Java Programming Tutorial
Programming Graphical User Interface (GUI)

1. Introduction

So far, we have covered most of the basic constructs of Java and introduced the important concept of Object-Oriented Programming (OOP). As discussed, OOP permits higher level of abstraction than the traditional procedural-oriented languages (such as C and Pascal). OOP lets you think in the problem space rather than the computer's bits and bytes and instruction sequences. You can create high-level abstract data types called *classes* to mimic real-life things and represent entities in the problem space. These classes are self-contained and are reusable.

In this article, I shall show you how you can reuse the graphics classes provided in JDK for constructing your own Graphical User Interface (GUI) applications. Writing your own graphics classes (re-inventing the wheels) will take you many years! These graphics classes, developed by expert programmers, are highly complex and involve many advanced Java concepts. However, re-using them are not so difficult, if you follow the API documentation, samples and templates provided.

I shall assume you have a good grasp of OOP, including inheritance and polymorphism (otherwise, read the earlier articles). I will describe another important concept called *nested class* (or *inner class*) in this article.

There are two sets of Java APIs for graphics programming: AWT (Abstract Windowing Toolkit) and Swing.

1. AWT API was introduced in JDK 1.0. Most of the AWT components have become obsolete and should be replaced by newer Swing components.

2. Swing API, a much more comprehensive set of graphics libraries that enhances the AWT, was introduced as part of Java Foundation Classes (JFC) after the release of JDK 1.1. JFC consists of Swing, Java2D, Accessibility, Internationalization, and Pluggable Look-and-Feel Support APIs. JFC was an add-on to JDK 1.1 but has been integrated into core Java since JDK 1.2.

Other than AWT/Swing Graphics APIs provided in JDK, others have also provided Graphics APIs that work with Java, such as Eclipse's Standard Widget Toolkit (SWT) (used in Eclipse), Google Web Toolkit (GWT) (used in Android), 3D Graphics API such as Java bindings for OpenGL (JOGL) and Java3D.

You need to check the JDK API specification (http://docs.oracle.com/javase/8/docs/api/index.html) for the AWT and Swing APIs while reading this chapter. The best online reference for Graphics programming is the "Swing Tutorial" @ http://docs.oracle.com/javase/tutorial/uiswing/. For advanced 2D graphics programming, read "Java 2D Tutorial" @ http://docs.oracle.com/javase/tutorial/2d/index.html.

2. Programming GUI with AWT

Java Graphics APIs - AWT and Swing - provide a huge set of reusable GUI components, such as button, text field, label, choice, panel and frame for building GUI applications. You can simply reuse these classes rather than re-invent the wheels. I shall start with the AWT classes before moving into Swing to give you a complete picture. I have to stress that AWT component classes are now obsoleted by Swing's counterparts.

2.1 AWT Packages

AWT is huge! It consists of 12 packages (Swing is even bigger, with 18 packages as of JDK 1.8). Fortunately, only 2 packages - `java.awt` and `java.awt.event` - are commonly-used.

1. The `java.awt` package contains the *core* AWT graphics classes:
   - GUI Component classes (such as `Button`, `TextField`, and `Label`),
   - GUI Container classes (such as `Frame`, `Panel`, `Dialog` and `ScrollPane`),
   - Layout managers (such as `FlowLayout`, `BorderLayout` and `GridLayout`),
   - Custom graphics classes (such as `Graphics`, `Color` and `Font`).
2. The `java.awt.event` package supports event handling:
   - Event classes (such as `ActionEvent`, `MouseEvent`, `KeyEvent` and `WindowEvent`),
   - Event Listener Interfaces (such as `ActionListener`, `MouseListener`, `KeyListener` and `WindowListener`),
   - Event Listener Adapter classes (such as `MouseAdapter`, `KeyAdapter`, and `WindowAdapter`).

AWT provides a *platform-independent* and *device-independent* interface to develop graphic programs that runs on all platforms, such as Windows, Mac, and Linux.

2.2 Containers and Components
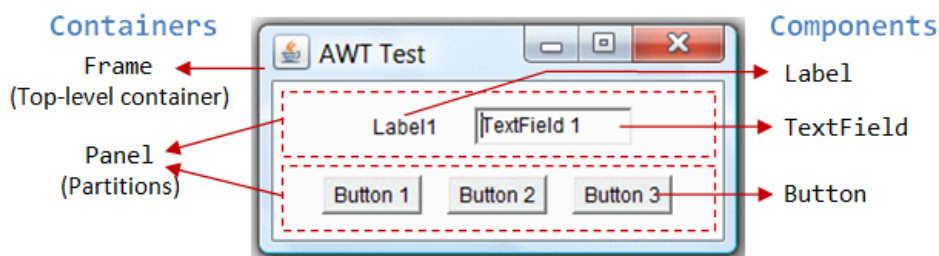
There are two types of GUI elements:

1. *Component*: Components are elementary GUI entities (such as `Button`, `Label`, and `TextField`.)
2. *Container*: Containers (such as `Frame` and `Panel`) are used to *hold components in a specific layout* (such as flow or grid). A container can also hold sub-containers.

GUI components are also called *controls* (Microsoft ActiveX Control), *widgets* (Eclipse's Standard Widget Toolkit, Google Web Toolkit), which allow users to interact with (or control) the application through these components (such as button-click and text-entry).

In the above figure, there are three containers: a `Frame` and two `Panels`. A `Frame` is the *top-level container* of an AWT program. A `Frame` has a title bar (containing an icon, a title, and the minimize/maximize/close buttons), an optional menu bar and the content display area. A `Panel` is a *rectangular area* used to group related GUI components in a certain layout. In the above figure, the top-level `Frame` contains two `Panels`. There are five components: a `Label` (providing description), a `TextField` (for users to enter text), and three `Buttons` (for user to trigger certain programmed actions).

In a GUI program, a component must be kept in a container. You need to identify a container to hold the components. Every container has a method called `add(Component c)`. A container (say aContainer) can invoke `aContainer.add(aComponent)` to add aComponent into itself. For example,

```
Panel panel = new Panel();        // Panel is a Container
Button btn = new Button("Press"); // Button is a Component
panel.add(btn);                   // The Panel Container adds a Button Component
```

2.3 AWT Container Classes

Top-Level Containers: `Frame`, `Dialog` **and** `Applet`

Each GUI program has a *top-level container*. The commonly-used top-level containers in AWT are `Frame`, `Dialog` and `Applet`:

- A `Frame` provides the "main window" for the GUI application, which has a title bar (containing an icon, a title, the minimize, maximize/restore-down and close buttons), an optional menu bar, and the content display area. To write a GUI program, we typically start with a subclass extending from `java.awt.Frame` to inherit the main window as follows:
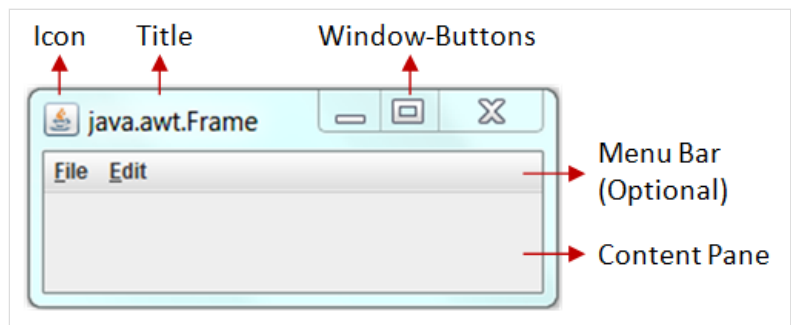
```java
import java.awt.Frame;  // Using Frame class in package java.awt

// A GUI program is written as a subclass of Frame - the top-level container
// This subclass inherits all properties from Frame, e.g., title, icon, buttons, content-pane
public class MyGUIProgram extends Frame {

   // Constructor to setup the GUI components
   public MyGUIProgram() { ...... }

   // Other methods
   ......
   ......

   // The entry main() method
   public static void main(String[] args) {
      // Invoke the constructor (to setup the GUI) by allocating an instance
      new MyGUIProgram();
   }
}
```

- An AWT `Dialog` is a "*pop-up window*" used for interacting with the users. A `Dialog` has a title-bar (containing an icon, a title and a close button) and a content display area, as illustrated.

- An AWT `Applet` (in package `java.applet`) is the top-level container for an applet, which is a Java program running inside a browser. Applet will be discussed in the later chapter.
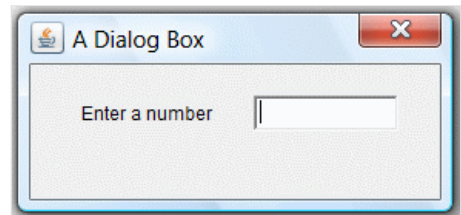
Secondary Containers: `Panel` and `ScrollPane`

Secondary containers are placed inside a top-level container or another secondary container. AWT also provide these secondary containers:

- `Panel`: a rectangular box under a higher-level container, used to *layout* a set of related GUI components in pattern such as grid or flow.

- `ScrollPane`: provides automatic horizontal and/or vertical scrolling for a single child component.

- others.

Hierarchy of the AWT `Container` Classes

The hierarchy of the AWT `Container` classes is as follows:
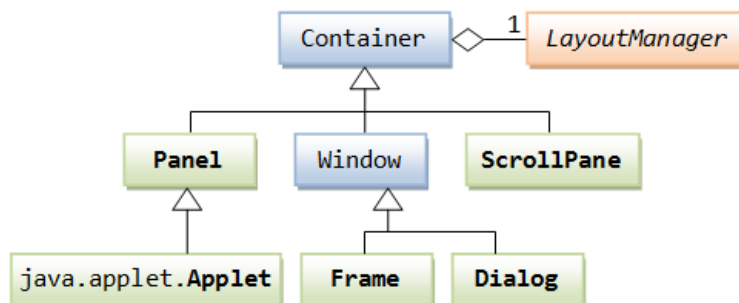
As illustrated, each `Container` has a layout.

2.4 AWT Component Classes

AWT provides many ready-made and reusable GUI components. The frequently-used are: `Button`, `TextField`, `Label`, `Checkbox`, `CheckboxGroup` (radio buttons), `List`, and `Choice`, as illustrated below.
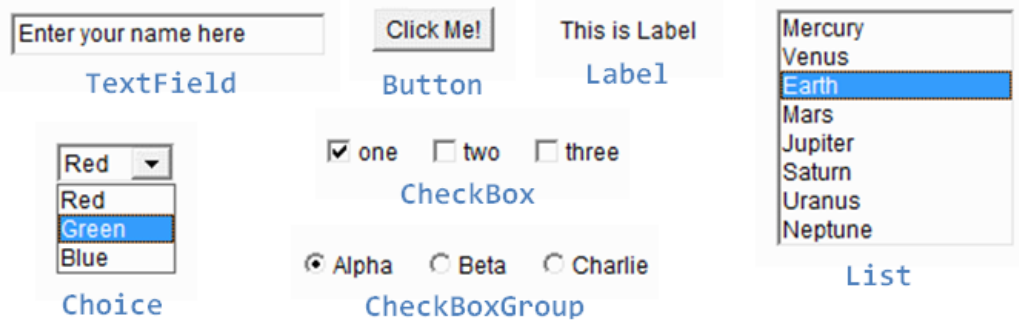
AWT GUI Component: `java.awt.Label`

A `java.awt.Label` provides a text description message. Take note that `System.out.println()` prints to the system console, not to the graphics screen. You could use a `Label` to label another component (such as text field) or provide a text description.

Check the JDK API specification for `java.awt.Label`.

Constructors

```
public Label(String strLabel, int alignment); // Construct a Label with the given text String, of the text alignment
public Label(String strLabel);                 // Construct a Label with the given text String
public Label();                                // Construct an initially empty Label
```

The `Label` class has three constructors:

1. The first constructor constructs a `Label` object with the given text string in the given alignment. Note that three `static` constants `Label.LEFT`, `Label.RIGHT`, and `Label.CENTER` are defined in the class for you to specify the alignment (rather than asking you to memorize arbitrary integer values).

2. The second constructor constructs a `Label` object with the given text string in default of left-aligned.

3. The third constructor constructs a `Label` object with an initially empty string. You could set the label text via the `setText()` method later.

Constants

```
public static final LEFT;     // Label.LEFT
public static final RIGHT;    // Label.RIGHT
public static final CENTER;   // Label.CENTER
```

These three constants are defined for specifying the alignment of the `Label`'s text.

Public Methods

```
// Examples
public String getText();
public void setText(String strLabel);
public int getAlignment();
public void setAlignment(int alignment);
```

The `getText()` and `setText()` methods can be used to read and modify the `Label`'s text. Similarly, the `getAlignment()` and `setAlignment()` methods can be used to retrieve and modify the alignment of the text.

Constructing a Component and Adding the Component into a Container

Three steps are necessary to create and place a GUI component:

1. Declare the component with an *identifier* (*name*);

2. Construct the component by invoking an appropriate constructor via the `new` operator;

3. Identify the container (such as `Frame` or `Panel`) designed to hold this component. The container can then add this component onto itself via *aContainer*.add(*aComponent*) method. Every container has a `add(Component)` method. Take note that it is the container that actively and explicitly adds a component onto itself, instead of the other way.

Example
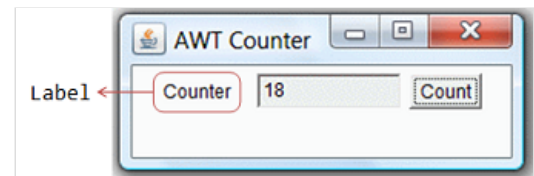
```
Label lblInput;                      // Declare an Label instance called lblInput
lblInput = new Label("Enter ID");    // Construct by invoking a constructor via the new operator
add(lblInput);                       // this.add(lblInput) - "this" is typically a subclass of Frame
lblInput.setText("Enter password"); // Modify the Label's text string
lblInput.getText();                  // Retrieve the Label's text string
```

An Anonymous Instance

You can create a `Label` without specifying an identifier, called *anonymous instance*. In the case, the Java compiler will assign an *anonymous identifier* for the allocated object. You will not be able to reference an anonymous instance in your program after it is created. This is usually alright for a `Label` instance as there is often no need to reference a `Label` after it is constructed.

Example

```
// Allocate an anonymous Label instance. "this" container adds the instance into itself.
// You CANNOT reference an anonymous instance to carry out further operations.
add(new Label("Enter Name: ", Label.RIGHT));

// Same as
Label lblXxx = new Label("Enter Name: ", Label.RIGHT)); // lblXxx assigned by compiler
add(lblXxx);
```

AWT GUI Component: `java.awt.Button`

A `java.awt.Button` is a GUI component that triggers a certain programmed *action* upon clicking.
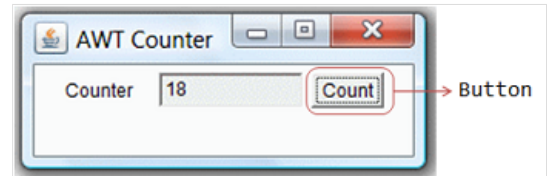
Constructors

```
public Button(String buttonLabel);
    // Construct a Button with the given label
public Button();
    // Construct a Button with empty label
```

The `Button` class has two constructors. The first constructor creates a `Button` object with the given label painted over the button. The second constructor creates a `Button` object with no label.

Public Methods

```
public String getLabel();
    // Get the label of this Button instance
public void setLabel(String buttonLabel);
    // Set the label of this Button instance
public void setEnable(boolean enable);
    // Enable or disable this Button. Disabled Button cannot be clicked.
```

The `getLabel()` and `setLabel()` methods can be used to read the current label and modify the label of a button, respectively.

Note: The latest Swing's `JButton` replaces `getLabel()`/`setLabel()` with `getText()`/`setText()` to be consistent with all the components. We will describe Swing later.

Event

Clicking a button fires a so-called `ActionEvent` and triggers a certain programmed action. I will explain event-handling later.

Example

```
Button btnColor = new Button("Red"); // Declare and allocate a Button instance called btnColor
add(btnColor);                       // "this" Container adds the Button
...
btnColor.setLabel("green");          // Change the button's label
btnColor.getLabel();                 // Read the button's label
...
add(Button("Blue"));      // Create an anonymous Button. It CANNOT be referenced later
```

AWT GUI Component: `java.awt.TextField`

A `java.awt.TextField` is single-line text box for users to enter texts. (There is a multiple-line text box called `TextArea`.) Hitting the "ENTER" key on a `TextField` object triggers an action-event.
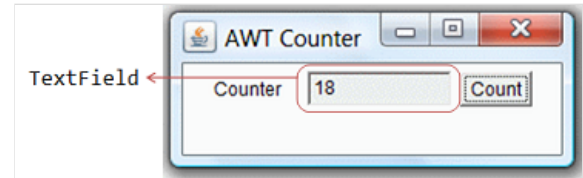
Constructors

```
public TextField(String strInitialText, int columns);
   // Construct a TextField instance with the given initial text string with the number of columns.
public TextField(String strInitialText);
   // Construct a TextField instance with the given initial text string.
public TextField(int columns);
   // Construct a TextField instance with the number of columns.
```

Public Methods

```
public String getText();
   // Get the current text on this TextField instance
public void setText(String strText);
   // Set the display text on this TextField instance
public void setEditable(boolean editable);
   // Set this TextField to editable (read/write) or non-editable (read-only)
```

Event

Hitting the "ENTER" key on a `TextField` fires a `ActionEvent`, and triggers a certain programmed action.

Example

```
TextField tfInput = new TextField(30); // Declare and allocate an TextField instance called tfInput
add(tfInput);                          // "this" Container adds the TextField
TextField tfResult = new TextField();  // Declare and allocate an TextField instance called tfResult
tfResult.setEditable(false) ;          // Set to read-only
add(tfResult);                         // "this" Container adds the TextField
......
// Read an int from TextField "tfInput", square it, and display on "tfResult".
// getText() returns a String, need to convert to int
int number = Integer.parseInt(tfInput.getText());
number *= number;
// setText() requires a String, need to convert the int number to String.
tfResult.setText(number + "");
```

Take note that getText()/SetText() operates on `String`. You can convert a `String` to a primitive, such as `int` or `double` via `static` method `Integer.parseInt()` or `Double.parseDouble()`. To convert a primitive to a `String`, simply concatenate the primitive with an empty `String`.

2.5 Example 1: `AWTCounter`

Let's assemble some components together into a simple GUI counter program, as illustrated. It has a top-level container Frame, which contains three components - a Label "Counter", a non-editable TextField to display the current count, and a "Count" Button. The TextField displays "0" initially.

Each time you click the button, the counter's value increases by 1.
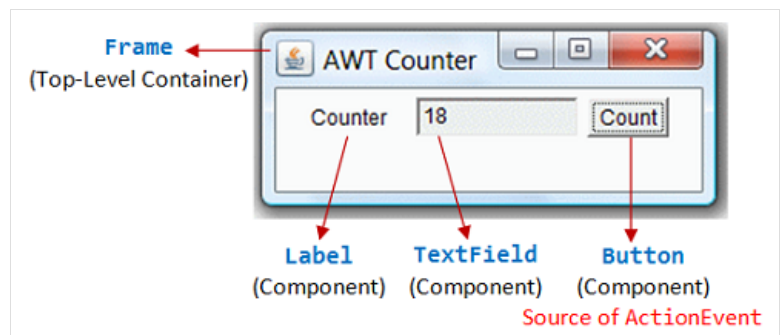

```
1   import java.awt.*;          // Using AWT container and component classes
2   import java.awt.event.*;  // Using AWT event classes and listener interfaces
3
4   // An AWT program inherits from the top-level container java.awt.Frame
5   public class AWTCounter extends Frame implements ActionListener {
6      private Label lblCount;     // Declare a Label component
7      private TextField tfCount; // Declare a TextField component
8      private Button btnCount;    // Declare a Button component
9      private int count = 0;      // Counter's value
```

```
10
11      // Constructor to setup GUI components and event handlers
12      public AWTCounter () {
13         setLayout(new FlowLayout());
14            // "super" Frame (a Container) sets its layout to FlowLayout, which arranges
15            // the components from left-to-right, and flow to next row from top-to-bottom.
16
17         lblCount = new Label("Counter");  // construct the Label component
18         add(lblCount);                    // "super" Frame adds Label
19
20         tfCount = new TextField("0", 10); // construct the TextField component
21         tfCount.setEditable(false);       // set to read-only
22         add(tfCount);                     // "super" Frame adds TextField
23
24         btnCount = new Button("Count");   // construct the Button component
25         add(btnCount);                    // "super" Frame adds Button
26
27         btnCount.addActionListener(this);
28            // btnCount is the source object that fires ActionEvent when clicked.
29            // The source add "this" instance as an ActionEvent listener, which provides
30            //  an ActionEvent handler called actionPerformed().
31            // Clicking btnCount invokes actionPerformed().
32
33         setTitle("AWT Counter");  // "super" Frame sets its title
34         setSize(250, 100);        // "super" Frame sets its initial window size
35
36         // For inspecting the components/container objects
37         // System.out.println(this);
38         // System.out.println(lblCount);
39         // System.out.println(tfCount);
40         // System.out.println(btnCount);
41
42         setVisible(true);         // "super" Frame shows
43
44         // System.out.println(this);
45         // System.out.println(lblCount);
46         // System.out.println(tfCount);
47         // System.out.println(btnCount);
48      }
49
50      // The entry main() method
51      public static void main(String[] args) {
52         // Invoke the constructor to setup the GUI, by allocating an instance
53         AWTCounter app = new AWTCounter();
54            // or simply "new AWTCounter();" for an anonymous instance
55      }
56
57      // ActionEvent handler - Called back upon button-click.
58      @Override
59      public void actionPerformed(ActionEvent evt) {
60         ++count; // increase the counter value
61         // Display the counter value on the TextField tfCount
62         tfCount.setText(count + ""); // convert int to String
63      }
64   }
```

To exit this program, you have to close the CMD-shell (or press "control-c" on the CMD console); or push the "red-square" close button in Eclipse's Application Console. This is because we have yet to write the handler for the Frame's close button. We shall do that in the later example.

Dissecting the `AWTCounter.java`

- The `import` statements (Lines 1-2) are needed, as AWT container and component classes, such as `Frame`, `Button`, `TextField`, and `Label`, are kept in the `java.awt` package; while AWT events and event-listener interfaces, such as `ActionEvent` and `ActionListener` are kept in the `java.awt.event` package.

- A GUI program needs a top-level container, and is often written as a subclass of `Frame` (Line 5). In other words, this class `AWTCounter` *is a* `Frame`, and inherits all the attributes and behaviors of a `Frame`, such as the title bar and content pane.

- Lines 12 to 46 define a constructor, which is used to setup and initialize the GUI components.

- In Line 13, the `setLayout()` (inherited from the superclass `Frame`) is used to set the layout of the container. `FlowLayout` is used which arranges the components in left-to-right and flows into next row in a top-to-bottom manner.

- A `Label`, `TextField` (non-editable), and `Button` are constructed. We invoke the `add()` method (inherited from the superclass `Frame`) to add these components into container.

- In Line 33-34, we invoke the `setSize()` and the `setTitle()` (inherited from the superclass `Frame`) to set the initial size and the title of the `Frame`. The `setVisible(true)` method (Line 42) is then invoked to show the display.

- The statement `btnCount.addActionListener(this)` (Line 27) is used to setup the event-handling mechanism, which will be discussed in length later. In brief, whenever the button is clicked, the `actionPerformed()` will be called. In the `actionPerformed()` (Lines 57-63), the counter value increases by 1 and displayed on the `TextField`.

- In the entry `main()` method (Lines 51-55), an instance of `AWTCounter` is constructed. The constructor is executed to initialize the GUI components and setup the event-handling mechanism. The GUI program then waits for the user input.

Inspecting Container/Components via `toString()`

It is interesting to inspect the GUI objects via the `toString()`, to gain an insight to these classes. (Alternatively, use a graphic debugger in Eclipse/NetBeans or study the JDK source code.) For example, if we insert the following code before and after the `setvisible()`:

```
System.out.println(this);
System.out.println(lblCount);
System.out.println(tfCount);
System.out.println(btnCount);

setVisible(true);  // "super" Frame shows

System.out.println(this);
System.out.println(lblCount);
System.out.println(tfCount);
System.out.println(btnCount);
```

The output (with my comments) are as follows. You could have an insight of the variables defined in the class.

```
// Before setVisible()
AWTCounter[frame0,0,0,250x100,invalid,hidden,layout=java.awt.FlowLayout,title=AWT Counter,resizable,normal]
      // name (assigned by compiler) is "frame0"; top-left (x,y) at (0,0); width/height is 250x100 (via setSize());
java.awt.Label[label0,0,0,0x0,invalid,align=left,text=Counter]
      // name is "Label0"; align is "Label.LEFT" (default); text is "Counter" (assigned in contructor)
java.awt.TextField[textfield0,0,0,0x0,invalid,text=0,selection=0-0]
      // name is "Textfield0"; text is "0" (assigned in contructor)
java.awt.Button[button0,0,0,0x0,invalid,label=Count]
      // name is "button0"; label text is "Count" (assigned in contructor)
      // Before setVisible(), all components are invalid (top-left (x,y), width/height are invalid)

// After setVisible(), all components are valid
AWTCounter[frame0,0,0,250x100,layout=java.awt.FlowLayout,title=AWT Counter,resizable,normal]
      // valid and visible (not hidden)
java.awt.Label[label0,20,41,58x23,align=left,text=Counter]
      // Top-left (x,y) at (20,41) relative to the parent Frame; width/height = 58x23
java.awt.TextField[textfield0,83,41,94x23,text=0,selection=0-0]
      // Top-left (x,y) at (83,41) relative to the parent Frame; width/height = 94x23; no text selected (0-0)
java.awt.Button[button0,182,41,47x23,label=Count]
      // Top-left (x,y) at (182,41) relative to the parent Frame; width/height = 47x23
```

2.6 Example 2: `AWTAccumulator`

In this example, the top-level container is again the typical `java.awt.Frame`, which contains 4 components: a `Label` "Enter an Integer", a `TextField` for accepting user input, another `Label` "The Accumulated Sum is", and another non-editable `TextField` for displaying the sum. The components are arranged in `FlowLayout`.

The program shall accumulate the number entered into the *input* `TextField` and display the sum in the *output* `TextField`.
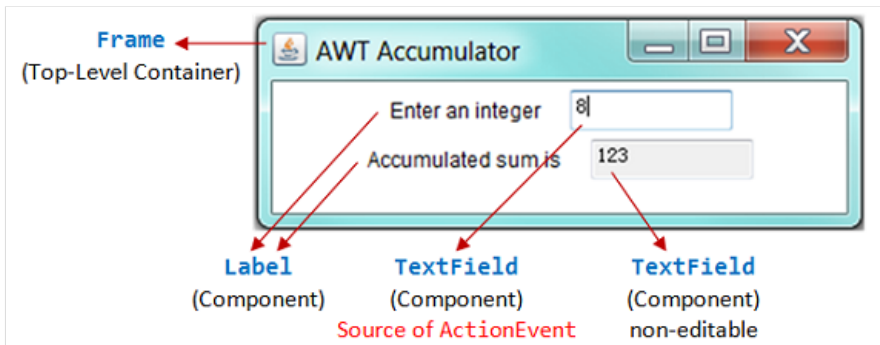
```
 1   import java.awt.*;          // Using AWT container and component classes
 2   import java.awt.event.*;    // Using AWT event classes and listener interfaces
 3
 4   // An AWT GUI program inherits from the top-level container java.awt.Frame
 5   public class AWTAccumulator extends Frame implements ActionListener {
 6      private Label lblInput;      // Declare input Label
 7      private Label lblOutput;     // Declare output Label
 8      private TextField tfInput;   // Declare input TextField
 9      private TextField tfOutput;  // Declare output TextField
10      private int sum = 0;         // Accumulated sum, init to 0
11
12      // Constructor to setup the GUI components and event handlers
13      public AWTAccumulator() {
14         setLayout(new FlowLayout());
15            // "super" Frame (a Container) sets layout to FlowLayout, which arranges
16            // the components from left-to-right, and flow to next row from top-to-bottom.
17
18         lblInput = new Label("Enter an Integer: "); // Construct Label
19         add(lblInput);                   // "super" Frame adds Label
20
21         tfInput = new TextField(10); // Construct TextField
22         add(tfInput);                    // "super" Frame adds TextField
23
24         tfInput.addActionListener(this);
25            // tfInput is the source object that fires ActionEvent when entered.
26            // The source add "this" instance as an ActionEvent listener, which provides
27            //   an ActionEvent handler called actionPerformed().
28            // Hitting enter on tfInput invokes actionPerformed().
29
30         lblOutput = new Label("The Accumulated Sum is: ");  // allocate Label
31         add(lblOutput);                  // "super" Frame adds Label
32
33         tfOutput = new TextField(10); // allocate TextField
34         tfOutput.setEditable(false);  // read-only
35         add(tfOutput);                   // "super" Frame adds TextField
36
37         setTitle("AWT Accumulator");  // "super" Frame sets title
38         setSize(350, 120);  // "super" Frame sets initial window size
39         setVisible(true);   // "super" Frame shows
40      }
41
42      // The entry main() method
43      public static void main(String[] args) {
44         // Invoke the constructor to setup the GUI, by allocating an anonymous instance
45         new AWTAccumulator();
46      }
47
48      // ActionEvent handler - Called back upon hitting enter key on TextField
49      @Override
50      public void actionPerformed(ActionEvent evt) {
51         // Get the String entered into the TextField tfInput, convert to int
52         int numberIn = Integer.parseInt(tfInput.getText());
53         sum += numberIn;       // Accumulate numbers entered into sum
54         tfInput.setText("");   // Clear input TextField
55         tfOutput.setText(sum + ""); // Display sum on the output TextField
56                                     // convert int to String
57      }
58   }
```

Dissecting the `AWTAccumulator.java`

1. An AWT GUI program extends from `java.awt.Frame` (Line 5) - the top-level window container.

2. In the constructor (Line 13), we constructs 4 components - 2 `java.awt.Label` and 2 `java.awt.TextFields`. The Frame adds the components, in `FlowLayout`.

3. tfInput (`TextField`) is the source object, which fires an `ActionEvent` upon hitting the Enter key. tfInput adds `this` instance as an `ActionEvent` handler (Line 24). The listener class (`this` or `AWTAccumulator`) needs to implement `ActionListener` interface and provides implementation to method `actionPerformed()`. Whenever an user hits Enter on the tfInput (`TextField`), the `actionPerformed()` will be invoked.

Inspecting Container/Components via `toString()`

Printing the `toString()` after `setVisible()` produces:

```
AWTAccumulator[frame0,0,0,350x120,layout=java.awt.FlowLayout,title=AWT Accumulator,resizable,normal]
java.awt.Label[label0,72,41,107x23,align=left,text=Enter an Integer: ]
java.awt.Label[label1,47,69,157x23,align=left,text=The Accumulated Sum is: ]
java.awt.TextField[textfield0,184,41,94x23,text=,editable,selection=0-0]
java.awt.TextField[textfield1,209,69,94x23,text=,selection=0-0]
```

3. AWT Event-Handling

Java adopts the so-called "Event-Driven" (or "Event-Delegation") programming model for event-handling, similar to most of the visual programming languages (such as Visual Basic and Delphi).

In event-driven programming, a piece of event-handling codes is executed (or called back by the graphics subsystem) when an event has been fired in response to an user input (such as clicking a mouse button or hitting the ENTER key). This is unlike the procedural model, where codes are executed in a sequential manner.

The AWT's event-handling classes are kept in package `java.awt.event`.

Three objects are involved in the event-handling: a *source*, *listener*(s) and an *event* object.

The *source* object (such as `Button` and `Textfield`) interacts with the user. Upon triggered, it creates an *event* object. This *event* object will be messaged to all the *registered listener* object(s), and an appropriate event-handler method of the listener(s) is called-back to provide the response. In other words, *triggering a source fires an event to all its listener(s), and invoke an appropriate handler of the listener(s)*.

To express interest for a certain source's event, the listener(s) must be registered with the source. In other words, the listener(s) "subscribes" to a source's event, and the source "publishes" the event to all its subscribers upon activation. This is known as *subscribe-publish* or *observable-observer* design pattern.
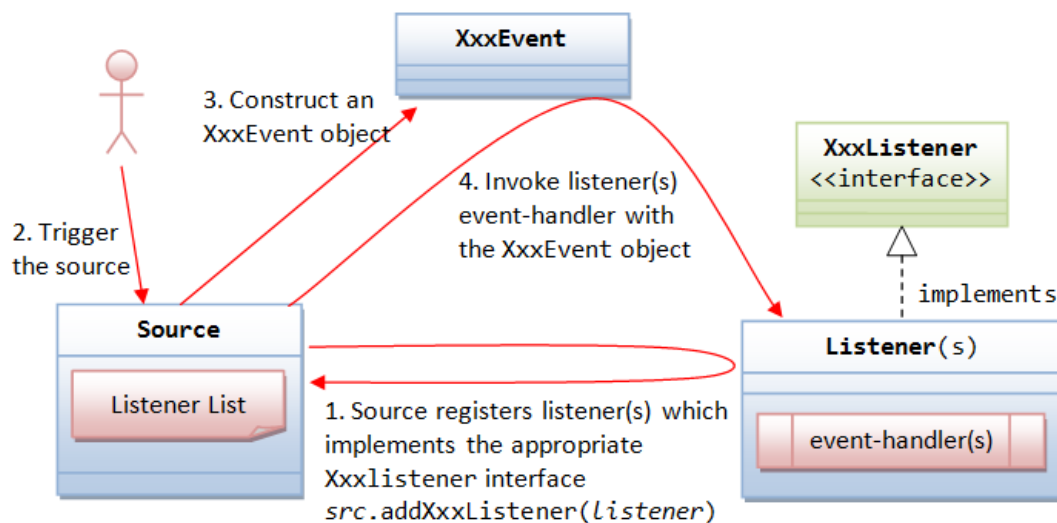
The sequence of steps is illustrated above:

1. *The source object registers its listener(s)* for a certain type of *event*.

    Source object fires event event upon triggered. For example, clicking an `Button` fires an `ActionEvent`, mouse-click fires `MouseEvent`, key-type fires `KeyEvent`, etc.

    How the source and listener understand each other? The answer is via an agreed-upon interface. For example, if a source is capable of firing an event called XxxEvent (e.g., MouseEvent) involving various operational modes (e.g., mouse-clicked, mouse-entered, mouse-exited, mouse-pressed, and mouse-released). Firstly, we need to declare an interface called XxxListener (e.g., MouseListener) containing the names of the handler methods. Recall that an `interface` contains only `abstract` methods without implementation. For example,

```
// A MouseListener interface, which declares the signature Jof the handlers
//   for the various operational modes.
public interface MouseListener {
    public void mousePressed(MouseEvent evt);  // Called back upon mouse-button pressed
    public void mouseReleased(MouseEvent evt); // Called back upon mouse-button released
```

```
      public void mouseClicked(MouseEvent evt);  // Called back upon mouse-button clicked (pressed and released)
      public void mouseEntered(MouseEvent evt);  // Called back when mouse pointer entered the component
      public void mouseExited(MouseEvent evt);   // Called back when mouse pointer exited the component
   }
```

Secondly, all the listeners interested in the XxxEvent must implement the XxxListener interface. That is, the listeners must provide their own implementations (i.e., programmed responses) to all the abstract methods declared in the XxxListener interface. In this way, the listenser(s) can response to these events appropriately. For example,

```
// An example of MouseListener,  which provides implementation to the handler methods
class MyMouseListener implement MouseListener {
   @Override
   public void mousePressed(MouseEvent e)  {
      System.out.println("Mouse-button pressed!");
   }

   @Override
   public void mouseReleased(MouseEvent e) {
      System.out.println("Mouse-button released!");
   }

   @Override
   public void mouseClicked(MouseEvent e)  {
      System.out.println("Mouse-button clicked (pressed and released)!");
   }

   @Override
   public void mouseEntered(MouseEvent e)  {
      System.out.println("Mouse-pointer entered the source component!");
   }

   @Override
   public void mouseExited(MouseEvent e)   {
      System.out.println("Mouse exited-pointer the source component!");
   }
}
```

Thirdly, in the source, we need to maintain a list of listener object(s), and define two methods: addXxxListener() and removeXxxListener() to add and remove a listener from this list. The signature of the methods are:

```
public void addXxxListener(XxxListener l);
public void removeXxxListener(XxxListener l);
```

Take note that all the listener(s) interested in the XxxEvent must implement the XxxListener interface. That is, they are sub-type of the XxxListener. Hence, they can be upcasted to XxxListener and passed as the argument of the above methods.

In summary, we identify the source, the event-listener interface, and the listener object. The listener must implement the event-listener interface. The source object then registers listener object via the addXxxListener() method:

```
aSource.addXxxListener(aListener);  // aSource registers aListener for XxxEvent
```

2. The source is triggered by a user.

3. The source create an XxxEvent object, which encapsulates the necessary information about the activation. For example, the (x, y) position of the mouse pointer, the text entered, etc.

4. Finally, for each of the listeners in the listener list, the source invokes the appropriate handler on the listener(s), which provides the programmed response.

In brief, *triggering a source fires an event to all its registered listeners, and invoke an appropriate handler of the listener*.

3.1 Revisit Example 1 AWTCounter: ActionEvent and ActionListener Interface

Clicking a Button (or hitting the "Enter" key on a TextField) fires an ActionEvent to all its ActionEvent listener(s). An ActionEvent listener must implement ActionListener interface, which declares one abstract method actionPerformed() as follow:

```
public interface ActionListener {
   public void actionPerformed(ActionEvent evt);
      // Called back upon button-click (on Button), enter-key pressed (on TextField)
}
```

Here are the event-handling steps:

- We identify btnCount (Button) as the *source* object.

- Clicking Button fires an ActionEvent to all its ActionEvent listener(s).

- The listener(s) is required to implement ActionListener interface, and override the actionPerformed() method to provide the response. For simplicity, we choose "this" object (AWTCounter) as the *listener* for the ActionEvent. Hence, "this" class is required to implement ActionListener interface and provide the programmed response in the actionPerformed().

```java
public class AWTCounter extends Frame implements ActionListener {
      // "this" is chosen as the ActionEvent listener, hence, it is required
      // to implement ActionListener interface
   ......

   // Implementing ActionListener interface requires this class to provide implementation
   // to the abstract method actionPerformed() declared in the interface.
   @Override
   public void actionPerformed(ActionEvent evt) {
      // Programmed response upon activation
      // Increment the count value and display on the TextField
      ++count;
      tfCount.setText(count + "");
   }
}
```

- The source object registers listener via the addActionListener(). In this example, the *source* btnCount (Button) adds "this" object as a *listener* via:

```java
btnCount.addActionListener(this);
```

Note that addActionListener() takes an argument of the type ActionListener. "this", which implements ActionListener interface (i.e., a subclass of ActionListener), is upcasted and passed to the addActionListener() method.

- Upon button-click, the btnCount creates an ActionEvent object, and calls back the actionPerformed(ActionEvent) method of all its registered listener(s) with the ActionEvent object created:

```java
ActionEvent evt = new ActionEvent( ...... );
listener.actionPerformed(evt);   // for all its listener(s)
```

The sequence diagram is as follows:
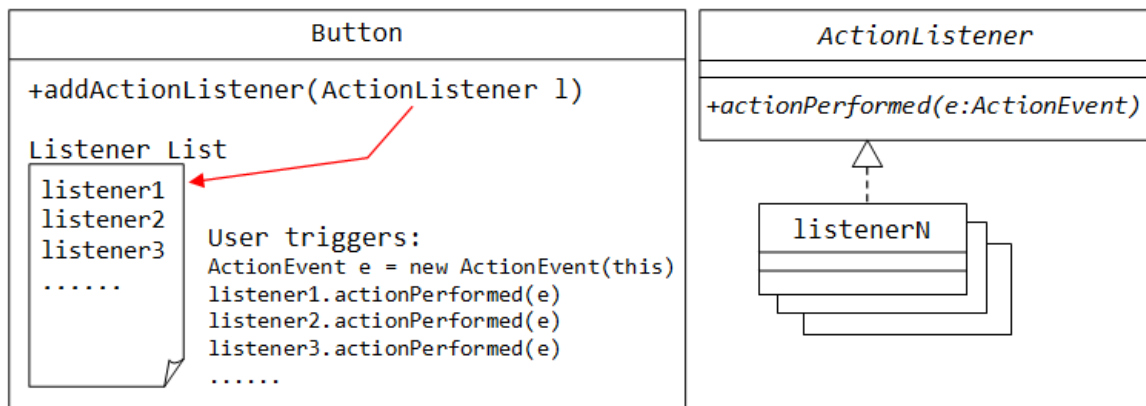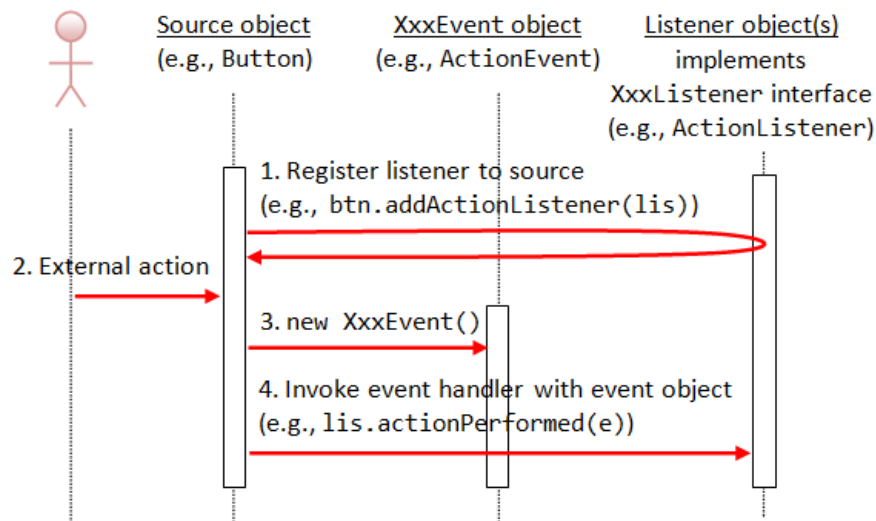
3.2 Revisit Example 2 `AWTAccumulator`: `ActionEvent` and `ActionListener` Interface

In this example,

1. We identify the `tfInput` (`TextField`) as the source object.

2. Hitting the "Enter" key on a `TextField` fires an `ActionEvent` to all its `ActionEvent` listener(s).

3. We choose `this` object as the `ActionEvent` listener (for simplicity).

4. The source object `tfInput` (`TextField`) registers the listener (`this` object) via the `tfInput.addActionListener(this)`.

5. The `ActionEvent` listener (`this` class) is required to implement the `ActionListener` interface, and override the `actionPerformed()` method to provide the programmed response upon activation.

3.3 Example 3: `WindowEvent` and `WindowListener` Interface

A `WindowEvent` is fired (to all its `WindowEvent` listeners) when a window (e.g., `Frame`) has been opened/closed, activated/deactivated, iconified/deiconified via the 3 buttons at the top-right corner or other means. The source of `WindowEvent` shall be a top-level window-container such as `Frame`.
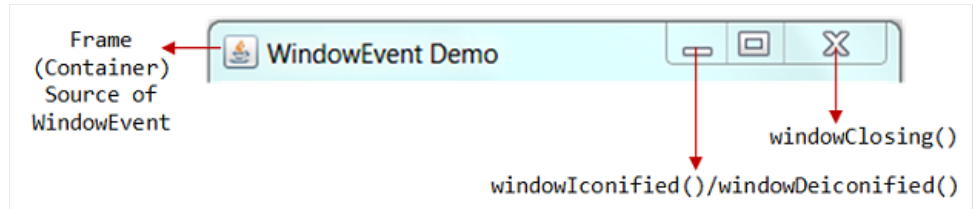
A `WindowEvent` listener must implement `WindowListener` interface, which declares 7 `abstract` event-handling methods, as follows. Among them, the `windowClosing()`, which is called back upon clicking the window-close button, is the most commonly-used.

```
public void windowClosing(WindowEvent evt)
   // Called-back when the user attempts to close the window by clicking the window close button.
   // This is the most-frequently used handler.
public void windowOpened(WindowEvent evt)
   // Called-back the first time a window is made visible.
public void windowClosed(WindowEvent evt)
   // Called-back when a window has been closed as the result of calling dispose on the window.
public void windowActivated(WindowEvent evt)
   // Called-back when the Window is set to be the active Window.
public void windowDeactivated(WindowEvent evt)
   // Called-back when a Window is no longer the active Window.
public void windowIconified(WindowEvent evt)
   // Called-back when a window is changed from a normal to a minimized state.
public void windowDeiconified(WindowEvent evt)
   // Called-back when a window is changed from a minimized to a normal state.
```

The following program added support for "close-window button" to Example 1: `AWTCounter`.

```
1   import java.awt.*;         // Using AWT containers and components
2   import java.awt.event.*;   // Using AWT events classes and listener interfaces
```

```
 3
 4    // An AWT GUI program inherits the top-level container java.awt.Frame
 5    public class WindowEventDemo extends Frame
 6          implements ActionListener, WindowListener {
 7          // This class acts as listener for ActionEvent and WindowEvent
 8          // A Java class can extend one superclass, but can implement multiple interfaces.
 9
10       private TextField tfCount;  // Declare a TextField component
11       private Button btnCount;    // Declare a Button component
12       private int count = 0;      // Counter's value
13
14       // Constructor to setup the GUI components and event handlers
15       public WindowEventDemo() {
16          setLayout(new FlowLayout()); // "super" Frame sets to FlowLayout
17
18          add(new Label("Counter"));   // "super" Frame adds an anonymous Label
19
20          tfCount = new TextField("0", 10); // Construct the TextField
21          tfCount.setEditable(false);       // read-only
22          add(tfCount);                     // "super" Frame adds TextField
23
24          btnCount = new Button("Count");   // Construct the Button
25          add(btnCount);                    // "super" Frame adds Button
26
27          btnCount.addActionListener(this);
28            // btnCount (source object) fires ActionEvent upon clicking
29            // btnCount adds "this" object as an ActionEvent listener
30
31          addWindowListener(this);
32            // "super" Frame (source object) fires WindowEvent.
33            // "super" Frame adds "this" object as a WindowEvent listener.
34
35          setTitle("WindowEvent Demo"); // "super" Frame sets title
36          setSize(250, 100);            // "super" Frame sets initial size
37          setVisible(true);             // "super" Frame shows
38       }
39
40       // The entry main() method
41       public static void main(String[] args) {
42          new WindowEventDemo();  // Let the construct do the job
43       }
44
45       /* ActionEvent handler */
46       @Override
47       public void actionPerformed(ActionEvent evt) {
48          ++count;
49          tfCount.setText(count + "");
50       }
51
52       /* WindowEvent handlers */
53       // Called back upon clicking close-window button
54       @Override
55       public void windowClosing(WindowEvent evt) {
56          System.exit(0);  // Terminate the program
57       }
58
59       // Not Used, but need to provide an empty body to compile.
60       @Override public void windowOpened(WindowEvent evt) { }
61       @Override public void windowClosed(WindowEvent evt) { }
62       @Override public void windowIconified(WindowEvent evt) { }
63       @Override public void windowDeiconified(WindowEvent evt) { }
64       @Override public void windowActivated(WindowEvent evt) { }
65       @Override public void windowDeactivated(WindowEvent evt) { }
66    }
```

In this example, we shall modify the earlier `AWTCounter` example to handle the `WindowEvent`. Recall that pushing the "close-window" button on the `AWTCounter` has no effect, as it did not handle the `WindowEvent` of `windowClosing()`. We included the `WindowEvent` handling codes in this example.

1. We identify super `Frame` as the source object.

2. The `Frame` fires the `WindowEvent` to all its registered `WindowEvent` listener(s).

3. We select `this` object as the `WindowEvent` listener (for simplicity)

4. We register `this` object as the `WindowEvent` listener to the source `Frame` via method `addWindowListener(this)`.

5. The `WindowEvent` listener (this class) is required to implement the `WindowListener` interface, which declares 7 abstract methods: `windowOpened()`, `windowClosed()`, `windowClosing()`, `windowActivated()`, `windowDeactivated()`, `windowIconified()` and `windowDeiconified()`.

6. We override the `windowClosing()` handler to terminate the program using `System.exit(0)`. We ignore the other 6 handlers, but required to provide an empty body.

The sequence diagram is as follow:
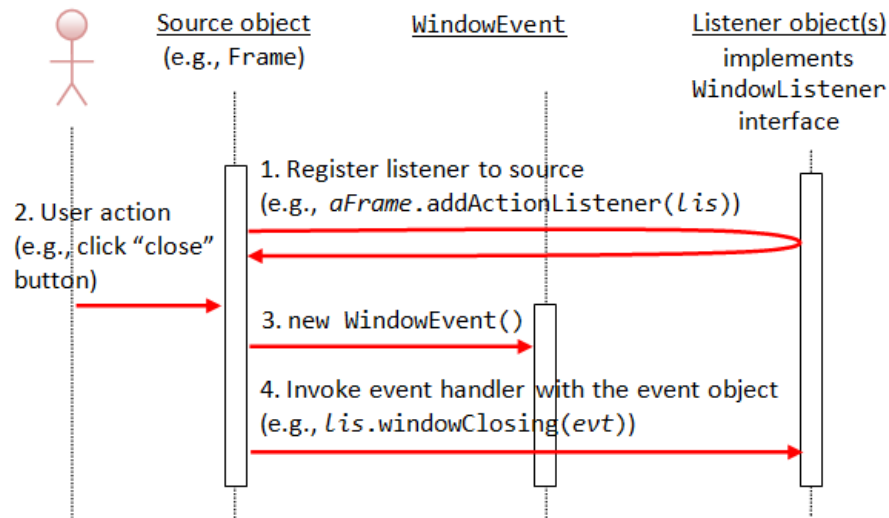
3.4 Example 4: `MouseEvent` and `MouseListener` Interface

A `MouseEvent` is fired to all its registered listeners, when you press, release, or click (press followed by release) a mouse-button (left or right button) at the source object; or position the mouse-pointer at (enter) and away (exit) from the source object.

A `MouseEvent` listener must implement the `MouseListener` interface, which declares the following five `abstract` methods:

```
public void mouseClicked(MouseEvent evt)
   // Called-back when the mouse-button has been clicked (pressed followed by released) on the source.
public void mousePressed(MouseEvent evt)
public void mouseReleased(MouseEvent evt)
   // Called-back when a mouse-button has been pressed/released on the source.
   // A mouse-click invokes mousePressed(), mouseReleased() and mouseClicked().
public void mouseEntered(MouseEvent evt)
public void mouseExited(MouseEvent evt)
   // Called-back when the mouse-pointer has entered/exited the source.
```
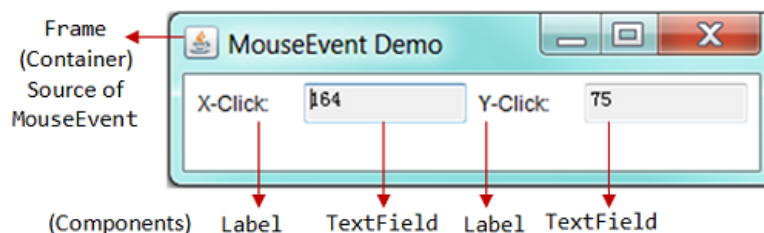


```
1    import java.awt.*;
2    import java.awt.event.*;
3
4    public class MouseEventDemo extends Frame implements MouseListener {
5        private TextField tfMouseX; // to display mouse-click-x
6        private TextField tfMouseY; // to display mouse-click-y
7
8        // Constructor - Setup the UI components and event handlers
9        public MouseEventDemo() {
```

```
10            setLayout(new FlowLayout()); // "super" frame sets its layout to FlowLayout
11
12            // Label (anonymous)
13            add(new Label("X-Click: ")); // "super" frame adds Label component
14
15            // TextField
16            tfMouseX = new TextField(10); // 10 columns
17            tfMouseX.setEditable(false);  // read-only
18            add(tfMouseX);                  // "super" frame adds TextField component
19
20            // Label (anonymous)
21            add(new Label("Y-Click: ")); // "super" frame adds Label component
22
23            // TextField
24            tfMouseY = new TextField(10);
25            tfMouseY.setEditable(false);  // read-only
26            add(tfMouseY);                  // "super" frame adds TextField component
27
28            addMouseListener(this);
29                // "super" frame (source) fires the MouseEvent.
30                // "super" frame adds "this" object as a MouseEvent listener.
31
32            setTitle("MouseEvent Demo"); // "super" Frame sets title
33            setSize(350, 100);           // "super" Frame sets initial size
34            setVisible(true);            // "super" Frame shows
35        }
36
37        public static void main(String[] args) {
38            new MouseEventDemo();  // Let the constructor do the job
39        }
40
41        /* MouseEvent handlers */
42        // Called back upon mouse clicked
43        @Override
44        public void mouseClicked(MouseEvent evt) {
45            tfMouseX.setText(evt.getX() + "");
46            tfMouseY.setText(evt.getY() + "");
47        }
48
49        // Not used - need to provide an empty body to compile.
50        @Override public void mousePressed(MouseEvent evt) { }
51        @Override public void mouseReleased(MouseEvent evt) { }
52        @Override public void mouseEntered(MouseEvent evt) { }
53        @Override public void mouseExited(MouseEvent evt) { }
54    }
```

In this example, we setup a GUI with 4 components (two Labels and two non-editable TextFields), inside a top-level container Frame, arranged in FlowLayout.

To demonstrate the MouseEvent:

1. We identity super Frame as the source object.

2. The Frame fires a MouseEvent to all its MouseEvent listener(s) when you click/press/release a mouse-button or enter/exit with the mouse-pointer.

3. We select this object as the MouseEvent listener (for simplicity).

4. We register this object as the MouseEvent listener to super Frame (source) via the method addMouseListener(this).

5. The listener (this class) is required to implement the MouseListener interface, which declares 5 abstract methods: mouseClicked(), mousePressed(), mouseReleased(), mouseEntered(), and mouseExit(). We override the mouseClicked() to display the (x, y) co-ordinates of the mouse click on the two displayed TextFields. We ignore all the other handlers (for simplicity - but you need to provide an empty body for compilation).

Try: Include a WindowListener to handle the close-window button.

## 3.5 Example 5: MouseEvent and MouseMotionListener Interface

A MouseEvent is also fired when you moved and dragged the mouse pointer at the source object. But you need to use MouseMotionListener to handle the mouse-move and mouse-drag. The MouseMotionListener interface declares the following two abstract methods:

```
public void mouseDragged(MouseEvent e)
   // Called-back when a mouse-button is pressed on the source component and then dragged.
public void mouseMoved(MouseEvent e)
   // Called-back when the mouse-pointer has been moved onto the source component but no buttons have been pushed.
```
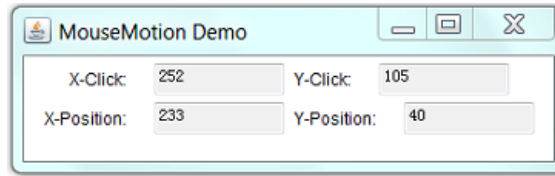


```
1    import java.awt.*;
2    import java.awt.event.*;
3
4    // An AWT GUI program inherits from the top-level container java.awt.Frame
5    public class MouseMotionDemo extends Frame
6          implements MouseListener, MouseMotionListener {
7          // This class acts as MouseListener and MouseMotionListener
8
9       // To display the (x, y) of the mouse-clicked
10      private TextField tfMouseClickX;
11      private TextField tfMouseClickY;
12      // To display the (x, y) of the current mouse-pointer position
13      private TextField tfMousePositionX;
14      private TextField tfMousePositionY;
15
16      // Constructor to setup the GUI components and event handlers
17      public MouseMotionDemo() {
18         setLayout(new FlowLayout()); // "super" frame sets to FlowLayout
19
20         add(new Label("X-Click: "));
21         tfMouseClickX = new TextField(10);
22         tfMouseClickX.setEditable(false);
23         add(tfMouseClickX);
24         add(new Label("Y-Click: "));
25         tfMouseClickY = new TextField(10);
26         tfMouseClickY.setEditable(false);
27         add(tfMouseClickY);
28
29         add(new Label("X-Position: "));
30         tfMousePositionX = new TextField(10);
31         tfMousePositionX.setEditable(false);
32         add(tfMousePositionX);
33         add(new Label("Y-Position: "));
34         tfMousePositionY = new TextField(10);
35         tfMousePositionY.setEditable(false);
36         add(tfMousePositionY);
37
38         addMouseListener(this);
39         addMouseMotionListener(this);
40            // "super" frame (source) fires MouseEvent.
41            // "super" frame adds "this" object as MouseListener and MouseMotionListener.
42
43         setTitle("MouseMotion Demo"); // "super" Frame sets title
44         setSize(400, 120);            // "super" Frame sets initial size
45         setVisible(true);             // "super" Frame shows
46      }
47
48      // The entry main() method
49      public static void main(String[] args) {
50         new MouseMotionDemo();  // Let the constructor do the job
51      }
52
53      /** MouseListener handlers */
54      // Called back when a mouse-button has been clicked
55      @Override
56      public void mouseClicked(MouseEvent evt) {
57         tfMouseClickX.setText(evt.getX() + "");
58         tfMouseClickY.setText(evt.getY() + "");
```

```
59        }
60
61        // Not Used, but need to provide an empty body for compilation
62        @Override public void mousePressed(MouseEvent evt) { }
63        @Override public void mouseReleased(MouseEvent evt) { }
64        @Override public void mouseEntered(MouseEvent evt) { }
65        @Override public void mouseExited(MouseEvent evt) { }
66
67        /** MouseMotionEvent handlers */
68        // Called back when the mouse-pointer has been moved
69        @Override
70        public void mouseMoved(MouseEvent evt) {
71           tfMousePositionX.setText(evt.getX() + "");
72           tfMousePositionY.setText(evt.getY() + "");
73        }
74
75        // Not Used, but need to provide an empty body for compilation
76        @Override public void mouseDragged(MouseEvent evt) { }
77    }
```

In this example, we shall illustrate both the MouseListener and MouseMotionListener.

1. We identify super Frame as the source, which fires the MouseEvent to its registered MouseListener and MouseMotionListener.

2. We select this object as the MouseListener and MouseMotionListner (for simplicity).

3. We register this object as the listener to super Frame via method addMouseListener(this) and addMouseMotionListener(this).

4. The MouseMotionListener (this class) needs to implement 2 abstract methods: mouseMoved() and mouseDragged() declared in the MouseMotionListener interface.

5. We override the mouseMoved() to display the (x, y) position of the mouse pointer. We ignore the MouseDragged() handler by providing an empty body for compilation.

Try: Include a WindowListener to handle the close-window button.

3.6 Example 6: KeyEvent and KeyListener Interface

A KeyEvent is fired (to all its registered KeyListeners) when you pressed, released, and typed (pressed followed by released) a key on the source object. A KeyEvent listener must implement KeyListener interface, which declares three abstract methods:

```
public void keyTyped(KeyEvent e)
   // Called-back when a key has been typed (pressed and released).
public void keyPressed(KeyEvent e)
public void keyReleased(KeyEvent e)
   // Called-back when a key has been pressed/released.
```
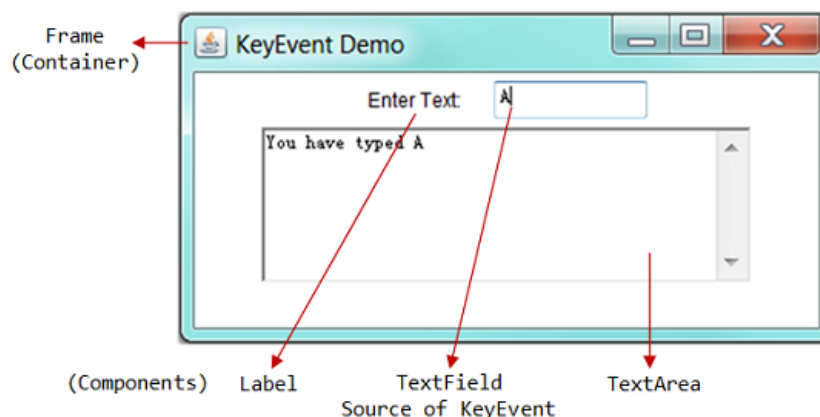


```
1   import java.awt.*;
2   import java.awt.event.*;
3
4   // An AWT GUI program inherits from the top-level container java.awt.Frame
5   public class KeyEventDemo extends Frame implements KeyListener {
6        // This class acts as KeyEvent Listener
7
8        private TextField tfInput;  // Single-line TextField to receive tfInput key
9
```

```
10        private TextArea taDisplay; // Multi-line TextArea to taDisplay result
11
12        // Constructor to setup the GUI components and event handlers
13        public KeyEventDemo() {
14            setLayout(new FlowLayout()); // "super" frame sets to FlowLayout
15
16            add(new Label("Enter Text: "));
17            tfInput = new TextField(10);
18            add(tfInput);
19            taDisplay = new TextArea(5, 40); // 5 rows, 40 columns
20            add(taDisplay);
21
22            tfInput.addKeyListener(this);
23                // tfInput TextField (source) fires KeyEvent.
24                // tfInput adds "this" object as a KeyEvent listener.
25
26            setTitle("KeyEvent Demo"); // "super" Frame sets title
27            setSize(400, 200);         // "super" Frame sets initial size
28            setVisible(true);          // "super" Frame shows
29        }
30
31        // The entry main() method
32        public static void main(String[] args) {
33            new KeyEventDemo();  // Let the constructor do the job
34        }
35
36        /** KeyEvent handlers */
37        // Called back when a key has been typed (pressed and released)
38        @Override
39        public void keyTyped(KeyEvent evt) {
40            taDisplay.append("You have typed " + evt.getKeyChar() + "\n");
41        }
42
43        // Not Used, but need to provide an empty body for compilation
44        @Override public void keyPressed(KeyEvent evt) { }
45        @Override public void keyReleased(KeyEvent evt) { }
    }
```

In this example:

1. We identify the `tfInput` (`TextField`) as the source object.

2. The source fires a `KeyEvent` when you press/release/type a key to all its `KeyEvent` listener(s).

3. We select `this` object as the `KeyEvent` listener.

4. We register `this` object as the `KeyEvent` listener to the source `TextField` via method `input.addKeyListener(this)`.

5. The `KeyEvent` listener (`this` class) needs to implement the `KeyListener` interface, which declares 3 abstract methods: `keyTyped()`, `keyPressed()`, `keyReleased()`.

6. We override the `keyTyped()` to display key typed on the display `TextArea`. We ignore the `keyPressed()` and `keyReleased()`.

4. (Advanced) Observer Design Pattern

Source (Observable)

Listeners (Observers)
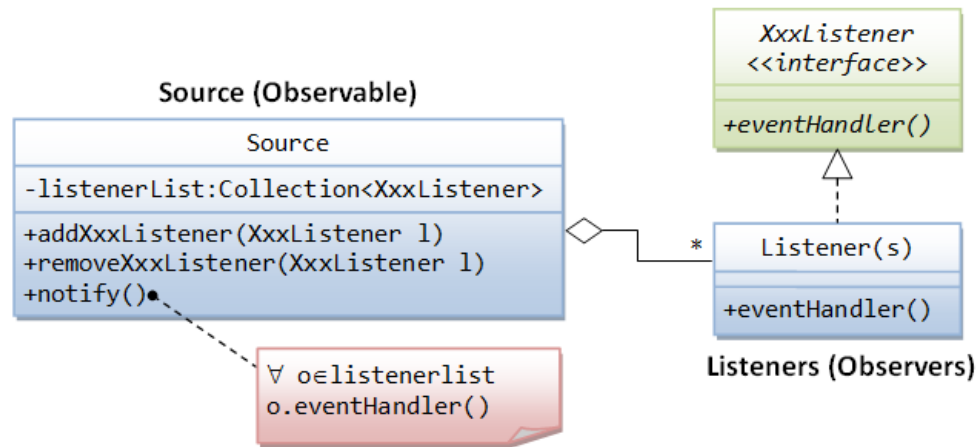
The *Observer* design pattern (aka *Publish-Subscribe* or *Observable-Observer*) is one of the 23 GoF's design patterns. Whenever the source's state changes, it notifies all its registered listener.

The source and listener are *bound* via the interface XxxListener, which defines a set of handlers. The source maintain a list of registered listeners, and two methods: addXxxListener() and removeXxxListener(). Both addXxxListener() and removeXxxListener() takes an argument of XxxListener. Hence, a listener object must implement XxxListener in order to be registered. Whenever the source's state changes, it invokes a particular handler of all the registered listeners. The interface guarantees the existence of such handler in the listener.

4.1 Example: Creating Your Own Event, Source and Listener

Suppose that we have a source called Light, with two states - on and off. This source is capable of notifying its registered listener(s), whenever its state changes.

- First, we define the LightEvent class (extends from java.util.EventObject).

- Next, we define a LightListener interface to *bind* the source and its listeners. This interface specifies the signature of the handlers, lightOn(LightEvent) and lightOff(LightEvent).

- In the source Light, we use an ArrayList to maintain its listeners, and create two methods: addLightListner(LightListener) and removeLightListener(LightListener). An method called notifyListener() is written to invoke the appropriate handlers of each of its registered listeners, whenever the state of the Light changes.

- A listener class called LightWatcher is written, which implements the LightListener interface and provides implementation for the handlers.

Event: `LightEvent.java`

```
1  /** LightEvent */
2  import java.util.EventObject;
3  public class LightEvent extends EventObject {
4     public LightEvent (Object src) {
5        super(src);
6     }
7  }
```

Listener Interface: `LightListener.java`

```
1  /** The LightListener interface define the expected behevior of LightEvent listeners */
2  import java.util.EventListener;
3  public interface LightListener extends EventListener {
4     public void lightOn(LightEvent evt);  // Called-back upon light on
5     public void lightOff(LightEvent evt); // Called-back upon light off
6  }
```

Source: `Light.java`

```
1  /** The Light Source, which maintains a list of listeners and fires LightEvent to its listeners */
2  import java.util.*;
3  public class Light {
4     // Status - on (true) or off (false)
5     private boolean on;
6     // Listener list
      private List<LightListener> listeners = new ArrayList<LightListener>();
```

```
 7
 8        /** Constructor */
 9        public Light() {
10           on = false;    // initially off
11           System.out.println("Light: constructed and off");
12        }
13
14        /** Add the given LightListener */
15        public void addLightListener(LightListener listener) {
16           listeners.add(listener);
17           System.out.println("Light: added a listener");
18        }
19
20        /** Remove the given LightListener */
21        public void removeLightListener(LightListener listener) {
22           listeners.remove(listener);
23           System.out.println("Light: removed a listener");
24        }
25
26        /** Turn on this light */
27        public void turnOn() {
28           if (!on) {
29              on = !on;
30              System.out.println("Light: turn on");
31              notifyListener();
32           }
33        }
34
35        /** Turn off this light */
36        public void turnOff() {
37           if (on) {
38              on = !on;
39              System.out.println("Light: turn off");
40              notifyListener();
41           }
42        }
43
44        /** Construct an LightEvent and trigger the appropriate handler on its registered listeners */
45        private void notifyListener() {
46           LightEvent evt = new LightEvent(this);
47           for (LightListener listener : listeners) {
48              if (on) {
49                 listener.lightOn(evt);
50              } else {
51                 listener.lightOff(evt);
52              }
53           }
54        }
55     }
56
```

A Listener: `LightWatcher.java`

```
 1    /** A LightEvent listener, which is required to implement LightListener interface. */
 2    public class LightWatcher implements LightListener {
 3       private int id;  // ID of this listener
 4
 5       /** Constructor */
 6       public LightWatcher(int id) {
 7          this.id = id;
 8          System.out.println("LightWatcher-" + id + ": created");
 9       }
10
11       /** LightEvent handlers - Called back with LightEvent is fired */
12       @Override
13       public void lightOn(LightEvent evt) {
14          System.out.println("LightWatcher-" + id
15             + ": I am notified that light is on");
16       }
17
```

```
18        @Override
19        public void lightOff(LightEvent evt) {
20           System.out.println("LightWatcher-" + id
21              + ": I am notified that light is off");
22        }
23    }
```

A Test Driver: `TestLight.java`

```
1    /** A Test Driver */
2    public class TestLight {
3        public static void main(String[] args) {
4            Light light = new Light();
5            LightWatcher lw1 = new LightWatcher(1);
6            LightWatcher lw2 = new LightWatcher(2);
7            LightWatcher lw3 = new LightWatcher(3);
8            light.addLightListener(lw1);
9            light.addLightListener(lw2);
10           light.turnOn();
11           light.addLightListener(lw3);
12           light.turnOff();
13           light.removeLightListener(lw1);
14           light.removeLightListener(lw3);
15           light.turnOn();
16       }
17   }
```

The expected outputs are:

```
Light: constructed and off
LightWatcher-1: created
LightWatcher-2: created
LightWatcher-3: created
Light: added a listener
Light: added a listener
Light: turn on
LightWatcher-1: I am notified that light is on
LightWatcher-2: I am notified that light is on
Light: added a listener
Light: turn off
LightWatcher-1: I am notified that light is off
LightWatcher-2: I am notified that light is off
LightWatcher-3: I am notified that light is off
Light: removed a listener
Light: removed a listener
Light: turn on
LightWatcher-2: I am notified that light is on
```

5. Nested (Inner) Classes

A *nested class* (or commonly called *inner class*) is *a class defined inside another class* - introduced in JDK 1.1. As an illustration, two nested classes `MyNestedClass1` and `MyNestedClass2` are defined *inside* the definition of an outer class called `MyOuterClass`.

```
public class MyOuterClass {    // outer class defined here
   ......
   private class MyNestedClass1 { ...... }  // an nested class defined inside the outer class
   public static class MyNestedClass2 { ...... }  // an "static" nested class defined inside the outer class
   ......
}
```

A nested class has these properties:

1. A nested class is a proper class. That is, it could contain constructors, member variables and member methods. You can create an instance of a nested class via the new operator and constructor.

2. A nested class is a *member* of the outer class, just like any member variables and methods defined inside a class.

3. Most importantly, a nested class can access the `private` members (variables/methods) of the enclosing outer class, as it is at the *same level* as these `private` members. This is the property that makes inner class useful.

4. A nested class can have `private`, `public`, `protected`, or the *default* access, just like any member variables and methods defined inside a class. A `private` inner class is only accessible by the enclosing outer class, and is not accessible by any other classes. [An top-level outer class cannot be declared `private`, as no one can use a `private` outer class.]

5. A nested class can also be declared `static`, `final` or `abstract`, just like any ordinary class.

6. A nested class is NOT a *subclass* of the outer class. That is, the nested class does not inherit the variables and methods of the outer class. It is an *ordinary* self-contained class. [Nonetheless, you could declare it as a subclass of the outer class, via keyword "`extends OuterClassName`", in the nested class's definition.]

The usages of nested class are:

1. To control visibilities (of the member variables and methods) between inner/outer class. The nested class, being defined inside an outer class, can access `private` members of the outer class.

2. To place a piece of class definition codes *closer* to where it is going to be used, to make the program clearer and easier to understand.

3. For namespace management.

5.1 Example 7: A Named Inner Class as Event Listener

A nested class is useful if you need a *small* class which relies on the enclosing outer class for its private variables and methods. It is ideal in an event-driven environment for implementing event handlers. This is because the event handling methods (in a listener) often require access to the private variables (e.g., a private `TextField`) of the outer class.

In this example (modified from Example 1 `AWTCounter`), instead of using "`this`" as the `ActionEvent` listener for the `Button`, we define a new class called `BtnCountListener`, and create an instance of `BtnCountListener` as the `ActionEvent` listener for the `btnCount`. The `BtnCountListener` needs to implement the `ActionListener` interface, and override the `actionPerformed()` handler. Since "`this`" is no long a `ActionListener`, we remove the "`implements ActionListener`" from "`this`" class's definition.

`BtnCountListener` needs to be defined as an inner class, as it needs to access private variables (`count` and `tfCount`) of the outer class.

```
1   import java.awt.*;
2   import java.awt.event.*;
3
4   // An AWT GUI program inherits from the top-level container java.awt.Frame
5   public class AWTCounterNamedInnerClass extends Frame {
6      // This class is NOT a ActionListener, hence, it does not implement ActionListener interface
7
8      // The event-handler actionPerformed() needs to access these "private" variables
9      private TextField tfCount;
10     private Button btnCount;
11     private int count = 0;
12
13     // Constructor to setup the GUI components and event handlers
14     public AWTCounterNamedInnerClass () {
15        setLayout(new FlowLayout());  // "super" Frame sets to FlowLayout
16        add(new Label("Counter"));    // An anonymous instance of Label
17        tfCount = new TextField("0", 10);
18        tfCount.setEditable(false);   // read-only
19        add(tfCount);                 // "super" Frame adds tfCount
20
21        btnCount = new Button("Count");
22        add(btnCount);                // "super" Frame adds btnCount
23
24        // Construct an anonymous instance of BtnCountListener (a named inner class).
25        // btnCount adds this instance as a ActionListener.
26        btnCount.addActionListener(new BtnCountListener());
27
28        setTitle("AWT Counter");
29        setSize(250, 100);
30        setVisible(true);
31     }
32
33     // The entry main method
34     public static void main(String[] args) {
35        new AWTCounterNamedInnerClass(); // Let the constructor do the job
36     }
37
38     /**
39      * BtnCountListener is a "named inner class" used as ActionListener.
```

```
40          * This inner class can access private variables of the outer class.
41          */
42         private class BtnCountListener implements ActionListener {
43            @Override
44            public void actionPerformed(ActionEvent evt) {
45               ++count;
46               tfCount.setText(count + "");
47            }
48         }
49      }
```

Dissecting the Program

- An inner class named `BtnCountListener` is used as the `ActionListner`.

- An anonymous instance of the `BtnCountListener` inner class is constructed. The `btnCount` source object adds this instance as a listener, as follows:

  ```
  btnCount.addActionListener(new BtnCountListener());
  ```

- The inner class can access the `private` variable `tfCount` and count of the outer class.

- Since "this" is no longer a listener, we remove the "`implements ActionListener`" from `this` class' definition.

- The inner class is compiled into `AWTCount$BtnCountListener.class`, in the format of *OuterClassName$InnerClassName*`.class`.

Using an Ordinary (Outer) Class as Listener

Try moving the `BtnCountListener` class outside, and define it as an ordinary class. You would need to pass a reference of the `AWTConnter` into the constructor of `BtnCountListener`, and use this reference to access variables `tfCount` and count, through `public` getters or granting them to `public` access.

```
// An ordinary outer class used as ActionListener for the Button
public class BtnCountListener implements ActionListener {
   AWTCounter frame;
   public BtnCountListener(AWTCounter frame) {
      this.frame = frame;
   }

   @Override
   public void actionPerformed(ActionEvent evt) {
      frame.count++;
      frame.tfCount.setText(frame.count + "");
   }
}
```

This code is messy!

5.2 Example 8: An Anonymous Inner Class as Event Listener

Instead of using a *named inner class* (called `BtnCountListner` in the previous example), we shall use an inner class without a name, known as *anonymous inner class* as the `ActionListener` in this example.

```
 1    import java.awt.*;
 2    import java.awt.event.*;
 3
 4    // An AWT GUI program inherits from the top-level container java.awt.Frame
 5    public class AWTCounterAnonymousInnerClass extends Frame {
 6       // This class is NOT a ActionListener, hence, it does not implement ActionListener interface
 7
 8       // The event-handler actionPerformed() needs to access these private variables
 9       private TextField tfCount;
10       private Button btnCount;
11       private int count = 0;
12
13       // Constructor to setup the GUI components and event handlers
14       public AWTCounterAnonymousInnerClass () {
15          setLayout(new FlowLayout());   // "super" Frame sets to FlowLayout
16          add(new Label("Counter"));     // An anonymous instance of Label
17          tfCount = new TextField("0", 10);
18          tfCount.setEditable(false);    // read-only
19          add(tfCount);                  // "super" Frame adds tfCount
```

```
20
21          btnCount = new Button("Count");
22          add(btnCount);                  // "super" Frame adds btnCount
23
24          // Construct an anonymous instance of an anonymous class.
25          // btnCount adds this instance as a ActionListener.
26          btnCount.addActionListener(new ActionListener() {
27             @Override
28             public void actionPerformed(ActionEvent evt) {
29                ++count;
30                tfCount.setText(count + "");
31             }
32          });
33
34          setTitle("AWT Counter");
35          setSize(250, 100);
36          setVisible(true);
37       }
38
39       // The entry main method
40       public static void main(String[] args) {
41          new AWTCounterAnonymousInnerClass(); // Let the constructor do the job
42       }
43    }
```

Dissecting the Program

- Again, "this" class is NOT used as the ActionEvent listener. Hence, we remove the "implements ActionListener" from this class' definition.

- The anonymous inner class is given a name generated by the compiler, and compiled into *OuterClassName*$*n*.class, where *n* is a running number of the inner classes of this outer class.

- An anonymous instance of an anonymous inner class is constructed, and passed as the argument of the addActionListener() method as follows:

```
btnCount.addActionListener(new ActionListener() {
   @Override
   public void actionPerformed(ActionEvent evt) {
      ++count;
      tfCount.setText(count + "");
   }
});
```

The above codes is equivalent to and compiled as:

```
private class N implements ActionListener {  // N is a running number of the inner classes created
   @Override
   public void actionPerformed(ActionEvent evt) {
      ++count;
      tfCount.setText(count + "");
   }
}
btnCount.addActionListener(new N());

// Or
N n = new N()
btnCount.addActionListener(n);
```

Properties of Anonymous Inner Class

1. The anonymous inner class is define inside a method, instead of a member of the outer class (class member). It is *local* to the method and cannot be marked with access modifier (such as public, private) or static, just like any local variable of a method.

2. An anonymous inner class must always extend a superclass or implement an interface. The keyword "extends" or "implements" is NOT required in its declaration. An anonymous inner class must implement all the abstract methods in the superclass or in the interface.

3. An anonymous inner class always uses the default (no-arg) constructor from its superclass to create an instance. If an anonymous inner class implements an interface, it uses the java.lang.Object().

4. An anonymous inner class is compiled into a class named *OuterClassName*$*n*.class, where *n* is a running number of inner classes within the outer class.

5. An instance of an anonymous inner class is constructed via this syntax:

```
new SuperClassName/InterfaceName() {  // extends superclass or implements interface
                                      // invoke the default no-arg constructor or Object[]
   // Implement abstract methods in superclass/interface
   // More methods if necessary
   ......
}
```

The created instance can be assigned to a variable or used as an argument of a method.

5.3 Example 9: An Anonymous Inner Class for Each Source

Let's modify our AWTCounter example to include 3 buttons for counting up, counting down, and reset the count, respectively. We shall attach an anonymous inner class as the listener to each of buttons.
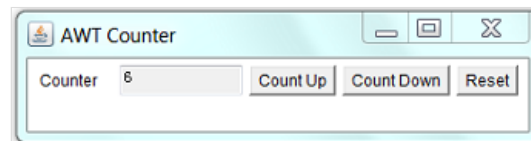

```
1   import java.awt.*;
2   import java.awt.event.*;
3
4   // An AWT GUI program inherits the top-level container java.awt.Frame
5   public class AWTCounter3Buttons extends Frame {
6      private TextField tfCount;
7      private Button btnCountUp, btnCountDown, btnReset;
8      private int count = 0;
9
10     // Constructor to setup the GUI components and event handlers
11     public AWTCounter3Buttons () {
12        setLayout(new FlowLayout());
13        add(new Label("Counter"));   // an anonymous instance of Label
14        tfCount = new TextField("0", 10);
15        tfCount.setEditable(false);  // read-only
16        add(tfCount);                // "super" Frame adds tfCount
17
18        btnCountUp = new Button("Count Up");
19        add(btnCountUp);
20        // Construct an anonymous instance of an anonymous inner class.
21        // The source Button adds the anonymous instance as ActionEvent listener
22        btnCountUp.addActionListener(new ActionListener() {
23           @Override
24           public void actionPerformed(ActionEvent evt) {
25              ++count;
26              tfCount.setText(count + "");
27           }
28        });
29
30        btnCountDown = new Button("Count Down");
31        add(btnCountDown);
32        btnCountDown.addActionListener(new ActionListener() {
33           @Override
34           public void actionPerformed(ActionEvent evt) {
35              count--;
36              tfCount.setText(count + "");
37           }
38        });
39
40        btnReset = new Button("Reset");
41        add(btnReset);
42        btnReset.addActionListener(new ActionListener() {
43           @Override
44           public void actionPerformed(ActionEvent evt) {
45              count = 0;
46              tfCount.setText("0");
47           }
48        });
49
50        setTitle("AWT Counter");
51        setSize(400, 100);
```

```
52          setVisible(true);
53       }
54
55       // The entry main method
56       public static void main(String[] args) {
57          new AWTCounter3Buttons();  // Let the constructor do the job
58       }
59    }
```

Dissecting the Program

    1. Each of the `Buttons` uses an anonymous instance of an anonymous inner class as its `ActionEvent` listener.

5.4 Example 10: Using the Same Listener Instance for All the Buttons

If you use the same instance as the listener for the 3 buttons, you need to determine which button has fired the event. It is because all the 3 buttons trigger the same event-handler method.

Using `ActionEvent's getActionCommand()`

In the following example, we use the same instance of a named inner class as the listener for all the 3 buttons. The listener needs to determine which button has fired the event. This can be accomplished via the `ActionEvent's getActionCommonad()` method, which returns the button's label.

```
 1    import java.awt.*;
 2    import java.awt.event.*;
 3
 4    // An AWT GUI program inherits the top-level container java.awt.Frame
 5    public class AWTCounter3Buttons1Listener extends Frame {
 6       private TextField tfCount;
 7       private Button btnCountUp, btnCountDown, btnReset;
 8       private int count = 0;
 9
10       // Constructor to setup the GUI components and event handlers
11       public AWTCounter3Buttons1Listener () {
12          setLayout(new FlowLayout());
13          add(new Label("Counter"));
14          tfCount = new TextField("0", 10);
15          tfCount.setEditable(false);
16          add(tfCount);
17
18          // Construct Buttons
19          btnCountUp = new Button("Count Up");
20          add(btnCountUp);
21          btnCountDown = new Button("Count Down");
22          add(btnCountDown);
23          btnReset = new Button("Reset");
24          add(btnReset);
25
26          // Allocate an instance of named inner class BtnListener.
27          BtnListener listener = new BtnListener();
28          // Use the same listener instance to all the 3 Buttons.
29          btnCountUp.addActionListener(listener);
30          btnCountDown.addActionListener(listener);
31          btnReset.addActionListener(listener);
32
33          setTitle("AWT Counter");
34          setSize(400, 100);
35          setVisible(true);
36       }
37
38       // The entry main method
39       public static void main(String[] args) {
40          new AWTCounter3Buttons1Listener();  // Let the constructor do the job
41       }
42
43       /**
44        * BtnListener is a named inner class used as ActionEvent listener for all the Buttons.
45        */
```

```
46      private class BtnListener implements ActionListener {
47         @Override
48         public void actionPerformed(ActionEvent evt) {
49            // Need to determine which button fired the event.
50            // the getActionCommand() returns the Button's label
51            String btnLabel = evt.getActionCommand();
52            if (btnLabel.equals("Count Up")) {
53               ++count;
54            } else if (btnLabel.equals("Count Down")) {
55               --count;
56            } else {
57               count = 0;
58            }
59            tfCount.setText(count + "");
60         }
61      }
62   }
```

Using `getSource()` of `EventObject`

Besides the getActionCommand(), which is only available for `ActionEvent`, you can use the `getSource()` method, which is available to all event objects, to retrieve a reference to the source object that has fired the event. getSource() returns a java.lang.Object. You may need to downcast it to the proper type of the source object. For example,

```
1    import java.awt.*;
2    import java.awt.event.*;
3
4    public class AWTCounter3ButtonsGetSource extends Frame {
5       private TextField tfCount;
6       private Button btnCountUp, btnCountDown, btnReset;
7       private int count = 0;
8
9       // Constructor to setup the GUI components and event handlers
10      public AWTCounter3ButtonsGetSource () {
11         setLayout(new FlowLayout());
12         add(new Label("Counter"));
13         tfCount = new TextField("0", 10);
14         tfCount.setEditable(false);
15         add(tfCount);
16
17         // Construct Buttons
18         btnCountUp = new Button("Count Up");
19         add(btnCountUp);
20         btnCountDown = new Button("Count Down");
21         add(btnCountDown);
22         btnReset = new Button("Reset");
23         add(btnReset);
24
25         // Allocate an instance of inner class BtnListener.
26         BtnListener listener = new BtnListener();
27         // Use the same listener instance to all the 3 Buttons.
28         btnCountUp.addActionListener(listener);
29         btnCountDown.addActionListener(listener);
30         btnReset.addActionListener(listener);
31
32         setTitle("AWT Counter");
33         setSize(400, 100);
34         setVisible(true);
35      }
36
37      // The entry main method
38      public static void main(String[] args) {
39         new AWTCounter3ButtonsGetSource();  // Let the constructor do the job
40      }
41
42      /**
43       * BtnListener is a named inner class used as ActionEvent listener for all the Buttons.
44       */
45      private class BtnListener implements ActionListener {
46         @Override
```

```
47        public void actionPerformed(ActionEvent evt) {
48            // Need to determine which button has fired the event.
49            Button source = (Button)evt.getSource();
50                // Get a reference of the source that has fired the event.
51                // getSource() returns a java.lang.Object. Downcast back to Button.
52            if (source == btnCountUp) {
53                ++count;
54            } else if (source == btnCountDown) {
55                --count;
56            } else {
57                count = 0;
58            }
59            tfCount.setText(count + "");
60        }
61     }
62  }
```

6. (Advanced) More on Nested Classes

There are 4 types of nested classes:

1. `static` nested class (as a outer class member),

2. non-`static` (instance) inner class (as a outer class member),

3. local inner class (defined inside a method),

4. anonymous local inner class (defined inside a method).

6.1 Static vs. Instance Nested Classes

A nested class can be declared `static` (belonging to the class instead of an instance). Recall that a `static` member can be used without instantiating the class and can be referenced via the classname in the form of *Classname.memberName* (e.g., `Math.PI`, `Integer.parseInt()`). Similarly, a `static` nested class can be used without instantiating the outer class and can be referenced via *OuterClassName.InnerClassName*.

On the other hand, a non-`static` nested class belongs to an instance of the outer class, just like any instance variable or method. It can be referenced via *outerClassInstanceName.innerClassInstanceName*. A non-`static` nested class is formally called an *inner class*.

Example of non-`static` (instance) inner class

In this example, a non-`static` (instance) inner class called `MyInnerClass` is defined inside the outer class. The inner class can access `private` members (variables/methods) of the outer class. This outer class also declares and constructs an instance of inner class as its member variable.

```
 1  public class MyOuterClassWithInnerClass {
 2     // Private member variable of the outer class
 3     private String msgOuter = "Hello from outer class";
 4
 5     // Define an inner class as a member of the outer class
 6     // This is merely an definition.
 7     // Not instantiation takes place when an instance of outer class is constructed
 8     public class MyInnerClass {
 9        // Private variable of the inner class
10        private String msgInner;
11        // Constructor of the inner class
12        public MyInnerClass(String msgInner) {
13           this.msgInner = msgInner;
14           System.out.println("Constructing an inner class instance: " + msgOuter);
15                // can access private member variable of outer class
16        }
17        // A method of inner class
18        public void printMessage() {
19           System.out.println(msgInner);
20        }
21     }
22
23     // Declare and construct an instance of the inner class, inside the outer class
24     MyInnerClass anInner = new MyInnerClass("Hi from inner class");
25  }
```

Two class files are produced: `MyOuterClassWithInnerClass.class` and `MyOuterClassWithInnerClass$MyInnerClass.class`.

The following test program:

1. Allocates an instance of outer class, which implicitly allocates an inner class (called `anInner`) as its member variable. You can access this inner class via *outerClassInstanceName.innerClassInstanceName*.

2. Explicitly constructs another instance of the inner class, under the same outer class instance created in the previous step.

3. Explicitly constructs one more instance of the inner class, under a new instance of outer class. This new outer class instance also implicitly allocates an inner class instance as its member, as seen from the output.

```
 1  public class TestInnerClass {
 2      public static void main(String[] args) {
 3          // Construct an instance of outer class, which create anInner
 4          MyOuterClassWithInnerClass anOuter = new MyOuterClassWithInnerClass();
 5          // Invoke inner class's method from this outer class instance
 6          anOuter.anInner.printMessage();
 7
 8          // Explicitly construct another instance of inner class
 9          MyOuterClassWithInnerClass.MyInnerClass inner2
10              = anOuter.new MyInnerClass("Inner class 2");
11          inner2.printMessage();
12
13          // Explicitly construct an instance of inner class, under another instance of outer class
14          MyOuterClassWithInnerClass.MyInnerClass inner3
15              = new MyOuterClassWithInnerClass().new MyInnerClass("Inner class 3");
16          inner3.printMessage();
17      }
18  }
```

```
Constructing an inner class instance: Hello from outer class
Hi from inner class
Constructing an inner class instance: Hello from outer class
Inner class 2
Constructing an inner class instance: Hello from outer class
Constructing an inner class instance: Hello from outer class
Inner class 3
```

An inner class definition is merely a definition of a class. The outer class does not create an inner class instance, when it is instantiated. Nonetheless, you could declare it as member of the outer class, as illustrated in the above example. In many situations, we declare the inner class `private`. In this cases, the inner class can only be used (declare and construct) within the outer class.

You can set the inner class to `private` access. In this case, the inner class can only be accessed within the outer class, and not by other classes.

Example of `static` nested class

In this example, a `static` nested class is defined inside the outer class, which can access the `private` `static` variables of the outer class.

```
 1  public class MyOuterClassWithStaticNestedClass {
 2      // Private "static" member variable of the outer class
 3      private static String msgOuter = "Hello from outer class";
 4
 5      // Define a "static" nested class as a member of the outer class
 6      // It can access private "static" variable of the outer class
 7      public static class MyStaticNestedClass {
 8          // Private variable of inner class
 9          private String msgInner;
10          // Constructor of inner class
11          public MyStaticNestedClass(String msgInner) {
12              this.msgInner = msgInner;
13              System.out.println(msgOuter); // access private member of the outer class
14          }
15          // A method of inner class
16          public void printMessage() {
17              System.out.println(msgInner);
18          }
19      }
20  }
```

You can access the `static` nested class via the outer classname, in the form of *OuterClassName.NestedClassName*, just like any static variables/methods (e.g., `Math.PI`, `Integer.parseInt()`). You can instantiate a `static` nested class without instantiate the outer class, as `static` members are associated with the class, instead of instances.

```
1    public class TestStaticNestedClass {
2       public static void main(String[] args) {
3          // Construct an instance of static nested class
4          // A "static" nested class, like other "static" members, can be accessed via
5          // the Classname.membername
6          MyOuterClassWithStaticNestedClass.MyStaticNestedClass aNestedInner =
7                new MyOuterClassWithStaticNestedClass.MyStaticNestedClass("Hi from inner class");
8          aNestedInner.printMessage();
9       }
10   }
```

```
Hello from outer class
Hi from inner class
```

As seen from the example, a `static` nested class is really like a top-level class with a modified name (*OuterClassname.InnerClassname*). It can be used as an extension to package for *namespace management*.

6.2 Local Inner Class Defined Inside a Method

Java allows you to define an inner class inside a method, just like defining a method's local variable. Like local variable, a local inner class does not exist until the method is invoked, and goes out of scope when the method exits.

A local inner class has these properties:

1. A local inner class cannot have access modifier (such as `private` or `public`). It also cannot be declared `static`.

2. A local inner class can access all the variables/methods of the enclosing outer class.

3. A local inner class can have access to the local variables of the enclosing method only if they are declared `final` (to prevent undesirable side-effects).

Example

```
1    public class MyOuterClassWithLocalInnerClass {
2       // Private member variable of the outer class
3       private String msgOuter = "Hello from outer class";
4
5       // A member method of the outer class
6       public void doSomething() {
7
8          // A local variable of the method
9          final String msgMethod = "Hello from method";
10
11         // Define a local inner class inside the method
12         class MyInnerClass {
13            // Private variable of the inner class
14            private String msgInner;
15            // Constructor of the inner class
16            public MyInnerClass(String msgInner) {
17               this.msgInner = msgInner;
18               System.out.println("Constructing an inner class instance: " + msgOuter);
19                  // can access private member variable of outer class
20               System.out.println("Accessing final variable of the method: " + msgMethod);
21                  // can access final variable of the method
22            }
23            // A method of inner class
24            public void printMessage() {
25               System.out.println(msgInner);
26            }
27         }
28
29         // Create an instance of inner class and invoke its method
30         MyInnerClass anInner = new MyInnerClass("Hi, from inner class");
31         anInner.printMessage();
32      }
33
34      // Test main() method
35      public static void main(String[] args) {
36          // Create an instance of the outer class and invoke the method.
37          new MyOuterClassWithLocalInnerClass().doSomething();
```

```
38      }
39    }
```

```
Constructing an inner class instance: Hello from outer class
Accessing final variable of the method: Hello from method
Hi, from inner class
```

6.3 An Anonymous Inner Class

An anonymous inner class is a local inner class (of a method) without assigning an explicit classname. It must either "extends" an existing superclass or "implements" an interface. It is declared and instantiated in one statement via the new keyword.

Example

```
1    public class MyOuterClassWithAnonymousInnerClass {
2       // Private member variable of the outer class
3       private String msgOuter = "Hello from outer class";
4
5       // A member method of the outer class
6       public void doSomething() {
7          // A local variable of the method
8          final String msgMethod = "Hello from method";
9
10         Thread thread = new Thread() { // create an instance of an anonymous inner class that extends Thread class
11            @Override
12            public void run() {
13               System.out.println("Constructing an inner class instance: " + msgOuter);
14                  // can access private member variable of outer class
15               System.out.println("Accessing final variable of the method: " + msgMethod);
16                  // can access final variable of the method
17               System.out.println("Hi, from inner class!");
18            }
19         };
20         thread.start();
21      }
22
23      // Test main() method
24      public static void main(String[] args) {
25         // Create an instance of the outer class and invoke the method.
26         new MyOuterClassWithAnonymousInnerClass().doSomething();
27      }
28   }
```

```
Constructing an inner class instance: Hello from outer class
Accessing final variable of the method: Hello from method
Hi, from inner class
```

The anonymous inner class definition is equivalent to:

```
public void doSomething()
   ......
   class OuterClassName.n extends Thread {  // where n is a running number of anonymous inner classes
      ......
   }
   Thread thread = new OuterClassName.n();  // create an instance of the anonymous inner class
   ......
}
```

Clearly, you can only create one instance for each anonymous inner class.

6.4 Example of Static Nested Class in JDK: Point2D, Point2D.Double, Point2D.Float, Point

The abstract class Point2D (in package java.awt.geom of Java 2D API), which models a 2D point, declares abstract methods such as getX() and getY(). The Point2D cannot be instantiated. Point2D does not define any instance variable, in particular, the x and y location of the point. This is because it is not sure about the *type* of x and y (which could be int, float, or double). The instance variables, therefore, are left to the implementation subclasses.

Three subclasses were implemented for types of int, float and double, respectively. Point2D cannot be designed as a pure abstract-method-only interface, as it contains non-abstract methods.

The subclass `Point` defines instance variables x and y in `int` precision and provides implementation to abstract methods such as `getX()` and `getY()`. `Point` (of int-precision) is a straight-forward implementation of inheritance and polymorphism. `Point` is a legacy class (since JDK 1.1) and retrofitted when Java 2D was introduced.

Two subclasses `Point2D.Float` and `Point2D.Double` define instance variables x and y in `float` and `double` precision, respectively. These two subclasses, are also declared as `public static` nested class of the outer class `Point2D`. Since they are `static`, they can be referenced as `Point2D.Double` and `Point2D.Float`. They are implemented as nested `static` subclasses within the `Point2D` outer class to keep the codes together and for namespace management. There is no access-control (of private variables of the outer class) involved.

```java
package java.awt.geom;
abstract public class Point2D  {

   // abstract methods
   abstract public double getX();
   abstract public double getY();
   abstract public void setLocation(double x, double y);

   public double distance(double x, double y) { ... }
   public double distance(Point2D p) { ... }
   public static double distance(double x1, double y1, double x2, double y2) { ... }
   ......

   public static class Double extends Point2D {
      public double x;
      public double y;

      public Double(double x, double y) { ... }
      @Override public double getX() { return x; }
      @Override public double getY() { return y; }
      @Override public void setLocation(double x, double y) { ... }
      ......
   }

   public static class Float extends Point2D {
      public float x;
      public float y;
      public Double(float x, float y) { ... }
      @Override public double getX() { ... }
      @Override public double getY() { ... }
      @Override public void setLocation(double x, double y) { ... }
      public void setLocation(float x, float y) { ... }
      ......
   }
}
```

```java
package java.awt.geom;
public class Point extends Point2D {
   public int x;
   public int y;
   public Point(int x, int y) { ... }
   @Override public double getX() { return x; }
   @Override public double getY() { return y; }
   @Override public void setLocation(double x, double y) { ... }
   ......
}
```

`Point2D.Double` and `Point2D.Float` are public `static` classes. In other words, they can be used directly without instantiating the outer class, just like any `static` variable or method (which can be referenced directly via the classname, e.g., `Math.PI`, `Math.sqrt()` and `Integer.parseInt()`). Since they are subclass of `Point2D`, they can be upcast to `Point2D`.

```java
Point2D.Double p1 = new Point2D.Double(1.1, 2.2);
Point2D.Float p2 = new Point2D.Float(1.1f, 2.2f);
Point p3 = new Point(1, 2);
// Using polymorphism
Point2D p1 = new Point2D.Double(1.1, 2.2);   // upcast
Point2D p2 = new Point2D.Float(1.1f, 2.2f); // upcast
Point2D p3 = new Point(1, 2);               // upcast
```

Note: These classes were designed before the introduction of generic in JDK 1.5, which supports the passing of type as argument.

6.5 "Cannot refer to a non-final variable inside an inner class defined in a different method"

Java specification 8.1.3: "Any local variable, formal method parameter or exception handler parameter used but not declared in an inner class must be declared final."

By allowing inner class to access non-`final` local variables inside a method, the local variable could be modified by the inner class, and causes a strange side-effect.

Solution:

1. Declare the variable `final` if permissible.

2. Declare the variable outside the method, e.g., as *member variables* of the class, instead of a local variable within a method. Both the method and the inner class could access the variable.

3. Use a wrapper class to wrap the variable inside a class. Declare the instance `final`.

6.6 Referencing Outer-class's "this" from Inner-class

Inside the inner class, "this" refers to the inner class. To refer to the "this" of the outer class, use "*OuterClassName*.this". But you can reference outer class's members directly without this clumsy syntax. For example,

```
......
public class MyOuterClassName {
   private String msg = "Hello";
   ......
   public MyOuterClassName() {  // constructor
      ......
      Button btn = new Button("TEST");
      btn.addActionListener(new ActionListener() {
         @Override
         public void actionPerformed(ActionEvent e) {
            // Need OuterClassName.this to refer to the outer class.
            // But can reference outer class members (e.g., msg) directly.
            JOptionPane.showMessageDialog(MyOuterClassName.this, msg);
         }
      });
   }
}
```

7. Event Listener's Adapter Classes

7.1 Example 11: `WindowAdapter` for `WindowListener`

Using `WindowListener` Interface

Refer to the `WindowEventDemo`, a `WindowEvent` listener is required to implement the `WindowListener` interface, which declares 7 `abstract` methods. Although we are only interested in `windowClosing()`, we need to provide an empty body to the other 6 `abstract` methods in order to compile the program. This is tedious, e.g., we can rewrite the `WindowEventDemo` using an inner class implementing `ActionListener` as follows:

```
1   import java.awt.*;
2   import java.awt.event.*;
3
4   // An AWT GUI program inherits the top-level container java.awt.Frame
5   public class WindowEventDemoWithInnerClass extends Frame {
6      private TextField tfCount;
7      private Button btnCount;
8      private int count = 0;
9
10     // Constructor to setup the GUI components and event handlers
11     public WindowEventDemoWithInnerClass () {
12        setLayout(new FlowLayout());
13        add(new Label("Counter"));
14        tfCount = new TextField("0", 10);
15        tfCount.setEditable(false);
16        add(tfCount);
17
18        btnCount = new Button("Count");
19        add(btnCount);
20        btnCount.addActionListener(new ActionListener() {
```

```
21            @Override
22            public void actionPerformed(ActionEvent evt) {
23               ++count;
24               tfCount.setText(count + "");
25            }
26         });

27
28         // Allocate an anonymous instance of an anonymous inner class
29         // that implements WindowListener.
30         // "super" Frame adds this instance as WindowEvent listener.
31         addWindowListener(new WindowListener() {
32            @Override
33            public void windowClosing(WindowEvent evt) {
34               System.exit(0);  // terminate the program
35            }
36            // Need to provide an empty body for compilation
37            @Override public void windowOpened(WindowEvent evt) { }
38            @Override public void windowClosed(WindowEvent evt) { }
39            @Override public void windowIconified(WindowEvent evt) { }
40            @Override public void windowDeiconified(WindowEvent evt) { }
41            @Override public void windowActivated(WindowEvent evt) { }
42            @Override public void windowDeactivated(WindowEvent evt) { }
43         });
44
45         setTitle("WindowEvent Demo");
46         setSize(250, 100);
47         setVisible(true);
48      }
49
50      // The entry main method
51      public static void main(String[] args) {
52         new WindowEventDemoWithInnerClass();   // Let the constructor do the job
53      }
54   }
```

Using `WindowAdapter` Superclass

An *adapter* class called `WindowAdapter` is therefore provided, which implements the `WindowListener` interface and provides default implementations to all the 7 abstract methods. You can then derive a subclass from `WindowAdapter` and override only methods of interest and leave the rest to their default implementation. For example,

```
1   import java.awt.*;
2   import java.awt.event.*;
3
4   // An AWT GUI program inherits the top-level container java.awt.Frame
5   public class WindowEventDemoAdapter extends Frame {
6      private TextField tfCount;
7      private Button btnCount;
8      private int count = 0;
9
10      // Constructor to setup the GUI components and event handlers
11      public WindowEventDemoAdapter () {
12         setLayout(new FlowLayout());
13         add(new Label("Counter"));
14         tfCount = new TextField("0", 10);
15         tfCount.setEditable(false);
16         add(tfCount);
17
18         btnCount = new Button("Count");
19         add(btnCount);
20         btnCount.addActionListener(new ActionListener() {
21            @Override
22            public void actionPerformed(ActionEvent evt) {
23               ++count;
24               tfCount.setText(count + "");
25            }
26         });
27
28         // Allocate an anonymous instance of an anonymous inner class
29         // that extends WindowAdapter.
```

```
30          // "super" Frame adds the instance as WindowEvent listener.
31          addWindowListener(new WindowAdapter() {
32             @Override
33             public void windowClosing(WindowEvent evt) {
34                System.exit(0);  // Terminate the program
35             }
36          });
37
38          setTitle("WindowEvent Demo");
39          setSize(250, 100);
40          setVisible(true);
41       }
42
43       /** The entry main method */
44       public static void main(String[] args) {
45          new WindowEventDemoAdapter();   // Let the constructor do the job
46       }
47    }
```

7.2 Other Event-Listener Adapter Classes

Similarly, adapter classes such as MouseAdapter, MouseMotionAdapter, KeyAdapter, FocusAdapter are available for MouseListener, MouseMotionListener, KeyListener, and FocusListener, respectively.

There is no ActionAdapter for ActionListener, because there is only one abstract method (i.e. actionPerformed()) declared in the ActionListener interface. This method has to be overridden and there is no need for an adapter.

8. Layout Managers and Panel

A container has a so-called *layout manager* to arrange its components. The layout managers provide a level of abstraction to map your user interface on all windowing systems, so that the layout can be *platform-independent*.

AWT provides the following layout managers (in package java.awt): FlowLayout, GridLayout, BorderLayout, GridBagLayout, BoxLayout, CardLayout, and others. (Swing added more layout manager in package javax.swing, to be described later.)

`Container's setLayout()`

A container has a setLayout() method to set its layout manager:

```
// java.awt.Container
public void setLayout(LayoutManager mgr)
```

To set up the layout of a Container (such as Frame, JFrame, Panel, or JPanel), you have to:

1. Construct an instance of the chosen layout object, via new and constructor, e.g., new FlowLayout())
2. Invoke the setLayout() method of the Container, with the layout object created as the argument;
3. Place the GUI components into the Container using the add() method in the correct order; or into the correct zones.

For example,

```
// Allocate a Panel (container)
Panel p = new Panel();
// Allocate a new Layout object. The Panel container sets to this layout.
p.setLayout(new FlowLayout());
// The Panel container adds components in the proper order.
p.add(new JLabel("One"));
p.add(new JLabel("Two"));
p.add(new JLabel("Three"));
......
```

`Container's getLayout()`

You can get the current layout via Container's getLayout().

```
Panel awtPanel = new Panel();
System.out.println(awtPanel.getLayout());
    // java.awt.FlowLayout[hgap=5,vgap=5,align=center]
```

Panel's Initial Layout

Panel (and Swing's JPanel) provides a constructor to set its initial layout manager. It is because a primary function of `Panel` is to layout a group of component in a particular layout.

```
public void Panel (LayoutManager layout)
   // Construct a Panel in the given layout
   // By default, Panel (and JPanel) has FlowLayout

// For example, create a Panel in BorderLayout
Panel mainPanel = new Panel(new BorderLayout());
```

8.1 FlowLayout

In the `java.awt.FlowLayout`, components are arranged from left-to-right inside the container in the order that they are added (via method `aContainer.add(aComponent)`). When one row is filled, a new row will be started. The actual appearance depends on the width of the display window.
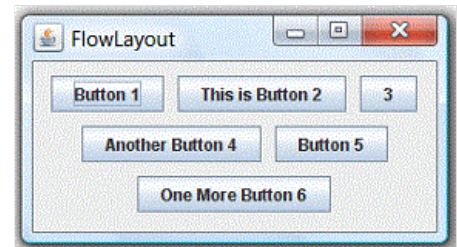
Constructors

```
public FlowLayout();
public FlowLayout(int align);
public FlowLayout(int align, int hgap, int vgap);
   // align: FlowLayout.LEFT (or LEADING), FlowLayout.RIGHT (or TRAILING), or FlowLayout.CENTER
   // hgap, vgap: horizontal/vertical gap between the components
   // By default: hgap=5, vgap=5, align=CENTER
```

Example

```
 1   import java.awt.*;
 2   import java.awt.event.*;
 3
 4   // An AWT GUI program inherits the top-level container java.awt.Frame
 5   public class AWTFlowLayoutDemo extends Frame {
 6      private Button btn1, btn2, btn3, btn4, btn5, btn6;
 7
 8      // Constructor to setup GUI components and event handlers
 9      public AWTFlowLayoutDemo () {
10         setLayout(new FlowLayout());
11            // "super" Frame sets layout to FlowLayout, which arranges the components
12            // from left-to-right, and flow from top-to-bottom.
13
14         btn1 = new Button("Button 1");
15         add(btn1);
16         btn2 = new Button("This is Button 2");
17         add(btn2);
18         btn3 = new Button("3");
19         add(btn3);
20         btn4 = new Button("Another Button 4");
21         add(btn4);
22         btn5 = new Button("Button 5");
23         add(btn5);
24         btn6 = new Button("One More Button 6");
25         add(btn6);
26
27         setTitle("FlowLayout Demo"); // "super" Frame sets title
28         setSize(280, 150);           // "super" Frame sets initial size
29         setVisible(true);            // "super" Frame shows
30      }
31
32      // The entry main() method
33      public static void main(String[] args) {
34         new AWTFlowLayoutDemo();  // Let the constructor do the job
```
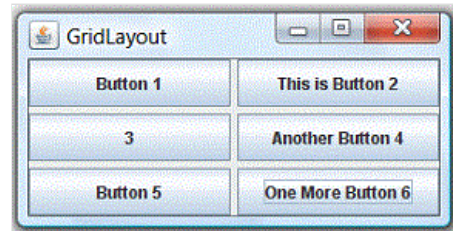
```
35      }
36  }
```

8.2 GridLayout

In java.awt.GridLayout, components are arranged in a grid (matrix) of rows and columns inside the Container. Components are added in a left-to-right, top-to-bottom manner in the order they are added (via method *aContainer*.add(*aComponent*)).

Constructors

```
public GridLayout(int rows, int columns);
public GridLayout(int rows, int columns, int hgap, int vgap);
    // By default: rows=1, cols=0, hgap=0, vgap=0
```

Example

```
 1  import java.awt.*;
 2  import java.awt.event.*;
 3
 4  // An AWT GUI program inherits the top-level container java.awt.Frame
 5  public class AWTGridLayoutDemo extends Frame {
 6     private Button btn1, btn2, btn3, btn4, btn5, btn6;
 7
 8     // Constructor to setup GUI components and event handlers
 9     public AWTGridLayoutDemo () {
10        setLayout(new GridLayout(3, 2, 3, 3));
11           // "super" Frame sets layout to 3x2 GridLayout, horizontal and verical gaps of 3 pixels
12
13        // The components are added from left-to-right, top-to-bottom
14        btn1 = new Button("Button 1");
15        add(btn1);
16        btn2 = new Button("This is Button 2");
17        add(btn2);
18        btn3 = new Button("3");
19        add(btn3);
20        btn4 = new Button("Another Button 4");
21        add(btn4);
22        btn5 = new Button("Button 5");
23        add(btn5);
24        btn6 = new Button("One More Button 6");
25        add(btn6);
26
27        setTitle("GridLayout Demo"); // "super" Frame sets title
28        setSize(280, 150);           // "super" Frame sets initial size
29        setVisible(true);            // "super" Frame shows
30     }
31
32     // The entry main() method
33     public static void main(String[] args) {
34        new AWTGridLayoutDemo();  // Let the constructor do the job
35     }
36  }
```

If rows or cols is 0, but not both, then any number of components can be placed in that column or row. If both the rows and cols are specified, the cols value is ignored. The actual cols is determined by the actual number of components and rows.

8.3 BorderLayout

In java.awt.BorderLayout, the container is divided into 5 zones: EAST, WEST, SOUTH, NORTH, and CENTER. Components are added using method *aContainer*.add(*acomponent, aZone*), where *azone* is either BorderLayout.NORTH (or PAGE_START), BorderLayout.SOUTH (or PAGE_END), BorderLayout.WEST (or LINE_START), BorderLayout.EAST (or LINE_END), or BorderLayout.CENTER. The method *aContainer*.add(*aComponent*) without specifying the zone adds the component to the CENTER.

You need not add components to all the 5 zones. The NORTH and SOUTH components may be stretched horizontally; the EAST and WEST components may be stretched vertically; the CENTER component may stretch both horizontally and vertically to fill any space left over.
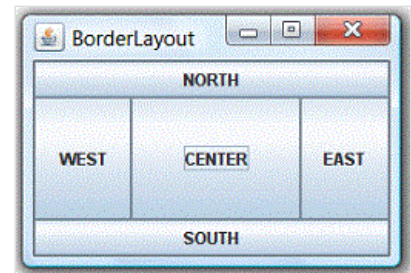
Constructors

```
public BorderLayout();
public BorderLayout(int hgap, int vgap);
      // By default hgap=0, vgap=0
```
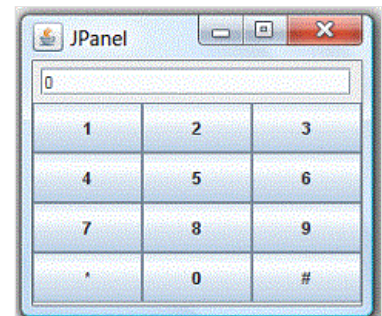
Example

```
 1   import java.awt.*;
 2   import java.awt.event.*;
 3
 4   // An AWT GUI program inherits the top-level container java.awt.Frame
 5   public class AWTBorderLayoutDemo extends Frame {
 6      private Button btnNorth, btnSouth, btnCenter, btnEast, btnWest;
 7
 8      // Constructor to setup GUI components and event handlers
 9      public AWTBorderLayoutDemo () {
10         setLayout(new BorderLayout(3, 3));
11            // "super" Frame sets layout to BorderLayout,
12            // horizontal and vertical gaps of 3 pixels
13
14         // The components are added to the specified zone
15         btnNorth = new Button("NORTH");
16         add(btnNorth, BorderLayout.NORTH);
17         btnSouth = new Button("SOUTH");
18         add(btnSouth, BorderLayout.SOUTH);
19         btnCenter = new Button("CENTER");
20         add(btnCenter, BorderLayout.CENTER);
21         btnEast = new Button("EAST");
22         add(btnEast, BorderLayout.EAST);
23         btnWest = new Button("WEST");
24         add(btnWest, BorderLayout.WEST);
25
26         setTitle("BorderLayout Demo"); // "super" Frame sets title
27         setSize(280, 150);             // "super" Frame sets initial size
28         setVisible(true);             // "super" Frame shows
29      }
30
31      // The entry main() method
32      public static void main(String[] args) {
33         new AWTBorderLayoutDemo();  // Let the constructor do the job
34      }
35   }
```

8.4 Using `Panels` as Sub-Container to Organize Components

An AWT `Panel` is a rectangular pane, which can be used as sub-container to organized a group of related components in a specific layout (e.g., `FlowLayout`, `BorderLayout`). Panels are secondary containers, which shall be added into a top-level container (such as `Frame`), or another `Panel`.

For example, the following figure shows a `Frame` (in `BorderLayout`) containing two `Panels`, `panelResult` in `FlowLayout` and `panelButtons` in `GridLayout`. `panelResult` is added to the `NORTH`, and `panelButtons` is added to the `CENTER`.

```
1   import java.awt.*;
2   import java.awt.event.*;
3
4   // An AWT GUI program inherits the top-level container java.awt.Frame
5   public class AWTPanelDemo extends Frame {
6      private Button[] btnNumbers;  // Array of 10 numeric Buttons
7      private Button btnHash, btnStar;
8      private TextField tfDisplay;
9
10     // Constructor to setup GUI components and event handlers
11     public AWTPanelDemo () {
12        // Set up display panel
13        Panel panelDisplay = new Panel(new FlowLayout());
14        tfDisplay = new TextField("0", 20);
15        panelDisplay.add(tfDisplay);
16
17        // Set up button panel
18        Panel panelButtons = new Panel(new GridLayout(4, 3));
19        btnNumbers = new Button[10];  // Construct an array of 10 numeric Buttons
20        btnNumbers[1] = new Button("1");  // Construct Button "1"
21        panelButtons.add(btnNumbers[1]);  // The Panel adds this Button
22        btnNumbers[2] = new Button("2");
23        panelButtons.add(btnNumbers[2]);
24        btnNumbers[3] = new Button("3");
25        panelButtons.add(btnNumbers[3]);
26        btnNumbers[4] = new Button("4");
27        panelButtons.add(btnNumbers[4]);
28        btnNumbers[5] = new Button("5");
29        panelButtons.add(btnNumbers[5]);
30        btnNumbers[6] = new Button("6");
31        panelButtons.add(btnNumbers[6]);
32        btnNumbers[7] = new Button("7");
33        panelButtons.add(btnNumbers[7]);
34        btnNumbers[8] = new Button("8");
35        panelButtons.add(btnNumbers[8]);
36        btnNumbers[9] = new Button("9");
37        panelButtons.add(btnNumbers[9]);
38              // You should use a loop for the above statements!!!
39        btnStar = new Button("*");
40        panelButtons.add(btnStar);
41        btnNumbers[0] = new Button("0");
42        panelButtons.add(btnNumbers[0]);
43        btnHash = new Button("#");
44        panelButtons.add(btnHash);
45
46        setLayout(new BorderLayout());  // "super" Frame sets to BorderLayout
47        add(panelDisplay, BorderLayout.NORTH);
48        add(panelButtons, BorderLayout.CENTER);
49
50        setTitle("BorderLayout Demo"); // "super" Frame sets title
51        setSize(200, 200);             // "super" Frame sets initial size
52        setVisible(true);              // "super" Frame shows
53     }
54
55     // The entry main() method
56     public static void main(String[] args) {
57        new AWTPanelDemo();  // Let the constructor do the job
58     }
59  }
```
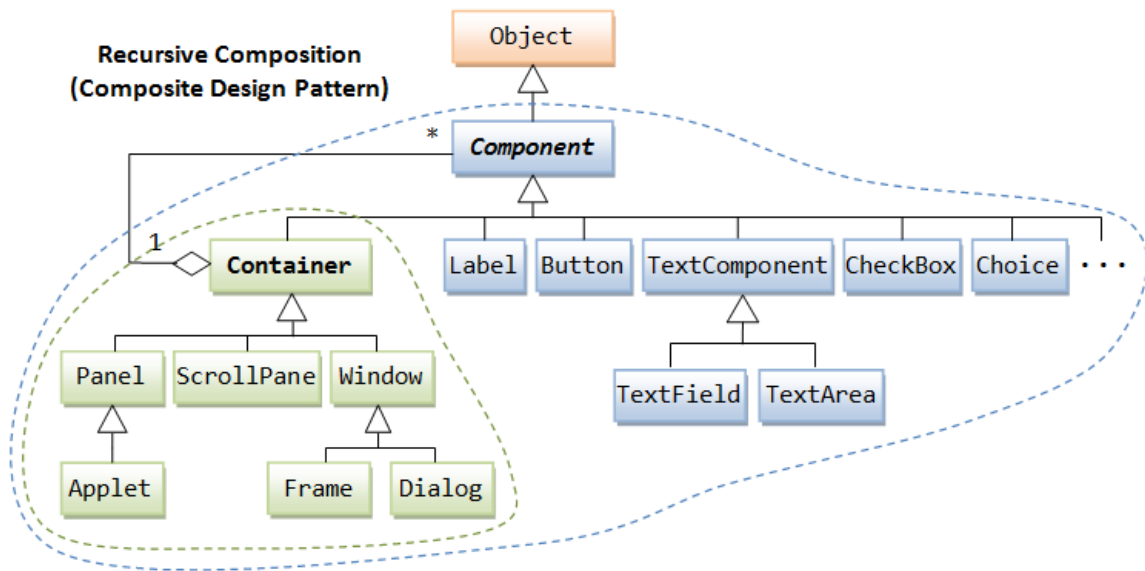
8.5 BoxLayout

BoxLayout arrange components in a single row or column. It respects components' requests on the minimum sizes.

[TODO] Example and diagram

9. (Advanced) Composite Design Pattern

Recursive Composition (Composite Design Pattern) class diagram showing Object → Component → Container, Label, Button, TextComponent, CheckBox, Choice; Container → Panel, ScrollPane, Window; Panel → Applet; Window → Frame, Dialog; TextComponent → TextField, TextArea.

As mentioned earlier, there are two groups of classes in the AWT hierarchy: containers and components. A container (e.g., `Frame`, `Panel`, `Dialog`, `java.applet.Applet`) holds components (e.g., `Label`, `Button`, `TextField`). A container (e.g., `Frame` and `Panel`) can also hold sub-containers (e.g. `Panel`). Hence, we have a situation that "a container can contain containers or components".
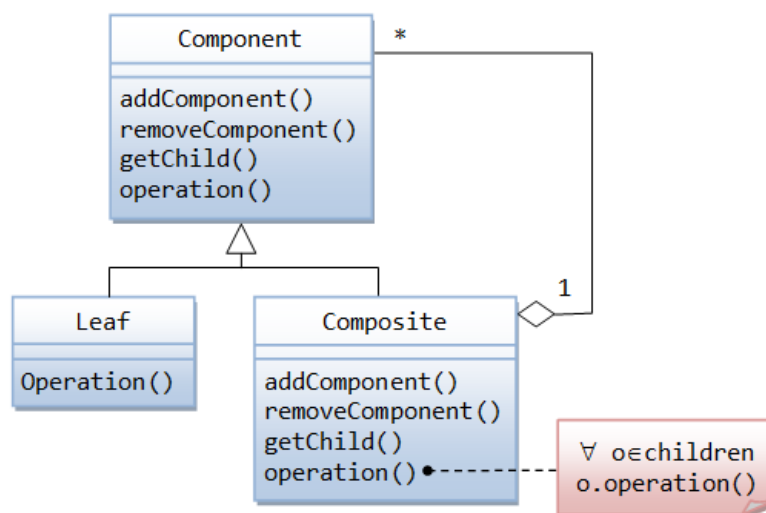
This is quite a common problem: e.g., a directory contains (sub)directories or files; a group contains (sub)groups or elementary elements; the tree structure. A *design pattern* has been proposed for this problem. A design pattern is a proven and possibly the best solution for a specific class of problems.

As shown in the class diagram, there are two sets of relationship between `Container` and `Component` classes.

1. *One-to-many aggregation*: A `Container` contains zero or more `Components`. Each `Component` is contained in exactly one `Container`.

2. *Generalization (or Inheritance)*: `Container` is a *subclass* of `Component`. In other words, a `Container` is a `Component`, which possesses all the properties of `Component` and can be substituted in place of a `Component`.

Combining both relationships, we have: A `Container` contains `Components`. Since a `Container` is a `Component`, a `Container` can also contain `Containers`. Consequently, a `Container` can contain `Containers` and `Components`.

The *Gof* calls this *recursive composition* class design "*composite design pattern*", which is illustrated as follows:

Composite design pattern diagram: Component class (addComponent(), removeComponent(), getChild(), operation()) with Leaf (Operation()) and Composite (addComponent(), removeComponent(), getChild(), operation()) subclasses. Note: ∀ o∈children o.operation()

10. Swing

10.1 Introduction

Swing is part of the so-called "Java Foundation Classes (JFC)" (have you heard of MFC?), which was introduced in 1997 after the release of JDK 1.1. JFC was subsequently included as an integral part of JDK since JDK 1.2. JFC consists of:

- Swing API: for advanced graphical programming.

- Accessibility API: provides assistive technology for the disabled.

- Java 2D API: for high quality 2D graphics and images.

- Pluggable look and feel supports.

- Drag-and-drop support between Java and native applications.

The goal of Java GUI programming is to allow the programmer to build GUI that looks good on ALL platforms. JDK 1.0's AWT was awkward and non-object-oriented (using many `event.getSource()`). JDK 1.1's AWT introduced event-delegation (event-driven) model, much clearer and object-oriented. JDK 1.1 also introduced inner class and JavaBeans – a component programming model for visual programming environment (similar to Visual Basic and Dephi).

Swing appeared after JDK 1.1. It was introduced into JDK 1.1 as part of an add-on JFC (Java Foundation Classes). Swing is a rich set of easy-to-use, easy-to-understand JavaBean GUI components that can be dragged and dropped as "GUI builders" in visual programming environment. Swing is now an integral part of Java since JDK 1.2.

10.2 Swing's Features

Swing is huge (consists of 18 API packages as in JDK 1.7) and has great depth. Compared with AWT, Swing provides a huge and comprehensive collection of reusable GUI components, as shown in the Figure below (extracted form Swing Tutorial).
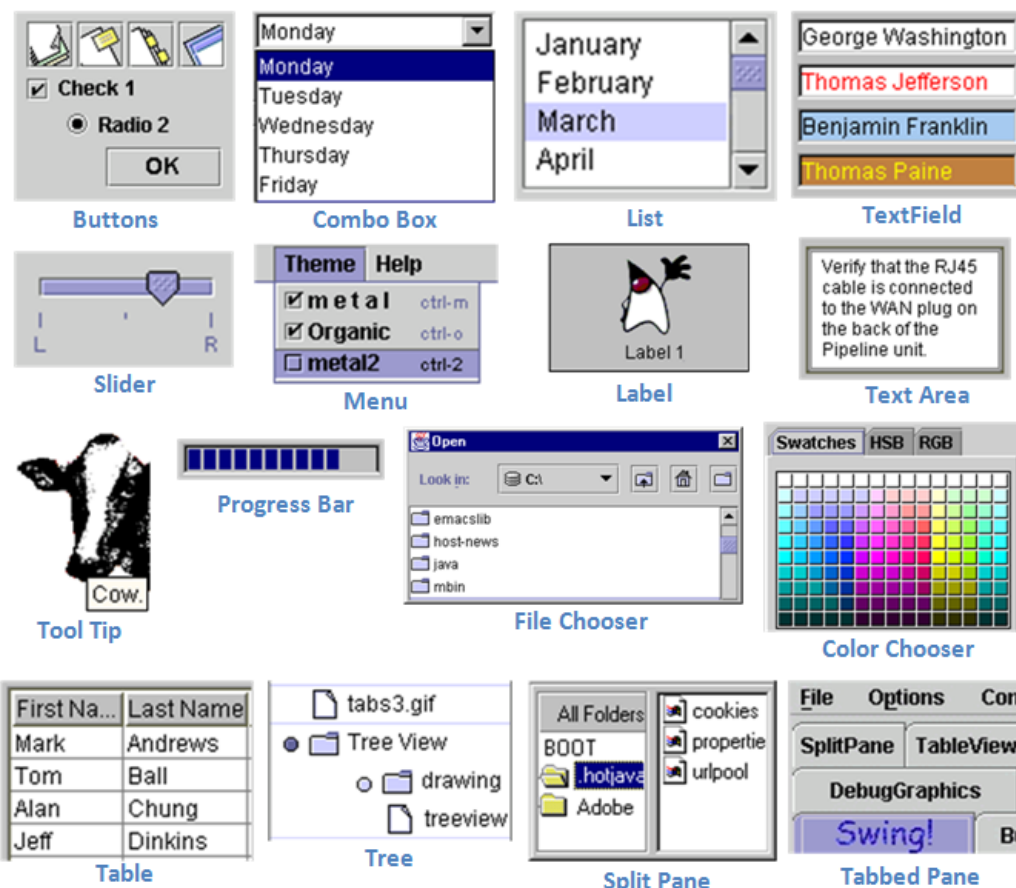
The main features of Swing are (extracted from the Swing website):

1. Swing is written in pure Java (except a few classes) and therefore is 100% portable.

2. Swing components are *lightweight*. The AWT components are *heavyweight* (in terms of system resource utilization). Each AWT component has its own opaque native display, and always displays on top of the lightweight components. AWT components rely heavily on the underlying windowing subsystem of the native operating system. For example, an AWT button ties to an actual button in the underlying native windowing subsystem, and relies on the native windowing subsystem for their rendering and processing. Swing components (`JComponents`) are written in Java. They are generally not "weight-down" by complex GUI considerations imposed by the underlying windowing subsystem.

3. Swing components support *pluggable look-and-feel*. You can choose between *Java look-and-feel* and the *look-and-feel of the underlying OS* (e.g., Windows, UNIX or Mac). If the later is chosen, a Swing button runs on the Windows looks like a Windows' button and feels like a Window's button. Similarly, a Swing button runs on the UNIX looks like a UNIX's button and feels like a UNIX's button.
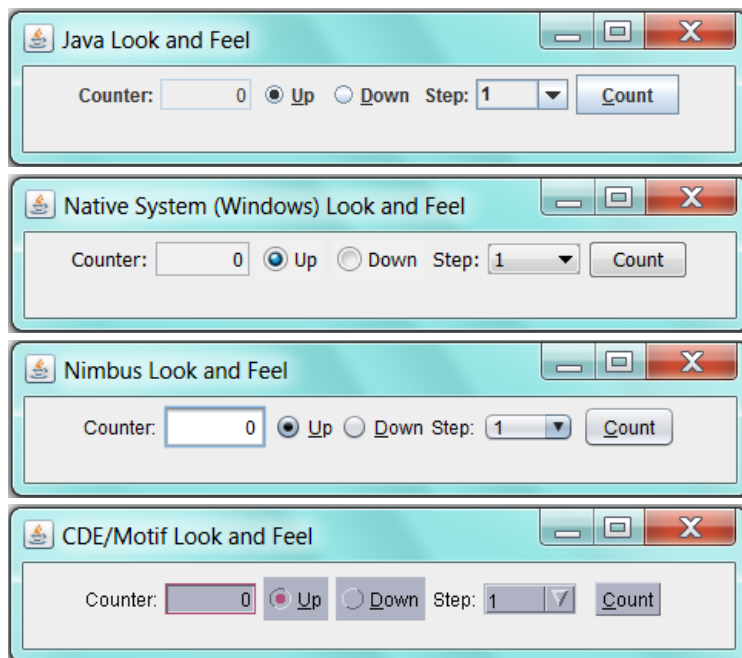
4. Swing supports *mouse-less operation*, i.e., it can operate entirely using keyboard.

5. Swing components support "tool-tips".

6. Swing components are *JavaBeans* – a Component-based Model used in Visual Programming (like Visual Basic). You can drag-and-drop a Swing component into a "design form" using a "GUI builder" and double-click to attach an event handler.

7. Swing application uses AWT event-handling classes (in package `java.awt.event`). Swing added some new classes in package `javax.swing.event`, but they are not frequently used.

8. Swing application uses AWT's layout manager (such as `FlowLayout` and `BorderLayout` in package `java.awt`). It added new layout managers, such as `Springs`, `Struts`, and `BoxLayout` (in package `javax.swing`).

9. Swing implements *double-buffering* and automatic repaint batching for smoother screen repaint.

10. Swing introduces `JLayeredPane` and `JInternalFrame` for creating Multiple Document Interface (MDI) applications.

11. Swing supports floating toolbars (in `JToolBar`), splitter control, "undo".

12. Others - check the Swing website.

10.3 Using Swing API

If you understood the AWT programming (such as container/component, event-handling, layout manager), switching over to Swing (or any other Graphics packages) is straight-forward.

Swing's Components

Compared with the AWT classes (in package `java.awt`), Swing component classes (in package `javax.swing`) begin with a prefix `"J"`, e.g., `JButton`, `JTextField`, `JLabel`, `JPanel`, `JFrame`, or `JApplet`.
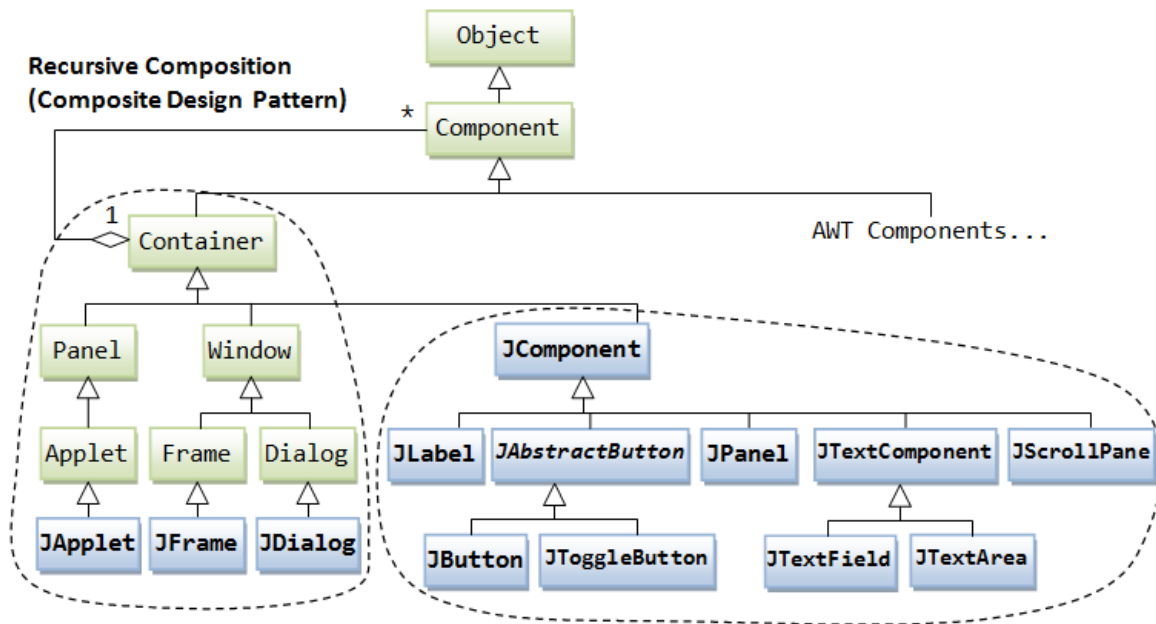
The above figure shows the class hierarchy of the swing GUI classes. Similar to AWT, there are two groups of classes: *container*s and *component*s. A container is used to hold components. A container can also hold containers because it is a (subclass of) component.

As a rule, do not mix heavyweight AWT components and lightweight Swing components in the same program, as the heavyweight components will always be painted *on top of* the lightweight components.

Swing's Top-Level and Secondary Containers

Just like AWT application, a Swing application requires a *top-level container*. There are three top-level containers in Swing:

1. `JFrame`: used for the application's main window (with an icon, a title, minimize/maximize/close buttons, an optional menu-bar, and a content-pane), as illustrated.

2. `JDialog`: used for secondary pop-up window (with a title, a close button, and a content-pane).

3. `JApplet`: used for the applet's display-area (content-pane) inside a browser's window.

Similarly to AWT, there are *secondary containers* (such as `JPanel`) which can be used to group and layout relevant components.
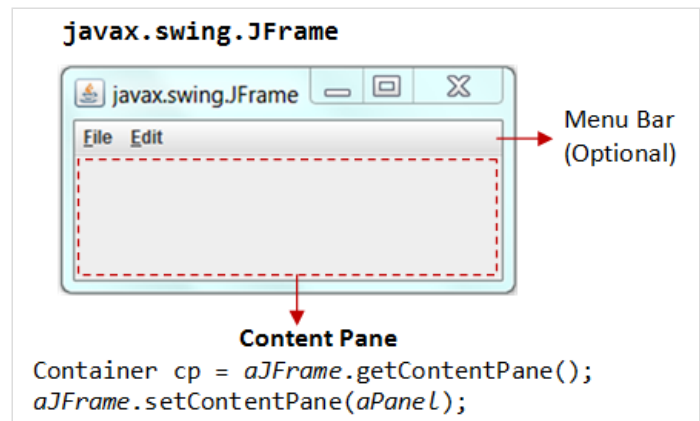
The Content-Pane of Swing's Top-Level Container

However, unlike AWT, the `JComponents` shall not be added onto the top-level container (e.g., `JFrame`, `JApplet`) directly because they are lightweight components. The `JComponents` must be added onto the so-called *content-pane* of the top-level container. Content-pane is in fact a `java.awt.Container` that can be used to group and layout components.

You could:

1. get the content-pane via `getContentPane()` from a top-level container, and add components onto it. For example,

```java
public class TestGetContentPane extends JFrame {
   // Constructor
   public TestGetContentPane() {
      // Get the content-pane of this JFrame, which is a java.awt.Container
      // All operations, such as setLayout() and add() operate on the content-pane
      Container cp = this.getContentPane();
      cp.setLayout(new FlowLayout());
      cp.add(new JLabel("Hello, world!"));
      cp.add(new JButton("Button"));
      ......
   }
   .......
}
```

2. set the content-pane to a JPanel (the main panel created in your application which holds all your GUI components) via JFrame's setContentPane().

```java
public class TestSetContentPane extends JFrame {
    // Constructor
    public TestSetContentPane() {
        // The "main" JPanel holds all the GUI components
        JPanel mainPanel = new JPanel(new FlowLayout());
        mainPanel.add(new JLabel("Hello, world!"));
        mainPanel.add(new JButton("Button"));

        // Set the content-pane of this JFrame to the main JPanel
        this.setContentPane(mainPanel);
        ......
    }
    .......
}
```

Notes: If a component is added directly into a JFrame, it is added into the content-pane of JFrame instead, i.e.,

```java
// "this" is a JFrame
add(new JLabel("add to JFrame directly"));
// is executed as
getContentPane().add(new JLabel("add to JFrame directly"));
```

Event-Handling in Swing

Swing uses the AWT event-handling classes (in package java.awt.event). Swing introduces a few new event-handling classes (in package javax.swing.event) but they are not frequently used.

Writing Swing Applications

In summary, to write a Swing application, you have:

1. Use the Swing components with prefix "J" in package javax.swing, e.g., JFrame, JButton, JTextField, JLabel, etc.

2. A top-level container (such as JFrame or JApplet) is needed. The JComponents should be added directly onto the top-level container. They shall be added onto the *content-pane* of the top-level container. You can retrieve a reference to the content-pane by invoking method getContentPane() from the top-level container, or set the content-pane to the main JPanel created in your program.

3. Swing applications uses AWT event-handling classes, e.g., ActionEvent/ActionListener, MouseEvent/MouseListener, etc.

4. Run the constructor in the Event Dispatcher Thread (instead of Main thread) for thread safety, as shown in the following program template.

10.4 Swing Program Template

```java
 1   import java.awt.*;        // Using AWT layouts
 2   import java.awt.event.*;  // Using AWT event classes and listener interfaces
 3   import javax.swing.*;     // Using Swing components and containers
 4
 5   // A Swing GUI application inherits from top-level container javax.swing.JFrame
 6   public class ...... extends JFrame {
 7
 8      // Private instance variables
 9      // ......
10
11      // Constructor to setup the GUI components and event handlers
12      public ......() {
13         // Retrieve the top-level content-pane
14         Container cp = this.getContentPane();
15
16         // Content-pane sets layout
17         cp.setLayout(new ....Layout());
18
19         // Allocate the GUI components
20         // .....
21
22         // Content-pane adds components
23         cp.add(....);
24
25         // Source object adds listener
```

```
26          // .....
27
28          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29             // Exit the program when the close-window button clicked
30          setTitle("......");   // "super" JFrame sets title
31          setSize(300, 150);    // "super" JFrame sets initial size (or pack())
32          setVisible(true);     // "super" JFrame shows
33       }
34
35       // The entry main() method
36       public static void main(String[] args) {
37          // Run GUI codes in Event-Dispatching thread for thread-safety
38          SwingUtilities.invokeLater(new Runnable() {
39             @Override
40             public void run() {
41                new ......();   // Let the constructor do the job
42             }
43          });
44       }
45    }
```

I will explain this template in the following Swing example.

10.5 Swing Example 1: `SwingCounter`

Let's convert the earlier AWT application example into Swing. Compare the two source files and note the changes (which are highlighted). The display is shown below. Note the differences in *look and feel* between the AWT GUI components and Swing's.
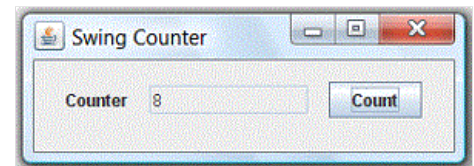

```
1    import java.awt.*;        // Using AWT layouts
2    import java.awt.event.*; // Using AWT event classes and listener interfaces
3    import javax.swing.*;     // Using Swing components and containers
4
5    // A Swing GUI application inherits from top-level container javax.swing.JFrame
6    public class SwingCounter extends JFrame {    // JFrame instead of Frame
7       private JTextField tfCount;  // Use Swing's JTextField instead of AWT's TextField
8       private JButton btnCount;     // Using Swing's JButton instead of AWT's Button
9       private int count = 0;
10
11      // Constructor to setup the GUI components and event handlers
12      public SwingCounter () {
13         // Retrieve the content-pane of the top-level container JFrame
14         // All operations done on the content-pane
15         Container cp = getContentPane();
16         cp.setLayout(new FlowLayout());   // The content-pane sets its layout
17
18         cp.add(new JLabel("Counter"));
19         tfCount = new JTextField("0", 10);
20         tfCount.setEditable(false);
21         cp.add(tfCount);
22
23         btnCount = new JButton("Count");
24         cp.add(btnCount);
25
26         // Allocate an anonymous instance of an anonymous inner class that
27         //  implements ActionListener as ActionEvent listener
28         btnCount.addActionListener(new ActionListener() {
29            @Override
30            public void actionPerformed(ActionEvent evt) {
31               ++count;
32               tfCount.setText(count + "");
33            }
34         });
35
36         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  // Exit program if close-window button clicked
37         setTitle("Swing Counter"); // "super" JFrame sets title
38         setSize(300, 100);         // "super" JFrame sets initial size
```

```
39          setVisible(true);          // "super" JFrame shows
40      }
41
42      // The entry main() method
43      public static void main(String[] args) {
44          // Run the GUI construction in the Event-Dispatching thread for thread-safety
45          SwingUtilities.invokeLater(new Runnable() {
46              @Override
47              public void run() {
48                  new SwingCounter(); // Let the constructor do the job
49              }
50          });
51      }
52  }
```

JFrame's Content-Pane

The JFrams's method getContentPane() returns the content-pane (which is a java.awt.Containter) of the JFrame. You can then set its layout (the default layout is BorderLayout), and add components into it. For example,

```
Container cp = getContentPane(); // Get the content-pane of this JFrame
cp.setLayout(new FlowLayout());  // content-pane sets to FlowLayout
cp.add(new JLabel("Counter"));   // content-pane adds a JLabel component
......
cp.add(tfCount);     // content-pane adds a JTextField component
......
cp.add(btnCount);    // content-pane adds a JButton component
```

You can also use the JFrame's setContentPane() method to directly set the content-pane to a JPanel (or a JComponent). For example,

```
JPanel displayPanel = new JPanel();
this.setContentPane(displayPanel);
      // "this" JFrame sets its content-pane to a JPanel directly
.....

// The above is different from:
this.getContentPane().add(displayPanel);
      // Add a JPanel into the content-pane. Appearance depends on the JFrame's layout.
```

JFrame's setDefaultCloseOperation()

Instead of writing a WindowEvent listener with a windowClosing() handler to process the "close-window" button, JFrame provides a method called setDefaultCloseOperation() to sets the default operation when the user initiates a "close" on this frame. Typically, we choose the option JFrame.EXIT_ON_CLOSE, which terminates the application via a System.exit().

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Running the GUI Construction Codes on the Event-Dispatching Thread

In the previous examples, we invoke the constructor directly in the entry main() method to setup the GUI components. For example,

```
// The entry main method
public static void main(String[] args) {
   // Invoke the constructor (by allocating an instance) to setup the GUI
   new SwingCounter();
}
```

The constructor will be executed in the so-called "Main-Program" thread. This may cause multi-threading issues (such as unresponsive user-interface and deadlock).

It is recommended to execute the GUI setup codes in the so-called "Event-Dispatching" thread, instead of "Main-Program" thread, for thread-safe operations. Event-dispatching thread, which processes events, should be used when the codes updates the GUI.

To run the constructor on the event-dispatching thread, invoke static method SwingUtilities.invokeLater() to asynchronously queue the constructor on the event-dispatching thread. The codes will be run after all pending events have been processed. For example,

```
public static void main(String[] args) {
   // Run the GUI codes in the Event-dispatching thread for thread-safety
   SwingUtilities.invokeLater(new Runnable() {
      @Override
      public void run() {
         new SwingCounter();  // Let the constructor do the job
```

```
        }
    });
}
```

Note: `javax.swing.SwingUtilities.invokeLater()` is a cover for `java.awt.EventQueue.invokeLater()` (which is used in the NetBeans' Visual GUI Builder).

At times, for example in game programming, the *constructor* or the `main()` may contains non-GUI codes. Hence, it is a common practice to create a dedicated method called `initComponents()` (used in NetBeans visual GUI builder) or `createAndShowGUI()` (used in Swing tutorial) to handle all the GUI codes (and another method called `initGame()` to handle initialization of the game's objects). This GUI init method shall be run in the event-dispatching thread.

Warning Message "The serialization class does not declare a static final serialVersionUID field of type long"

This warning message is triggered because `java.awt.Frame` (via its superclass `java.awt.Component`) implements the `java.io.Serializable` interface. This interface enables the object to be written out to an output stream *serially* (via method `writeObject()`); and read back into the program (via method `readObject()`). The serialization runtime uses a number (called `serialVersionUID`) to ensure that the object read into the program is compatible with the class definition, and not belonging to another version.

You have these options:

1. Simply ignore this warning message. If a `serializable` class does not explicitly declare a `serialVersionUID`, then the serialization runtime will calculate a default `serialVersionUID` value for that class based on various aspects of the class.

2. Add a `serialVersionUID` (Recommended), e.g.

```
private static final long serialVersionUID = 1L;  // version 1
```

3. Suppress this particular warning via annotation @SuppressWarmomgs (in package `java.lang`) (JDK 1.5):

```
@SuppressWarnings("serial")
public class MyFrame extends JFrame { ...... }
```

10.6 Swing Example 2: `SwingAccumulator`

```
1   import java.awt.*;        // Using layouts
2   import java.awt.event.*; // Using AWT event classes and listener interfaces
3   import javax.swing.*;     // Using Swing components and containers
4
5   // A Swing GUI application inherits the top-level container javax.swing.JFrame
6   public class SwingAccumulator extends JFrame {
7      private JTextField tfInput, tfOutput;
8      private int sum = 0;     // accumulated sum, init to 0
9
10     // Constructor to setup the GUI components and event handlers
11     public SwingAccumulator() {
12        // Retrieve the content-pane of the top-level container JFrame
13        // All operations done on the content-pane
14        Container cp = getContentPane();
15        cp.setLayout(new GridLayout(2, 2, 5, 5));  // The content-pane sets its layout
16
17        add(new JLabel("Enter an Integer: "));
18        tfInput = new JTextField(10);
19        add(tfInput);
20        add(new JLabel("The Accumulated Sum is: "));
21        tfOutput = new JTextField(10);
22        tfOutput.setEditable(false);  // read-only
23        add(tfOutput);
24
25        // Allocate an anonymous instance of an anonymous inner class that
26        //  implements ActionListener as ActionEvent listener
27        tfInput.addActionListener(new ActionListener() {
28           @Override
29           public void actionPerformed(ActionEvent evt) {
30              // Get the String entered into the input TextField, convert to int
31              int numberIn = Integer.parseInt(tfInput.getText());
32              sum += numberIn;      // accumulate numbers entered into sum
33              tfInput.setText("");  // clear input TextField
34              tfOutput.setText(sum + ""); // display sum on the output TextField
35           }
```

```
36              });
37
38          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  // Exit program if close-window button clicked
39          setTitle("Swing Accumulator"); // "super" Frame sets title
40          setSize(350, 120);  // "super" Frame sets initial size
41          setVisible(true);   // "super" Frame shows
42       }
43
44       // The entry main() method
45       public static void main(String[] args) {
46          // Run the GUI construction in the Event-Dispatching thread for thread-safety
47          SwingUtilities.invokeLater(new Runnable() {
48             @Override
49             public void run() {
50                new SwingAccumulator(); // Let the constructor do the job
51             }
52          });
53       }
54    }
```

11. Using Visual GUI Builder - NetBeans/Eclipse

If you have a complicated layout for your GUI application, you should use a GUI Builder, such as NetBeans or Eclipse to layout your GUI components in a drag-and-drop manner, similar to the popular visual languages such as Visual Basic and Dephi.

11.1 NetBeans

For using NetBeans GUI Builder, read my "Writing Java GUI (AWT/Swing) Application in NetBeans"; or Swing Tutorial's "Learning Swing with the NetBeans IDE".

11.2 Eclipse

For using Eclipse GUI Builder, read "Writing Swing Applications using Eclipse GUI Builder".

LINK TO JAVA REFERENCES & RESOURCES

MORE REFERENCES & RESOURCES

1. "Creating a GUI With JFC/Swing" (aka "The Swing Tutorial") @ http://docs.oracle.com/javase/tutorial/uiswing/.
2. JFC Demo (under JDK demo "jfc" directory).
3. Java2D Tutorial @ http://docs.oracle.com/javase/tutorial/2d/index.html.
4. JOGL (Java Binding on OpenGL) @ http://java.net/projects/jogl/.
5. Java3D (@ http://java3d.java.net/).

Latest version tested: JDK 1.8.0
Last modified: April, 2016