Understanding Action Objects in Java

Baldwin explains the theory behind, and illustrates the use of the Action interface.

Published: May 25, 2002
By Richard G. Baldwin

Java Programming Notes # 107

- Preface
- Discussion and Sample Code
- Run the Program
- Complete Program Listing

Preface

The main purpose of this lesson is to help you to understand the use of Java **Action** objects, with particular emphasis on the changes introduced in Java version 1.3. Along the way, you will also learn something about *bound properties* in Java.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different figures and listings while you are reading about them.

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online Java tutorials. You will find those lessons published at Gamelan.com. However, as of the date of this writing, Gamelan doesn't maintain a consolidated index of my Java tutorial lessons, and sometimes they are difficult to locate there. You will find a consolidated index at Baldwin's Java Programming Tutorials.

Discussion and Sample Code

What is an Action object?

An **Action** object is an object that is instantiated from any class that implements the **Action** interface.

> *(To avoid confusion at the outset, let me point out that the **Action** interface is not the same as the **ActionListener** interface.)*

This leads to the obvious question, *"What is the **Action** interface?"*

Sun summarizes the purpose of the **Action** interface in the following way:

> *"The **Action** interface provides a useful extension to the **ActionListener** interface in cases where the same functionality may be accessed by several controls."*

A real-world example

Most text editors and word processors provide a *copy* action. Typically, the *copy* action makes it possible to highlight some text, and to copy it to the system clipboard.

The *copy* action can often be initiated in several ways. There will usually be a *copy* item on the *Edit* menu. Usually, there will also be a *copy* button on a toolbar.

> *(Typically also, the copy item on the menu will have an accelerator keystroke such as Ctrl+C, but that is the topic of a different article.)*

Need the same behavior

Regardless of whether the user initiates the *copy* action by selecting the menu item or by clicking the toolbar button, the behavior should be the same. *(That behavior is usually to copy highlighted text onto the system clipboard.)*

Copy action can be disabled

With the editor that I am using, whenever no text has been selected, the *copy* action is disabled. Whenever text is selected, the *copy* action becomes enabled. It is important that the *copy* item on the menu and the *copy* button on the toolbar become enabled and disabled in unison. *(It would be very confusing if the copy menu item were to become disabled while the copy button on the toolbar remains enabled.)*

ActionListener objects

In Java, the behavior of menu items and toolbar buttons is normally controlled by registering **ActionListener** objects on those sources, and defining the behavior in the **actionPerformed** method of the **ActionListener** object.

> *(If you are unfamiliar with the use of Action listeners, you will find numerous tutorial lessons that discuss the Delegation or JavaBeans event model at Baldwin's Java Programming Tutorials.)*

It is a straightforward matter to instantiate an **ActionListener** object and to register it on more than one source *(a menu item and a toolbar button, for example).* This will guarantee that action events will be handled in an identical way regardless of which source fires the event.

Enable and disable

It is somewhat less straightforward to guarantee that all of the sources are enabled and disabled in unison, particularly from a software maintenance viewpoint. Whenever the enabled state changes, it is necessary to individually enable or disable all of the sources on which the **ActionListener** object is registered.

Short of examining the source code, there is no easy way to determine how many and which sources a given **ActionListener** object has been registered on. You could, of course, create and maintain a list of such sources, but that would entail extra programming effort.

Action object to the rescue

This is where an **Action** object shines. The **Action** interface extends the **ActionListener** interface. Therefore, an **Action** object is an *action listener*, which provides a definition for the **actionPerformed** method.

In addition, and this is very important, the **Action** object is also a source of **PropertyChange** events. *(An ordinary **ActionListener** object is not normally a source of **PropertyChange** events.)*

Here is part of what Sun has to say about **PropertyChange** events:

> *"A "PropertyChange" event gets delivered whenever a bean changes a "bound" or "constrained" property. A PropertyChangeEvent object is sent as an argument to the PropertyChangeListener and VetoableChangeListener methods."*

What are bound and constrained properties?

If you are unfamiliar with the concept of *bound properties* in JavaBeans components, you will find several tutorial lessons on JavaBeans components at Baldwin's Java Programming Tutorials.

In a nutshell, a bound property is a property that will:

- Register other objects that want to be notified when the value of the property changes
- Notify them when the value changes

(Constrained properties don't come into play here, so there is no need to discuss them further in this lesson.)

Two-way communication

An **Action** object has a bound property, of type **boolean**, named **enabled**. Whenever an **Action** object is *set* or *registered* on an *action-aware* component, at least two important things happen.

Registered as an ActionListener object

First, the **Action** object is registered as an **ActionListener** object on the *action-aware* component. The behavior of the **actionPerformed** method, as defined in the **Action** object, handles **Action** events fired by the component.

What does action-aware mean?

An *action-aware* class is a class that either defines or inherits the **setAction** method shown in Figure 1. An *action-aware* component is an object instantiated from such a class.

public void **setAction**(Action a)

Sets the Action for the ActionEvent source.

Parameters:
a - the Action for the AbstractButton, or null

Figure 1

This is the method that sets up the two-way communication described earlier *(and sometimes does some other things as well, which I will discuss later).*

What are the action-aware components?

A review of the Java API documentation indicates that as of Java version 1.4.0, the following classes are *action-aware* according to the definition given above. *(Note, however, not all of these classes are suitable for instantiating typical **Action** objects. For example, I can't think of a good reason to use a **MetalScrollButton** as an **Action** object, but there may be such a reason that I haven't thought of.)*

- AbstractButton
  - JButton
    - BasicArrowButton
      - MetalScrollButton
    - MetalComboBoxButton
  - JMenuItem
    - JCheckBoxMenuItem
    - JMenu
    - JRadioButtonMenuItem
  - JToggleButton
    - JCheckBox
    - JRadioButton
- JTextField
  - DefaultTreeCellEditor.DefaultTextField
  - JPasswordField
  - JFormattedTextField
- JComboBox

Establishes behavior of ActionEvent handler

The invocation of the **setAction** method on an *action-aware* component sets the **Action** property for the **ActionEvent** source. In other words, it establishes the behavior that will be exhibited when the *action-aware* component fires an **Action** event.

The new **Action** replaces any previously set **Action** objects. However, it does not affect any **ActionListener** objects that were independently added using the **addActionListener** method *(I will demonstrate this later).* Also, if the **Action** object is already a registered **ActionListener**, it is not re-registered.

Sets certain component properties

Another important result of setting the **Action** property on the *action-aware* component is that certain other properties belonging to the component are also set from the values encapsulated in the **Action** object. The properties that get set may differ for different component classes. For example, if the component is a **JButton**, the properties that get set include:

- The **enabled** property
- The property that controls the text on the face of the button
- The property that controls the **Icon** that appears on the button
- The text for the tool tip
- The Mnemonic for the button

You can learn more about this process by taking a look at the documentation for the **configurePropertiesFromAction** method for the class of interest. This is the method that handles the configuration process for each different component.
*(Note that this documentation for version 1.4.0 is considerably different from the documentation for version 1.3.1 with respect to the **AbstractButton**, **JButton**, and **JMenuItem** classes. In particular, the list of properties that get set has been expanded in all three cases. I don't know if this is the result of changes, or earlier typographical errors.)*
For example, here is what Sun has to say about the **configurePropertiesFromAction** method for the **AbstractButton** class *(v 1.4.0):*
"The properties which are set may differ for subclasses. By default, the properties which get set are Text, Icon, Enabled, ToolTipText, ActionCommand, and Mnemonic."
The Sun documentation for the **configurePropertiesFromAction** method for the J**Button** class says essentially the same thing as the documentation for the **AbstractButton** class.

In comparison, here is what Sun has to say about the **configurePropertiesFromAction** method for the **JMenuItem** class:

> *"By default, this method sets the same properties as AbstractButton.configurePropertiesFromAction(), plus Accelerator."*

In other words, a **JMenuItem** object can set and track one property, **Accelerator**, which is not tracked by a **JButton** object.

PropertyChange notification

Earlier I stated that whenever an **Action** object is *set* or *registered* on an *action-aware* component, at least two important things happen. I have been discussing the first of the two things that happen. It is now time to discuss the second important thing that happens.

Setting the **Action** object on an *action-aware* component causes the component to be registered as a **PropertyChange** listener on the **Action** object. Thereafter, whenever the value of the **enabled** property of the **Action** object changes, the component is notified of the change.

> *(It is the responsibility of the component to respond to that notification and to cause its **enabled** property to track the value of the **enabled** property of the **Action** object.)*

Keyed properties

Certain other properties belonging to the **Action** object are also tracked and updated on the component as the **Action** object's *keyed properties* change.

> *(Disclaimer: Even though I refer to these values as keyed properties, it is probably not technically correct to call them properties or bound properties, because the methods for setting and getting their values do not conform to the JavaBeans design patterns for properties. However, they behave like bound properties, meaning that the component is notified when the values of the keyed properties change.)*

A PropertyChangeListener

This update process is handled by causing the component to *act like* a **PropertyChangeListener** registered on the **Action** object.

> *(The components can't actually be PropertyChangeListeners in their own right, because they neither define nor inherit the method named **PropertyChange**.)*

Some smoke and mirrors is being used here. This process is handled by the method named **createActionPropertyChangeListener**.
*(I'm not certain exactly how this is handled, but my guess is that the invocation of the **setAction** method on a particular component causes the invocation of the **createActionPropertyChangeListener** method on that component. I suspect that this method instantiates an object from a class that implements the **PropertyChangeListener** interface. This **PropertyChangeListener** object is registered on the **Action** object so that it is notified when a property change occurs. My guess is that it also holds a reference to the component for which it was instantiated. When it is notified of a **PropertyChange** event on the **Action** object, it updates the component's properties using current values from the **Action** object. This causes the component to behave like a **PropertyChangeListener**.)*

Overridden createActionPropertyChangeListener

Different overridden versions of the **createActionPropertyChangeListener** method exist for different component classes, so the properties that are tracked and updated may differ from one component class to the next.

For example, if the component is a **JButton**, the properties that are tracked and automatically updated include those in the following list:

- The **enabled** property
- The property that controls the text on the face of the button
- The property that controls the **Icon** that appears on the button
- The text for the tool tip
- The Mnemonic for the button

Notification mechanism

In typical fashion, the **propertyChange** method shown in Figure 2 is invoked on every registered listener when the **Action** object fires a **PropertyChange** event.

public void
propertyChange(PropertyChangeEvent evt)

This method gets called when a bound property is changed.

Parameters:
evt - A PropertyChangeEvent object describing the
event source and the property that has changed.

Figure 2

It is the responsibility of the **propertyChange** method in each registered listener object to use this notification to cause the enabled state of the component to track the value of the **enabled** property of the **Action** object. This method also causes other properties discussed above to track the values of the corresponding *keyed properties* of the **Action** object.

Two different programming styles

Two different programming styles can be used to accomplish the above behavior. The first of those programming styles existed prior to the release of JDK 1.3. The second approach was released with JDK 1.3 and updated in JDK 1.4.0.

The earlier approach, *(which is still supported in version 1.4.0 but is not the preferred approach),* makes use of a specialized version of the **add** method for three specific container types: **JMenu**, **JPopupMenu**, and **JToolBar**. *(This method is limited to use with those three*

container types). The signature of the specialized version of the **add** method for each of the three container types respectively is:

- `public JMenuItem add(Action a)`
- `public JMenuItem add(Action a)`
- `public JButton   add(Action a)`

The specialized add method

This specialized version of the **add** method is defined for the container classes **JMenu**, **JPopupMenu**, and **JToolBar**. Here are descriptions for this method as defined for those three classes respectively:

- **JMenu** - Creates a new menu item attached to the specified **Action** object and appends it to the end of this menu. Returns **JMenuItem**.
- **JPopupMenu** - Appends a new menu item to the end of the menu, which dispatches the specified **Action** object. Returns **JMenuItem**.
- **JToolBar** - Adds a new **JButton**, which dispatches the action. Returns **JButton**.

Not the preferred method

Although this method is still supported as of JDK 1.4.0, in all three cases, Sun has something like the following to say:

> *As of JDK 1.3, this is no longer the preferred method for adding Actions to a container. Instead it is recommended to configure a control with an action using* ***setAction****, and then add that control directly to the Container.*

However ...

Even though this is not the preferred approach, you are very likely to encounter code that uses this approach. Therefore, I am providing a brief description of this approach, and will compare it with the preferred approach.

> *(All of the discussion, from the beginning of the article down to the beginning of the above discussion on the specialized* ***add*** *method, was based on the preferred approach.)*

Use is restricted to three container types

It is important to emphasize that this approach can only be used with the following container classes *(because other container classes neither define nor inherit the specialized version of the* ***add*** *method):*

- **JMenu**
- **JPopupMenu**
- **JToolBar**

The preferred method, on the other hand, can be used with just about any of the standard container classes.
*(For example, you could place an action-aware component in a **JButton** if you had a reason to do so. However, you may want to avoid using AWT container classes in order to avoid the problems inherent in mixing heavyweight and lightweight components.)*

The specialized add method

When the specialized **add** method is invoked on one of the three containers listed above, passing an **Action** object as a parameter, the following things happen:

- The container creates a component that is appropriate for that container:
  - **JMenu** creates a **JMenuItem**
  - **JPopupMenu** creates a **JMenuItem**
  - **JToolBar** creates a **JButton**
- The container gets the appropriate properties *(such as enabled and the icon)* from the **Action** object to customize the component.
- The container checks the initial state of the **Action** object to determine if it is enabled or disabled, and renders the component in the same state.
- The container registers a **PropertyChangeListener** with the **Action** object so that it is notified of state changes. When the **Action** object changes from enabled to disabled, or back, the container makes the appropriate revisions and renders the component accordingly.

The major differences

The major differences between this approach and the preferred approach seem to be:

- This approach is limited to only three types of containers, whereas the preferred approach can be used with just about any container.
- This approach can only be used with one specific component type for each of the three containers. The preferred approach can pair any of a large number of component types with just about any container.

Much more general

The bottom line is that the preferred approach is the more general of the two. For example, if you had a reason to do so, you could put an *action-aware* **JTextField** object in a **JMenu** object, or an *action-aware* **JRadioButton** object in a **JButton** object.

A sample program

Now it is time to take a look at some code. A complete listing of the sample program named **ActionObj02** is shown in Listing 22 near the end of this article.

This program uses the preferred approach to place two *action-aware* components in each of the following three container types: **JMenu**, **JToolBar**, and **JPanel**.

*(Note that the earlier approach did not support the **JPanel** container type.)*

Two different **Action** objects are used to populate the three containers. Within each container, one of the components is registered with one of the **Action** objects, and the other component is registered with the other **Action** object.

The **JMenu** container is populated with two **JMenuItem** objects. The **JToolBar** container is populated with two **JButton** objects. The **JPanel** container is populated with one **JButton** object and one **JMenuItem** object.

View at startup

Figure 3 shows a view of the program as it appears at startup *(the **JMenu** object is hidden from view).*

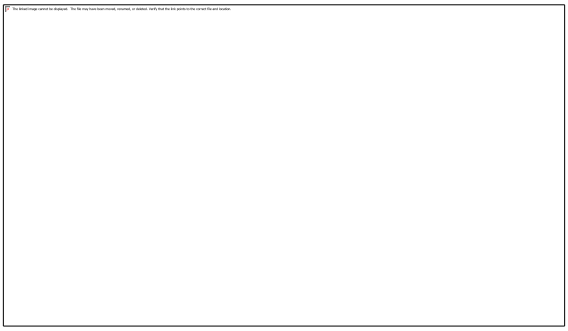

Figure 3 View of Program at Startup

The title for the hidden menu is shown at the top in Figure 3. A tool bar object is shown immediately below the menu bar. A **JPanel** object is shown at the bottom.

The four checkboxes in the center are used to manipulate the *enabled* and *icon* properties of the two **Action** objects. This, in turn, causes the *enabled* property of each of the components to change, and also causes the icon displayed on each of the components to change, when the checkboxes are checked and cleared.

The expanded menu

Figure 4 shows the screen display of the program with the menu expanded, but with no other changes.

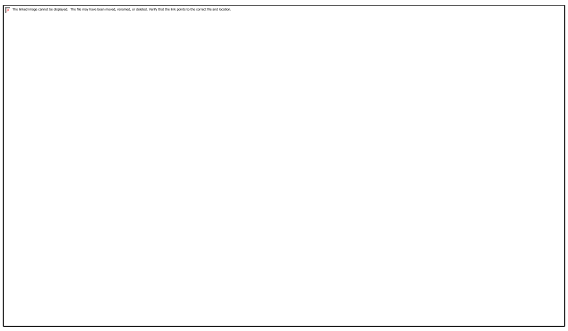

Figure 4 View with Menu Expanded

As you can see, the two menu items represent the same **Action** objects as the components on the tool bar and the components in the **JPanel**.

Manipulating the Action objects

The checkboxes in the center can be used to manipulate the **Action** objects. For example, Figure 5 shows the output with one of the **Action** objects disabled by checking the upper-left checkbox. *(Note that the text on the checkbox also changes when it is checked.)*

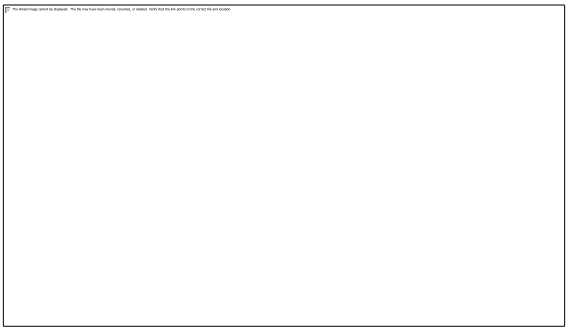

Figure 5 View with One Action Item Disabled

As you can see in Figure 5, disabling one of the **Action** objects causes both of the visible *action-aware* components registered *(as PropertyChange listeners)* on that object to be disabled.

> *(The menu also contains a component registered as a PropertyChange listener on that **Action** object. If you could see that component, you would see that it is disabled also.)*

In a similar fashion, checking the upper-right checkbox would disable the other **Action** object and the three components registered as *PropertyChange listeners* on it.

Toggle between two icons

The **enabled** property is clearly a property in the sense that its *setter* and *getter* methods match the JavaBeans design pattern for properties.

As we will see later, the property that specifies the icon to be displayed on the components doesn't use the standard *setter* and *getter* methods for properties. Regardless, changing the property that specifies the icon in the **Action** object causes the components registered on that object to display a different icon.

Figure 6 shows the result of checking the lower-left checkbox. Checking and clearing this checkbox toggles the specified icon in the **Action** object between two different gif files.

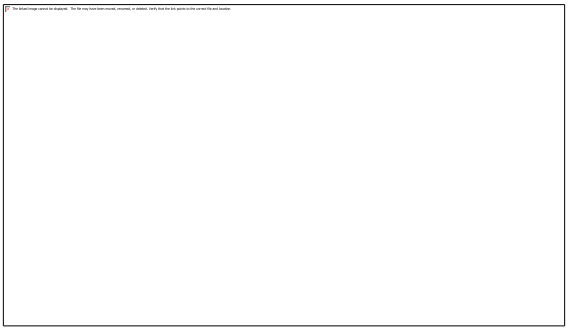

Figure 6 View with a Different Icon

As you can see in Figure 6, changing the property that specifies the icon in one of the **Action** objects causes the icon displayed on registered components to change from a red ball to a blue ball. Again, if you could see the corresponding item on the menu, you would see that it has changed also.

The Action interface

According to Sun,

> *"The Action interface provides a useful extension to the ActionListener interface in cases where the same functionality may be accessed by several controls."*

Previous sections in this article have discussed how such a capability may be used.

Here is a quotation from John Zukowski, which may help to illuminate the purpose of the **Action** interface.

> *"The Action interface is an extension to the ActionListener interface that's very flexible for defining shared event handlers independent of the components that act as the triggering agents. ..., the interface implements ActionListener and defines a lookup table data structure whose keys act as bound properties. Then, when an Action is associated with a component, these display properties are automatically carried over to it."*

The actionPerformed method and other things

Because the **Action** interface extends the **ActionListener** interface, it declares the **actionPerformed** method. In addition, this interface makes it possible to define the following items in a single location:

- Text strings that describe the properties of *action-aware* components associated with an **Action** object.
- Icons that may be displayed on *action-aware* components associated with an **Action** object.
- The *enabled/disabled* state that should be tracked by *action-aware* components registered as **PropertyChange** listeners on an **Action** object

The enabled/disabled state

In addition to the **actionPerformed** method inherited from the **ActionListener** interface, the **Action** interface declares six methods in support of the functionality described above.

The two methods shown in Figure 7 are the standard JavaBeans *setter* and *getter* methods for the **enabled** property.

isEnabled() - Returns the enabled state of the Action.

setEnabled(boolean b) - Sets the enabled state of the Action.

Figure 7

Although it isn't obvious in the method declarations, the **enabled** property is a *bound* property as described earlier in this article. *(In fact, the word bound doesn't appear anywhere in the JavaDocs description of javax.swing Interface Action for Java 2 Platform SE v1.4.0.)*

PropertyChangeListener registration

The two methods shown in Figure 8 are the standard JavaBeans methods for registering and unregistering a **PropertyChangeListener** object.

addPropertyChangeListener(PropertyChangeListener listener) - Adds a PropertyChange listener.

removePropertyChangeListener(PropertyChangeListener listener)- Removes a PropertyChange listener.

Figure 8

All objects that are registered as a **PropertyChangeListener** on the **Action** object are notified when the value of any *bound* property changes.

Keyed properties

The two methods shown in Figure 9 are used to *put* and *get* the values of properties that I will refer to as *keyed properties*. These methods behave like typical *put* and *get* methods for a hashtable object.

void **putValue**(String key, Object value) - Sets one of this object's properties using the associated key.

Object **getValue**(String key) - Gets one of this object's properties using the associated key.

Figure 9

The values of these properties consist of:

- Text strings that describe the properties of *action-aware* components associated with an **Action** object
- References to **ImageIcon** objects that may be displayed on *action-aware* components associated with an **Action** object.

The keys

The keys associated with these values are declared by the **Action** interface as **public static final** fields of type **String**, having the following names and purposes.

- **NAME** - Used for storing the name for the action. Appears at text on the face of a tool bar button or a menu item.
- **SHORT_DESCRIPTION** - Used for storing a short description for the action. This description is used for tooltip text.
- **SMALL_ICON** - Used for storing a small icon for the action. This icon appears on toolbar buttons and menu items.
- **ACCELERATOR_KEY** - Used for storing a **KeyStroke** to be used as the accelerator for the action.
- **MNEMONIC_KEY** - Used for storing a key code used as the mnemonic for the action.
- **ACTION_COMMAND_KEY** - Used to determine the command string for an **ActionEvent** object.
- **DEFAULT** - Can be used as the storage-retrieval key when setting or getting one of this object's properties.
- **LONG_DESCRIPTION** - Used for storing a longer description for the action.

The key values

Because these fields are declared in an interface, they are inherently **final**. They are all references to **String** objects. Although the documentation doesn't specify the values encapsulated in the **String** objects, it can be determined experimentally that as of version 1.4.0, the string values encapsulated in the **String** objects are:

- Name
- ShortDescription
- SmallIcon
- AcceleratorKey
- MnemonicKey
- ActionCommandKey
- Default
- LongDescription

Whenever possible you should refer to the constants symbolically *(as in Action.NAME)* and avoid using the encapsulated values explicitly. That way, if the encapsulated values change in some later version of Java, it will be less likely that your code will become broken.

Key-value pairs

The values associated with these keys *(in a key-value pair sense)* in any particular program are dependent on the program. You will see how some of these keys are used in the sample program to be discussed later, so I won't discuss them further at this time.

Keyed properties are bound properties

Apparently each of these keyed properties behaves like a *bound* property; meaning that all registered **PropertyChangeListener** objects are notified whenever the value of one of these properties changes. Whether or not that listener object takes any specific action as a result of the change is determined by the author of the class from which the **PropertyChangeListener** object is instantiated.

Implementing the Action interface

As is usually the case, you can start from scratch and define a new class that implements the **Action** interface. Among other things, this will require you to provide **PropertyChange** support.

If you elect to take this route, I recommend that you consider using an object instantiated from the **PropertyChangeSupport** class or the **SwingPropertyChangeSupport** class to handle the registration and notification of **PropertyChangeListener** objects.

However, in most cases, you won't need to start from scratch. Rather, you can define a new class that extends the **AbstractAction** class, and override any methods whose behavior needs to be different from that defined in that class.

The AbstractAction class

According to Sun,

> *"This class provides default implementations for the JFC **Action** interface. Standard behaviors like the get and set methods for **Action** object properties (icon, text, and enabled) are defined here. The developer need only subclass this abstract class and define the **actionPerformed** method."*

Often, by extending this class, instead of defining a new class that implements the **Action** interface, you can save yourself a lot of programming effort.
*(As a side note regarding the importance of the **Action** interface, more than two-dozen classes in v1.4.0 extend the **AbstractAction** class.)*

Fields of AbstractAction class

In addition to the **final** fields inherited from the **Action** interface, this class defines the following two fields:

- SwingPropertyChangeSupport **changeSupport**
- boolean **enabled**

The first field contains a reference to an object instantiated from the **SwingPropertyChangeSupport** class. As I mentioned earlier, this is a helper class designed to register and notify **PropertyChange** listeners

The second field is of type **boolean**, and specifies whether the action is enabled at a particular point in time. The default value is true.

Methods of AbstractAction class

The **AbstractAction** class provides default implementations for each of the six methods declared in the **Action** interface *(See Figures 7, 8, and 9.)*

In addition, the **AbstractAction** class defines the four methods shown in Figure 10.

clone() - Clones the abstract action. **firePropertyChange**(String propertyName, Object oldValue, Object newValue) - Supports reporting bound property changes.

getKeys() - Returns an array of Objects which are keys for which values have been set for this AbstractAction, or null if no keys have values set.

getPropertyChangeListeners() - Returns an array of all the PropertyChangeListeners added to this AbstractAction with addPropertyChangeListener().

Figure 10

The behavior of the **clone** method should be self-explanatory.

firePropertyChange method

This method can be used in a subclass to fire a **PropertyChange** event. When a **PropertyChange** event is fired, the **SwingPropertyChangeSupport** object will send the appropriate **PropertyChangeEvent** to all registered **PropertyChangeListener** objects.

getKeys method

Returns an array of **Object** references which are the keys for which values have been set. Returns null if no keys have their values set. Can be used in an *action-aware* component to identify and get the values for keyed properties.

getPropertyChangeListeners()

Returns an array of all the PropertyChangeListeners added to this **AbstractAction** object with **addPropertyChangeListener()**. The array is empty if no listeners have been added. Can be used in a subclass to get a list of listeners.

Now for some code

As is my practice, I will discuss the program named **ActionObj02** in fragments. A complete listing of the program is shown in Listing 22 near the end of the lesson. The first fragment, shown in Listing 1, simply uses the **main** method to instantiate an object of the **GUI** class. I will discuss the definition of the **GUI** class later.

```
public class ActionObj02{
  public static void main(
                        String[]
args){
    new GUI();
  }//end main
```

```
}//end ActionObj02

Listing 1
```

The MyAction class

Listing 2 shows the entire class definition for the **MyAction** class from which **Action** objects will be instantiated.

```
class MyAction extends AbstractAction{
  public void actionPerformed(
                        ActionEvent e){
    System.out.println("Action: " +
       ((AbstractButton)e.getSource()).
                   getActionCommand());
  }//end actionPerformed
}//end class MyAction

Listing 2
```

Note that the **MyAction** class doesn't implement the **Action** interface directly. Rather, it extends the **AbstractAction** class and implements the **Action** interface through inheritance. This causes it to inherit the default behavior of the methods defined in the **AbstractAction** class and discussed earlier in this lesson.

Overrides actionPerformed

The **MyAction** class overrides the **actionPerformed** method. The overridden behavior of the method is to get and display the *action command* for any source object *(on which it is registered)* that fires an **ActionEvent**. By default, the *action command* is simply the text that is visible on the face of the source object. Thus, clicking any of the six components located on the **JMenu**, the **JToolBar**, or the **JPanel** *(see Figure 4)* will cause the text on the face of that component to be displayed.

The GUI class

Listing 3 shows the beginning of the **GUI** class.

```
class GUI extends JFrame{
  JMenuBar menuBar = new JMenuBar();
  JMenu menu = new JMenu("Menu");
  JToolBar toolBar = new JToolBar();
  JPanel panel = new JPanel();

Listing 3
```

The code in Listing 3 creates the following container objects, which are visible in Figure 4:

- JMenu
- JToolBar
- JPanel

Two Action objects

Listing 4 creates two **Action** objects *(instances of the **MyAction** class discussed above).* These two **Action** objects will be registered on various *action-aware* components later.

```
  Action actionObj01 = new MyAction();
  Action actionObj02 = new MyAction();

Listing 4
```

Control

Listing 5 creates the four check boxes and a control panel shown in the center of Figure 3.

```
  JCheckBox ckBox01 = new JCheckBox(
                          "Disable01");
  JCheckBox ckBox02 = new JCheckBox(
                          "Disable02");
  JCheckBox ckBox03 = new JCheckBox(
                        "Toggle Icon01");
  JCheckBox ckBox04 = new JCheckBox(
                        "Toggle Icon02");
  JPanel controlPanel = new JPanel();

Listing 5
```

The four check boxes will be used to manipulate the behavior of the two **Action** objects, and hence the behavior of the six associated *action-aware* components. Note that the text shown on the face of the checkboxes will be modified when those checkboxes are checked and cleared.

The GUI constructor

Listing 6 shows the beginning of the constructor for the **GUI** class.

```
  GUI(){//constructor
    menuBar.add(menu);

    menuBar.setBorder(
                new BevelBorder(
```

```
                    BevelBorder.RAISED));

    setJMenuBar(menuBar);

Listing 6
```

The code in Listing 6 adds the **JMenu** object to the **JMenuBar** object, which is the standard way to construct a menu in Java.

Then it invokes the **setBorder** method on the menu bar to cause its edges to become visible. I did this to make it easy to visually separate the menu bar from the tool bar that appears immediately below it in Figure 3. *(You can learn more about Swing borders at Baldwin's Java Programming Tutorials.)*

Set the JMenuBar

Finally, the code in Listing 6 adds the menu bar to the **JFrame**. Note, however, that this is accomplished by invoking the **setJMenuBar** method on the **JFrame** object instead of invoking an *add* method, *(which is the way that other components are normally added to a **JFrame** container.)*

A decorated JToolBarcontainer

Listing 7 decorates the **JToolBar** object with raised bevel borders, and places it in the NORTH position of the **JFrame** object. This causes the toolbar to appear immediately below the menu bar in Figure 3.

```
    toolBar.setBorder(new BevelBorder(
                    BevelBorder.RAISED));
    getContentPane().add(
            toolBar,BorderLayout.NORTH);

Listing 7
```

*(If need be, you can also learn more about the **BorderLayout** manager and the **getContentPane** method at Baldwin's Java Programming Tutorials.)*

A decorated JPanel container

Listing 8 decorates the **JPanel** object with raised bevel borders, and places it in the SOUTH position of the **JFrame** object. This causes it to appear at the bottom of the **JFrame** in Figure 3.

```
    panel.setBorder(new BevelBorder(
                    BevelBorder.RAISED));
    getContentPane().add(
```

```
            panel,BorderLayout.SOUTH);

Listing 8
```

Setting keyed values

Except for the definition of the **MyAction** class discussed earlier, very little of the code discussed so far has been specific to the **Action** interface. However, the code in Listing 9 is very specific to the **Action** interface.

```
    actionObj01.putValue(
            Action.NAME,"actionObj01");

Listing 9
```

Listing 9 invokes the **putValue** method on one of the **Action** objects to set the *value* for one of the *keyed properties.* As mentioned earlier, each **Action** object has a container for *keyed properties* that behaves much as a hashtable behaves. *Values* are stored against *keys*, and can be accessed later by specifying the *key.*

Values are stored in that container by invoking the **putValue** method and passing a *key* and a *value* as parameters. In this case, one of the constants *(final variables)* defined in the **Action** interface is used as the *key*. The literal **String** object *"actionObj01"* is passed as the value.

> *(Note that as discussed earlier, the constant is referred to symbolically as **Action.NAME**, and no explicit use is made of the actual **String** value encapsulated in that constant.)*

The remaining keyed properties

Listing 10 sets the values for each of the remaining *keyed properties.*

```
    actionObj01.putValue(
       Action.SMALL_ICON,new ImageIcon(
                        "redball.gif"));
    actionObj01.putValue(
            Action.SHORT_DESCRIPTION,
            "ToolTip for actionObj01");

    actionObj02.putValue(
            Action.NAME,"actionObj02");
    actionObj02.putValue(
       Action.SMALL_ICON,new ImageIcon(
                         "bulb2.gif"));
    actionObj02.putValue(
            Action.SHORT_DESCRIPTION,
            "ToolTip for actionObj02");
```

```
Listing 10
```

As mentioned earlier, these *keyed properties* behave as *bound properties*, meaning that registered objects are notified when their values change. Checking and clearing two of the checkboxes in this program causes the values of the **SMALL_ICON** properties to change, thus causing the icons displayed on the components registered on the respective **Action** objects to change accordingly.

Build the menu

Code that I discussed earlier created a menu bar and placed a menu on that bar. Now the time has come to place some menu items on that menu. This process begins in Listing 11.

```
    JMenuItem mnuA1 = new JMenuItem();
    mnuA1.setAction(actionObj01);
    menu.add(mnuA1);

Listing 11
```

Invoke the setAction method

The code in Listing 11 is very significant relative to the use of **Action** objects. Listing 11 instantiates a new **JMenuItem** object and then invokes the **setAction** method on that object, passing a reference to one of the **Action** objects as a parameter.

As discussed earlier, this causes the **Action** object to be registered as an action listener on the **JMenuItem** object.

PropertyChange listener registration

More importantly, however, for the purpose of this discussion, invocation of the **setAction** method causes the **JMenuItem** object to be registered as a **PropertyChange** listener on the **Action** object. Following this, when the *enabled* property, or any of the *keyed properties* of the **Action** object change, the **JMenuItem** object will be notified of the change.

The **JMenuItem** class is designed such that this will cause some of the corresponding properties on the **JMenuItem** object to change in order to track the change on the **Action** object. *(A **JMenuItem** object doesn't set or track all of the indexed properties.)*

After the **JMenuItem** object has been properly prepared, it is added to the menu in Listing 11.

An alternative to invoking setAction

Listing 12 shown an alternative approach to invoking the **setAction** method on the new **JMenuItem** object. (*This approach appears to achieve the same result as the invocation of the **setAction** method.).*

```
    //Put a JMenuItem on the menu. Set
    // its Action object in the
    // constructor. This is a
    // different approach.
    JMenuItem mnuA2 =
            new
JMenuItem(actionObj02);
    menu.add(mnuA2);

Listing 12
```

The **JMenuItem** class has a constructor that accepts a reference to an **Action** object as a parameter, and according to the Sun documentation, *"Creates a menu item whose properties are taken from the specified Action."*

An understatement

It appears that the statement in the Sun documentation is something of an understatement. In particular, it appears that constructing the **JMenuItem** object in this manner causes the **action** property of that object to be set to the **Action** object, just as though the **setAction** method were invoked on that object as in Listing 11.

As explained in the discussion of Listing 11, this involves much more than the Sun documentation statement would imply. In particular, the **Action** object becomes an *action listener* on the **JMenuItem** object, and the **JMenuItem** also becomes a *change listener* on the **Action** object.

Registration of two action listeners

Registering an **Action** object on an *action-aware* component does not prohibit the registration of ordinary **ActionListener** objects on the same component. This is illustrated in Listing 13.

```
    JButton butB1 = new JButton();

    butB1.addActionListener(
      new ActionListener(){
        public void actionPerformed(
                        ActionEvent
e){
          System.out.println(
           "Ordinary Action
Listener");
        }//end actionPerformed()
```

```
      }//end ActionListener
    );//end addActionListener

    butB1.setAction(actionObj01);

    toolBar.add(butB1);

Listing 13
```

Listing 13 instantiates a new **JButton** object, and then uses an anonymous inner class to register an ordinary **ActionListener** object on that component.

> *(If you are unfamiliar with anonymous inner classes, see Baldwin's Java Programming Tutorials.)*

Then it invokes the **setAction** method on the same **JButton** object, which causes the specified **Action** object to also be registered as an action listener on the **JButton** object.

Two actionPerformed methods are invoked

Thereafter, when the **JButton** object is clicked with the mouse, causing an action event to be fired by the **JButton** object, the following two lines of text appear on the screen:

```
Action: actionObj01
Ordinary Action Listener
```

The first line of output text is produced by the **actionPerformed** method defined in the **Action** object *(see Listing 2).*

The second line of output text is produced by the **actionPerformed** method defined for the ordinary **ActionListener** object *(see Listing 13).*

Finish populating the toolbar

Listing 14 finishes populating the toolbar by placing another **JButton** object on the toolbar.

```
    JButton butB2 = new JButton();
    butB2.setAction(actionObj02);
    toolBar.add(butB2);

Listing 14
```

Listing 14 invokes the **setAction** method on the **JButton** object to register the other **Action** object on that **JButton**. Thus, two different **Action** objects are registered on the two **JButton** objects on the toolbar.

Synchronized with menu items

Furthermore, the same two **Action** objects that are registered on the **JButton** objects on the tool bar are also registered on the two **JMenuItem** objects on the menu. Thus, the menu items and the toolbar buttons are tied together *(in pairs)* so as to behave in a synchronous fashion.

When one of the **Action** objects becomes disabled, the corresponding menu item and the corresponding toolbar button also become disabled. When the other **Action** object becomes disabled, the other menu item and the other toolbar button become disabled.

When an **Action** object becomes enabled, the corresponding menu item and the corresponding toolbar button become enabled.

When the value of the *keyed property* of an **Action** object identified with the key *Action.SMALL_ICON* changes, the corresponding menu item and the corresponding toolbar button track the change. This causes the image displayed on both of those components to change.

Various containers can be used

Probably the most common use of the **Action** interface is to synchronize the behavior of items on a menu with buttons on a toolbar. However, the use of the **Action** interface is not confined to those two types of containers. Rather, the **Action** interface can be used to synchronize the behavior of *action-aware* components in just about any kind of container. This is illustrated in Listing 15.

> *(You may want to avoid AWT containers to avoid the problems involved in mixing heavyweight and lightweight components. For example, it appears that you cannot control icons on a **JButton** that is placed in an AWT **Panel**.)*

```
    JButton butC = new JButton();
    butC.setAction(actionObj01);
    panel.add(butC);

    JMenuItem mnuC = new JMenuItem();
    mnuC.setAction(actionObj02);
    panel.add(mnuC);

Listing 15
```

Populate the JPanel

Listing 15 creates a new **JButton** object and a new **JMenuItem** object and places them in a common **JPanel** container object. The **setAction** method is used, along with the same two **Action** objects discussed earlier, to synchronize the behavior of these two components with the two components on the menu, and the two components on the toolbar.

> *(Hopefully by now you will have compiled and run this program. Due to the limitations of my verbal descriptions of behavior, running the program may ultimately be necessary for you to fully understand how this program behaves.)*

Construct control panel, set size, etc.

Listing 16 constructs the control panel containing check boxes that will be used to manipulate the properties of the two **Action** objects. Listing 16 also sets the size of the GUI and makes it visible.

```
    //Construct the control panel and
    // put it in the Center
    controlPanel.setLayout(
              new GridLayout(2,2,3,3));
    controlPanel.add(ckBox01);
    controlPanel.add(ckBox02);
    controlPanel.add(ckBox03);
    controlPanel.add(ckBox04);
    getContentPane().add(controlPanel,
                  BorderLayout.CENTER);

    //Set the size and make the GUI
    // visible
    setSize(350,200);
    setVisible(true);
    setTitle("Copyright 2002, " +
                        "R.G.Baldwin");
```

Listing 16

The code in Listing 16 is completely straightforward and doesn't deserve further discussion in this article.

Register checkbox event handlers to manipulate Action properties

The anonymous inner listener class in Listing 17 makes it possible for the user to manipulate the **enabled** property of one of the **Action** objects and its associated visual components.

The code in Listing 17, *(and the code that follows in other listings)*, registers **ActionListener** objects on each of the checkboxes.

> *(Note that these ActionListener objects registered on the checkboxes are completely independent of the **Action** objects registered on the action-aware components.)*

This control structure makes it possible for the user to disable and enable the two **Action** objects independently of one another simply by checking or clearing the checkboxes.

This control structure also makes it possible to toggle the icons between two different images on each **Action** object when the **Action** object is enabled.

```
    ckBox01.addActionListener(
      new ActionListener(){
        public void actionPerformed(
                        ActionEvent e){
          if(e.getActionCommand().
                  equals("Disable01")){
            ckBox01.setText(
                          "Enable01");
            actionObj01.setEnabled(
                                 false);
            //Disable ability to toggle
            // the IconImage.
            ckBox03.setEnabled(false);
          }else{
            ckBox01.setText(
                        "Disable01");
            actionObj01.setEnabled(
                                  true);
            //Enable ability to toggle
            // the IconImage.
            ckBox03.setEnabled(true);
          }//end else
        }//end actionPerformed()
      }//end ActionListener
    );//end addActionListener

Listing 17
```

Two important statements

Although the code in Listing 17 appears to be long and complex, it is relatively straightforward.

The **actionPerformed** method shown in Listing 17 is invoked when the checkbox fires an action event *(when the user checks or clears the checkbox).* Let me draw your attention to the two boldface statements in Listing 17. These two statements use the current state of the checkbox object to make a decision between two alternatives, and then to change the value of the **enabled** property of the **Action** object referred to by **actionObj01**. *(The state of the checkbox object is also changed accordingly.)*

When the value of the **enabled** property of the **Action** object is changed, the three components registered earlier as *change listeners* on the **Action** object are notified of the change. The code in the classes from which those components were instantiated causes them to become enabled or disabled accordingly. In other words, the *enabled* state of all three components tracks the **enabled** property of the **Action** object.

The other Action object

The code in Listing 18 uses a different checkbox object to provide essentially the same *enabled/disabled* control for the other **Action** object and its associated components. Since this code replicates the code in Listing 17, I deleted most of it for brevity. You can view all of the

code in Listing 22 near the end of this lesson.

```
    ckBox02.addActionListener(
      //...code deleted for brevity
    );//end addActionListener

Listing 18
```

Manipulate the icons

Listing 19 is an anonymous inner listener class that makes it possible for the user to manipulate a *keyed property* value of an **Action** object, the value of which specifies the icon to be displayed on registered components.

Basically, this code causes the value of the property that specifies the icon displayed on the components to toggle between two different **ImageIcon** objects based on two different *gif* files.

```
    ckBox03.addActionListener(
      new ActionListener(){
        public void actionPerformed(
                        ActionEvent e){
          //Get file name for the
          // ImageIcon object.
          String gif = (actionObj01.
                 getValue(
                   Action.SMALL_ICON)).
                            toString();

          if((gif).equals(
                       "redball.gif")){
            actionObj01.putValue(
                      Action.SMALL_ICON,
                       new ImageIcon(
                       "blueball.gif"));
          }else{
            actionObj01.putValue(
               Action.SMALL_ICON,
                          new ImageIcon(
                        "redball.gif"));
          }//end else

        }//end actionPerformed()
      }//end ActionListener
    );//end addActionListener

Listing 19
```

Change the displayed icon

The logic in Listing 19 is straightforward. When the checkbox fires an action event, the code in the **actionPerformed** method uses the **getValue** method of the **Action** interface to get the name of the *gif* file that specifies the icon. Then it changes the value of that *keyed property* in the **Action** object to an **ImageIcon** object based on the other *gif* file, thereby toggling the value between two **ImageIcon** objects based on the two different *gif* files.

When the value of this *keyed property* changes, the three components that were earlier registered as *change listeners* on the **Action** object are notified of the change. The code in the classes from which the three components were instantiated causes the icon that is displayed on the component to track the corresponding property value of the **Action** object.

Do it again for the other Action object

Listing 20 provides the same icon toggling capability for the other **Action** object. Once again, since this code replicates the code in Listing 19, I deleted most of it for brevity. You can view all of the code in Listing 22 near the end of this lesson.

```
    ckBox04.addActionListener(
      //... code deleted for brevity
    );//end addActionListener

Listing 20
```

Finally, the end of the program

Listing 21 is a simple WindowListener that terminates the program when the user clicks the *close* button on the **JFrame** object. This is relatively standard material, which should not require a discussion in this context.

```
    this.addWindowListener(
      new WindowAdapter(){
        public void windowClosing(
                        WindowEvent
e){
          System.exit(0);
        }//end windowClosing()
      }//end WindowAdapter
    );//end addWindowListener
  }//end constructor
}//end GUI class

Listing 21
```

Listing 21 is also the end of the **GUI** class definition.

Run the Program

If you haven't done so already, I encourage you to copy the code from Listing 22 into your text editor. Then compile and run it. Make changes and experiment with it in order to get a feel for the use of the **Action** interface.
*(Note that you will need a couple of gif files containing small images to use as **ImageIcon** objects. Any small image will do. Just be sure to handle the file names properly.)*

Complete Program Listing

A complete listing of the program is shown in Listing 22 below.

```
/*ActionObj02.java
Rev 03/30/02
Copyright 2002, R.G.Baldwin

Illustrates use of action objects.
Uses the setAction method that was
released with V1.3

Tested using JDK 1.4.0 under
Win2000.
**************************************/
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.awt.event.*;

public class ActionObj02{
  public static void main(
                        String[] args){
    new GUI();
  }//end main
}//end ActionObj02
//====================================//

class GUI extends JFrame{
  //Create three containers
  JMenuBar menuBar = new JMenuBar();
  JMenu menu = new JMenu("Menu");
  JToolBar toolBar = new JToolBar();
  JPanel panel = new JPanel();

  //Create two Action objects
  Action actionObj01 = new MyAction();
  Action actionObj02 = new MyAction();

  //Create four check boxes and a
  // control panel, which will be used
  // to manipulate the Action objects
  // and their visual components.
  JCheckBox ckBox01 = new JCheckBox(
```

```
                          "Disable01");
  JCheckBox ckBox02 = new JCheckBox(
                          "Disable02");
  JCheckBox ckBox03 = new JCheckBox(
                         "Toggle Icon01");
  JCheckBox ckBox04 = new JCheckBox(
                         "Toggle Icon02");
  JPanel controlPanel = new JPanel();

  GUI(){//constructor
    //Construct and decorate the menu
    // and put it in place.
    menuBar.add(menu);
    menuBar.setBorder(
               new BevelBorder(
                  BevelBorder.RAISED));
    setJMenuBar(menuBar);

    //Decorate the JToolBar and place
    // it in the North position.
    toolBar.setBorder(new BevelBorder(
                  BevelBorder.RAISED));
    getContentPane().add(
           toolBar,BorderLayout.NORTH);

    //Decorate the JPanel and place it
    // in the South position.
    panel.setBorder(new BevelBorder(
                  BevelBorder.RAISED));
    getContentPane().add(
             panel,BorderLayout.SOUTH);

    //Set some keyed properties for
    // each of the two Action objects.
    actionObj01.putValue(
            Action.NAME,"actionObj01");
    actionObj01.putValue(
       Action.SMALL_ICON,new ImageIcon(
                        "redball.gif"));
    actionObj01.putValue(
            Action.SHORT_DESCRIPTION,
            "ToolTip for actionObj01");

    actionObj02.putValue(
            Action.NAME,"actionObj02");
    actionObj02.putValue(
       Action.SMALL_ICON,new ImageIcon(
                         "bulb2.gif"));
    actionObj02.putValue(
            Action.SHORT_DESCRIPTION,
            "ToolTip for actionObj02");

    //Put a JMenuItem on the menu.  Set
    // its Action object using the
    // setAction method. This is
    // one approach.
```

```
    JMenuItem mnuA1 = new JMenuItem();
    mnuA1.setAction(actionObj01);
    menu.add(mnuA1);

    //Put a JMenuItem on the menu. Set
    // its Action object in the
    // constructor. This is a
    // different approach.
    JMenuItem mnuA2 =
            new JMenuItem(actionObj02);
    menu.add(mnuA2);


    //Put a JButton on the toolbar.
    // Set its Action object using the
    // setAction method.  Also register
    // an ordinary Action listener
    // on it.
    JButton butB1 = new JButton();
    butB1.addActionListener(
      new ActionListener(){
        public void actionPerformed(
                        ActionEvent e){
          System.out.println(
           "Ordinary Action Listener");
        }//end actionPerformed()
      }//end ActionListener
    );//end addActionListener
    butB1.setAction(actionObj01);
    toolBar.add(butB1);

    //Put a JButton on the toolbar.
    // Set its Action object.
    JButton butB2 = new JButton();
    butB2.setAction(actionObj02);
    toolBar.add(butB2);

    //Put a JButton on the JPanel.
    // Set its Action object.
    JButton butC = new JButton();
    butC.setAction(actionObj01);
    panel.add(butC);

    //Put a JMenuItem on the JPanel.
    // Set its Action object.
    JMenuItem mnuC = new JMenuItem();
    mnuC.setAction(actionObj02);
    panel.add(mnuC);


    //Construct the control panel and
    // put it in the Center
    controlPanel.setLayout(
              new GridLayout(2,2,3,3));
    controlPanel.add(ckBox01);
    controlPanel.add(ckBox02);
```

```
    controlPanel.add(ckBox03);
    controlPanel.add(ckBox04);
    getContentPane().add(controlPanel,
                  BorderLayout.CENTER);

    //Set the size and make the GUI
    // visible
    setSize(350,200);
    setVisible(true);
    setTitle("Copyright 2002, " +
                        "R.G.Baldwin");

    //The following anonymous inner
    // allow the user to manipulate the
    // Action objects and their
    // associated visual components.

    //Register ActionListener objects
    // on each of the check boxes.
    // This makes it possible to
    // disable and enable the two
    // Actions objects independently of
    // one another.  It also makes it
    // possible to toggle the icons
    // between two different images
    // on each Action object when the
    // Action object is enabled.
    ckBox01.addActionListener(
      new ActionListener(){
        public void actionPerformed(
                          ActionEvent e){
          if(e.getActionCommand().
                  equals("Disable01")){
            ckBox01.setText(
                           "Enable01");
            actionObj01.setEnabled(
                                 false);
            //Disable ability to toggle
            // the IconImage.
            ckBox03.setEnabled(false);
          }else{
            ckBox01.setText(
                         "Disable01");
            actionObj01.setEnabled(
                                  true);
            //Enable ability to toggle
            // the IconImage.
            ckBox03.setEnabled(true);
          }//end else
        }//end actionPerformed()
      }//end ActionListener
    );//end addActionListener

    ckBox02.addActionListener(
      new ActionListener(){
        public void actionPerformed(
```

```
                        ActionEvent e){
        if(e.getActionCommand().
                equals("Disable02")){
          ckBox02.setText(
                          "Enable02");
          actionObj02.setEnabled(
                                false);
          //Disable ability to toggle
          // the IconImage.
          ckBox04.setEnabled(false);
        }else{
          ckBox02.setText(
                         "Disable02");
          actionObj02.setEnabled(
                                true);
          //Enable ability to toggle
          // the IconImage.
          ckBox04.setEnabled(true);
        }//end else
      }//end actionPerformed()
    }//end ActionListener
  );//end addActionListener

  ckBox03.addActionListener(
    new ActionListener(){
      public void actionPerformed(
                        ActionEvent e){
        //Get file name for the
        // ImageIcon object.
        String gif = (actionObj01.
               getValue(
                 Action.SMALL_ICON)).
                          toString();

        if((gif).equals(
                       "redball.gif")){
          actionObj01.putValue(
                    Action.SMALL_ICON,
                     new ImageIcon(
                     "blueball.gif"));
        }else{
          actionObj01.putValue(
             Action.SMALL_ICON,
                       new ImageIcon(
                      "redball.gif"));
        }//end else

      }//end actionPerformed()
    }//end ActionListener
  );//end addActionListener

  ckBox04.addActionListener(
    new ActionListener(){
      public void actionPerformed(
                        ActionEvent e){
        //Get file name for the
```

```
          // ImageIcon object.
          String gif = (actionObj02.
                 getValue(
                   Action.SMALL_ICON)).
                            toString();

          if((gif).equals(
                       "bulb2.gif")){
            actionObj02.putValue(
               Action.SMALL_ICON,
                         new ImageIcon(
                       "bulb1.gif"));
          }else{
            actionObj02.putValue(
               Action.SMALL_ICON,
                         new ImageIcon(
                         "bulb2.gif"));
          }//end else

        }//end actionPerformed()
      }//end ActionListener
    );//end addActionListener

    //Create a WindowListener used
    // to terminate the program
    this.addWindowListener(
      new WindowAdapter(){
        public void windowClosing(
                         WindowEvent e){
          System.exit(0);
        }//end windowClosing()
      }//end WindowAdapter
    );//end addWindowListener
  }//end constructor
}//end GUI class

//=================================//

//This is the class from which the
// Action objects are instantiated.
class MyAction extends AbstractAction{
  public void actionPerformed(
                        ActionEvent e){
    System.out.println("Action: " +
       ((AbstractButton)e.getSource()).
                    getActionCommand());
  }//end actionPerformed
}//end class MyAction
```

Listing 22

About the author

Richard Baldwin *is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is a combination of Java, C#, and XML. In addition to the many platform and/or language independent benefits of Java and C# applications, he believes that a combination of Java, C#, and XML will become the primary driving force in the delivery of structured information on the Web.*

Richard has participated in numerous consulting projects, and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which has gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

baldwin.richard@iname.com

-end-