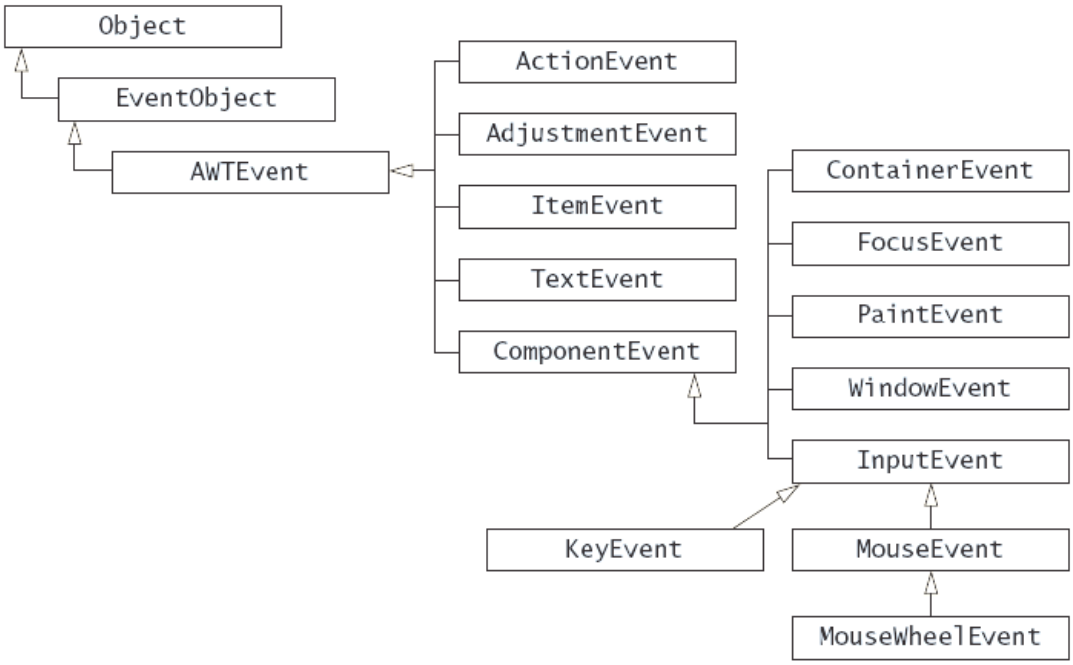
Event Handling

1. Events

Events come from User Controls. When you define a user interface, you will usually have some way to get user input. For example, buttons, menus, sliders, mouse clicks, ... all generate **events** when the user does something with them.
User interface **event objects** are passed from an event source, such as a button or mouse click, to an event listener, a user method which will process them.

The figure below illustrates the inheritance hierarchy for events.



Every Input Control (JButton, JSlider, ...) Needs an Event Listener. If you want a control to do something when the user alters the control, you must have a listener.

There are several kinds of events. The most common are:

User Control	***addXXXListener***	***method in listener***

`JButton`
`JTextField`
`JMenuItem`
	`addActionListener()`	`actionPerformed(ActionEvent e)`

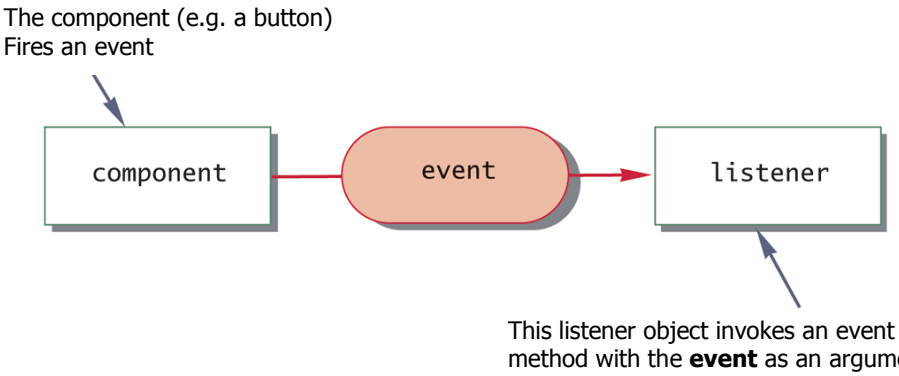
`JSlider`	`addChangeListener()`	`stateChanged(ChangeEvent e)`
`JCheckBox`	`addItemListener()`	`itemstateChanged()`
key on component	`addKeyListener()`	`keyPressed(), keyReleased(),` **`keyTyped`**`()`
mouse on component	`addMouseListener()`	**`mouseClicked`**`(), mouseEntered(), mouseExited(), mousePressed(), mouseReleased()`
mouse on component	`addMouseMotionListener()`	`mouseMoved(), mouseDragged()`

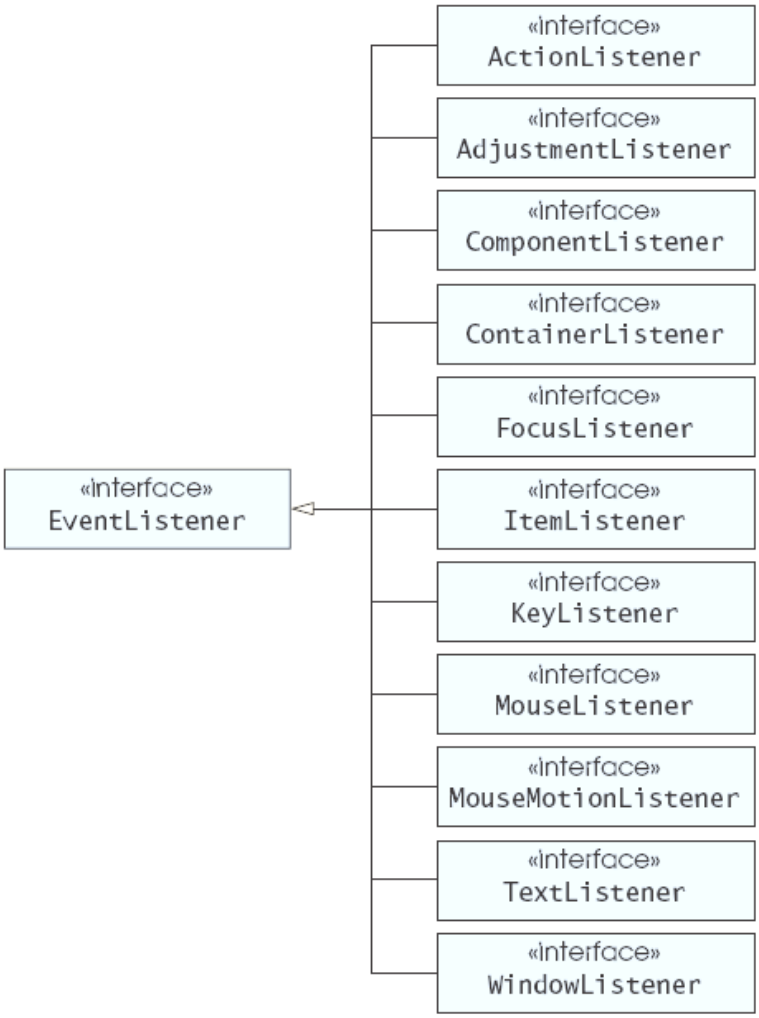
`import` Statements. To use events, you must have these import statements:

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

2. Event firing

The figure below illustrates what happens when an event is fired.





3. Listeners

A *listener* is called when the user does something to the user interface that causes an *event*. Although these events usually come from the user interface, they can have other sources (e.g., a Timer).

The figure on the left shows the eventßlistener interfaces of **`java.awt.event`**.

After a button is created, you will add a listener to it. E.g.,

b.addActionListener(
`listener_object`);

When the button is clicked, a call is made to the `actionPerformed()` method defined in the class of the listener object. An `ActionEvent` object is passed as a parameter to it .

3.1. `actionPerformed`

The listener method must be called `actionPerformed`.

Restriction: a single actionPerform per class. Because there can only be one `actionPerformed` method in a class, a new class is needed for every separate listener, or you have to share an `actionPerformed` method and use the ugly technique of figuring out who caused the call.

You need to create an instance of the class which defines the `actionPerformed` method for the button, and it is this instance that becomes the button listener.

3.2. Common Listener Strategies

Here are common ways to write listeners:

1. **Named Inner Class Listeners** are one of the most common ways to write small programs. They are convenient to write and and be shared with several components.
2. **Anonymous Inner Class Listeners** are sometimes used to associate a different listener with each button or other control. This is a little quicker to write, but lacks some of the flexibility of inner class listeners.
3. **Top-level Listeners** (this) are commonly used where there is only one source of an event. For example, if you are defining a subclass of `JPanel` to use for drawing with a mouse, an instance of this new class will typically also be the mouse event listener. Similarly with a `JPanel` used for animation – the panel itself may serve as the `ActionListener` for the `Timer`. Do not use "this" as a listener for buttons, menus, etc. because all controls must then share that one action listener.
4. **Action and AbstractAction** objects are action listeners. In some ways they are more powerful than other action listener strategies in that they can easily be used to enable or disable multiple controls. An action object can be shared with many controls. E.g., the same action is sometimes accomplished with a menu item and a button or toolbar tool. They encapsulate the name, description, enabled status, listener, and icon information. This is a good choice for large interfaces, and quite compatible with the MVC pattern (see below).
5. **External Listeners** - To implement the Model-View-Controller pattern, all listeners will effectively be in the Controller module. There are a number of ways to split the controller actions from the View part of the interface. One example is in Model-View-Controller (MVC) Structure.
6. **Subclassing a component and overriding *processActionEvent()***. Do not do this.

1. Model-View-Controller (MVC) Structure

The calculator is organized according the Model-View-Controller (MVC)pattern. The idea is to separate the user interface (the Presentation in the previous example) into a View (creates the display interacting with the Model as necessary), and Controller (responds to user requests, interacting with both the View and Controller as necessary). The literature on MVC leaves room for a number of variations, but they all follow this basic idea. This model is simple and can be used with simple method calls. If there are more complex interactions (e.g., the Model is asynchronously updated), an Observer pattern (listeners) may be required.

Main program

The main program initializes everything and ties everything together.

```
// stucture/calc-mvc/CalcMVC.java -- Calculator in MVC pattern.
// Fred Swartz -- December 2004

import javax.swing.*;

public class CalcMVC {
    //... Create model, view, and controller.  They are
    //    created once here and passed to the parts that
    //    need them so there is only one copy of each.
    public static void main(String[] args) {
```

```
        CalcModel      model      = new CalcModel();
        CalcView       view       = new CalcView(model);
        CalcController controller = new CalcController(model, view);

        view.setVisible(true);
    }
}
```

View

This View doesn't know about the Controller, except that it provides methods for registering a Controller's listeners. Other organizations are possible (eg, the Controller's listeners are non-private variables that can be referenced by the View, the View calls the Controller to get listeners, the View calls methods in the Controller to process actions, ...).

```
// structure/calc-mvc/CalcView.java - View component
//    Presentation only.  No user actions.
// Fred Swartz -- December 2004

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

class CalcView extends JFrame {
    //... Constants
    private static final String INITIAL_VALUE = "1";

    //... Components
    private JTextField m_userInputTf = new JTextField(5);
    private JTextField m_totalTf     = new JTextField(20);
    private JButton    m_multiplyBtn = new JButton("Multiply");
    private JButton    m_clearBtn    = new JButton("Clear");

    private CalcModel m_model;

    //================================================== constructor
    /** Constructor */
    CalcView(CalcModel model) {
        //... Set up the logic
        m_model = model;
        m_model.setValue(INITIAL_VALUE);

        //... Initialize components
        m_totalTf.setText(m_model.getValue());
        m_totalTf.setEditable(false);

        //... Layout the components.
        JPanel content = new JPanel();
        content.setLayout(new FlowLayout());
        content.add(new JLabel("Input"));
        content.add(m_userInputTf);
        content.add(m_multiplyBtn);
        content.add(new JLabel("Total"));
        content.add(m_totalTf);
        content.add(m_clearBtn);

        //... finalize layout
        this.setContentPane(content);
        this.pack();

        this.setTitle("Simple Calc - MVC");
        // The window closing event should probably be passed to the
        // Controller in a real program, but this is a short example.
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    void reset() {
        m_totalTf.setText(INITIAL_VALUE);
```

```
    }

    String getUserInput() {
        return m_userInputTf.getText();
    }

    void setTotal(String newTotal) {
        m_totalTf.setText(newTotal);
    }

    void showError(String errMessage) {
        JOptionPane.showMessageDialog(this, errMessage);
    }

    void addMultiplyListener(ActionListener mal) {
        m_multiplyBtn.addActionListener(mal);
    }

    void addClearListener(ActionListener cal) {
        m_clearBtn.addActionListener(cal);
    }
}
```

The Controller

The controller process the user requests. It is implemented here as an Observer pattern -- the Controller registers listeners that are called when the View detects a user interaction. Based on the user request, the Controller calls methods in the View and Model to accomplish the requested action.

```
// stucture/calc-mvc/CalcController.java - Controller
//    Handles user interaction with listeners.
//    Calls View and Model as needed.
// Fred Swartz -- December 2004

import java.awt.event.*;

public class CalcController {
    //... The Controller needs to interact with both the Model and View.
    private CalcModel m_model;
    private CalcView  m_view;

    //========================================================== constructor
    /** Constructor */
    CalcController(CalcModel model, CalcView view) {
        m_model = model;
        m_view  = view;

        //... Add listeners to the view.
        view.addMultiplyListener(new MultiplyListener());
        view.addClearListener(new ClearListener());
    }


    ////////////////////////////////////// inner class MultiplyListener
    /** When a multiplication is requested.
     *  1. Get the user input number from the View.
     *  2. Call the model to multiply by this number.
     *  3. Get the result from the Model.
     *  4. Tell the View to display the result.
     * If there was an error, tell the View to display it.
     */
    class MultiplyListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String userInput = "";
            try {
                userInput = m_view.getUserInput();
                m_model.multiplyBy(userInput);
                m_view.setTotal(m_model.getValue());
```

```
            } catch (NumberFormatException nfex) {
                m_view.showError("Bad input: '" + userInput + "'");
            }
        }
    }//end inner class MultiplyListener


    ////////////////////////////////////////// inner class ClearListener
    /**  1. Reset model.
     *   2. Reset View.
     */
    class ClearListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            m_model.reset();
            m_view.reset();
        }
    }// end inner class ClearListener
}
```

Model

The model is independent of the user interface. It doesn't know if it's being used from a text-based, graphical, or web interface.

```
// structure/calc-mvc/CalcModel.java
// Fred Swartz - December 2004
// Model
//     This model is completely independent of the user interface.
//     It could as easily be used by a command line or web interface.

import java.math.BigInteger;

public class CalcModel {
    //... Constants
    private static final String INITIAL_VALUE = "0";

    //... Member variable defining state of calculator.
    private BigInteger m_total;  // The total current value state.

    //================================================================ constructor
    /** Constructor */
    CalcModel() {
        reset();
    }

    //====================================================================== reset
    /** Reset to initial value. */
    public void reset() {
        m_total = new BigInteger(INITIAL_VALUE);
    }

    //================================================================ multiplyBy
    /** Multiply current total by a number.
    *@param operand Number (as string) to multiply total by.
    */
    public void multiplyBy(String operand) {
        m_total = m_total.multiply(new BigInteger(operand));
    }

    //================================================================== setValue
    /** Set the total value.
    *@param value New value that should be used for the calculator total.
    */
    public void setValue(String value) {
        m_total = new BigInteger(value);
```

```
    }

    //================================================================ getValue
    /** Return current calculator total. */
    public String getValue() {
        return m_total.toString();
    }
}
```

3.3. Top-level Listeners

3.3.1. Using `this` as a listener

A common way to write *simple* applets is to use the use the applet itself as a listener (referred to as `this`). For example

```
...
public class Hello extends JApplet implements ActionListener {
   JButton b;
   ...
   public void init() {
      JButton b = new JButton("Hello");
      b.addActionListener(this);
      ...
   }

   public void actionPerformed(ActionEvent e) {
      message = "Hi";
      repaint();
   }
   ...
}
```

This doesn't seem to be a problem for small programs, but what happens if there is more than one button? There can be only one `actionPerformed()` method in a class.

3.3.2. Problem: One listener for many components. Solution: Use inner class listeners

Inside the listener method, it's possible to check the parameter to find out which component caused the event. For example,

```
JButton b1, b2;
. . .
public MyClass() {  // constructor
    ...
    b1.addActionListener(this);
    ...
    b2.addActionListener(this);  // SAME listener!
    ...
}//end constructor

public void actionPerformed(ActionEvent e) {
    Object obj = e.getSource();    // get the control that caused the event
if (o instanceof JButton) {   // make sure it's a button.
        JButton b = (JButton)obj;  // downcast to a button
        if (b == b1) {   // UGLY, DON'T DO THIS
           // do something for button b1
        } else if (b == b2) {
           // do something for button b2
        }
    } else if (o instanceof JTextField) {
       ...
    ...
```

Using one listener makes the response slower, and forces all events to be handled in the same place. This uses the event model which was introduced in Java 1.1, but it has all the problems of the old Java 1.0 event model. Although you will see this style used in some Java books, **don't use it**. It doesn't scale up from one control to many. Buttons, menu items, toolbar items, etc. use action listeners; imagine what this method would look like for Microsoft Word!
Grouping all separate action listeners together in the source code is a good idea to make the code more comprehensible, but don't try to put all processing in one method!

3.4. Inner-class Listeners

3.4.1. Named inner class

Defining an inner class listener to handle events is a very popular style.

- **Access**. Use an inner class rather than an outer class to access instance variables of the enclosing class. In the example below, the `myGreetingField` can be referenced by the listener class. Because simple program listeners typically get or set values of other widgets in the interface, it is very convenient to use an inner class.
- **Reuse**. Unlike annonymous inner class listeners, it's easy to reuse the same listener for more than one control, e.g., the click of a button might perform the same action as the equivalent menuitem, and might be the same as hitting the enter key in a text field.
- **Organization**. It's easier to group all the listeners together with inner classes than with anonymous inner class listeners.

```
class MyPanel extends JPanel {
. . .
    private JButton myGreetingButton = new JButton("Hello");
    private JTextField myGreetingField = new JTextField(20);

    //=== Constructor
    public MyPanel() {
        ActionListener doGreeting = new GreetingListener();
        myGreetingButton.addActionListener(doGreeting);
        myGreetingMenuitem.addActionListener(doGreeting);
        . . .
    }

. . .
    //===This class is defined inside MyPanel
    private class GreetingListener extends ActionListener {
        new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                myGreetingField.setText("Guten Tag");
            }
        }
    }
}
```

3.5. Getting the name or actionCommand from a button

If you have a listener for a button, you might wonder why you need to get the text that labels the button. A common reason is that you have a number of buttons (e.g., the numeric keypad on a calculator) that do almost the same thing. You can create one listener for all the keys, then use the text from the button as the value.
getActionCommand(). By default the *getActionCommand()* method of an *ActionEvent* will return the text on a button. However, if you internationalize your buttons so that the show different text depending on the user locale, then you can explicitly set the "actionCommmand" text to something else.
Let's say that you want to put the button value at the end of the `inField` field as you might want to do for a calculator. You could do something like below (Although checking for overflow).

```
 JTextField inField = new JTextField(10);
 . . .
// Create an action listener that adds the key name to a field
ActionListener keyIn = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // Get the name of the button
        String keyNum = e.getActionCommand(); // "1", "2", ...
        inField.setText(inField.getText() + keyNum);
    }
};

// Create many buttons with the same listener
JButton key1 = new JButton("1");
key1.addActionListener(keyIn);
JButton key2 = new JButton("2");
key2.addActionListener(keyIn);
JButton key3 = new JButton("3");
key3.addActionListener(keyIn);
 . . .
```

3.6. Anonymous Listeners

3.6.1. Using anonymous inner class listeners - a common idiom

There is no need to define a named class simply to add a listener object to a button. Java has a somewhat obscure syntax for creating an *anonymous innner class* listener that implements an interface. There is no need to memorize the syntax; just copy and paste it each time. For example,

```
class myPanel extends JPanel {
. . .
    public MyPanel() {
        . . .  //in the constructor
        JButton b1 = new JButton("Hello");
        b1.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    // do something for button b1
                }
            }
        );
```

The above example creates a subclass of `Object` that `implements` the `ActionListener` interface. The compiler creates names for anonymous classes. The JDK typically uses the enclosing class name followed by `$` followed by a number, e.g., you may see a `MyPanel$1.class` file generated by the compiler.

4. Action, AbstractAction

The `javax.swing.Action` interface, and the corresponding class, `javax.swing.AbstractAction`, provide a useful mechanism to implement action listeners that can be shared and coordinated.

- Actions can be used with most buttons, including toobox buttons and menu items, text fields, etc.
- They can be shared with all controls which do the same thing.
- Actions can be dis-/enabled, and they will then dis-/enable all corresponding controls.
- They can specify text, icons, tooltip text, accelerator, and mnemonic keys. Mnemonic keys work almost like accelerators; they activate a menu with keystrokes. The difference here is that an accelerator will immediately trigger the attached action. A mnemonic key will only trigger the menu action it is attached to when its menu is visible.

Subclassing. You *must* subclass `AbstractAction` (the hint is the word "abstract" in the class name). The minimum you need to do is override `actionPerformed` to specify what you want the `Action` to do. See examples below.

4.1. `AbstractAction` constructors, methods, and fields

Constructors		
act =	`new AbstractAction(String` *name*`);`	Specifies *name* for buttons, etc.
act =	`new AbstractAction(String` *name*`, Icon` *smallIcon*`);`	Specifies *name* and an icon (e.g., that will appear on a toolbar buttons).
Some Methods		
b =	*act*.**`isEnabled`**`()`	Returns true if this Action *act* is enabled.
	act.**`setEnabled`**`(boolean` *enabled*`)`	Sets the status of this Action.
	act.**`putValue`**`(String` *key*`, Object` *value*`)`	Sets the value of property key to value.
obj =	*act*.**`getValue`**`(String` *key*`)`	Gets the value of property key.
Some Property Fields (use ***`putValue()`*** to explicitly set these fields)		
	`ACCELERATOR_KEY`	Accelerator key.
	`MNEMONIC_KEY`	Mnemonic key.
	`NAME`	Name for buttons and menu items.
	`SHORT_DESCRIPTION`	Used as tooltip text.
	`SMALL_ICON`	Used for toolbars.

4.2. Actions Usage

Actions can be used directly in the `add()` method of some containers (e.g., menus and toolbars), or in constructors for buttons and menu items.

```
fileMenu.add(exitAction);  // Add directly to menu.  Uses Action's text, icon.
```

If you don't want all the functionality of an Action, create the desired component from the Action, then modify it.

```
JMenuItem exitItem = new JMenuItem(exitAction);  // Use to create component.
exitItem.setIcon(null);                          // Modify to suppress the icon.
fileMenu.add(exitItem);
```

4.2.1. Example - Simple anonymous class

```
Action openAction = new AbstractAction("Open...") {
    public void actionPerformed(ActionEvent e) {
        openFile();  // Do what you want here.
    }
};
. . .
fileMenu.add(openAction);  // Add action to menu
```

The **Simple Editor** example shows the use of anonymous subclassing.

4.2.2. Example - Defining subclass to get additional functionality

```
Action openAction = new OpenAction("Open...", folderIcon, "Open file.",
KeyEvent.VK_O);
. . .
fileMenu.add(openAction);  // Add action to menu
. . .
class OpenAction extends AbstractAction {
    //- Define a constructor which takes parameters you want to set.
```

```
    public OpenAction(String text, ImageIcon icon, String tooltip, int mnemonic)
{
        super(text, icon); //  AbstractAction constructor takes only two params.
        putValue(SHORT_DESCRIPTION, tooltip);  // Will appear as tooltip text.
        putValue(MNEMONIC_KEY, new Integer(mnemonic));
    }

    //- Override actionPerformed to do what you want.
    public void actionPerformed(ActionEvent e) {
        openFile();  // Do what you want here.
    }
}
```

Example - Simple Editor

This simple text editor uses a `JTextArea` with *Actions* to implement the menu items. Actions are a good way to implement ActionListeners.

- Actions can be used with most buttons, including toobox buttons and menu items, text fields, etc.
- They can be shared so that all controls which do the same thing can share an Action.
- Actions can be dis-/enabled, and they will then dis-/enable all corresponding controls.
- They can be used to set icons, tooltip text, accelerator and mnemonic keys.

```
// editor/NutPad.java -- A very simple text editor -- Fred Swartz - 2004-08
// Illustrates use of AbstractActions for menus.

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

public class NutPad extends JFrame {
    //-- Components
    private JTextArea mEditArea;
    private JFileChooser mFileChooser = new JFileChooser(".");

    //-- Actions
    private Action mOpenAction;
    private Action mSaveAction;
    private Action mExitAction;

    //================================================================== main
    public static void main(String[] args) {
        new NutPad().setVisible(true);
    }//end main

    //============================================================ constructor
    public NutPad() {
        createActions();
        this.setContentPane(new contentPanel());
        this.setJMenuBar(createMenuBar());
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setTitle("NutPad");
        this.pack();
    }//end constructor

    ////////////////////////////////////////////////////// class contentPanel
    private class contentPanel extends JPanel {
        //======================================================== constructor
        contentPanel() {
            //-- Create components.
            mEditArea = new JTextArea(15, 80);
                mEditArea.setBorder(BorderFactory.createEmptyBorder(2,2,2,2));
                mEditArea.setFont(new Font("monospaced", Font.PLAIN, 14));
            JScrollPane scrollingText = new JScrollPane(mEditArea);
```

```
            //-- Do layout
            this.setLayout(new BorderLayout());
            this.add(scrollingText, BorderLayout.CENTER);
        }//end constructor
    }//end class contentPanel

    //============================================================ createMenuBar
    /** Utility function to create a menubar. */
    private JMenuBar createMenuBar() {
        JMenuBar menuBar = new JMenuBar();
        JMenu fileMenu = menuBar.add(new JMenu("File"));
        fileMenu.add(mOpenAction);  // Note use of actions, not text.
        fileMenu.add(mSaveAction);
        fileMenu.addSeparator();
        fileMenu.add(mExitAction);
        return menuBar;
    }//end createMenuBar

    //============================================================ createActions
    /** Utility function to define actions. */
    private void createActions() {
        mOpenAction = new AbstractAction("Open...") {
            public void actionPerformed(ActionEvent e) {
                int retval = mFileChooser.showOpenDialog(NutPad.this);
                if (retval == JFileChooser.APPROVE_OPTION) {
                    File f = mFileChooser.getSelectedFile();
                    try {
                        FileReader reader = new FileReader(f);
                        mEditArea.read(reader, "");  // Use TextComponent read
                    } catch (IOException ioex) {
                        System.out.println(e);
                        System.exit(1);
                    }
                }
            }
        };

        mSaveAction = new AbstractAction("Save") {
            public void actionPerformed(ActionEvent e) {
                int retval = mFileChooser.showSaveDialog(NutPad.this);
                if (retval == JFileChooser.APPROVE_OPTION) {
                    File f = mFileChooser.getSelectedFile();
                    try {
                        FileWriter writer = new FileWriter(f);
                        mEditArea.write(writer);  // Use TextComponent write
                    } catch (IOException ioex) {
                        System.out.println(e);
                        System.exit(1);
                    }
                }
            }
        };

        mExitAction = new AbstractAction("Exit") {
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        };
    }//end createActions
}//end class NutPad
```