Note from the Authors: This article is an excerpt from Chapter 2, Section 2.3 of *Advanced Java™ 2 Platform How to Program*. This article discusses Java™ Swing actions of GUI components. Readers should be familiar with Swing and event handling. The code examples included in this article show readers examples using the DEITEL™ signature *LIVE-CODE™ Approach*, which presents all concepts in the context of complete, working programs followed by the screen shots of the actual inputs and outputs.

2.3 Swing **Action**s

Applications often provide users with several different ways to perform a given task. For example, in a word processor there might be an **Edit** menu with menu items for cutting, copying and pasting text. There also might be a toolbar that has buttons for cutting, copying and pasting text. There also might be a pop-up menu to allow users to right click on a document to cut, copy or paste text. The functionality the application provides is the same in each case—the developer provides the various interface components for the user's convenience. However, the same GUI component instance (e.g., a **JButton** for cutting text) cannot be used for menus and toolbars and pop-up menus, so the developer must code the same functionality three times. If there were many such interface items, repeating this functionality would become tedious and error-prone.

The *Command design pattern* solves this problem by enabling developers to define the functionality (e.g., copying text) once in a reusable object that the developer then can add to a menu, toolbar or pop-up menu. This design pattern is called Command because it defines a user command or instruction. The **Action** interface defines required methods for the Java Swing implementation of the Command design pattern.

An **Action** represents user-interface logic and properties for GUI components that represent that logic, such as the label for a button, the text for a tool tip and the mnemonic key for keyboard access. The logic takes the form of an **actionPerformed** method that the event mechanism invokes in response to the user activating an interface component (e.g., the user clicking a **JButton**). Interface **Action** extends interface **ActionListener**, which enables **Action**s to process **ActionEvent**s generated by GUI components. Once a developer defines an **Action**, the developer can add that **Action** to a **JMenu** or **JToolBar**, just as if the **Action** were a **JMenuItem** or **JButton**. For example, when a developer adds an **Action** to a **JMenu**, the **JMenu** creates a **JMenuItem** for the **Action** and uses the **Action** properties to configure the **JMenuItem**.

Actions provide an additional benefit in that the developer can enable or disable all GUI components associated with an **Action** by enabling or disabling the **Action** itself. For example, copying text from a document first requires that the user select the text to be copied. If there is no selected text, the program should not allow the user to perform a copy operation. If the application used a separate **JMenuItem** in a **JMenu** and **JButton** in a **JToolBar** for copying text, the developer would need to disable each of these GUI components individually. Using **Action**s, the developer could disable the **Action** for copying text, which also would disable all associated GUI components.

ActionSample (Fig. 2.5) demonstrates two **Action**s. Lines 15–16 declare **Action** references **sampleAction** and **exitAction**. Lines 24–35 create an anonymous inner class that extends class **AbstractAction** and assigns the instance to reference **sampleAction**. Class **AbstractAction** facilitates creating **Action** objects. Class **AbstractAction** implements interface **Action**, but is marked **abstract** because class **AbstractAction** does not provide an implementation for method **actionPerformed**. Lines 26–34 implement method **actionPerformed**. The Swing event mechanism invokes method **actionPerformed** when the user activates a GUI component associated with **sampleAction**. We show how to create these GUI components shortly. Lines 29–30 in method **actionPerformed** display a **JOptionPane** message dialog to inform the user that **sampleAction** was invoked. Line 33 then invokes method **setEnabled** of interface **Action** on the **exitAction** reference. This enables the **exitAction** and its associated GUI components. Note that **Action**s are enabled by default. We disabled the **exitAction** (line 80) to demonstrate that this disables the GUI components associated with that **Action**.

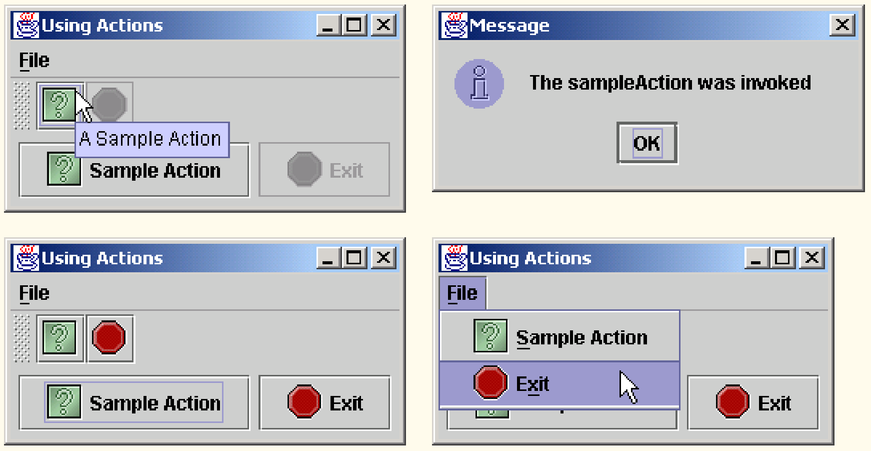
```java
 1  // ActionSample.java
 2  // Demonstrating the Command design pattern with Swing Actions.
 3  package com.deitel.advjhtp1.gui.actions;
 4
 5  // Java core packages
 6  import java.awt.*;
 7  import java.awt.event.*;
 8
 9  // Java extension packages
10  import javax.swing.*;
11
12  public class ActionSample extends JFrame {
13
14     // Swing Actions
15     private Action sampleAction;
16     private Action exitAction;
17
18     // ActionSample constructor
19     public ActionSample()
20     {
21        super( "Using Actions" );
22
23        // create AbstractAction subclass for sampleAction
24        sampleAction = new AbstractAction() {
25
26           public void actionPerformed( ActionEvent event )
27           {
28              // display message indicating sampleAction invoked
29              JOptionPane.showMessageDialog( ActionSample.this,
30                 "The sampleAction was invoked" );
31
32              // enable exitAction and associated GUI components
33              exitAction.setEnabled( true );
34           }
35        };
36
37        // set Action name
38        sampleAction.putValue( Action.NAME, "Sample Action" );
39
40        // set Action Icon
41        sampleAction.putValue( Action.SMALL_ICON, new ImageIcon(
42           getClass().getResource( "images/Help24.gif" ) ) );
43
44        // set Action short description (tooltip text)
45        sampleAction.putValue( Action.SHORT_DESCRIPTION,
46           "A Sample Action" );
```

Fig. 2.5 **ActionSample** application demonstrating the Command design pattern with Swing **Action**s (part 1 of 4).

```

      // set Action mnemonic key
      sampleAction.putValue( Action.MNEMONIC_KEY,
         new Integer( 'S' ) );

      // create AbstractAction subclass for exitAction
      exitAction = new AbstractAction() {

         public void actionPerformed( ActionEvent event )
         {
            // display message indicating exitAction invoked
            JOptionPane.showMessageDialog( ActionSample.this,
               "The exitAction was invoked" );
            System.exit( 0 );
         }
      };

      // set Action name
      exitAction.putValue( Action.NAME, "Exit" );

      // set Action icon
      exitAction.putValue( Action.SMALL_ICON, new ImageIcon(
         getClass().getResource( "images/EXIT.gif" ) ) );

      // set Action short description (tooltip text)
      exitAction.putValue( Action.SHORT_DESCRIPTION,
         "Exit Application" );

      // set Action mnemonic key
      exitAction.putValue( Action.MNEMONIC_KEY,
         new Integer( 'x' ) );

      // disable exitAction and associated GUI components
      exitAction.setEnabled( false );

      // create File menu
      JMenu fileMenu = new JMenu( "File" );

      // add sampleAction and exitAction to File menu to
      // create a JMenuItem for each Action
      fileMenu.add( sampleAction );
      fileMenu.add( exitAction );

      fileMenu.setMnemonic( 'F' );

      // create JMenuBar and add File menu
      JMenuBar menuBar = new JMenuBar();
      menuBar.add( fileMenu );
      setJMenuBar( menuBar );

      // create JToolBar
      JToolBar toolBar = new JToolBar();
```

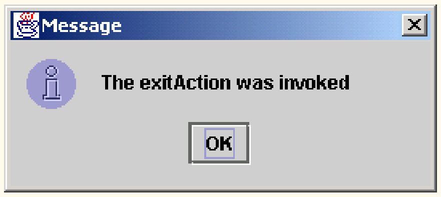
Fig. 2.5 **ActionSample** application demonstrating the Command design pattern with Swing **Action**s (part 2 of 4).

```java

      // add sampleAction and exitAction to JToolBar to create
      // JButtons for each Action
      toolBar.add( sampleAction );
      toolBar.add( exitAction );

      // create JButton and set its Action to sampleAction
      JButton sampleButton = new JButton();
      sampleButton.setAction( sampleAction );

      // create JButton and set its Action to exitAction
      JButton exitButton = new JButton( exitAction );

      // lay out JButtons in JPanel
      JPanel buttonPanel = new JPanel();
      buttonPanel.add( sampleButton );
      buttonPanel.add( exitButton );

      // add toolBar and buttonPanel to JFrame's content pane
      Container container = getContentPane();
      container.add( toolBar, BorderLayout.NORTH );
      container.add( buttonPanel, BorderLayout.CENTER );
   }

   // execute application
   public static void main( String args[] )
   {
      ActionSample sample = new ActionSample();
      sample.setDefaultCloseOperation( EXIT_ON_CLOSE );
      sample.pack();
      sample.setVisible( true );
   }
}
```



The sampleAction was invoked

Fig. 2.5 **ActionSample** application demonstrating the Command design pattern with Swing **Action**s (part 3 of 4).



The exitAction was invoked

Fig. 2.5 **`ActionSample`** application demonstrating the Command design pattern with Swing **`Action`**s (part 4 of 4).

After instantiating an **`AbstractAction`** subclass to create **`sampleAction`**, lines 38–50 repeatedly invoke method **`putValue`** of interface **`Action`** to configure **`sampleAction`** properties. Each property has a key and a value. Interface **`Action`** defines the keys as **`public`** constants, which we list in Fig. 2.6. GUI components associated with **`sampleAction`** use the property values we assign for GUI component labels, icons, tooltips and so on. Line 38 invokes method **`putValue`** of interface **`Action`** with arguments **`Action.NAME`** and **`"Sample Action"`**. This assigns **`sampleAction`**'s name, which GUI components use as their label. Lines 41–42 invoke method **`putValue`** of interface **`Action`** with key **`Action.SMALL_ICON`** and an **`ImageIcon`** value, which GUI components use as their **`Icon`**. Lines 45–46 set the **`Action`**'s tool tip using key **`Action.SHORT_DESCRIPTION`**. Lines 49–50 set the **`Action`**'s mnemonic key using key **`Action.MNEMONIC_KEY`**. When the **`Action`** is placed in a **`JMenu`**, the mnemonic key provides keyboard access to the **`Action`**. Lines 53–80 create the **`exitAction`** in a similar way to **`sampleAction`**, with an appropriate name, icon, description and mnemonic key. Line 80 invokes method **`setEnabled`** of interface **`Action`** with argument **`false`** to disable the **`exitAction`**. We use this to demonstrate that disabling an **`Action`** also disables the **`Action`**'s associated GUI components.

Line 83 creates the **`fileMenu`** **`JMenu`**, which contains **`JMenuItem`**s corresponding to **`sampleAction`** and **`exitAction`**. Class **`JMenu`** overloads method **`add`** with a version that takes an **`Action`** argument. This overloaded **`add`** method returns a reference to the **`JMenuItem`** that it creates. Lines 87–88 invoke method **`add`** of class **`JMenu`** to add **`sampleAction`** and **`exitAction`** to the menu. We have no need for the **`JMenuItem`** references that method **`add`** returns, so we ignore them. Line 90 sets the **`fileMenu`** mnemonic key, and lines 93–95 add the **`fileMenu`** to a **`JMenuBar`** and invoke method **`setJMenuBar`** of class **`JFrame`** to add the **`JMenuBar`** to the application.

Line 98 creates a new **`JToolBar`**. Like **`JMenu`**, **`JToolBar`** also provides overloaded method **`add`** for adding **`Action`**s to **`JToolBar`**s. Method **`add`** of class **`JToolBar`** returns a reference to the **`JButton`** created for the given **`Action`**. Lines 102–103 invoke method **`add`** of class **`JToolBar`** to add the **`sampleAction`** and **`exitAction`** to the **`JToolBar`**. We have no need for the **`JButton`** references that method **`add`** returns, so we ignore them.

Class **`JButton`** provides method **`setAction`** for configuring a **`JButton`** with properties of an **`Action`**. Line 106 creates **`JButton`** **`sampleButton`**. Line 107 invokes method **`setAction`** of class **`JButton`** with a **`sampleAction`** argument to configure **`sampleButton`**. Line 110 demonstrates an alternative way to configure a **`JButton`** with properties from an **`Action`**. The **`JButton`** constructor is overloaded to accept an

Action argument. The constructor configures the **JButton** using properties from the given **Action**.

Software Engineering Observation 2.1

*According to the Java 2 SDK documentation, it is preferable to create **JButton**s and **JMenuItem**s, invoke method **setAction** then add the **JButton** or **JMenuItem** to its container, rather than adding the **Action** to the container directly. This is because most GUI-building tools do not support adding **Action**s to containers directly.*

Lines 113–120 add the newly created **JButton**s to a **JPanel** and lay out the **JToolBar** and **JPanel** in the **JFrame**'s content pane. Note that in the first screen capture of Fig. 2.5, the **JButton**s for **exitAction** appear grayed-out. This is because the **exitAction** is disabled. After invoking the **sampleAction**, the **exitAction** is **enabled** and appears in full color. Note also the tool tips, icons and labels on each GUI component. Each of these items was configured using properties of the respective **Action** object.

Figure 2.6 summarizes **Action** properties. Each property name is a **static** constant in interface **Action** and acts as a key for setting or retrieving the property value.

Subsequent sections of this chapter in *Advanced Java 2 Platform How to Program* demonstrate two alternative ways to create Swing **Action** instances. The first uses named inner classes. The second defines a generic **AbstractAction** subclass that provides a constructor for commonly used properties and *set* methods for each individual **Action** property.

Name	Description
NAME	Name to be used for GUI-component labels.
SHORT_DESCRIPTION	Descriptive text for use in tooltips.
SMALL_ICON	**Icon** for displaying in GUI-component labels.
MNEMONIC_KEY	Mnemonic key for keyboard access (e.g., for accessing menus and menu items using the keyboard).
ACCELERATOR_KEY	Accelerator key for keyboard access (e.g., using the *Ctrl* key).
ACTION_COMMAND_KEY	Key for retrieving command string to be used in **ActionEvent**s.
LONG_DESCRIPTION	Descriptive text, e.g., for application help.

Fig. 2.6 **Action** class **static** keys for **Action** properties.