FIT ICT Software Development

Lecture 10

Lecturer : Charles Tyner

In this Chapter you'll learn:

- The design principles of graphical user interfaces (GUIs).
- How to use Java's new, elegant, cross-platform Nimbus look-and-feel.
- To build GUIs and handle events generated by user interactions with GUIs.
- To understand the packages containing GUI components, event-handling classes and interfaces.
- To create and manipulate buttons, labels, lists, text fields and panels.
- To handle mouse events and keyboard events.
- To use layout managers to arrange GUI components

Introduction

- A graphical user interface (GUI) presents a user-friendly mechanism for interacting with an application.
  - Pronounced “GOO-ee”
  - Gives an application a distinctive “look” and “feel.”
  - Consistent, intuitive user-interface components give users a sense of familiarity
  - Learn new applications more quickly and use them more productively.

Introduction

- Built from GUI components.
  - Sometimes called controls or widgets—short for window gadgets.
- User interacts via the mouse, the keyboard or another form of input, such as voice recognition.
- IDEs (Integrated Development Environments)
  - Provide GUI design tools to specify a component's exact size and location in a visual manner by using the mouse.
  - Generates the GUI code for you.
  - Greatly simplifies creating GUIs, but each IDE has different capabilities and generates different code.

Introduction

- Example of a GUI: SwingSet3 application (Fig. 14.1) on the following slide
- title bar at top contains the window's title.
- menu bar contains menus (File and View).
- In the top-right region of the window is a set of buttons
  - Typically, users press buttons to perform tasks.
- In the GUI Components area of the window is a combo box;
  - User can click the down arrow at the right side of the box to select from a list of items.

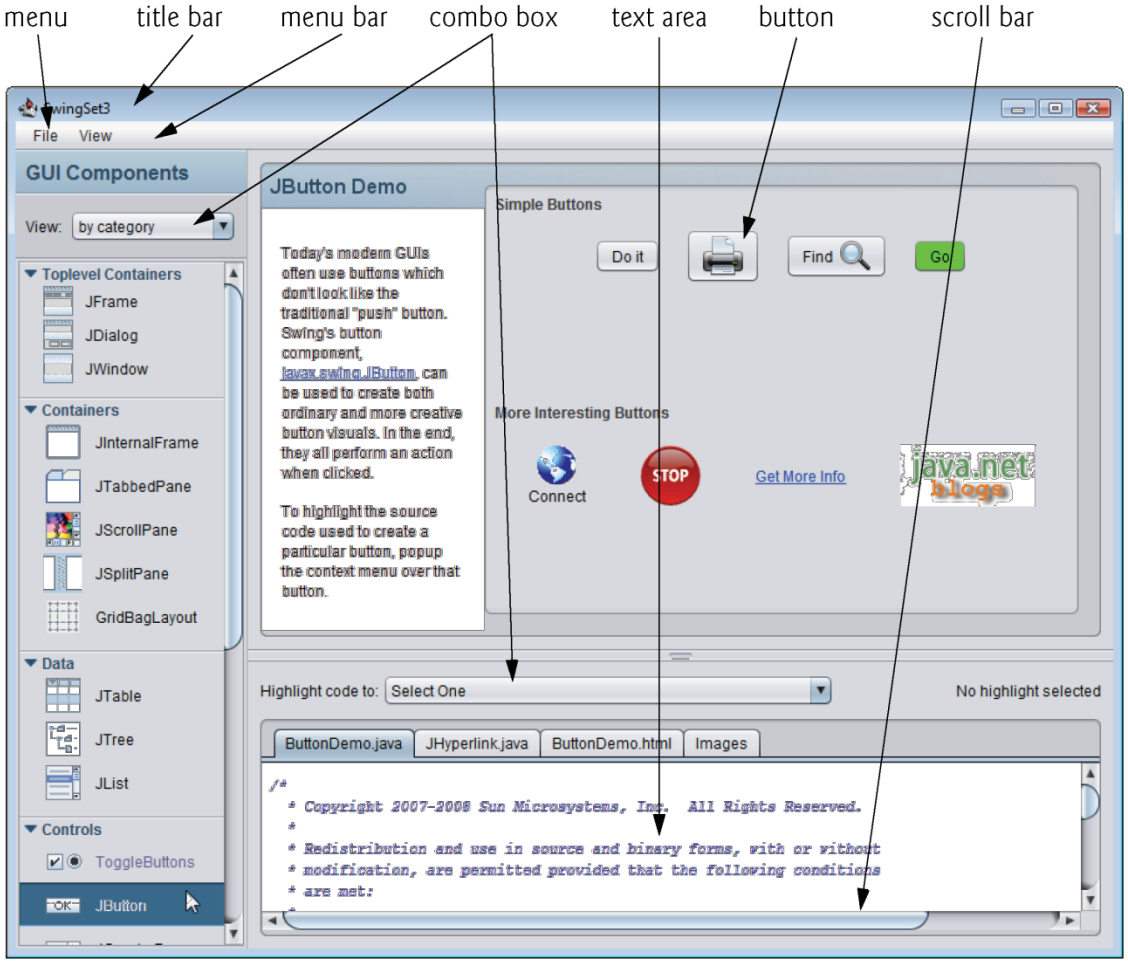


Fig. 14.1 | SwingSet3 application demonstrates many Swing GUI components.

Java's New Nimbus Look-and-Feel

- Java SE 6 update 10
- New, elegant, cross-platform look-and-feel known as Nimbus.
- We can configure our systems to use Nimbus as the default look-and-feel.

Java's New Nimbus Look-and-Feel

- Three ways to use Nimbus:
  - Set it as the default for all Java applications that run on your computer.
  - Set it as the look-and-feel when you launch an application by passing a command-line argument to the java command.
  - Set it as the look-and-feel programatically in your application

Java's New Nimbus Look-and-Feel (cont.)

- To set Nimbus as the default for all Java applications:
  - Create a text file named `swing.properties` in the `lib` folder of both your JDK installation folder and your JRE installation folder.
  - Place the following line of code in the file:

```
swing.defaultlaf=
   com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

- For more information on locating these installation folders visit
  - http://java.sun.com/javase/6/webnotes/install/index.html
- [Note: In addition to the standalone JRE, there is a JRE nested in your JDK's installation folder. If you are using an IDE that depends on the JDK (e.g., NetBeans), you may also need to place the `swing.properties` file in the nested `jre` folder's `lib` folder.]

Java's New Nimbus Look-and-Feel

- To select Nimbus on an application-by-application basis:
  - When running java from the command prompt…place the following command-line argument after the `java` command and before the application's name when you run the application:

```
-Dswing.defaultlaf=
  com.sun.java.swing.plaf.nimbus.
  NimbusLookAndFeel
```

Simple GUI-Based Input/Output with `JOptionPane`

- Most applications use windows or dialog boxes (also called dialogs) to interact with the user.
- `JOptionPane` (package `javax.swing`) provides prebuilt dialog boxes for input and output
  - Displayed via `static` `JOptionPane` methods.
- Figure 14.2 uses two input dialogs to obtain integers from the user and a message dialog to display the sum of the integers the user enters.

```
1  // Fig. 14.2: Addition.java
2  // Addition program that uses JOptionPane for input and output.
3  import javax.swing.JOptionPane; // program uses JOptionPane
4
5  public class Addition
6  {
7     public static void main( String[] args )
8     {
9        // obtain user input from JOptionPane input dialogs
10       String firstNumber =
11          JOptionPane.showInputDialog( "Enter first integer" );
12       String secondNumber =
13          JOptionPane.showInputDialog( "Enter second integer" );
14
15       // convert String inputs to int values for use in a calculation
16       int number1 = Integer.parseInt( firstNumber );
17       int number2 = Integer.parseInt( secondNumber );
18
19       int sum = number1 + number2; // add numbers
20
```

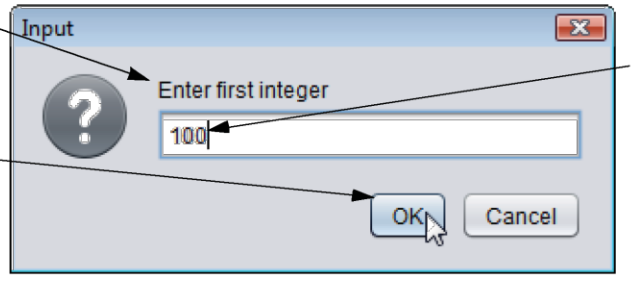
Displays an input dialog and returns a typed by the user

Fig. 14.2 | Addition program that uses `JOptionPane` for input and output. (Part 1 of 3.)

```
21          // display result in a JOptionPane message dialog
22          JOptionPane.showMessageDialog( null, "The sum is " + sum,
23             "Sum of Two Integers", JOptionPane.PLAIN_MESSAGE );
24       } // end method main
25    } // end class Addition
```

Displays a message dialog centered on the screen with no icon.

(a) Input dialog displayed by lines 10–11



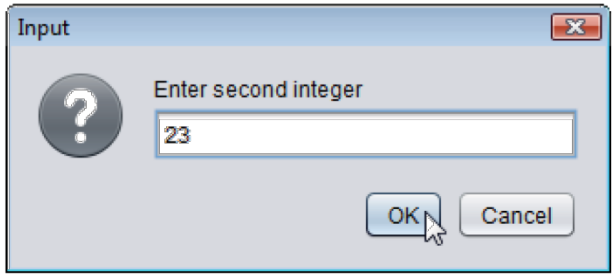
Prompt to the user

When the user clicks **OK**, `showInputDialog` returns to the program the **100** typed by the user as a `String`. The program must convert the `String` to an `int`

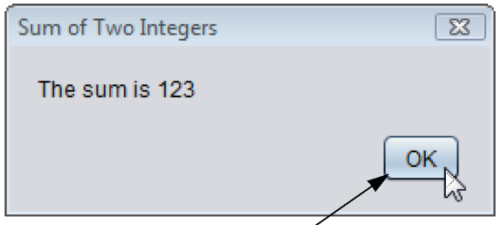
Text field in which the user types a value

Fig. 14.2 | Addition program that uses `JOptionPane` for input and output. (Part 2 of 3.)

(b) Input dialog displayed by lines 12–13



(c) Message dialog displayed by lines 22–23



When the user clicks **OK**, the message dialog is dismissed (removed from the screen).

Fig. 14.2 | Addition program that uses `JOptionPane` for input and output. (Part 3 of 3.)

Simple GUI-Based Input/Output with `JOptionPane` (cont.)

- `JOptionPane` `static` method `showInputDialog` displays an input dialog, using the method's `String` argument as a prompt.
  - The user types characters in the text field, then clicks OK or presses the *Enter* key to submit the `String` to the program.
  - Clicking OK dismisses (hides) the dialog.
  - Can input only `String`s. Typical of most GUI components.
  - If the user clicks Cancel, returns `null`.
  - `JOptionPane` dialog are dialog—the user cannot interact with the rest of the application while dialog is displayed.

Look-and-Feel Observation 14.2

The prompt in an input dialog typically uses ***sentence-style capitalization****—a style that capitalizes only the first letter of the first word in the text unless the word is a proper noun (for example, Jones).*

Look-and-Feel Observation 14.3

Do not overuse modal dialogs as they can reduce the usability of your applications. Use a modal dialog only when it's necessary to prevent users from interacting with the rest of an application until they dismiss the dialog. 14.3

Simple GUI-Based Input/Output with `JOptionPane` (cont.)

- Converting `String`s to `int` Values
  - `Integer` class's `static` method `parseInt` converts its `String` argument to an `int` value.
- Message Dialogs
  - `JOptionPane` `static` method `showMessageDialog` displays a message dialog.
  - The first argument helps determine where to position the dialog.
    - If `null`, the dialog box is displayed at the center of your screen.
  - The second argument is the message to display.
  - The third argument is the `String` that should appear in the title bar at the top of the dialog.
  - The fourth argument is the type of message dialog to display.

Simple GUI-Based Input/Output with `JOptionPane` (cont.)

- Message Dialogs
  - A `JOption-Pane.PLAIN_MESSAGE` dialog does not display an icon to the left of the message.
- `JOptionPane` online documentation:
  - `http://java.sun.com/javase/6/docs/api/javax/swing/JOptionPane.html`

Look-and-Feel Observation 14.4

The title bar of a window typically uses ***book-title capitalization****—a style that capitalizes the first letter of each significant word in the text and does not end with any punctuation (for example, Capitalization in a Book Title).*

Message dialog type	Icon	Description
`ERROR_MESSAGE`		Indicates an error to the user.
`INFORMATION_MESSAGE`		Indicates an informational message to the user.
`WARNING_MESSAGE`		Warns the user of a potential problem.
`QUESTION_MESSAGE`		Poses a question to the user. This dialog normally requires a response, such as clicking a **Yes** or a **No** button.
`PLAIN_MESSAGE`	no icon	A dialog that contains a message, but no icon.

Fig. 14.3 | `JOptionPane static` constants for message dialogs.

Overview of Swing Components

- Swing GUI components located in package `javax.swing`.
- Most are pure Java components
  - Written, manipulated and displayed completely in Java.
  - Part of the Java Foundation Classes (JFC) for cross-platform GUI development.
  - JFC and Java desktop technologies: http://java.sun.com/javase/technologies/desktop/

Component	Description
`JLabel`	Displays uneditable text or icons.
`JTextField`	Enables user to enter data from the keyboard. Can also be used to display editable or uneditable text.
`JButton`	Triggers an event when clicked with the mouse.
`JCheckBox`	Specifies an option that can be selected or not selected.
`JComboBox`	Provides a drop-down list of items from which the user can make a selection by clicking an item or possibly by typing into the box.
`JList`	Provides a list of items from which the user can make a selection by clicking on any item in the list. Multiple elements can be selected.
`JPanel`	Provides an area in which components can be placed and organized. Can also be used as a drawing area for graphics.

Fig. 14.4 | Some basic GUI components.

Overview of Swing Components (cont.)

- Abstract Window Toolkit (AWT) in package `java.awt` is another set of GUI components in Java.
  - When a Java application with an AWT GUI executes on different Java platforms, the application's GUI components display differently on each platform.
- Together, the appearance and the way in which the user interacts with the application are known as that application's look-and-feel.
- Swing GUI components allow you to specify a uniform look-and-feel for your application across all platforms or to use each platform's custom look-and-feel.

Portability Tip 14.1

Swing components are implemented in Java, so they are more portable and flexible than the original Java GUI components from package `java.awt`*, which were based on the GUI components of the underlying platform. For this reason, Swing GUI components are generally preferred.*

Overview of Swing Components (cont.)

- Most Swing components are not tied to actual GUI components of the underlying platform.
  - Known as lightweight components.
- AWT components are tied to the local platform and are called heavyweight components, because they rely on the local platform's windowing system to determine their functionality and their look-and-feel.
- Several Swing components are heavyweight components.

Portability Tip 14.2

The look-and-feel of a GUI defined with heavyweight GUI components from package `java.awt` *may vary across platforms. Because heavyweight components are tied to the local-platform GUI, the look-and-feel varies from platform to platform.*

Software Engineering Observation 14.1

Study the attributes and behaviors of the classes in the class hierarchy of Fig. 14.5. These classes declare the features that are common to most Swing components.

Overview of Swing Components (cont.)

- Class `Component` (package `java.awt`) declares many of the attributes and behaviors common to the GUI components in packages `java.awt` and `javax.swing`.
- Most GUI components extend class `Component` directly or indirectly.
- Online documentation:
  - http://java.sun.com/javase/6/docs/api/java/awt/Component.html

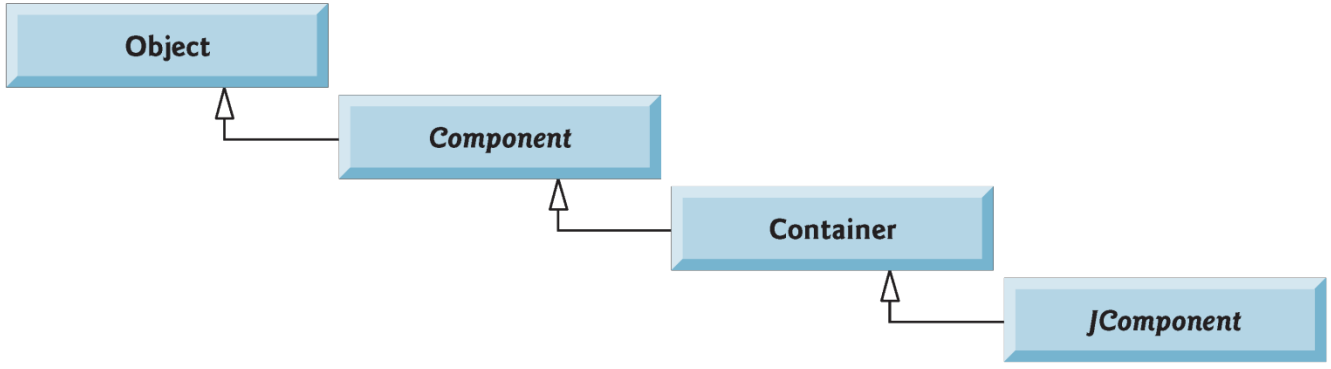


Fig. 14.5 | Common superclasses of many of the Swing components.

Overview of Swing Components (cont.)

- Class `Container` (package `java.awt`) is a subclass of `Component`.
- `Component`s are attached to `Container`s so that they can be organized and displayed on the screen.
- Any object that *is a* `Container` can be used to organize other `Component`s in a GUI.
- Because a `Container` *is a* `Component`, you can place `Container`s in other `Container`s to help organize a GUI.
- Online documentation:
  - http://java.sun.com/javase/6/docs/api/java/awt/Container.html

Overview of Swing Components (cont.)

- Class `JComponent` (package `javax.swing`) is a subclass of `Container`.
- `JComponent` is the superclass of all lightweight Swing components, all of which are also `Container`s.

Overview of Swing Components (cont.)

- Some common lightweight component features supported by `JComponent` include:
  - pluggable look-and-feel
  - Shortcut keys (called mnemonics)
  - Common event-handling capabilities for components that initiate the same actions in an application.
  - tool tips
  - Support for accessibility
  - Support for user-interface localization
- Online documentation:
  - http://java.sun.com/javase/6/docs/api/javax-/swing/JComponent.html

Displaying Text and Images in a Window (cont.)

- Most windows that can contain Swing GUI components are instances of class `JFrame` or a subclass of `JFrame`.
- `JFrame` is an indirect subclass of class `java.awt.Window`
- Provides the basic attributes and behaviors of a window
  - a title bar at the top
  - buttons to minimize, maximize and close the window
- Most of our examples will consist of two classes
  - a subclass of `JFrame` that demonstrates new GUI concepts
  - an application class in which `main` creates and displays the application's primary window.

```
1  // Fig. 14.6: LabelFrame.java
2  // Demonstrating the JLabel class.
3  import java.awt.FlowLayout; // specifies how components are arranged
4  import javax.swing.JFrame; // provides basic window features
5  import javax.swing.JLabel; // displays text and images
6  import javax.swing.SwingConstants; // common constants used with Swing
7  import javax.swing.Icon; // interface used to manipulate images
8  import javax.swing.ImageIcon; // loads images
9
10 public class LabelFrame extends JFrame
11 {
12    private JLabel label1; // JLabel with just text
13    private JLabel label2; // JLabel constructed with text and icon
14    private JLabel label3; // JLabel with added text and icon
15
16    // LabelFrame constructor adds JLabels to JFrame
17    public LabelFrame()
18    {
19       super( "Testing JLabel" );
20       setLayout( new FlowLayout() ); // set frame layout
21
```

Custom GUIs are often built in classes that extend `JFrame`

Sets the `JFrame`'s title bar text to the specified `String`

Fig. 14.6 | `JLabel`s with text and icons. (Part 1 of 2.)

```
22          // JLabel constructor with a string argument
23          label1 = new JLabel( "Label with text" );
24          label1.setToolTipText( "This is label1" );
25          add( label1 ); // add label1 to JFrame
26
27          // JLabel constructor with string, Icon and alignment arguments
28          Icon bug = new ImageIcon( getClass().getResource(  "bug1.png" ) );
29          label2 = new JLabel( "Label with text and icon", bug,
30             SwingConstants.LEFT );
31          label2.setToolTipText( "This is label2" );
32          add( label2 ); // add label2 to JFrame
33
34          label3 = new JLabel(); // JLabel constructor no arguments
35          label3.setText( "Label with icon and text at bottom" );
36          label3.setIcon( bug ); // add icon to JLabel
37          label3.setHorizontalTextPosition( SwingConstants.CENTER );
38          label3.setVerticalTextPosition( SwingConstants.BOTTOM );
39          label3.setToolTipText( "This is label3" );
40          add( label3 ); // add label3 to JFrame
41       } // end LabelFrame constructor
42    } // end class LabelFrame
```

Create a `JLabel` with the specified text then set its tooltip

Load in icon from the same location as class `LabelFrame`, then create a `JLabel` with text and an icon and set the `JLabel`'s tooltip text.

Create an empty `JLabel` then use set methods to change its characteristics.

Fig. 14.6 | `JLabel`s with text and icons. (Part 2 of 2.)

```
1  // Fig. 14.7: LabelTest.java
2  // Testing LabelFrame.
3  import javax.swing.JFrame;
4
5  public class LabelTest
6  {
7     public static void main( String[] args )
8     {
9        LabelFrame labelFrame = new LabelFrame(); // create LabelFrame
10       labelFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       labelFrame.setSize( 260, 180 ); // set frame size
12       labelFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class LabelTest
```

Program should terminate when the user clicks the window's close button.

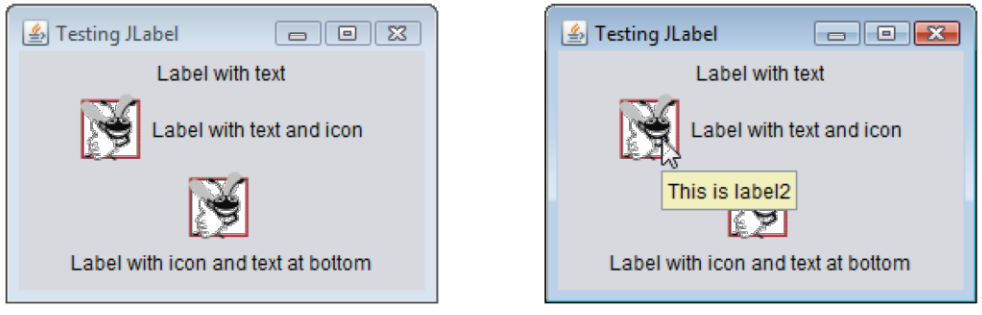


Fig. 14.7 | Test class for `LabelFrame`.

Displaying Text and Images in a Window (cont.)

- In a large GUI
  - Difficult to identify the purpose of every component.
  - Provide text stating each component's purpose.
- Such text is known as a label and is created with class `JLabel`—a subclass of `JComponent`.
  - Displays read-only text, an image, or both text and an image.

Displaying Text and Images in a Window (cont.)

- `JFrame`'s constructor uses its `String` argument as the text in the window's title bar.
- Must attach each GUI component to a container, such as a `JFrame`.
- You typically must decide where to position each GUI component.
  - Known as specifying the layout of the GUI components.
  - Java provides several layout managers that can help you position components.

Displaying Text and Images in a Window (cont.)

- Many IDEs provide GUI design tools in which you can specify the exact size and location of a component
- IDE generates the GUI code for you
- Greatly simplifies GUI creation
- We use Java's layout managers in our GUI examples instead of using an IDE as we need to know how to create this code ourselves

Displaying Text and Images in a Window (cont.)

- `FlowLayout`
  - GUI components are placed on a container from left to right in the order in which the program attaches them to the container.
  - When there is no more room to fit components left to right, components continue to display left to right on the next line.
  - If the container is resized, a `FlowLayout` reflows the components to accommodate the new width of the container, possibly with fewer or more rows of GUI components.
- Method `setLayout` is inherited from class `Container`.
  - argument must be an object of a class that implements the `LayoutManager` interface (e.g., `FlowLayout`).

Displaying Text and Images in a Window (cont.)

- `JLabel` constructor can receive a `String` specifying the label's text.
- Method `setToolTipText` (inherited by `JLabel` from `JComponent`) specifies the tool tip that is displayed when the user positions the mouse cursor over a `JComponent` (such as a `JLabel`).
- You attach a component to a container using the `add` method, which is inherited indirectly from class `Container`.

Displaying Text and Images in a Window (cont.)

- Icons enhance the look-and-feel of an application and are also commonly used to indicate functionality.
- An icon is normally specified with an `Icon` argument to a constructor or to the component's `setIcon` method.
- An `Icon` is an object of any class that implements interface `Icon` (package `javax.swing`).
- `ImageIcon` (package `javax.swing`) supports several image formats, including Graphics Interchange Format (GIF), Portable Network Graphics (PNG) and Joint Photographic Experts Group (JPEG).

Displaying Text and Images in a Window (cont.)

- `getClass().getResource("bug1.png")`
  - Invokes method `getClass` (inherited indirectly from class `Object`) to retrieve a reference to the `Class` object that represents the `LabelFrame` class declaration.
  - Next, invokes `Class` method `getResource`, which returns the location of the image as a URL.
  - The `ImageIcon` constructor uses the URL to locate the image, then loads it into memory.
  - The class loader knows where each class it loads is located on disk. Method `getResource` uses the `Class` object's class loader to determine the location of a resource, such as an image file.

Displaying Text and Images in a Window (cont.)

- A `JLabel` can display an `Icon`.
- `JLabel` constructor can receive text and an `Icon`.
  - The last constructor argument indicates the justification of the label's contents.
  - Interface `SwingConstants` (package `javax.swing`) declares a set of common integer constants (such as `SwingConstants.LEFT`) that are used with many Swing components.
  - By default, the text appears to the right of the image when a label contains both text and an image.
  - The horizontal and vertical alignments of a `JLabel` can be set with methods `setHorizontalAlignment` and `setVerticalAlignment`, respectively.

Constant	Description
Horizontal-position constants	
`SwingConstants.LEFT`	Place text on the left.
`SwingConstants.CENTER`	Place text in the center.
`SwingConstants.RIGHT`	Place text on the right.
Vertical-position constants	
`SwingConstants.TOP`	Place text at the top.
`SwingConstants.CENTER`	Place text in the center.
`SwingConstants.BOTTOM`	Place text at the bottom.

Fig. 14.8 | Positioning constants.

Displaying Text and Images in a Window (cont.)

- Class `JLabel` provides methods to change a label's appearance after it has been instantiated.
- Method `setText` sets the text displayed on the label.
- Method `getText` retrieves the current text displayed on a label.
- Method `setIcon` specifies the `Icon` to display on a label.
- Method `getIcon` retrieves the current `Icon` displayed on a label.
- Methods `setHorizontalTextPosition` and `setVerticalTextPosition` specify the text position in the label.

Displaying Text and Images in a Window (cont.)

- By default, closing a window simply hides the window.
- Calling method `setDefaultCloseOperation` (inherited from class `JFrame`) with the argument `JFrame.EXIT_ON_CLOSE` indicates that the program should terminate when the window is closed by the user.
- Method `setSize` specifies the width and height of the window in pixels.
- Method `setVisible` with the argument `true` displays the window on the screen.

Text Fields and an Introduction to Event Handling with Nested Classes

- GUIs are event driven.
- When the user interacts with a GUI component, the interaction—known as an event—drives the program to perform a task.
- The code that performs a task in response to an event is called an event handler, and the overall process of responding to events is known as event handling.

Text Fields and an Introduction to Event Handling with Nested Classes (cont.)

- `JTextFields` and `JPasswordFields` (package `javax.swing`).
- `JTextField` extends class `JTextComponent` (package `javax.swing.text`), which provides many features common to Swing's text-based components.
- Class `JPasswordField` extends `JTextField` and adds methods that are specific to processing passwords.
- `JPasswordField` shows that characters are being typed as the user enters them, but hides the actual characters with an echo character.

```
1  // Fig. 14.9: TextFieldFrame.java
2  // Demonstrating the JTextField class.
3  import java.awt.FlowLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JTextField;
8  import javax.swing.JPasswordField;
9  import javax.swing.JOptionPane;
10
11 public class TextFieldFrame extends JFrame
12 {
13    private JTextField textField1; // text field with set size
14    private JTextField textField2; // text field constructed with text
15    private JTextField textField3; // text field with text and size
16    private JPasswordField passwordField; // password field with text
17
18    // TextFieldFrame constructor adds JTextFields to JFrame
19    public TextFieldFrame()
20    {
21       super( "Testing JTextField and JPasswordField" );
22       setLayout( new FlowLayout() ); // set frame layout
23
```

Fig. 14.9 | JTextFields and JPasswordFields. (Part 1 of 4.)

```
24          // construct textfield with 10 columns
25          textField1 = new JTextField( 10 );
26          add( textField1 ); // add textField1 to JFrame
27
28          // construct textfield with default text
29          textField2 = new JTextField( "Enter text here" );
30          add( textField2 ); // add textField2 to JFrame
31
32          // construct textfield with default text and 21 columns
33          textField3 = new JTextField( "Uneditable text field", 21 );
34          textField3.setEditable( false ); // disable editing
35          add( textField3 ); // add textField3 to JFrame
36
37          // construct passwordfield with default text
38          passwordField = new JPasswordField( "Hidden text" );
39          add( passwordField ); // add passwordField to JFrame
40
41          // register event handlers
42          TextFieldHandler handler = new TextFieldHandler();
43          textField1.addActionListener( handler );
44          textField2.addActionListener( handler );
45          textField3.addActionListener( handler );
46          passwordField.addActionListener( handler );
47       } // end TextFieldFrame constructor
```

Width of the `JTextField` is based on the component's current font unless a layout manager overrides that size.

Width of the `JTextField` is based on the default text unless a layout manager overrides that size.

Width based on second argument unless a layout manager overrides that size.

Text in this component will be hidden by asterisks (*) by default.

`TextFieldHandler` inner class implements `ActionListener` interface, so it can respond to `JTextField` events. Lines 43–46 register the object `handler` to respond to each component's events.

Fig. 14.9 | `JTextFields` and `JPasswordFields`. (Part 2 of 4.)

```
48
49      // private inner class for event handling
50      private class TextFieldHandler implements ActionListener
51      {
52         // process text field events
53         public void actionPerformed( ActionEvent event )
54         {
55            String string = ""; // declare string to display
56
57            // user pressed Enter in JTextField textField1
58            if ( event.getSource() == textField1 )
59               string = String.format( "textField1: %s",
60                  event.getActionCommand() );
61
62            // user pressed Enter in JTextField textField2
63            else if ( event.getSource() == textField2 )
64               string = String.format( "textField2: %s",
65                  event.getActionCommand() );
66
67            // user pressed Enter in JTextField textField3
68            else if ( event.getSource() == textField3 )
69               string = String.format( "textField3: %s",
70                  event.getActionCommand() );
71
```

A `TextFieldHandler` is an `ActionListener`.

Called when the user presses *Enter* in a `JTextField` or `JPasswordField`.

`getSource` specifies which component the user interacted with

Obtains the text the user typed in the textfield.

Fig. 14.9 | `JTextFields` and `JPasswordFields`. (Part 3 of 4.)

```
72            // user pressed Enter in JTextField passwordField
73            else if ( event.getSource() == passwordField )
74               string = String.format( "passwordField: %s",
75                  event.getActionCommand() );
76
77            // display JTextField content
78            JOptionPane.showMessageDialog( null, string );
79         } // end method actionPerformed
80      } // end private inner class TextFieldHandler
81   } // end class TextFieldFrame
```

Fig. 14.9 | `JTextFields` and `JPasswordFields`. (Part 4 of 4.)

```
1  // Fig. 14.10: TextFieldTest.java
2  // Testing TextFieldFrame.
3  import javax.swing.JFrame;
4
5  public class TextFieldTest
6  {
7     public static void main( String[] args )
8     {
9        TextFieldFrame textFieldFrame = new TextFieldFrame();
10       textFieldFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       textFieldFrame.setSize( 350, 100 ); // set frame size
12       textFieldFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class TextFieldTest
```

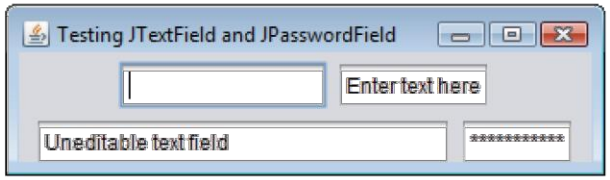


Fig. 14.10 | Test class for `TextFieldFrame`. (Part 1 of 3.)

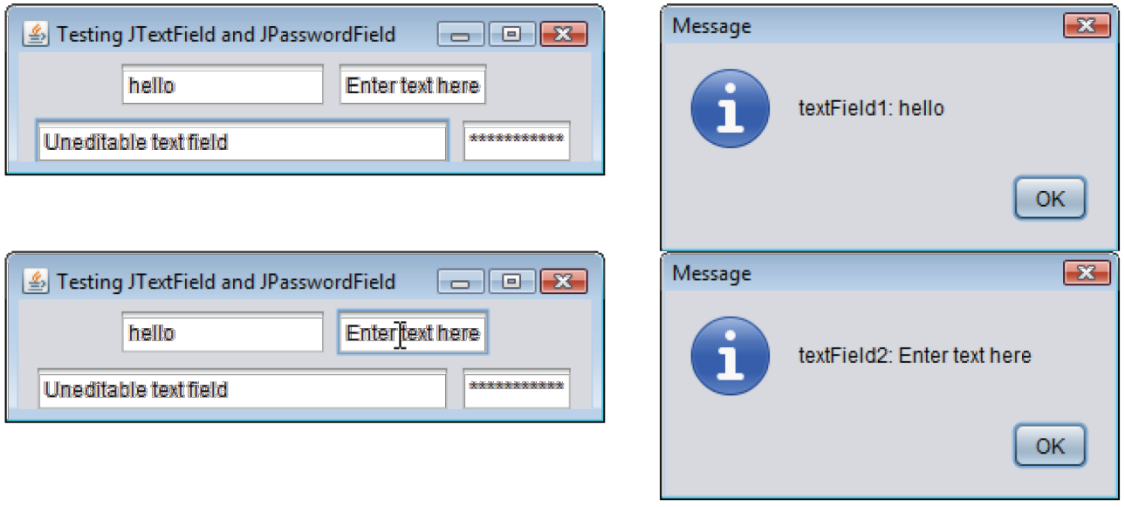


Fig. 14.10 | Test class for `TextFieldFrame`. (Part 2 of 3.)

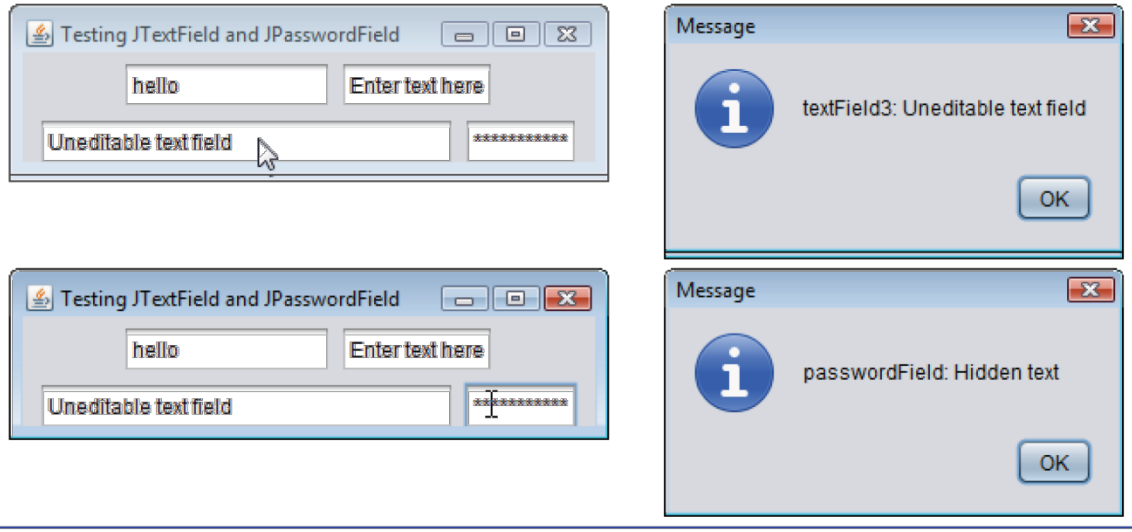


Fig. 14.10 | Test class for `TextFieldFrame`. (Part 3 of 3.)

Text Fields and an Introduction to Event Handling with Nested Classes (cont.)

- When the user types data into a `JTextField` or a `JPasswordField`, then presses *Enter*, an event occurs.
- You can type only in the text field that is "in focus."
- A component receives the focus when the user clicks the component.

Text Fields and an Introduction to Event Handling with Nested Classes (cont.)

- Before an application can respond to an event for a particular GUI component, you must perform several coding steps:
  - Create a class that represents the event handler.
  - Implement an appropriate interface, known as an event-listener interface, in the class from Step 1.
  - Indicate that an object of the class from Steps 1 and 2 should be notified when the event occurs. This is known as registering the event handler.

Text Fields and an Introduction to Event Handling with Nested Classes (cont.)

- All the classes discussed so far were so-called top-level classes—that is, they were not declared inside another class.
- Java allows you to declare classes inside other classes—these are called nested classes.
  - Can be `static` or non-`static`.
  - Non-`static` nested classes are called inner classes and are frequently used to implement event handlers.

Text Fields and an Introduction to Event Handling with Nested Classes (cont.)

- Before an object of an inner class can be created, there must first be an object of the top-level class that contains the inner class.
- This is required because an inner-class object implicitly has a reference to an object of its top-level class.
- There is also a special relationship between these objects—the inner-class object is allowed to directly access all the variables and methods of the outer class.
- A nested class that is `static` does not require an object of its top-level class and does not implicitly have a reference to an object of the top-level class.

Text Fields and an Introduction to Event Handling with Nested Classes (cont.)

- Inner classes can be declared `public`, `protected` or `private`.
- Since event handlers tend to be specific to the application in which they are defined, they are often implemented as `private` inner classes.

Text Fields and an Introduction to Event Handling with Nested Classes (cont.)

- GUI components can generate many events in response to user interactions.
- Each event is represented by a class and can be processed only by the appropriate type of event handler.
- Normally, a component's supported events are described in the Java API documentation for that component's class and its superclasses.

Text Fields and an Introduction to Event Handling with Nested Classes (cont.)

- When the user presses *Enter* in a `JTextField` or `JPasswordField`, an `ActionEvent` (package `java.awt.event`) occurs.
- Processed by an object that implements the interface `ActionListener` (package `java.awt.event`).
- To handle `ActionEvent`s, a class must implement interface `ActionListener` and declare method `actionPerformed`.
  - This method specifies the tasks to perform when an `ActionEvent` occurs.

Software Engineering Observation 14.3

The event listener for an event must implement the appropriate event-listener interface.

Common Programming Error 14.2

Forgetting to register an event-handler object for a particular GUI component's event type causes events of that type to be ignored.

Text Fields and an Introduction to Event Handling with Nested Classes (cont.)

- Must register an object as the event handler for each text field.
- `addActionListener` registers an `ActionListener` object to handle `ActionEvent`s.
- After an event handler is registered the object listens for events.

Text Fields and an Introduction to Event Handling with Nested Classes (cont.)

- The GUI component with which the user interacts is the event source.
- `ActionEvent` method `getSource` (inherited from class `EventObject`) returns a reference to the event source.
- `ActionEvent` method `getActionCommand` obtains the text the user typed in the text field that generated the event.
- `JPasswordField` method `getPassword` returns the password's characters as an array of type `char`.

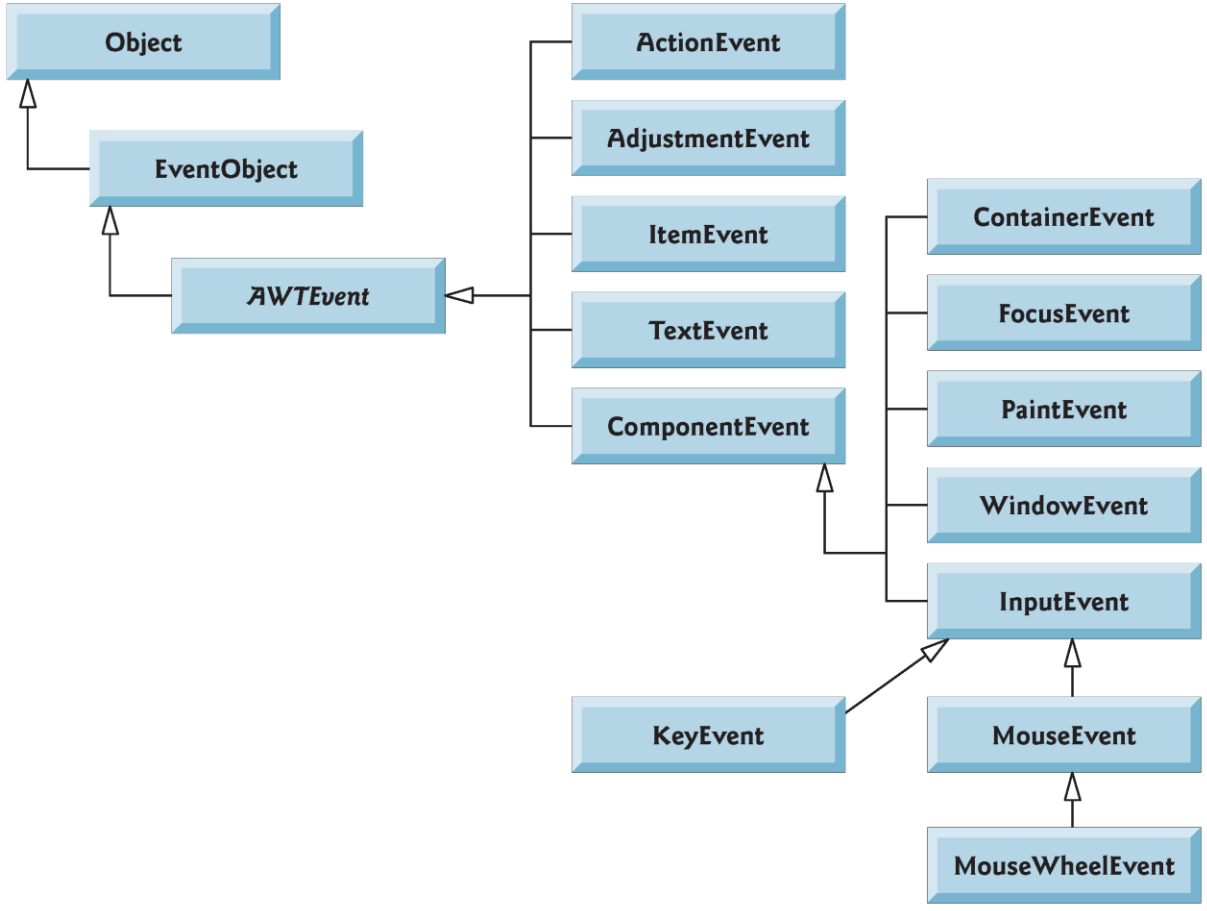


Fig. 14.11 | Some event classes of package `java.awt.event`.

Common GUI Event Types and Listener Interfaces

- Figure 14.11 illustrates a hierarchy containing many event classes from the package `java.awt.event`.
- Used with both AWT and Swing components.
- Additional event types that are specific to Swing GUI components are declared in package `javax.swing.event`.

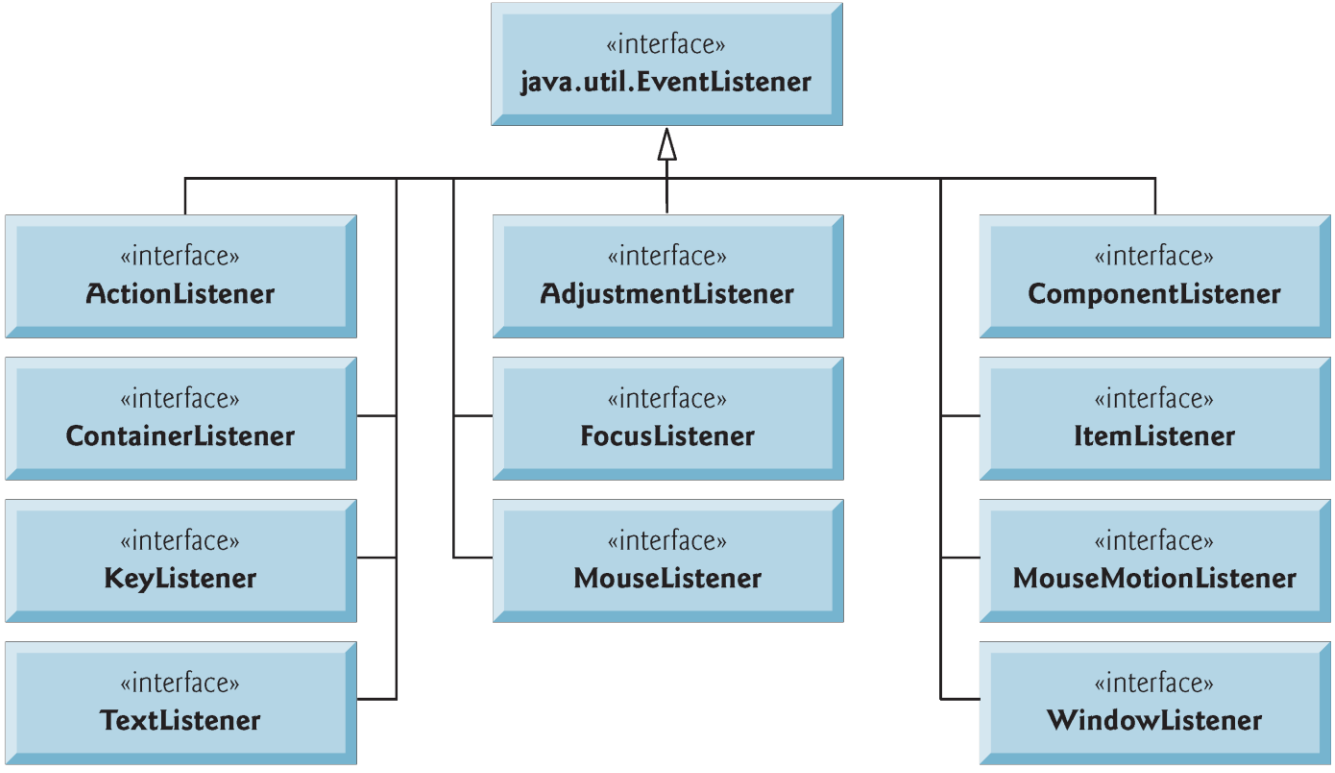


Fig. 14.12 | Some common event-listener interfaces of package `java.awt.event`.

Common GUI Event Types and Listener Interfaces (cont.)

- Delegation event model—an event's processing is delegated to an object (the event listener) in the application.
- For each event-object type, there is typically a corresponding event-listener interface.
- Many event-listener types are common to both Swing and AWT components.
  - Such types are declared in package `java.awt.event`, and some of them are shown in Fig. 14.12.
- Additional event-listener types that are specific to Swing components are declared in package `javax.swing.event`.

Common GUI Event Types and Listener Interfaces (cont.)

- Each event-listener interface specifies one or more event-handling methods that must be declared in the class that implements the interface.
- When an event occurs, the GUI component with which the user interacted notifies its registered listeners by calling each listener's appropriate event-handling method.

How Event Handling Works

- How the event-handling mechanism works:
- Every `JComponent` has a variable `listenerList` that refers to an `EventListenerList` (package `javax.swing.event`).
- Maintains references to registered listeners in the `listenerList`.
- When a listener is registered, a new entry is placed in the component's `listenerList`.
- Every entry also includes the listener's type.

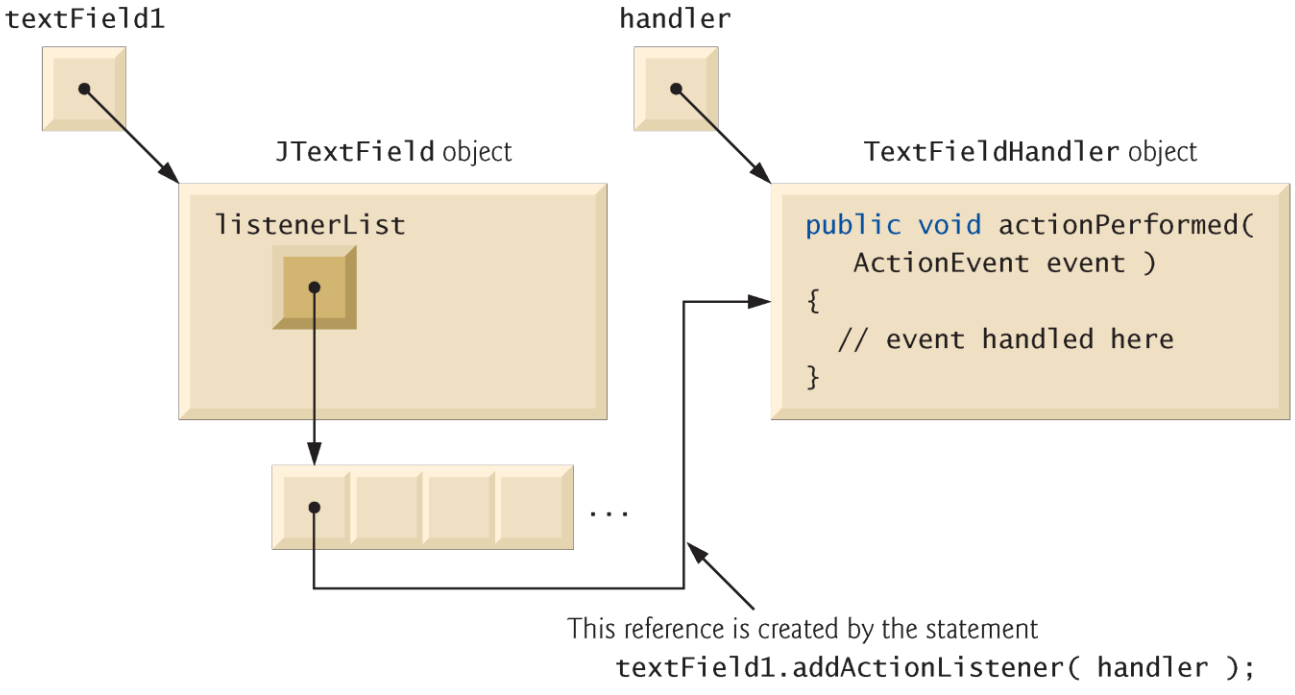


Fig. 14.13 | Event registration for `JTextField textField1`.

How Event Handling Works (cont.)

- How does the GUI component know to call `actionPerformed` rather than another method?
  - Every GUI component supports several event types, including mouse events, key events and others.
  - When an event occurs, the event is dispatched only to the event listeners of the appropriate type.
  - Dispatching is simply the process by which the GUI component calls an event-handling method on each of its listeners that are registered for the event type that occurred.

How Event Handling Works (cont.)

- Each event type has one or more corresponding event-listener interfaces.
  - `ActionEvent`s are handled by `ActionListener`s
  - `MouseEvent`s are handled by `MouseListener`s and `MouseMotionListener`s
  - `KeyEvent`s are handled by `KeyListener`s
- When an event occurs, the GUI component receives (from the JVM) a unique event ID specifying the event type.
  - The component uses the event ID to decide the listener type to which the event should be dispatched and to decide which method to call on each listener object.

How Event Handling Works (cont.)

- For an `ActionEvent`, the event is dispatched to every registered `ActionListener`'s `actionPerformed` method.
- For a `Mouse-Event`, the event is dispatched to every registered `MouseListener` or `MouseMotionListener`, depending on the mouse event that occurs.
  - The `MouseEvent`'s event ID determines which of the several mouse event-handling methods are called.

JButton

- A button is a component the user clicks to trigger a specific action.
- Several types of buttons
  - command buttons
  - checkboxes
  - toggle buttons
  - radio buttons
- Button types are subclasses of `AbstractButton` (package `javax.swing`), which declares the common features of Swing buttons.

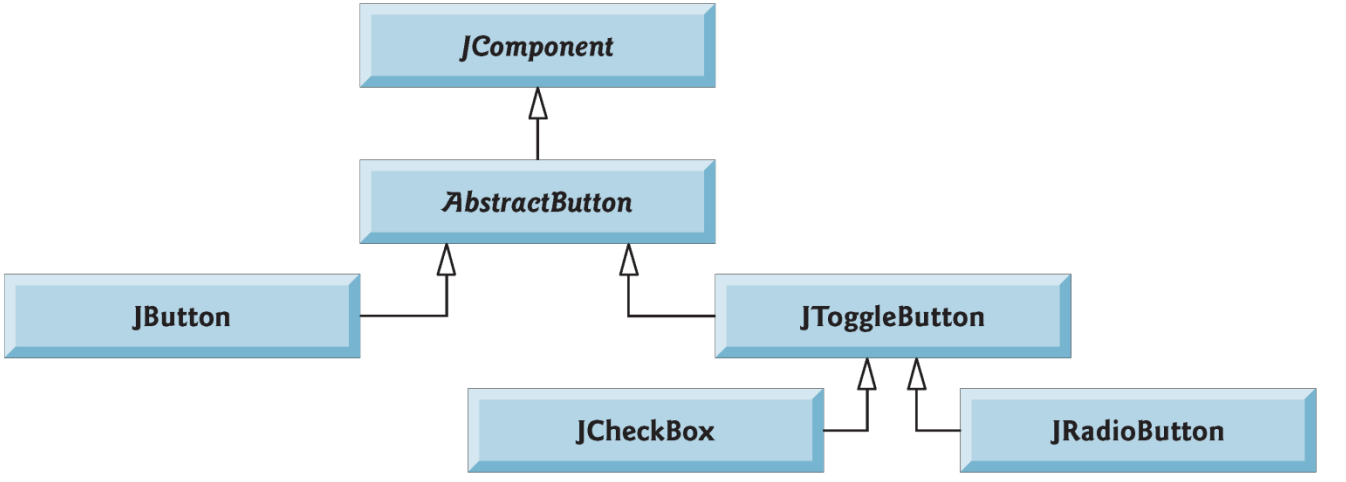


Fig. 14.14 | Swing button hierarchy.

`JButton` (cont.)

- A command button generates an `ActionEvent` when the user clicks it.
- Command buttons are created with class `JButton`.
- The text on the face of a `JButton` is called a button label.

```
1  // Fig. 14.15: ButtonFrame.java
2  // Creating JButtons.
3  import java.awt.FlowLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JButton;
8  import javax.swing.Icon;
9  import javax.swing.ImageIcon;
10 import javax.swing.JOptionPane;
11
12 public class ButtonFrame extends JFrame
13 {
14    private JButton plainJButton; // button with just text
15    private JButton fancyJButton; // button with icons
16
17    // ButtonFrame adds JButtons to JFrame
18    public ButtonFrame()
19    {
20       super( "Testing Buttons" );
21       setLayout( new FlowLayout() ); // set frame layout
22
23       plainJButton = new JButton( "Plain Button" ); // button with text
24       add( plainJButton ); // add plainJButton to JFrame
```

Creates a `JButton` with the specified text as its label.

Fig. 14.15 | Command buttons and action events. (Part 1 of 2.)

```
25
26          Icon bug1 = new ImageIcon( getClass().getResource( "bug1.gif" ) );
27          Icon bug2 = new ImageIcon( getClass().getResource( "bug2.gif" ) );
28          fancyJButton = new JButton( "Fancy Button", bug1 ); // set image
29          fancyJButton.setRolloverIcon( bug2 ); // set rollover image
30          add( fancyJButton ); // add fancyJButton to JFrame
31
32          // create new ButtonHandler for button event handling
33          ButtonHandler handler = new ButtonHandler();
34          fancyJButton.addActionListener( handler );
35          plainJButton.addActionListener( handler );
36       } // end ButtonFrame constructor
37
38       // inner class for button event handling
39       private class ButtonHandler implements ActionListener
40       {
41          // handle button event
42          public void actionPerformed( ActionEvent event )
43          {
44             JOptionPane.showMessageDialog( ButtonFrame.this, String.format(
45                "You pressed: %s", event.getActionCommand() ) );
46          } // end method actionPerformed
47       } // end private inner class ButtonHandler
48    } // end class ButtonFrame
```

Load two images from the same location as class `ButtonFrame`, then use the first as the default icon on the `JButton` and the second as the rollover icon.

Create object of inner class `ButtonHandler` and register it to handle the `ActionEvent`s for both `JButton`s.

Objects of this class can respond to `ActionEvent`s.

`ButtonFrame.this` is special notation that enables the inner class to access the `this` reference from the top-level class `ButtonFrame`.

Fig. 14.15 | Command buttons and action events. (Part 2 of 2.)

```
1  // Fig. 14.16: ButtonTest.java
2  // Testing ButtonFrame.
3  import javax.swing.JFrame;
4
5  public class ButtonTest
6  {
7     public static void main( String[] args )
8     {
9        ButtonFrame buttonFrame = new ButtonFrame(); // create ButtonFrame
10       buttonFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       buttonFrame.setSize( 275, 110 ); // set frame size
12       buttonFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class ButtonTest
```

Fig. 14.16 | Test class for `ButtonFrame`. (Part 1 of 2.)

Fig. 14.16 | Test class for `ButtonFrame`. (Part 2 of 2.)

JButton (cont.)

- A JButton can display an Icon.
- A JButton can also have a rollover Icon
  - displayed when the user positions the mouse over the JButton.
  - The icon on the JButton changes as the mouse moves in and out of the JButton's area on the screen.
- AbstractButton method setRolloverIcon specifies the image displayed on the JButton when the user positions the mouse over it.

Exercises

- Get all the code in this lecture up and running
- Comment and fully understand code