第 4 章　事件模型与事件处理

【本章导读】

本章首先介绍事件处理的由来，并描述基于窗口的事件驱动程序的流程，接着详细阐述 AWT 事件处理模型，主要涉及三类对象：事件源、监听器和事件处理方法，并说明事件处理机制，具体介绍事件类、事件监听器、AWT 事件及其相应的监听器接口、事件适配器和常用的四种对事件的响应，最后介绍开发一个基于 Java 平台事件驱动模型的记事本的案例。

【本章要点】

- 事件处理的流程和机制
- 事件类
- 事件监听器
- AWT 事件及其相应的监听器接口
- 事件适配器
- 常用的四种对事件的响应：将对事件的响应信息显示在 Applet 的状态栏上；将对事件响应信息显示在弹出窗口上；打开另一个对话窗口；显示另一个窗口界面。

4.1　事件处理概述

在 Java 的 GUI 编程中，如何处理鼠标及键盘的点击及输入等动作事件是非常重要的。只有实现了事件编程，才能算是真正实现了 GUI 编程。在 Java 中，事件表示程序和用户之间的交互，例如，在文本框中输入，在列表框或组合框中选择，单击复选框和单选框，单击按钮等。而对事件的响应，对用户的交互或者对事件的处理是事件处理程序完成的。当事件发生时，系统会自动捕捉到这一事件，创建表示动作的事件对象并把它们分派给程序内的事件处理程序代码。这种代码确定了如何处理此事件以使用户得到相应的回答。

对于什么事件要处理，也就是要如何响应，对什么样的事件不必处理，是由程序员在编写代码时确定的。在用户界面中，文本框、列表框、组合框、复选框和单选按钮都是数据输入接口组件，一般不需要做出响应，而对按钮必须做出响应。对我们来说，最重要的是要知道采用什么样的事件处理机制。图 4-1 描述了基于窗口的事件驱动程序的流程。

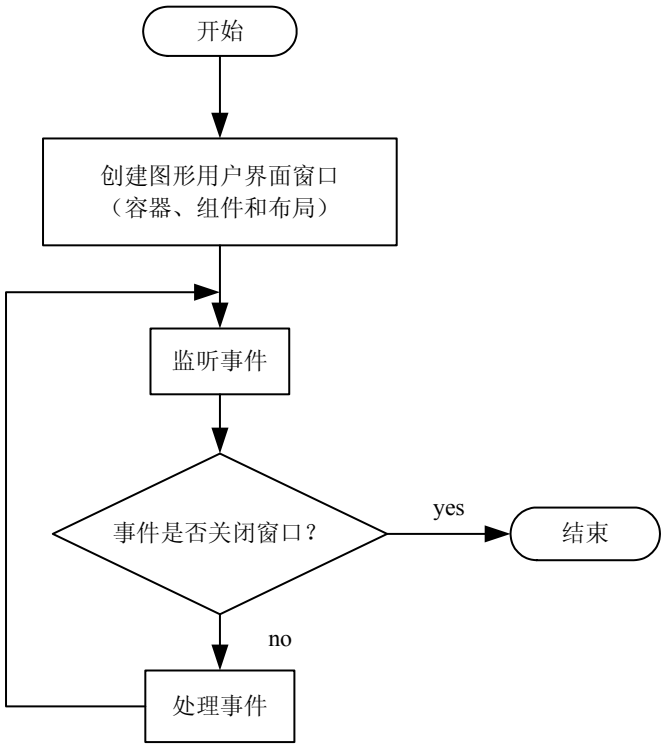


图 4-1　基于窗口的事件驱动程序的流程

4.2　AWT 事件处理模型

上一章中的主要内容是如何放置各种组件，使图形界面更加丰富多彩，但是还不能响应用户的任何操作，要能够让图形界面接收用户的操作，就必须给各个组件加上事件处理机制。在事件处理的过程中，主要涉及三类对象。

- 事件源：能够接收外部事件的源体，产生事件的地方（单击鼠标，单击按钮、选择项目等产生动作的对象）。
- 监听器：能够接收事件源通知的对象。监听程序必须注册一个事件源，才能接收这个事件，这个过程是自动的，监听程序必须实现接受和处理这个事件的方法。
- 事件处理方法：用于处理事件的对象，事件源产生一个事件，并把这个事件发送到一个或多个监听程序，监听程序只是等待这个事件并处理它，然后返回。即程序把事件的处理“委托”给一段“代码”。这段代码就是事件处理方法，也叫事件处理程序。

例如，如果用户用鼠标单击按钮对象 button，则该按钮 button 就是事件源，而 Java 运行时系统会生成 ActionEvent 类的对象 actionE，该对象中描述了该单击事件发生时的一些信息，然后，事件处理者对象将接收由 java 运行时系统传递过来的事件对象 actionE 并进行相应的处理。

Java 最常用的可视化编程当属 Java Swing 技术, Java Swing 为开发者提供了很多现成的组件，如按钮（JButton）、单选按钮（JRadioButton）、菜单（JMenu）、菜单项（JMenuItem）等。为了管理用户与组成程序图形用户界面的组件间的交互，必须理解在 Java 中如何处理事

件，如图 4-2 所示。

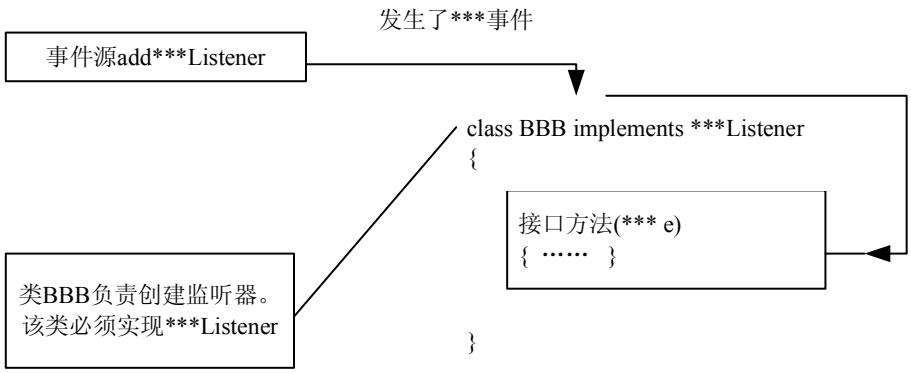


图 4-2　事件的处理机制

下面是 Java 中事件模型处理机制的描述：

- 监听器对象是一个实现了专门监听接口的类的实例。
- 可以向事件源注册相应事件的监听器。
- 当事件发生时，事件源能够把事件对象发给已注册的相应监听器。
- 监听器对象中的事件处理方法会使用事件中的信息决定对事情的反应。
- 简而言之就是当监听器监听到事件源发生某事时将做出相应的反应。

由于同一个事件源上可能发生多种事件，因此 Java 采取了授权处理机制（Delegation Model），事件源可以把在其自身所有可能发生的事件分别授权给不同的事件处理者来处理。比如在 Canvas 对象上既可能发生鼠标事件，也可能发生键盘事件，该 Canvas 对象就可以授权给事件处理者一来处理鼠标事件，同时授权给事件处理者二来处理键盘事件。有时也将事件处理者称为监听器，主要原因也在于监听器时刻监听着事件源上发生的所有事件类型，一旦该事件类型与自己所负责处理的事件类型一致，就马上进行处理。授权模型把事件的处理委托给外部的处理实体进行处理，实现了将事件源和监听器分开的机制。事件处理者（监听器）通常是一个类，该类如果要能够处理某种类型的事件，就必须实现与该事件类型相对的接口。例 4-1 中类 ButtonEventDemo 之所以能够处理 ActionEvent 事件，原因在于它实现了与 ActionEvent 事件对应的接口 ActionListener。每个事件类都有一个与之相对应的接口。

【例 4-1】程序实现一个简单的单击按钮改变页面背景颜色的界面。

```
import java.awt.*;
import java.awt.event.*;
public class ButtonEventDemo extends Frame implements ActionListener
{
    static ButtonEventDemo frm=new ButtonEventDemo();
    static Button btn=new Button("Click Me");
    public static void main(String args[])
    {
        btn.addActionListener(frm); // 把 frm 向 btn 注册
        frm.setLayout(new FlowLayout());
        frm.setTitle("Action Event");
```

```
        frm.setSize(200,150);
        frm.add(btn);
        frm.setVisible(true);
    }
    public void actionPerformed(ActionEvent e) // 事件发生的处理操作
    {
        frm.setBackground(Color.yellow);
    }
}
```

程序运行结果如图 4-3 所示（左边为初始界面，右边为单击按钮后的界面）。

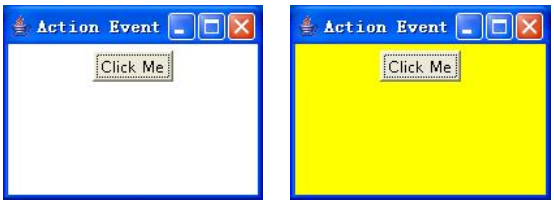


图 4-3　ButtonEventDemo.java 的运行结果

使用授权处理模型进行事件处理的一般方法归纳如下：

（1）对于某种类型的事件 XXXEvent，要想接收并处理这类事件，必须定义相应的事件监听器类，该类需要实现与该事件相对应的接口 XXXListener。

（2）事件源实例化以后，必须进行授权，注册该类事件的监听器，使用 addXXXListener（XXXListener）方法注册监听器。

4.2.1　事件类

与 AWT 有关的所有事件类都由 java.awt.AWTEvent 类派生，它也是 EventObject 类的子类。AWT 事件共有 10 类，可以归为两大类：低级事件和高级事件，如图 4-4 所示。

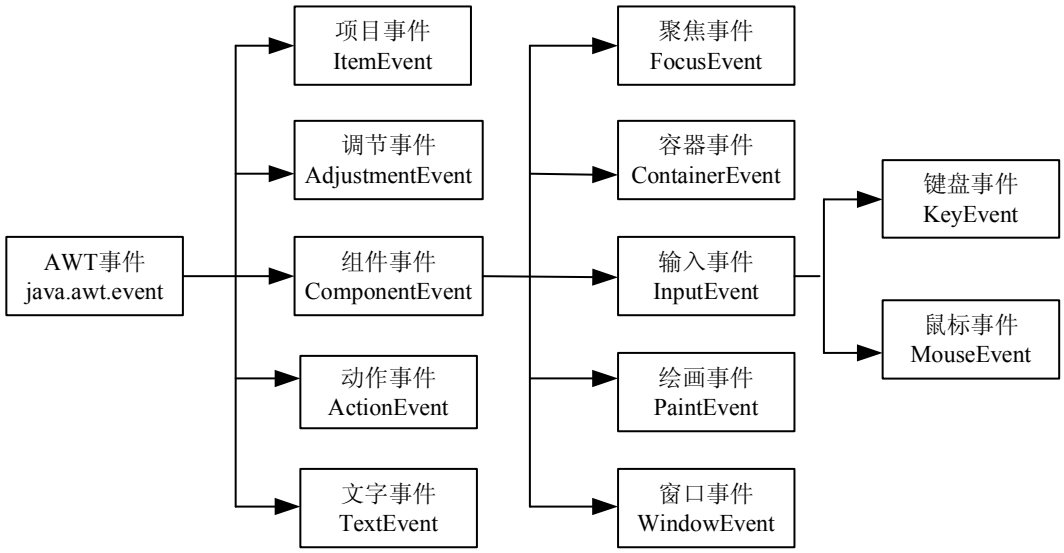


图 4-4　事件类的层次结构图

java.util.EventObject 类是所有事件对象的基础父类，所有事件都是由它派生出来的。AWT 的相关事件继承于 java.awt.AWTEvent 类，这些 AWT 事件分为两大类：低级事件和高级事件，低级事件是指基于组件和容器的事件，当一个组件上发生事件，如鼠标的进入、单击、拖放等，或组件的窗口开关等，都会触发组件事件。高级事件是基于语义的事件，它可以不和特定的动作相关联，而依赖于触发此事件的类，如在 TextField 中按 Enter 键会触发 ActionEvent 事件，滑动滚动条会触发 AdjustmentEvent 事件，或是选中项目列表的某一条就会触发 ItemEvent 事件。

1. 低级事件

ComponentEvent：组件事件，组件尺寸的变化、移动。

ContainerEvent：容器事件，组件增加、移动。

WindowEvent：窗口事件，关闭窗口、窗口闭合、图标化。

FocusEvent：焦点事件，焦点的获得和丢失。

KeyEvent：键盘事件，键按下、释放。

MouseEvent：鼠标事件，鼠标单击、移动。

2. 高级事件（语义事件）

ActionEvent：动作事件，按钮按下，TextField 中按 Enter 键。

AdjustmentEvent：调节事件，在滚动条上移动滑块以调节数值。

ItemEvent：项目事件，选择项目，例如选择“项目改变”。

TextEvent：文本事件，文本对象改变。

4.2.2　事件监听器

每类事件都有对应的事件监听器，监听器是接口，根据动作来定义方法。

例如，与键盘事件 KeyEvent 相对应的接口是：

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

注意到在本接口中有三个方法，Java 运行时系统何时调用哪个方法呢？其实根据这三个方法的方法名就能够知道应该是什么时候调用哪个方法执行了。当按键刚按下去时，将调用 keyPressed()方法执行，当按键抬起来时，将调用 keyReleased()方法执行，当键敲击一次时，将调用 keyTyped()方法执行。

又例如窗口事件接口：

```
public interface WindowListener extends EventListener{
    public void windowClosing(WindowEvent e);
    //把退出窗口的语句写在本方法中
    public void windowOpened(WindowEvent e);
    //窗口打开时调用
    public void windowIconified(WindowEvent e);
    //窗口图标化时调用
    public void windowDeiconified(WindowEvent e);
```

```
        //窗口非图标化时调用
        public void windowClosed(WindowEvent e);
        //窗口关闭时调用
        public void windowActivated(WindowEvent e);
        //窗口激活时调用
        public void windowDeactivated(WindowEvent e);
        //窗口非激活时调用
}
```

AWT 的组件类中提供注册和注销监听器的方法。

（1）注册监听器。

```
public void add<ListenerType> (<ListenerType>listener);
```

（2）注销监听器。

```
public void remove<ListenerType> (<ListenerType>listener);
```

例如 Button 类：

```
public class Button extends Component {
        ……
        public synchronized void addActionListener(ActionListener l);
        public synchronized void removeActionListener(ActionListener l);
……}
```

4.2.3 AWT 事件及其相应的监听器接口

表 4-1 列出了所有 AWT 事件及其相应的监听器接口，共包括 10 类事件，11 个接口。下面这张表应能牢牢记住。

表 4-1 AWT 事件及其相应的监听器接口

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)

续表

事件类别	描述信息	接口名	方法
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)

【例 4-2】在例 4-1 的基础上编程实现事件处理模型的应用。

```
import java.awt.*;
import java.awt.event.*;        //事件包
import javax.swing.*;
class EventDemo extends JFrame
{
    public EventDemo(String title)
    {
      super("Event Demo");
      addWindowListener(new MywinEvent(this));   //注册窗口监听程序
      /*new MywinEvent()继承了 WindowAdapter 适配器
      WindowAdapter 来实现 WindowListener 接口*/
    }
    JButton jbt;
    Color cl=Color.RED;
    void setMyMenu()
    {
        JMenuBar jmb=new JMenuBar();
        JMenu jmn=new JMenu("文件");
        JMenuItem jmi=new JMenuItem("退出  " );
        jmn.add(jmi);
        jmb.add(jmn);
        setJMenuBar(jmb);
        jmi.addActionListener(new MyactionEvent(this));     //注册动作事件
```

```
    }
    void setMybutton()
    {
        Container ct=super.getContentPane();
        jbt=new JButton("单击");
        jbt.setBackground(cl);
        ct.add(jbt,BorderLayout.CENTER);
        jbt.addMouseListener(new MymouseEvent(this));     //注册鼠标事件
    }
    public static void main(String args[])
    {
      EventDemo ed=new EventDemo("事件--演示");
      ed.setMyMenu();
      ed.setMybutton();
      ed.pack();
      ed.setVisible(true);
    }
}
//定义窗口监听程序
class MywinEvent extends WindowAdapter
{
    EventDemo mywin;
    public MywinEvent(EventDemo mywin)
    {
      this.mywin=mywin;
    }
    public void windowClosing(WindowEvent we)
    {
     System.exit(0);
    }
}
//定义动作监听程序
class MyactionEvent implements ActionListener
{
  EventDemo mywin;
  public MyactionEvent(EventDemo mywin)
  {
    this.mywin=mywin;
   }
  public void actionPerformed(ActionEvent ae)
  {
   System.exit(0);
    }
  }
//定义鼠标监听程序
class MymouseEvent implements MouseListener
```

```
{
   EventDemo mywin;
   public MymouseEvent(EventDemo mywin)
   {
      this.mywin=mywin;
      }
      public void mouseClicked(MouseEvent me)
     {
         if(me.getSource()==mywin.jbt)
         if(mywin.jbt.getBackground()!=Color.BLUE)
         {
            mywin.jbt.setBackground(Color.BLUE);
            mywin.repaint();
            }
          else
         {
            mywin.jbt.setBackground(Color.RED);
            mywin.repaint();
          }
      }
     public void mouseEntered(MouseEvent me){} //对其不感兴趣的方法可以方法体为空
     public void mouseExited(MouseEvent me){}
     public void mousePressed(MouseEvent me){}
     public void mouseReleased(MouseEvent me){}
}
```

程序的运行结果如图 4-5 所示（左边为初始界面，右边为单击按钮后的结果）。

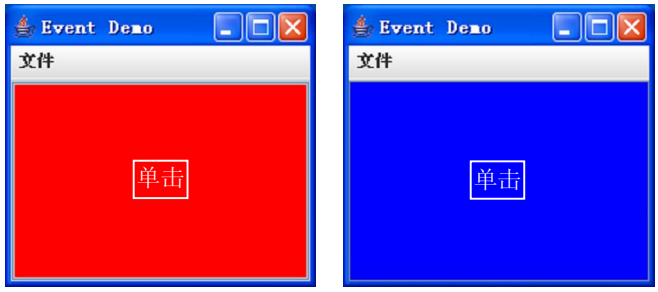


图 4-5　EventDemo.java 的运行结果

说明：Java 的事件模型结构基本上是这样的：继承适配器，可以重载需要的方法，一个、两个、更多个都可以。但是，实现接口必须实现这个接口提供的所有方法，哪怕是空方法都必须实现。

Java 语言类的层次非常分明，因而只支持单继承，为了实现多重继承的能力，Java 用接口来实现，一个类可以实现多个接口，这种机制比多重继承具有更简单、更灵活、更强的功能。在 AWT 中就经常用到声明和实现多个接口。记住无论实现了几个接口，接口中已定义的方法必须一一实现，如果对某事件不感兴趣，可以不具体实现其方法，而用空的方法体来代替。但必须所有方法都要写上。

4.2.4 事件适配器

Java 语言为一些 Listener 接口提供了适配器（Adapter）类。可以通过继承事件所对应的 Adapter 类，重写需要的方法，无关方法不用实现。事件适配器提供了一种简单的实现监听器的手段，可以缩短程序代码。但是，由于 Java 的单一继承机制，当需要多种监听器或此类已有父类时，就无法采用事件适配器了。

1. 事件适配器 EventAdapter

下例中采用了鼠标适配器：

```
import java.awt.*;
import java.awt.event.*;
public class MouseClickHandler extends MouseAdaper{
    public void mouseClicked(MouseEvent e) //只实现需要的方法
     { ……}
}
```

java.awt.event 包中定义的事件适配器类包括以下几个：

- 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）。

2. 用内部类实现事件处理

内部类（inner class）是被定义于另一个类中的类，使用内部类的主要原因是：

- 一个内部类的对象可访问外部类的成员方法和变量，包括私有的成员。
- 实现事件监听器时，采用内部类、匿名类编程非常容易实现其功能。
- 编写事件驱动程序，内部类很方便。

能够应用内部类的地方往往是在 AWT 的事件处理机制中。

【例 4-3】编写程序实现内部类的例子。

```
import java.awt.*;
import java.awt.event.*;
public class InnerDemo
{
    private Frame f;
    private TextField tf;
    public InnerDemo()
    {
        f=new Frame("Inner Demo");
        tf=new TextField(30);
    }
    public void loginFrame()
    {
```

```
        Label label=new Label("Clike and drag the mouse");
        f.add(label,BorderLayout.NORTH);
        f.add(tf,BorderLayout.SOUTH);
        f.addMouseMotionListener(new MyMouseMotionListener());    /*参数为内部类对象*/
        f.setSize(200,200);
        f.setVisible(true);
    }
    class MyMouseMotionListener extends MouseMotionAdapter
    {   /*内部类开始*/
        public void mouseDragged(MouseEvent e)
        {
            String s="Mouse Gragging: x="+e.getX()+","+"Y="+e.getY();
            tf.setText(s);
        }
    };
    public static void main(String[] args)
    {
        InnerDemo obj=new InnerDemo();
        obj.loginFrame();
    }
}
```

程序的运行结果如图 4-6 所示（左边为初始界面，右边为鼠标移动的记录界面）。

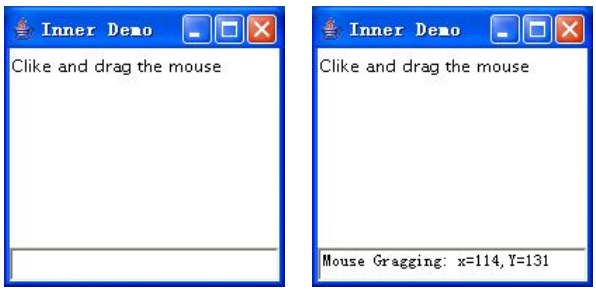


图 4-6　InnerDemo.java 的运行结果

3. 匿名类（Anonymous Class）

当一个内部类的类声明只是在创建此类对象时用了一次，而且要产生的新类需继承于一个已有的父类或实现一个接口，才能考虑用匿名类，由于匿名类本身无名，因此它也就不存在构造方法，它需要显式地调用一个无参的父类的构造方法，并且重写父类的方法。所谓的匿名就是该类连名字都没有，只是显式地调用一个无参的父类的构造方法。

【例 4-4】程序实现一个匿名类的例子。

```
import java.awt.*;
import java.awt.event.*;
public class AnonymouseDemo
{
    private Frame f;
    private TextField tf;
```

```
    public AnonymouseDemo()
    {
        f=new Frame("Anony Demo");
        tf=new TextField(30);
    }
    public void loginFrame()
    {
        Label label=new Label("click and drag the mouse");
        f.add(label,BorderLayout.NORTH);
        f.add(tf,BorderLayout.SOUTH);
        f.addMouseMotionListener(new MouseMotionAdapter(){
        public void mouseDragged(MouseEvent e)
        {
            String s="Mouse Gragging: x="+e.getX()+","+"Y="+e.getY();
            tf.setText(s);
        }
    });
        f.setSize(200,200);
        f.setVisible(true);
        }
    public static void main(String[] args)
    {
        AnonymouseDemo obj=new AnonymouseDemo();
        obj.loginFrame();
    }
}
```

程序的运行结果如图 4-7 所示（左边为初始界面，右边为鼠标移动的记录界面）。

图 4-7　AnonymouseDemo.java 的运行结果

4.2.5　对事件的响应

在事件编程中经常会遇到弹出另一个窗口、对话框或者切换到另一个窗口。下面介绍常用的四种对事件的响应方法：

- 将对事件的响应信息显示在 Applet 的状态栏上。
- 将对事件的响应信息显示在弹出窗口上。
- 打开另一个对话窗口。

- 显示另一个窗口界面。

1. 在 Applet 的状态栏上显示信息

状态栏在窗口的下方，如 Internet 浏览器、AppletViewer 程序查看器。状态栏是用来显示信息的，可以将对事件响应的信息显示在 Applet 的状态栏中。java.swing.JApplet 类是 java.applet.Applet 类的子类。java.applet.Applet 类有两种方法实现在状态栏上显示信息：

- AppletContext getAppletContext()：取决于 Applet 的上下文，允许 Applet 询问和影响其运行环境。
- void showStatus(String msg)：将字符串参数 msg 显示在状态窗口上。

【例 4-5】编写程序实现在 Applet 的状态栏上显示信息。

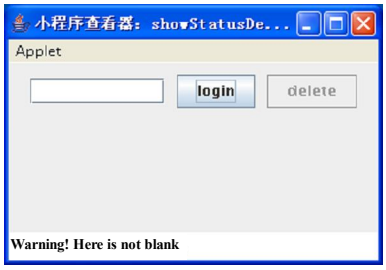
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class showStatusDemo extends JApplet
{
    JPanel p;
    JButton b1,b2;
    JTextField tf;

    public void init()
    {
        p=new JPanel();
        getContentPane().add(p);
        FlowLayout fl=new FlowLayout(FlowLayout.CENTER,10,10);
        p.setLayout(fl);
        tf=new JTextField(10);
        b1=new JButton("login");
        b2=new JButton("delete");
        b2.setEnabled(false);
        p.add(tf);
        p.add(b1);
        p.add(b2);
        ButtonEvent bObj=new ButtonEvent();
        b1.addActionListener(bObj);
        b2.addActionListener(bObj);
    }
    class ButtonEvent implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            Object obj=e.getSource();
            if(obj==b1)
            {
                String s=tf.getText();
                if(s.length()==0)
                {
```

```
                    getAppletContext().showStatus("Warning! Here is not blank.");
                    return;
                }
                getAppletContext().showStatus("Inside the textField is      "+s);
                b1.setEnabled(false);
                b2.setEnabled(true);
            }
            if(obj==b2)
            {
                b2.setEnabled(false);
                b1.setEnabled(true);
                tf.setText("");
                getAppletContext().showStatus("Click the Delete, delete the TextField!");
            }
        }
    }
}
```

程序的运行结果如图 4-8 所示。图 4-8（a）是初始界面，图 4-8（b）是单击 login 按钮后的界面，图 4-8（c）是在文本框中输入 sunpine 并单击 login 按钮后的界面，图 4-8（d）单击 delete 按钮后的界面。

（a）初始界面



（b）单击 login 按钮后的界面

（c）输入数据，单击按钮后的界面

（d）单击 delete 按钮后的界面

图 4-8　运行结果

2. 弹出式窗口

JOptionPane 用于弹出要求用户提供值或向其发出通知的标准对话框。JOptionPane 类的构造方法有：

- JOptionPane()：创建一个带有测试消息的 JOptionPane。

- JOptionPane(Object message)：创建一个显示消息的 JOptionPane 的实例，使其使用 UI 提供的普通消息类型和默认选项。
- JOptionPane(Object message, int messageType)：创建一个显示消息的 JOptionPane 的实例，使其具有指定的消息类型和默认选项。
- JOptionPane(Object message, int messageType, int optionType)：创建一个显示消息的 JOptionPane 的实例，使其具有指定的消息类型和选项类型。
- JOptionPane(Object message, int messageType, int optionType, Icon icon)：创建一个显示消息的 JOptionPane 的实例，使其具有指定的消息类型、选项类型和图标。
- JOptionPane(Object message, int messageType, int optionType, Icon icon, Object[] options)：创建一个显示消息的 JOptionPane 的实例，使其具有指定的消息类型、选项类型、图标和选项。
- JOptionPane(Object message, int messageType, int optionType, Icon icon, Object[] options, Object initialValue)：在指定最初选择的选项的前提下，创建一个显示消息的 JOptionPane 的实例，使其具有指定的消息类型、选项类型、图标和选项。
- JOptionPane 类的方法都是使用表 4-2 中静态 showXxxDialog 方法之一来实现的。

表 4-2　JOptionPane 类的实现方法

方法	功能描述
public void showConfirmDialog	询问一个确认问题，如 yes、no、cancel
public void showInputDialog	提示要求某些输入
public void showMessageDialog	告知用户某事已发生
public void showOptionDialog	上述三项的综合运用

这些方法的参数有：

（1）parentComponent：定义该对话框的父对话框的 Component。通过两种方式使用此参数：包含它的 Frame 可以用作对话框的父 Frame，在对话框的位置使用其屏幕坐标。一般情况下，将对话框紧靠组件置于其之下。此参数可以为 null，在这种情况下，默认的 Frame 用作父级，并且对话框将居中位于屏幕上。

（2）message。放置对话框中的描述消息。在最常见的应用中，message 就是一个 String 或 String 常量。此参数的类型实际上是 Object。其解释依赖于其类型：

- Object[]：对象数组被解释为在纵向堆栈中排列的一系列 message（每个对象一个）。解释是递归式的，即根据其类型解释数组中的每个对象。
- Component：该组件在对话框中显示。
- Icon：包装在 JLabel 中并在对话框中显示。该对象通过调用其 toString 方法被转换为 String。
- messageType：定义 message 的样式。布局管理器根据此值对对话框进行不同的布置，并且通常提供默认图标。可能的值为：
  - ERROR_MESSAGE
  - INFORMATION_MESSAGE

- WARNING_MESSAGE
- QUESTION_MESSAGE
- PLAIN_MESSAGE

（3）optionType：在对话框的底部显示的选项按钮的集合：

- DEFAULT_OPTION
- YES_NO_OPTION
- YES_NO_CANCEL_OPTION
- OK_CANCEL_OPTION

（4）options：使用 options 参数可以提供想使用的任何按钮。在对话框底部显示选项按钮集合的更详细描述。options 参数的常规值是 String 数组，但是参数类型是 Object 数组。根据对象的以下类型为每个对象创建一个按钮：

- Component：直接添加到按钮行中。
- Icon：创建的按钮以此图标作为其标签。

（5）Icon：对话框中的装饰性图标。图标的默认值由 messageType 参数确定。

（6）Title：对话框的标题。

（7）initialValue：默认选择（输入值）。当选择更改时，调用生成 PropertyChangeEvent 的 setValue 方法。如果已为所有输入 setWantsInput 配置了 JOptionPane，还可以监听绑定属性 JOptionPane.INPUT_VALUE_PROPERTY，以确定何时用户输入或选择了值。

当其中一个 showXxxDialog 方法返回整数时，可能的值为：YES_OPTION、NO_OPTION、CANCEL_OPTION、OK_OPTION 及 CLOSED_OPTION。

举例：

- 显示一个错误对话框，该对话框显示的 message 为'alert'：

```
JOptionPane.showMessageDialog(null, "alert", "alert", JOptionPane.ERROR_MESSAGE);
```

- 显示一个内部信息对话框，其 message 为'information'：

```
JOptionPane.showInternalMessageDialog(frame,
        "information","information",JOptionPane.INFORMATION_MESSAGE);
```

- 显示一个信息面板，其 options 为"yes/no"，message 为"choose one"：

```
JOptionPane.showConfirmDialog(null, "choose one", "choose one", JOptionPane.YES_NO_OPTION);
```

- 显示一个内部信息对话框，其 options 为"yes/no/cancel"，message 为"please choose one"，并具有 title 信息：

```
JOptionPane.showInternalConfirmDialog(frame, "please choose one", "information",
        JOptionPane.YES_NO_CANCEL_OPTION, JOptionPane.INFORMATION_MESSAGE);
```

- 显示一个警告对话框，其 options 为 OK、CANCEL，title 为"Warning"，message 为"Click OK to continue"：

```
Object[] options = { "OK", "CANCEL" };
JOptionPane.showOptionDialog(null, "Click OK to continue", "Warning", JOptionPane.DEFAULT_OPTION,
        JOptionPane.WARNING_MESSAGE, null, options, options[0]);
```

- 显示一个要求用户键入 String 的对话框：

```
String inputValue = JOptionPane.showInputDialog("Please input a value");
```

- 显示一个要求用户选择 String 的对话框：

```
        Object[] possibleValues = { "First", "Second", "Third" };
        Object selectedValue = JOptionPane.showInputDialog(null, "Choose one", "Input",
            JOptionPane.INFORMATION_MESSAGE, null,possibleValues, possibleValues[0]);
```

【例 4-6】编写程序实现一个弹出式窗口。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
public class OptionPaneDemo
{
    public static void main(String [] args)
    {
        DialogFrame df=new DialogFrame();
        df.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        df.setVisible(true);
    }
}
class DialogFrame extends JFrame
{
    public DialogFrame()
    {
        setTitle("Dailog Test");
        setSize(new Dimension(200,200));
        Container cc=this.getContentPane();
        JPanel btnPanel=new JPanel();
        JButton logoutBtn=new JButton("退出");
        btnPanel.add(logoutBtn);
        logoutBtn.addActionListener(new LogoutAction());
        cc.add(btnPanel,BorderLayout.SOUTH);
    }
    private class LogoutAction implements ActionListener
    {
        public void actionPerformed(ActionEvent ae)
        {
            int select=JOptionPane.showConfirmDialog(DialogFrame.this, "Are You Sure?","Logout",
JOptionPane.OK_CANCEL_OPTION,JOptionPane.WARNING_MESSAGE);
            if(select==JOptionPane.OK_OPTION)
                System.exit(0);
        }
    }
}
```

程序的运行结果如图 4-9 所示（左边为初始界面，右边为单击按钮后的弹出窗口）。

图 4-9　OptionPaneDemo.java 的运行结果

3. 对话窗口

JDialog 类和 Frame 类一样都是 Window 的子类，它必须依赖于某个窗口或组件。当其所依赖的窗口或组件消失时，它会随之消失；当其所依赖的窗口或组件可见时，对话框又会自动恢复。因此，可以将其理解为可以反复使用的资源。该类是一个容器类，可以像创建 JFrame 或 Applet 一样来创建，可以在对话框窗口上添加组件，设置布局管理器，设计事件处理等。其类的构造方法有：

- JDialog()：创建一个没有标题并且没有指定 Frame 所有者的无模式对话框。
- JDialog(Dialog owner)：创建一个没有标题但将指定的 Dialog 作为其所有者的无模式对话框。
- JDialog(Dialog owner, boolean modal)：创建一个具有指定所有者 Dialog 和模式的对话框。
- JDialog(Dialog owner, String title)：创建一个具有指定标题和指定所有者对话框的无模式对话框。
- JDialog(Dialog owner, String title, boolean modal)：创建一个具有指定标题、模式和指定所有者 Dialog 的对话框。
- JDialog(Dialog owner, String title, boolean modal, GraphicsConfiguration gc)：创建一个具有指定标题、所有者 Dialog、模式和 GraphicsConfiguration 的对话框。
- JDialog(Frame owner)：创建一个没有标题但将指定的 Frame 作为其所有者的无模式对话框。
- JDialog(Frame owner, boolean modal)：创建一个具有指定所有者 Frame、模式和空标题的对话框。
- JDialog(Frame owner, String title)：创建一个具有指定标题和指定所有者窗体的无模式对话框。
- JDialog(Frame owner, String title, boolean modal)：创建一个具有指定标题、所有者 Frame 和模式的对话框。
- JDialog(Frame owner, String title, boolean modal, GraphicsConfiguration gc)：创建一个具有指定标题、所有者 Frame、模式和 GraphicsConfiguration 的对话框。
- JDialog(Window owner)：创建一个具有指定所有者 Window 和空标题的无模式对话框。
- JDialog(Window owner, Dialog.ModalityType modalityType)：创建一个具有指定所有者 Window、模式和空标题的对话框。

- JDialog(Window owner, String title)：创建一个具有指定标题和所有者 Window 的无模式对话框。
- JDialog(Window owner, String title, Dialog.ModalityType modalityType)：创建一个具有指定标题、所有者 Window 和模式的对话框。
- JDialog(Window owner, String title, Dialog.ModalityType modalityType, GraphicsConfiguration gc)：创建一个具有指定标题、所有者 Window、模式和 GraphicsConfiguration 的对话框。

其中 frame 类型的参数表示对话框的拥有者，boolean 类型的参数用于控制对话框的工作方式。如果为 true，则对话框为可视时，其他构件不能接受用户的输入，此时的对话框称为静态的；如果为 false，则对话框和所属窗口可以互相切换，彼此之间没有顺序上的联系。String 类型的参数作为对话框的标题，在构造对话框之后，就可以添加其他构件。

【例 4-7】编写程序实现一个 JDialog 的例子。

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class JDialogDemo extends JFrame implements ActionListener{
    public JDialogDemo(){
        Container contentPane=this.getContentPane();
        JButton jButton1=new JButton("显示对话框");
        jButton1.addActionListener(this);
        contentPane.add(jButton1);
        this.setTitle("JDialogDemo");
        this.setSize(300,300);
        this.setLocation(400,400);
        this.setVisible(true);
    }
    /* 响应窗体的按钮事件 */
    public void actionPerformed(ActionEvent e){
        if(e.getActionCommand().equals("显示对话框")){
            HelloDialog hw=new HelloDialog(this);
        }
    }
        /* 窗体按钮监听器 */
        class HelloDialog implements ActionListener{
            JDialog jDialog1=null; //创建一个空的对话框对象
            HelloDialog(JFrame jFrame){
    /* 初始化 jDialog1 指定对话框的拥有者为 jFrame，标题为“Dialog”，当对话框为可视时，其他构件不能接受用户的输入（静态对话框）*/
                jDialog1=new JDialog(jFrame,"Dialog",true);
                //创建一个按钮对象,该对象被添加到对话框中
                JButton jButton1=new JButton("关闭");
                jButton1.addActionListener(this);
                //将“关闭”按钮对象添加至对话框容器中
                jDialog1.getContentPane().add(jButton1);
```

```
            /* 设置对话框的初始大小 */
            jDialog1.setSize(80,80);
             /* 设置对话框初始显示在屏幕中的位置 */
            jDialog1.setLocation(450,450);
            /* 设置对话框为可见（前提是生成了 HelloDialog 对象）*/
            jDialog1.setVisible(true);
        }
        //响应对话框中的按钮事件
        public void actionPerformed(ActionEvent e){
            if(e.getActionCommand().equals("关闭")){
                // 以下语句等价于 jDialog1.setVisible(false);
                /* public void dispose()释放由此 Window、其子组件及其拥有的所有子组件所使用的所有本机屏幕资源。即这些 Component 的资源将被破坏，它们使用的所有内存都将返回到操作系统，并将它们标记为不可显示。 */
                jDialog1.dispose();
            }
        }
    }
    public static void main(String[] args){
        JDialogDemo test=new JDialogDemo();
    }
}
```

程序的运行结果如图 4-10 所示。图 4-10（a）为初始界面，图 4-10（b）为单击“显示对话框”控件后的界面。

（a）初始界面

（b）单击“显示对话框”后的界面

图 4-10 运行结果

4. 显示另一个窗口

在程序应用中经常需要执行一个命令时有很多操作步骤，往往是从一个窗口切换到另一个界面，或者从一个对话框切换到下一个对话框。这些窗口没有父子关系，每个窗口都是一个独立的界面，包括自己的组件、布局和事件。

【例 4-8】程序实现从一个窗口开始，执行操作后显示另一个窗口。

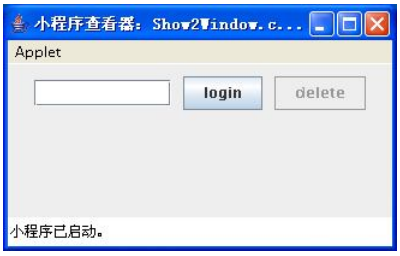
```
import javax.swing.*;
```

```
import java.awt.*;
import java.awt.event.*;
public class Show2Window extends JApplet
{
    JPanel p1,p2,p3;   //创建 3 个面板
    JButton b1,b2,b3,b4;
    JTextField tf;
    JList list;
    JScrollPane sp;
    FlowLayout fl;
    CardLayout cl;
    int i=0;
    String strList[]={"     ","     ","     ","     ","     ","     ","     ","     ","     ","     ","     "};
    public void init()
    {
    p1=new JPanel();
    getContentPane().add(p1);
   fl=new FlowLayout(FlowLayout.CENTER,10,10);
    tf=new JTextField(10);
    b1=new JButton("login");
   b2=new JButton("delete");
   b1.setEnabled(true);
   b2.setEnabled(false);
   list=new JList(strList);
   list.setEnabled(false);
   list.setFixedCellWidth(120);
   list.setVisibleRowCount(4);
   sp=new JScrollPane(list,JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,JScrollPane.HORIZONTAL_
SCROLLBAR_NEVER);
   b3=new JButton("edit");
   b4=new JButton("save");
   p2=new JPanel();
   p2.setLayout(fl);   //卡片窗口使用流布局管理器
   p2.add(tf);
   p2.add(b1);
   p2.add(b2);
   p3=new JPanel();
   p3.setLayout(fl);   //卡片窗口使用流布局管理器
   p3.add(sp);
   p3.add(b3);
   p3.add(b4);
   cl=new CardLayout();   //主面板使用卡片布局管理器
   p1.setLayout(cl);
   p1.add("Window1",p2);
   p1.add("Window2",p3);
   cl.show(p1,"Window1");
```

```
ButtonEvent bObj=new ButtonEvent();  //创建事件类的对象
b1.addActionListener(bObj);    //注册动作监听
b2.addActionListener(bObj);    //注册动作监听
b3.addActionListener(bObj);    //注册动作监听
b4.addActionListener(bObj);    //注册动作监听
}
class ButtonEvent implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        Object obj=e.getSource();
        if(obj==b1)
        {
            String s=tf.getText();
            if(s.length()==0)
            {
            getAppletContext().showStatus("Warning! Here is not blank.");
            return;
        }
        strList[i]=i+1+"."+tf.getText();
        cl.show(p1,"Window2");  //切换到第 2 个窗口
        getAppletContext().showStatus("Clike the login\"login\",Input the list.");
        b1.setEnabled(false);
        b2.setEnabled(true);
        }
        if(obj==b2)
        {
            b1.setEnabled(true);
            b2.setEnabled(false);
            tf.setText("");
            getAppletContext().showStatus("Click the Delete\"delete\", delete the TextField!");
        }
            if(obj==b3)
        {
            cl.show(p1,"Window1");  //切换到第 1 个窗口
            b1.setEnabled(true);
            b2.setEnabled(false);
            getAppletContext().showStatus("Click the edit\"edit\", edit the TextField!");
        }
            if(obj==b4)
        {
            cl.show(p1,"Window1");    //切换到第 1 个窗口
            b1.setEnabled(false);
            b2.setEnabled(true);
            getAppletContext().showStatus("Click the save\"save\", delete the TextField!");
            i++;
```

```
            }
        }
    }
}
```

程序的运行结果如图 4-11 所示。



（a）初始界面

（b）输入数据后，单击 login 的界面按钮

（c）单击 edit 的界面按钮

（d）输入数据后，单击 login 的界面按钮

（e）单击 save 按钮后的界面

（f）再次输入数据的界面

图 4-11　运行结果

实例一：开发一个基于 Java 平台事件驱动模型的记事本。如图 4-12 所示，记事本包括四大主菜单，其中“文件”包括新建文件、打开文件、保存文件、文件另存、退出等功能；“编辑”包括撤消、恢复、剪切、复制、粘贴、删除、全选、日期和时间等功能；“格式”包括自动换行等功能；“帮助”包括关于记事本（版权所有）等功能。

以下是对记事本主要功能的描述。

- 新建文件：如果记事本中有内容，并且没有保存，则单击“新建文件”后出现提示语“是否保存当前文件”，单击“是”则保存文件，单击“否”则将以前记事本中的内容去掉，如果已经保存则不出现提示语，直接将以前记事本中的内容去掉。流程图如图 4-13 所示。

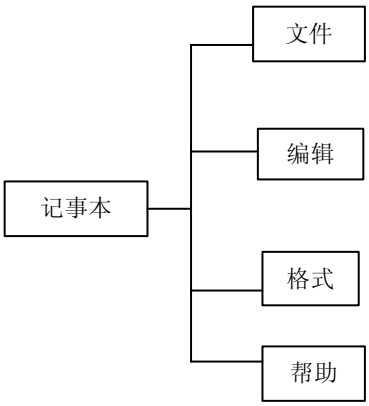


图 4-12　记事本的组成

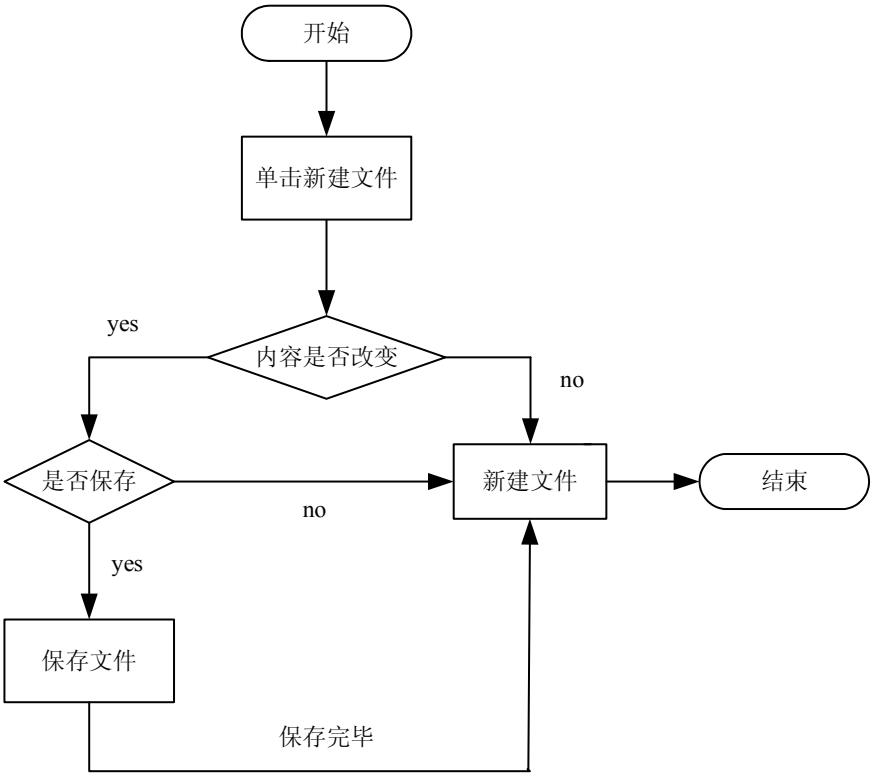


图 4-13　保存文件的流程图

- 打开文件：如果记事本中有内容，并且没有保存，则单击“打开文件”后出现提示语“是否保存当前文件”，单击“是”则保存文件，单击“否”则以前记事本中的内容将不复存在并出现文件选择器，打开相应的文件，如果已经保存则不出现提示语，直接出现文件选择器打开相应的文件。
- 保存文件：将已经编写好的文档写入计算机，单击“打开文件”则先出现文件选择器，命名后即可存入计算机，其方法是获取文本内容，再按路径存入指定的地方。保存后

将标志文档是否已保存的变量作相应修改，以便判断该文件内容是否已保存，是否发生改变。保存过后的文件再次单击“保存”是没有反应的，因为已经保存过，要再次保存则要单击“文件另存”，它可以无数次地保存。

- 文件另存：它可以无数地保存，这就是与保存的区别。
- 退出：先看当前文件是否保存，如果已保存则安全退出，如果没有保存则出现提示框，询问是否需要保存，“是”则保存，“否”则退出。

以下是系统中的关键函数及其功能。

- fileArea. copy()：复制。
- fileArea. paste()：粘贴。
- fileArea. cut()：剪切。
- fileArea. replaceSelection (null)：删除
- undo- actionPerformed (ActionEvent e1)：撤消。
- redo- actionPerformed (ActionEvent e1)：恢复。
- fileArea. selectAll ()：全选。
- fileSave- actionPerformed (ActionEvent e1)：保存文件。
- fileOpen- actionPerformed (ActionEvent e)：打开文件。
- fileCreate - actionPerformed (ActionEvent e)：新建文件。
- exit- actionPerformed (ActionEvent e)：退出。
- fileOtherSave - actionPerformed(ActionEvent e)：另存为。

当单击打开文件、新建文件、保存文件、文件另存、退出、复制、粘贴、剪切、删除、撤消、恢复、全选时都能依次实现那些功能。

下面是以上所有功能中最有代表性的“保存文件”功能的代码：

```
public class mainMenu extends JFrame
{
    JMenuBar jMenuBarOne = new JMenuBar();
    JTextArea fileArea = new JTextArea ();                  //创建写字板文本区
    JMenu fileMenu = new JMenu ("文件");                    //创建 JMENU 对象，即文件菜单
    JMenuItem save= new JMenuItem ("保存文件");             //创建文件的菜单项
    public mainMenu ()   //构造函数
    {
        save. addActionListener ( new fileSaveActionListener ()) ;   //对保存的监听
        jMenuBarOne. add(fileMenu );
        fileMenu. add (save);                               //添加 FILE 中的 JMenuItem 对象
        this. getContentPane (). add(new JScrollPane (fileArea )) ; //添加写字板
        setJMenuBar (jMenuBarOne );
        fileArea. setLineWrap (true);                           //设置自动换行
        this. setTitle ("无标题- 记事本");                      //设置标题
        this. setBounds (100,100,550,450 );                     //设置大小位置
        this. setVisible (true);                                //显示窗口
    }                                                           //构造函数结束
    public class fileSaveActionListener implements ActionListener
    {
```

```
        public void actionPerformed (ActionEvent e)
        {
                fileSave-actionPerformed (e1);
        }
    }                                                    //监听器
    fileSave-actionPerformed (ActionEvent e1)      //保存文件声明事件处理方法
    {
          ......
    }
    public static void main (String args [] )
    {
        mainMenu jsb= new mainMenu ();
    }
}
```

以上是利用事件驱动模型的例子。文件菜单和编辑菜单中的功能都是鼠标触发事件，本例事件源是 Save（菜单项），事件处理程序就是 public void actionPerformed（ActionEvent e），监听器就是 fileSaveActionListener 类。

本章小结

（1）事件表示用户与应用程序交互所得到的响应。

（2）事件编程必须实现某一事件接口并重写接口中所有的方法或继承事件类，重写类中的相关的方法。

（3）事件主要涉及三类对象：事件源、监听器和事件处理方法。

（4）同一个事件源上可能发生多种事件。

（5）使用授权处理模型进行事件处理的一般方法是：对于某种类型的事件 XXXEvent，要想接收并处理这类事件，必须定义相应的事件监听器类，该类需要实现与该事件相对应的接口 XXXListener；事件源实例化以后，必须进行授权，注册该类事件的监听器，使用 addXXXListener(XXXListener)方法来注册监听器。

（6）与 AWT 有关的所有事件类都由 java.awt.AWTEvent 类派生，它也是 EventObject 类的子类。AWT 事件共有 10 类，可以归为两大类：低级事件和高级事件。

（7）每类事件都有对应的事件监听器，监听器是接口，根据动作来定义方法。

（8）Java 的事件模型结构基本上是这样的：继承适配器，可以重载需要的方法，一个、两个、更多个都可以。但实现接口必须实现这个接口提供的所有方法，哪怕是空方法都必须实现。

（9）内部类（inner class）是被定义于另一个类中的类，使用内部类的主要原因是：一个内部类的对象可访问外部类的成员方法和变量，包括私有的成员；实现事件监听器时，采用内部类、匿名类编程非常容易实现其功能；编写事件驱动程序，内部类很方便。

（10）当一个内部类的类声名只是在创建此类对象时用了一次，而且要产生的新类需继承于一个已有的父类或实现一个接口，才能考虑用匿名类。

（11）常用的四种对事件的响应：将对事件的响应信息显示在 Applet 的状态栏上；将对事件响应信息显示在弹出窗口上；打开另一个对话窗口；显示另一个窗口界面。

习题四

一、简答题

1．叙述事件处理的机制，并画出流程图。

2．什么是监听器，如何进行事件注册？

3．事件分为哪些类，对应的接口是什么？

4．为什么需要 Adapter 类？

5．Anonymous Inner Class（匿名内部类）是否可以 extends（继承）其他类，是否可以 implements（实现）interface（接口）？

二、 实践题

1．编写程序 keyeventDemo.java。当窗体获得焦点时按下键盘，窗体中将实时显示按下的是哪一个键。

2．修改例 4-6 的程序代码，以包含窗体关闭事件，并通过事件适配器简化窗体事件处理方法。

3．编写一个 Applet 程序，跟踪鼠标的移动，并把鼠标的当前位置用不同的颜色显示在鼠标所在的位置上，同时监测所有的鼠标事件，把监测到的事件名称显示在 Applet 的状态条中。

4．编写一个 Applet 程序，首先捕捉用户的一次鼠标单击，然后记录单击的位置，从这个位置开始复制用户的按键。试一下，如果不单击鼠标而直接敲击键盘，能否捕捉到键盘事件？为什么？