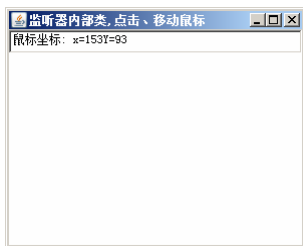


图 12-13 ListenerDemo.java 运行结果

编写完程序后,使用 javac 命令编译该文件产生 class 文件,然后使用 java 命令运行该 class 文件,运行结果如图 12-13 所示。

程序 12.11 中,窗口对象 f 添加 MouseMotionListener 监听器,参数是 MouseMotionAdapter 类的匿名类。细心的读者可能会发现,前面的两个程序实现的都是完全一样的功能,只是采取了不同的实现方式。其中一个实例的事件处理类是一个内部类,而另一个实例的事件处理类是匿名类。

注意: 当一个内部类只创建一个对象,并且该类继承一个父类或者实现一个接口的时候,才考虑使用匿名类。

12.6 案例——AWT 记事本

本节主要提供一个综合案例供读者参考,该程序是基于 AWT 的简单记事本。

12.6.1 域和构造方法

记事本类的域主要包括菜单栏、主文本输入区、菜单、菜单项、工具包、剪切板以及对话框等。NoteBook 的构造方法完成对域的初始化、组件位置安排和为组件添加事件监听器接。

```
MenuBar menuBar = new MenuBar();           //创建，并初始化菜单栏 menuBar
TextArea textArea = new TextArea();         //创建，并初始化文本区 textArea
//文件菜单
Menu fileMenu = new Menu("文件");           //创建，并初始化文件菜单 fileMenu
MenuItem newItem = new MenuItem("新建");    //创建，并初始化“新建”菜单项
MenuItem openItem = new MenuItem("打开");   //创建，并初始化“打开”菜单项
MenuItem saveItem = new MenuItem("保存");   //创建，并初始化“保存”菜单项
MenuItem saveAsItem = new MenuItem("另存"); //创建，并初始化“另存”菜单项
MenuItem exitItem = new MenuItem("退出");   //创建，并初始化“退出”菜单项
//编辑菜单
Menu editMenu = new Menu("编辑");           //创建，并初始化编辑菜单 editMenu
MenuItem selectItem = new MenuItem("全选"); //创建，并初始化“全选”菜单项
MenuItem copyItem = new MenuItem("复制");   //创建，并初始化“复制”菜单项
MenuItem cutItem = new MenuItem("剪切");    //创建，并初始化“剪切”菜单项
MenuItem pasteItem = new MenuItem("粘贴");  //创建，并初始化“粘贴”菜单项
String fileName = "NoName.txt";             //设置默认的文件名
Toolkit toolkit = Toolkit.getDefaultToolkit(); //获取默认工具包
Clipboard clipBoard = toolkit.getSystemClipboard(); //获取剪切板对象
//创建，并初始化打开文件对话框&保存文件对话框
private FileDialog openFileDialog = new FileDialog(this, "Open File", FileDialog.LOAD);
private FileDialog saveAsFileDialog = new FileDialog(this, "Sava File As", FileDialog.SAVE);
//实现 NoteBook 构造方法
public NoteBook(){
    setTitle("NotePad");                    //设置窗口的标题
    setFont(new Font("Times New Roman", Font.PLAIN, 12)); //设置字体
    setBackground(Color.white);             //设置背景色
    setSize(600, 400);                      //设置窗口大小
    fileMenu.add(newItem);                   //在菜单中添加“新建”菜单项
    fileMenu.add(openItem);                 //在菜单中添加“打开”菜单项
    fileMenu.addSeparator();                //在菜单中添加分割线
    fileMenu.add(saveItem);                 //在菜单中添加“保存”菜单项
    fileMenu.add(saveAsItem);               //在菜单中添加“另存”菜单项
    fileMenu.addSeparator();                //在菜单中添加分割线
    fileMenu.add(exitItem);                 //在菜单中添加“退出”菜单项
    editMenu.add(selectItem);                //在 editMenu 菜单中添加“选择”菜单项
    editMenu.addSeparator();                //在 editMenu 菜单中添加分割线
    editMenu.add(copyItem);                 //在 editMenu 菜单中添加“复制”菜单项
    editMenu.add(cutItem);                  //在 editMenu 菜单中添加“剪切”菜单项
    editMenu.add(pasteItem);                //在 editMenu 菜单中添加“粘贴”菜单项
    menuBar.add(fileMenu);                  //将菜单 fileMenu 添加到菜单项 menuBar
    menuBar.add(editMenu);                  //将菜单 editMenu 添加到菜单项 menuBar
    setMenuBar(menuBar);                   //为窗口添加菜单栏
    add(textArea);                          //在窗口中添加 textArea
    addWindowListener(new WindowAdapter){
```

```

        public void windowClosing(WindowEvent e){
            System.exit(0);           //程序退出
        }
    });
    newItem.addActionListener(this);           //为菜单项 newItem 添加监听器
    openItem.addActionListener(this);          //为菜单项 openItem 添加监听器
    saveItem.addActionListener(this);          //为菜单项 saveItem 添加监听器
    saveAsItem.addActionListener(this);        //为菜单项 saveAsItem 添加监听器
    exitItem.addActionListener(this);          //为菜单项 exitItem 添加监听器
    selectItem.addActionListener(this);        //为菜单项 selectItem 添加监听器
    copyItem.addActionListener(this);          //为菜单项 copyItem 添加监听器
    cutItem.addActionListener(this);           //为菜单项 cutItem 添加监听器
    pasteItem.addActionListener(this);         //为菜单项 pasteItem 添加监听器
}

```

构造方法首先设置窗口的标题、字体、背景色、窗口大小，然后在菜单中添加菜单项，并将菜单添加于菜单栏中，接着设置窗口的菜单项，最后为菜单项添加事件监听器。

12.6.2 事件处理方法

当用户为某个组件添加了一个事件监听器，必须定义特定的事件处理方法。当相应的事件发生时，就会调用该事件处理方法。

```

public void actionPerformed(ActionEvent e){
    Object eventSource = e.getSource();
    //判断事件源是那一个菜单项
    if(eventSource == newItem)           //设置文本区的文本内容
    {
        textArea.setText("");
    } else if(eventSource == openItem)   //当事件源为“打开”菜单项
    {
        openFileDialog.setVisible(true); //显示打开对话框
        fileName = openFileDialog.getDirectory()+openFileDialog.getFile(); //获得文件名字符串
        if(fileName != null)             //若文件名不为空
            readFile(fileName);          //读文件操作
    } else if(eventSource == saveItem)   //当事件源为“保存”菜单项
    {
        if(fileName != null)             //若文件名不为空
            writeFile(fileName);         //写文件操作
    } else if(eventSource == saveAsItem) //当事件源为“另存”菜单项
    {
        saveAsFileDialog.setVisible(true); //显示“另存”为对话框
        fileName = saveAsFileDialog.getDirectory()+saveAsFileDialog.getFile(); //获得文件名字符串
        if(fileName != null)             //若文件名不为空
            writeFile(fileName);         //写文件操作
    } else if(eventSource == selectItem) //当事件源为“选择”菜单项
    {
        textArea.selectAll();           //全选
    } else if(eventSource == copyItem)   //当事件源为“复制”菜单项
    {
        String text = textArea.getSelectedText(); //复制
    }
}

```

```

        StringSelection selection = new StringSelection(text); //创建能传输指定 String 的
Transferable
        clipBoard.setContents(selection, null); //将剪贴板的当前内容设置到指定的
transferable 对象
    }else if(eventSource == cutItem) //当事件源为“剪切”菜单项
    {
        String text = textArea.getSelectedText(); //获取文本中所选文本
        StringSelection selection = new StringSelection(text); //创建能传输指定 String 的
Transferable
        clipBoard.setContents(selection, null); //将剪贴板的当前内容设置到指定的
transferable 对象
        //用空字符串代替指定起始位置的文本
        textArea.replaceRange("",textArea.getSelectionStart(),textArea.getSelectionEnd());
    }else if(eventSource == pasteItem) //当事件源为“粘贴”菜单项
    {
        Transferable contents = clipBoard.getContents(this);//返回表示剪贴板当前内容的
transferable 对象
        if(contents==null) return; //如果返回内容为空，返回该方法
        String text; //声明字符串对象 text
        text = ""; //初始化 text 为空字符串
        try{
            //返回一个对象，表示将要被传输的数据
            text=(String)contents.getTransferData(DataFlavor.stringFlavor);
        }catch(Exception exception){ //捕获异常 Exception
        }
        //用 text 字符串代替指定起始位置的文本
        textArea.replaceRange(text,textArea.getSelectionStart(),textArea.getSelectionEnd());
    }else if(eventSource == exitItem){ //当事件源为“退出”菜单项
        System.exit(0); //退出程序
    }
}

```

由于这个案例为菜单项添加了 ActionListener 接口，所以必须定义方法 actionPerformed。actionPerformed 方法的参数为 ActionEvent 的对象 e，调用 e.getSource()方法返回事件源，程序将会根据事件源做出相应的处理。

12.6.3 文件读写方法

记事本必须具有读写文件的能力，因此必须提供相应的方法。readFile 方法用于文件读取操作，writeFile 用于保存文件操作。本程序中读写文件使用 FileReader 和 FileWriter 对象。

```

//读文件
public void readFile(String fileName){
    try{
        File file = new File(fileName); //创建，初始化 File 对象 file
        FileReader readIn = new FileReader(file); //由 file 对象创建 FileReader 对象
        int size = (int)file.length(); //返回文件长度
        int charsRead = 0; //创建，初始化整型数据 charsRead
        char[] content = new char[size]; //创建字符类型数组
        while(readIn.ready()) //循环读流数据
            charsRead += readIn.read(content, charsRead, size-charsRead); //读出文件字符
    }
}

```

```

数据
        readIn.close(); //关闭 readIn 对象
        textArea.setText(new String(content,0,charsRead)); //设置文本区内容
    }catch(IOException e){
        System.out.println("Error opening File"); //输出字符串信息
    }
}
//写文件
public void writeFile(String fileName){
    try{
        File file = new File(fileName); //创建, 初始化 File 对象 file
        FileWriter writeOut = new FileWriter(file); //由 file 对象创建 FileWriter 对象
        writeOut.write(textArea.getText()); //将文本区内容写入文件
        writeOut.close(); //关闭 writeOut
    }catch(IOException e){ //捕获异常
        System.out.println("Error writing file"); //输出字符串信息
    }
}
}

```

12.6.4 主方法

本程序的主方法先创建一个 NoteBook 对象, 并通过 Toolkit.getDefaultToolkit().getScreenSize()方法获取屏幕大小, 计算并设置窗口的显示位置。

```

public static void main(String[] args) {
    Frame frame = new NoteBook(); //创建, 并初始化 NoteBook 窗口
    Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize(); //获取屏幕大小
    Dimension frameSize = frame.getSize(); //获取窗口大小
    if(frameSize.height > screenSize.height){ //若窗口高度大于屏幕高度
        frameSize.height = screenSize.height; //设置窗口大小高
    }
    if(frameSize.width > screenSize.width){ //若窗口宽大于屏幕宽
        frameSize.width = screenSize.width; //设置窗口大小的宽
    }
    //将组件移动到指定位置
    frame.setLocation((screenSize.width - frameSize.width)/2, (screenSize.height -
frameSize.height)/2);
    frame.setVisible(true); //显示窗口
}
}

```

编写完程序后, 使用 javac 命令编译该文件产生 class 文件, 然后使用 java 命令运行该 class 文件, 运行结果如图 12-14 所示。

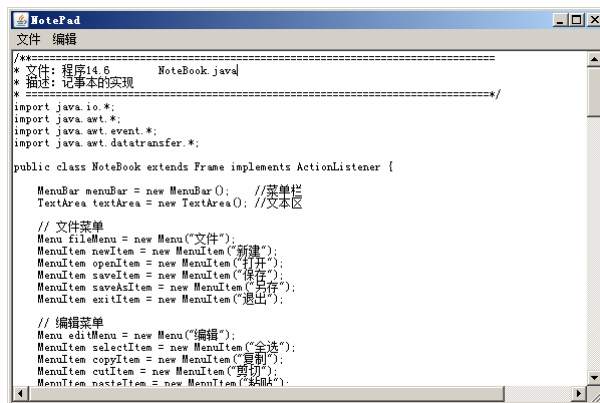


图 12-14 NoteBook.java 运行结果

NoteBook 类是一个记事本的基本类, 该类的对象具有记事本的基本功能, 包括打开文件、保存文件以及文本的复制、粘贴、删除等。该程序主要包括为窗口添加菜单栏、在菜单栏中添加菜单以及为菜单栏添加菜单项, 并且为每一个菜单项添加相应的监听器完成相应的操作。另外, 还包括文件内容的读取和保存, 这部分内容将在第 16 章详细讲解。

12.7 小结

本章主要介绍了 Java 事件处理模型和事件处理相关的基础知识, 这些内容在图形用户界面设计中应用比较广泛。事件处理方式通常包括实现接口以及内部类和匿名类的方式, 本章最后提供了一个涉及知识点比较集中的案例供读者参考。