UNIT- III

What is an Event?

Change in the state of an object is known as event i.e. event describes the change in state of source. Events are generated as result of user interaction with the graphical user interface components. For example, clicking on a button, moving the mouse, entering a character through keyboard,selecting an item from list, scrolling the page are the activities that causes an event to happen.

Types of Event

The events can be broadly classified into two categories:

- **Foreground Events** - Those events which require the direct interaction of user.They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard,selecting an item from list, scrolling the page etc.
- **Background Events** - Those events that require the interaction of end user are known as background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

What is Event Handling?

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism have the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.Let's have a brief introduction to this model.

The Delegation Event Model has the following key participants namely:

- **Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.
- **Listener** - It is also known as event handler.Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event. The user interface element is able to delegate the processing of an event to the separate piece of code. In this model, Listener needs to be registered with the source object so that the listener can receive the event notification. This is an efficient way of handling the event because the event notifications are sent only to those listener that want to receive them.

Steps involved in event handling

- The User clicks the button and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.

- the method is now get executed and returns.

Points to remember about listener

- In order to design a listener class we have to develop some listener interfaces. These Listener interfaces forecast some public abstract callback methods which must be implemented by the listener class.
- If you do not implement the any if the predefined interfaces then your class can not act as a listener class for a source object.

Callback Methods

These are the methods that are provided by API provider and are defined by the application programmer and invoked by the application developer. Here the callback methods represents an event method. In response to an event java jre will fire callback method. All such callback methods are provided in listener interfaces.

If a component wants some listener will listen to it's events the the source must register itself to the listener.

Event Handling Example

Create the following java program using any editor of your choice in say **D:/ > AWT > com > tutorialspoint > gui >**

AwtControlDemo.java

```
package com.tutorialspoint.gui;

import java.awt.*;
import java.awt.event.*;

public class AwtControlDemo {

  private Frame mainFrame;
  private Label headerLabel;
  private Label statusLabel;
  private Panel controlPanel;

  public AwtControlDemo(){
    prepareGUI();
  }

  public static void main(String[] args){
    AwtControlDemo  awtControlDemo = new AwtControlDemo();
    awtControlDemo.showEventDemo();
  }

  private void prepareGUI(){
    mainFrame = new Frame("Java AWT Examples");
    mainFrame.setSize(400,400);
    mainFrame.setLayout(new GridLayout(3, 1));
```

```
    mainFrame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent windowEvent){
        System.exit(0);
      }
    });
    headerLabel = new Label();
    headerLabel.setAlignment(Label.CENTER);
    statusLabel = new Label();
    statusLabel.setAlignment(Label.CENTER);
    statusLabel.setSize(350,100);

    controlPanel = new Panel();
    controlPanel.setLayout(new FlowLayout());

    mainFrame.add(headerLabel);
    mainFrame.add(controlPanel);
    mainFrame.add(statusLabel);
    mainFrame.setVisible(true);
  }

  private void showEventDemo(){
    headerLabel.setText("Control in action: Button");

    Button okButton = new Button("OK");
    Button submitButton = new Button("Submit");
    Button cancelButton = new Button("Cancel");

    okButton.setActionCommand("OK");
    submitButton.setActionCommand("Submit");
    cancelButton.setActionCommand("Cancel");

    okButton.addActionListener(new ButtonClickListener());
    submitButton.addActionListener(new ButtonClickListener());
    cancelButton.addActionListener(new ButtonClickListener());

    controlPanel.add(okButton);
    controlPanel.add(submitButton);
    controlPanel.add(cancelButton);

    mainFrame.setVisible(true);
  }

  private class ButtonClickListener implements ActionListener{
    public void actionPerformed(ActionEvent e) {
      String command = e.getActionCommand();
      if( command.equals( "OK" ))  {
        statusLabel.setText("Ok Button clicked.");
      }
      else if( command.equals( "Submit" ) )  {
        statusLabel.setText("Submit Button clicked.");
      }
```

```
        else {
          statusLabel.setText("Cancel Button clicked.");
        }
      }
    }
}
```

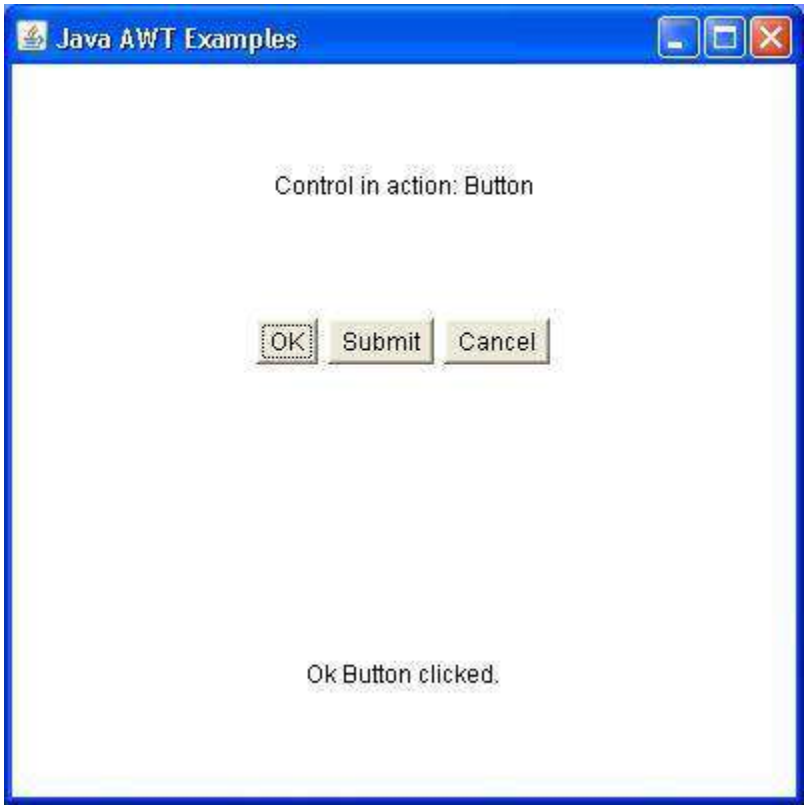
Compile the program using command prompt. Go to **D:/ > AWT** and type the following command.

D:\AWT>javac com\tutorialspoint\gui\AwtControlDemo.java

If no error comes that means compilation is successful. Run the program using following command.

D:\AWT>java com.tutorialspoint.gui.AwtControlDemo

Verify the following output



Event Handlers

What is an event handler?
In programming, an event handler is a callback routine that operates asynchronously once an event takes place. It dictates the action that follows the event. The programmer writes a code for this action to take place.

An event is an action that takes place when a user interacts with a program. For example, an event can be a mouse click or pressing a key on the keyboard. An event can be a single element or even an HTML document.

Let's say a mouse click on a button displays the message "DOM Event." The event handler will be the programmed routine whereby each time a user clicks on the button, the message "DOM Event" is displayed.

An event is a meaningful element of application information from an underlying development framework. There are many types of events, either from a graphical user interface (GUI) toolkit or some input-type routine.

On the GUI side, events include keystrokes, mouse activity, action selections or timer expirations. On the input side, events include opening or closing files and data streams, reading data and so forth.

Here are some common examples of an event handler:

- A notification pops up on a webpage when a new tab is opened.
- A form is submitted when the submit button is clicked.
- The background color changes on a mouse click.

Adapter Classes

A Java adapter class allows listener interfaces to be implemented by default. The Delegation Event Model gave rise to the concept of listener interfaces. It is one of the various strategies used to handle events in Graphical User Interface (GUI) programming languages like Java.

In most event-driven GUI development, the user interacts with the system through related images and graphics. This means any action done by the user is regarded as a separate event, such as moving the mouse pointer on the screen to change its coordinates, clicking a button, or scrolling a page.

These different event activities are inextricably linked to a code segment that iterates the application's response to the user. The path is pretty straightforward. The user generates an event that is sent to one or more listener interfaces. When an event potential is received, the Listener Interface analyses it and responds appropriately.

This path validates the event handling process. A Java adapter class implements an interface with a set of dummy methods. When programmers inherit a Java adapter class, they are not obligated to implement all of the methods mentioned under each listener interface. The adapter class can also be subclassed, allowing the programmer to override only the required methods. In other words, a programmer can quickly create their listener interface to field events by utilizing a Java adapter class—this aids in the reduction of code.

Read More About, Basics of Java

Relationship Between Java Adapter Class and Listener Interface

Listeners are used when a programmer intends to use the majority of the methods given in the interface. If a class implements a listener interface directly, it must implement all of the interface's functions, resulting

in overly long code. The use of a Java adapter class can aid in the resolution of this issue. A Java adapter class comes in helpful when an event requires a few specialized methods.

To use a Java adapter class, the programmer only needs to create a subclass of it and override the interest methods. Java Adapter classes are advantageous for Java listener interfaces with several methods.

Types of Java Adapter Classes

Java adapter classes can be found in java.awt.event, java.awt.dnd, and javax.swing.event packages. The adapter classes are listed below, along with their listener interfaces.

java.awt.event Adapter classes

Java adapter class	**Java Listener interface**
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
HierarchyBoundsAdapter	HierarchyBoundsListener

java.awt.dnd Adapter classes

Java adapter class	**Java Listener interface**
DragSourceAdapter	DragSourceListener
DragTargetAdapter	DragTargetListener

javax.swing.event Adapter classes

Java Adapter class	Java Listener interface
MouseInputAdapter	MouseInputListener
InternalFrameAdapter	InternalFrameListener

Examples

Java WindowAdapter

The WindowAdapter class of AWT is implemented in the following example, and one of its methods, windowClosing() is used to close the frame window.

```
// importing the necessary libraries
import java.awt.*;
import java.awt.event.*;

public class AdapterExample {
// object of Frame
   Frame f;

// class constructor
   AdapterExample() {
// creating a frame with the title
      f = new Frame ("Window Adapter");
// adding the WindowListener to the frame
// overriding the windowClosing() method
      f.addWindowListener (new WindowAdapter() {
         public void windowClosing (WindowEvent e) {
            f.dispose();
         }
      });
       // setting the size, layout and
      f.setSize (400, 400);
```

```
        f.setLayout (null);
        f.setVisible (true);
    }

// main method
public static void main(String[] args) {
    new AdapterExample();
}
}
```

You can also try this code with Online Java Compiler

Run Code

Java MouseMotionAdapter

The MouseMotionAdapter class and its various methods are implemented in the following example to listen to mouse motion events in the Frame window.

```
// importing the necessary libraries
import java.awt.*;
import java.awt.event.*;

// class which inherits the MouseMotionAdapter class
public class MouseMotionAdapterExample extends MouseMotionAdapter {

// object of Frame class
    Frame f;

// class constructor
    MouseMotionAdapterExample() {
// creating the frame with the title
        f = new Frame ("Mouse Motion Adapter");
// adding MouseMotionListener to the Frame
```

```
        f.addMouseMotionListener (this);
 // setting the size, layout and visibility of the frame
        f.setSize (300, 300);
        f.setLayout (null);
        f.setVisible (true);
    }

// overriding the mouseDragged() method
public void mouseDragged (MouseEvent e) {
// creating the Graphics object and fetching them from the Frame object using getGraphics() method
    Graphics g = f.getGraphics();
// setting the color of graphics object
    g.setColor (Color.ORANGE);
// setting the shape of graphics object
    g.fillOval (e.getX(), e.getY(), 20, 20);
}

public static void main(String[] args) {
    new MouseMotionAdapterExample();
}
}
```

Output:

Advantages of Java Adapter Class

The advantages of Java adapter class area as follows:-

- With the help of the Java adapter class, unrelated classes can work together.
- With the help of a Java adapter class, the same class can be used in multiple ways.
- Users can construct apps with the help of a pluggable kit. As a result, class is reused a lot.
- Java adapter class helps to make classes more visible.
- Java adapter classes allow us to put patterns in a class together.
- Java adapter classes allow courses that are not connected to collaborate more easily.
- Java adapter classes give us the flexibility to use classes in a variety of ways.
- Java adapter classes help to make classes more visible.
- Java adapter classes permit similar patterns to be included in the class.
- Java adapter class comes with a pluggable app development kit.
- Java adapter class increases the reusability of the class.

Actions

Java ActionListener Interface

The Java ActionListener is notified whenever you click on the button or menu item. It is notified against ActionEvent. The ActionListener interface is found in java.awt.event package. It has only one method: actionPerformed().

actionPerformed() method

The actionPerformed() method is invoked automatically whenever you click on the registered component.

```
1. public abstract void actionPerformed(ActionEvent e);
```

How to write ActionListener

The common approach is to implement the ActionListener. If you implement the ActionListener class, you need to follow 3 steps:

1) Implement the ActionListener interface in the class:

```
1. public class ActionListenerExample Implements ActionListener
```

2) Register the component with the Listener:

```
1. component.addActionListener(instanceOfListenerclass);
```

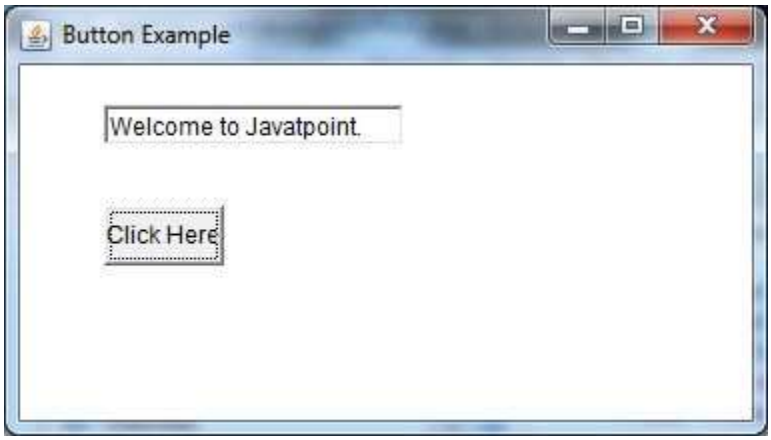
3) Override the actionPerformed() method:

```
1. public void actionPerformed(ActionEvent e){
2.         //Write the code here
3. }
```

Java ActionListener Example: On Button click

```
1. import java.awt.*;
2. import java.awt.event.*;
3. //1st step
4. public class ActionListenerExample implements ActionListener{
5. public static void main(String[] args) {
6.    Frame f=new Frame("ActionListener Example");
7.    final TextField tf=new TextField();
8.    tf.setBounds(50,50, 150,20);
9.    Button b=new Button("Click Here");
10.   b.setBounds(50,100,60,30);
11.   //2nd step
12.   b.addActionListener(this);
13.   f.add(b);f.add(tf);
14.   f.setSize(400,400);
15.   f.setLayout(null);
16.   f.setVisible(true);
17. }
18. //3rd step
19. public void actionPerformed(ActionEvent e){
20.         tf.setText("Welcome to Javatpoint.");
21. }
22. }
```

Output:



Java ActionListener Example: Using Anonymous class

We can also use the anonymous class to implement the ActionListener. It is the shorthand way, so you do not need to follow the 3 steps:

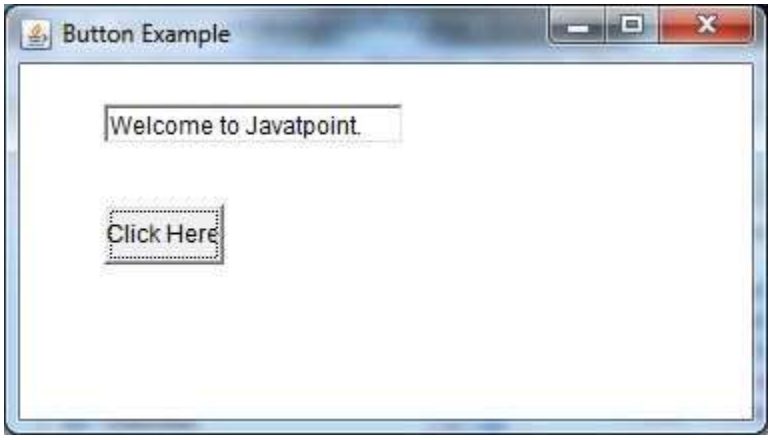
```
1. b.addActionListener(new ActionListener(){
2. public void actionPerformed(ActionEvent e){
3.         tf.setText("Welcome to Javatpoint.");
4. }
5. });
```

Let us see the full code of ActionListener using anonymous class.

```
1. import java.awt.*;
2. import java.awt.event.*;
3. public class ActionListenerExample {
4. public static void main(String[] args) {
5.     Frame f=new Frame("ActionListener Example");
6.     final TextField tf=new TextField();
7.     tf.setBounds(50,50, 150,20);
8.     Button b=new Button("Click Here");
9.     b.setBounds(50,100,60,30);
10.
11.    b.addActionListener(new ActionListener(){
12.    public void actionPerformed(ActionEvent e){
13.         tf.setText("Welcome to Javatpoint.");
14.    }
```

```
15.     });
16.     f.add(b);f.add(tf);
17.     f.setSize(400,400);
18.     f.setLayout(null);
19.     f.setVisible(true);
20. }
21. }
```

Output:



Java MouseListener Interface

The Java MouseListener is notified whenever you change the state of mouse. It is notified against MouseEvent. The MouseListener interface is found in java.awt.event package. It has five methods.

Methods of MouseListener interface

The signature of 5 methods found in MouseListener interface are given below:

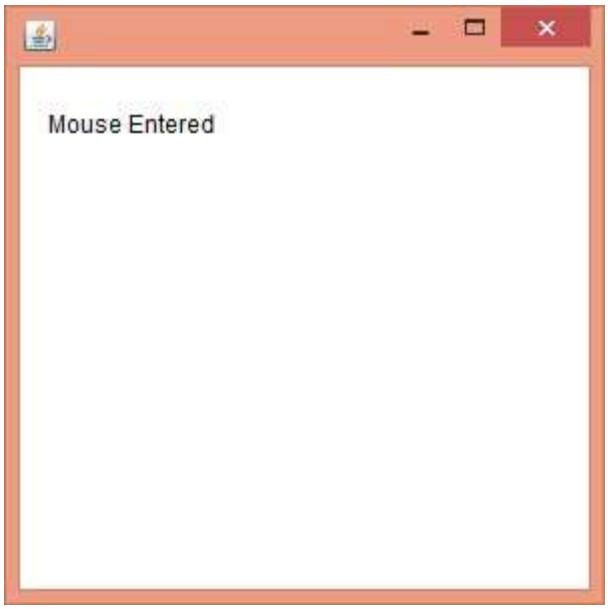
1. **public abstract void** mouseClicked(MouseEvent e);
2. **public abstract void** mouseEntered(MouseEvent e);
3. **public abstract void** mouseExited(MouseEvent e);
4. **public abstract void** mousePressed(MouseEvent e);
5. **public abstract void** mouseReleased(MouseEvent e);

Java MouseListener Example

```
1. import java.awt.*;
2. import java.awt.event.*;
```

```
3.  public class MouseListenerExample extends Frame implements MouseListener{
4.      Label l;
5.      MouseListenerExample(){
6.          addMouseListener(this);
7.
8.          l=new Label();
9.          l.setBounds(20,50,100,20);
10.         add(l);
11.         setSize(300,300);
12.         setLayout(null);
13.         setVisible(true);
14.     }
15.     public void mouseClicked(MouseEvent e) {
16.         l.setText("Mouse Clicked");
17.     }
18.     public void mouseEntered(MouseEvent e) {
19.         l.setText("Mouse Entered");
20.     }
21.     public void mouseExited(MouseEvent e) {
22.         l.setText("Mouse Exited");
23.     }
24.     public void mousePressed(MouseEvent e) {
25.         l.setText("Mouse Pressed");
26.     }
27.     public void mouseReleased(MouseEvent e) {
28.         l.setText("Mouse Released");
29.     }
30. public static void main(String[] args) {
31.     new MouseListenerExample();
32. }
33. }
```

Output:

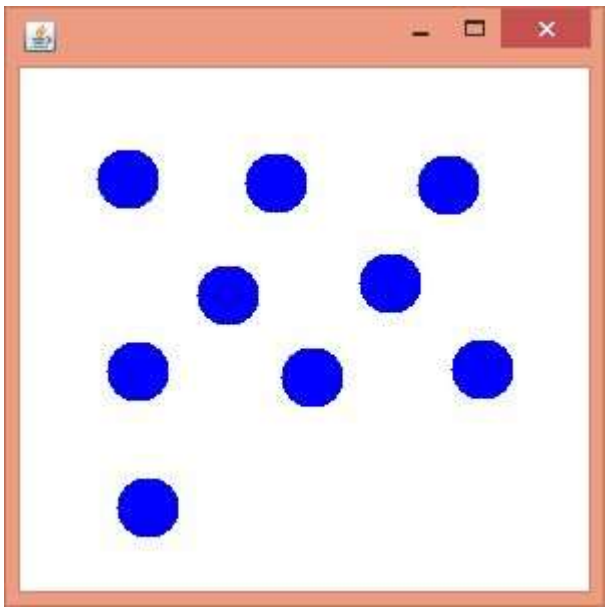


Java MouseListener Example 2

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  public class MouseListenerExample2 extends Frame implements MouseListener{
4.     MouseListenerExample2(){
5.        addMouseListener(this);
6.
7.        setSize(300,300);
8.        setLayout(null);
9.        setVisible(true);
10.    }
11.    public void mouseClicked(MouseEvent e) {
12.       Graphics g=getGraphics();
13.       g.setColor(Color.BLUE);
14.       g.fillOval(e.getX(),e.getY(),30,30);
15.    }
16.    public void mouseEntered(MouseEvent e) {}
17.    public void mouseExited(MouseEvent e) {}
18.    public void mousePressed(MouseEvent e) {}
19.    public void mouseReleased(MouseEvent e) {}
```

```
20.
21. public static void main(String[] args) {
22.     new MouseListenerExample2();
23. }
24. }
```

Output:

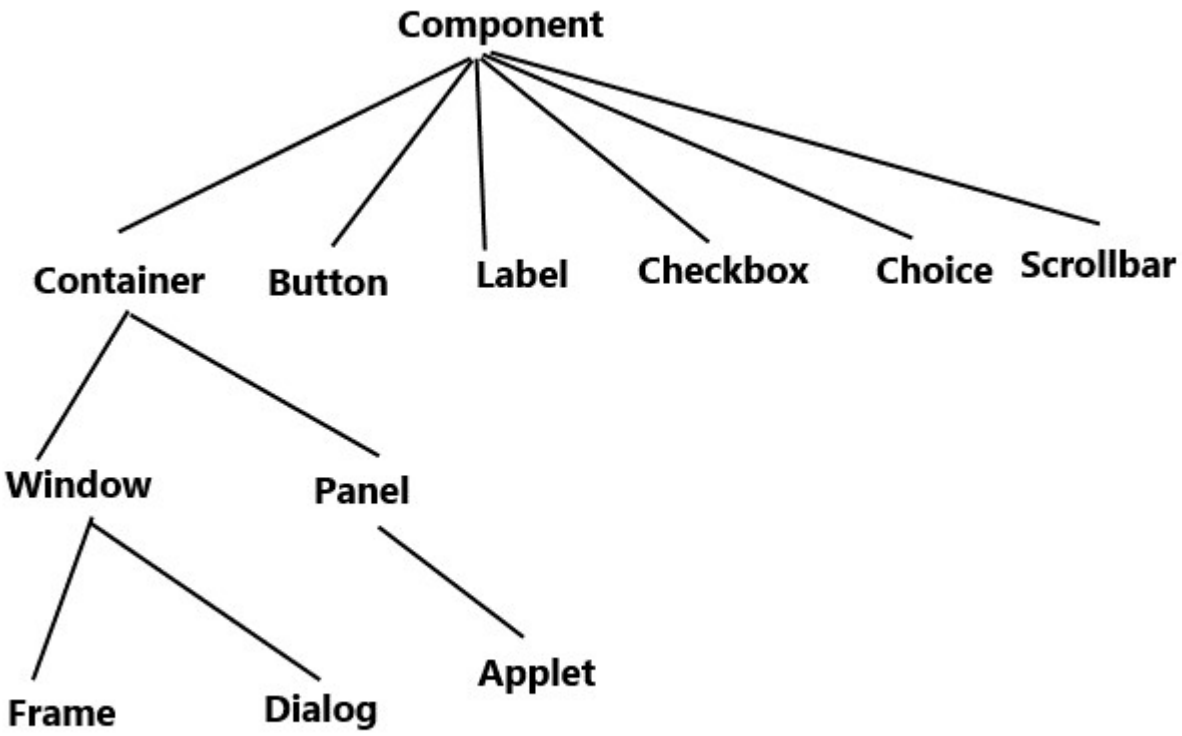


Introduction to AWT in Java

Java AWT is also known as Abstract Window Toolkit is an API that is used to develop either GUI or window-based applications in Java. Java AWT components are platform-dependent which implies that they are displayed according to the view of the operating system. It is also heavyweight implying that its components are using the resources of the Operating System. java. awt package provides classes for AWT api. For example, TextField, CheckBox, Choice, Label, TextArea, Radio Button, List, etc.

AWT hierarchy

Following is AWT hierarchy:



Container

The Container is one of the components in AWT that contains other components like buttons, text fields, labels, etc. The classes that extend the Container class are known as containers such as Frame, Dialog, and Panel as shown in the hierarchy.

Types of containers

As demonstrated above, container refers to the location where components can be added like text field, button, checkbox, etc. There are in total, four types of containers available in AW, that is, Window, Frame, Dialog, and Panel. As shown in the hierarchy above, Frame and Dialog are subclasses of the Window class.

1. Window: The window is a container that does not have borders and menu bars. In order to create a window, you can use frame, dialog or **another window**.

2. Panel: The Panel is the container/class that doesn't contain the title bar and menu bars. It has other components like buttons, text fields, etc.

3. Dialog: The Dialog is the container or class having a border and title. We cannot create an instance of the Dialog class without an associated instance of the respective Frame class.

4. Trim: The Frame is the container or class containing the title bar and might also have menu bars. It can also have other components like text field, button, etc.

Why AWT is platform dependent?

Java Abstract Window Toolkit calls native platform I.e., Operating system's subroutine in order to create components like text box, checkbox, button, etc. For example, an AWT GUI containing a button would be having varied look- and -feel in various platforms like Windows, Mac OS, and Unix, etc. since these platforms have different look and feel for their respective native buttons and then AWT would directly call their native subroutine that is going to create the button. In simple words, an application build on AWT would look more like a windows application when being run on Windows, however, that same application would look like a Mac application when being run on Mac Operating System.

Basic Methods of Component Class

- **public void add(Component c):** This method would insert a component on this component.
- **public void setSize(int width, int height):** This method would set the size (width and height) of the particular component.
- **public void setVisible(boolean status):** This method would change the visibility of the component, which is by default false.
- **public void setLayout(LayoutManager m):** This method would define the layout manager for the particular component.

Java AWT Example

We can create a GUI using Frame in two ways:

Let's show this by both examples, first extending Frame Class :

```
import java.awt.*;/* Extend the Frame class here,
*thus our class "Example" would behave
*like a Frame
*/public class Example extends Frame
{Example()
{Button b=new Button("Button!!");
//setting button position on screen
b.setBounds(50,50,50,50);
//adding button into frame
```

```
add(b);
//Setting  width and height
setSize(500,300);
//Setting title of Frame
setTitle("This is First AWT example");
//Setting the layout for the Frame
setLayout(new FlowLayout());
/*By default frame is not visible so
*we are setting the visibility to true
*to make it visible.
*/
setVisible(true);
}
public static void main(String args[]){
//Creating the instance of Frame
Example fr=new Example();
}
}
```

Example:

```
import java.awt.*;
public class Example {
Example()
{
//Creating Frame
Frame f=new Frame();
//Creating a label
Label l = new Label("User: ");
//adding label to the frame
```

```
f.add(l);
//Creating Text Field
TextField t = new TextField();
//adding text field to the frame
f.add(t);
//setting frame size
f.setSize(500, 300);
//Setting the layout for the Frame
f.setLayout(new FlowLayout());
f.setVisible(true);
}
public static void main(String args[])
{Example ex = new Example();
}
}
```

Layouts in AWT

There are 2 layouts in AWT which are as follows :

Flow layout is the default layout, which implies when you don't set any layout in your code then the particular layout would be set to Flow by default. Flow layout would put components like text fields, buttons, labels, etc in a row form and if horizontal space is not long enough to hold all components then it would add them in the next row and cycle goes on. Few points about Flow Layout

- All the rows in Flow layout are aligned center by default. But, if required we can set the alignment from left or right.
- The horizontal and vertical gap between all components is 5 pixels by default.
- By default, the orientation of the components is left to right, which implies that the components would be added from left to right as required, but we can change it from right to left when needed.

Border layout wherein we can add components like text fields, buttons, labels, etc to specific five These regions are known as PAGE_START, LINE_START, CENTER, LINE_END, PAGE_END.

Method for border layout is:

```
public BorderLayout(int hgap,int vgap)
```

It would construct a border layout with the gaps specified between components. The horizontal gap is specified by hgap and the vertical gap is specified by vgap.
Parameters are :

- **hgap:** The horizontal gap.
- **vgap:** The vertical gap.

We can also achieve the same by using setHgap(int hgap) method for the horizontal gap between components and setVgap(int vgap) method for the vertical gap.

What is Java Swing?

Java Swing is a popular and powerful Graphical User Interface (GUI) toolkit that is used for developing desktop applications. It is a part of the Java Foundation Classes (JFC) and provides a rich set of components and layout managers for creating a variety of GUIs. Java Swing is platform-independent and can be used on any operating system that supports Java.

It provides a set of lightweight components that are not only easy to use but also customizable. Some of the commonly used components in Swing are buttons, text fields, labels, menus, and many more.

Java Swing provides a pluggable look and feels that allows developers to customize the GUI according to the user's preferences. It also provides a robust event-handling mechanism that allows developers to handle events generated by the graphical components.

Some of the commonly used layout managers in Java Swing are BorderLayout, FlowLayout, GridLayout, CardLayout, and BoxLayout. These layout managers allow developers to create complex and intuitive GUIs that are easy to use and navigate.

Features of Java Swing

Some of the notable features of Java Swing are:

1. **Platform Independence:** Platform independence is one of Java Swing's most remarkable features. It can run on any platform that supports Java. Thus, Swing-based applications can run on Windows, Mac, Linux, or any other Java-compatible operating system.

2. **Lightweight Components:** Java Swing provides a set of lightweight components that are easy to use and customizable. These components are designed to consume less memory and use less processing power, making Swing-based applications run efficiently.

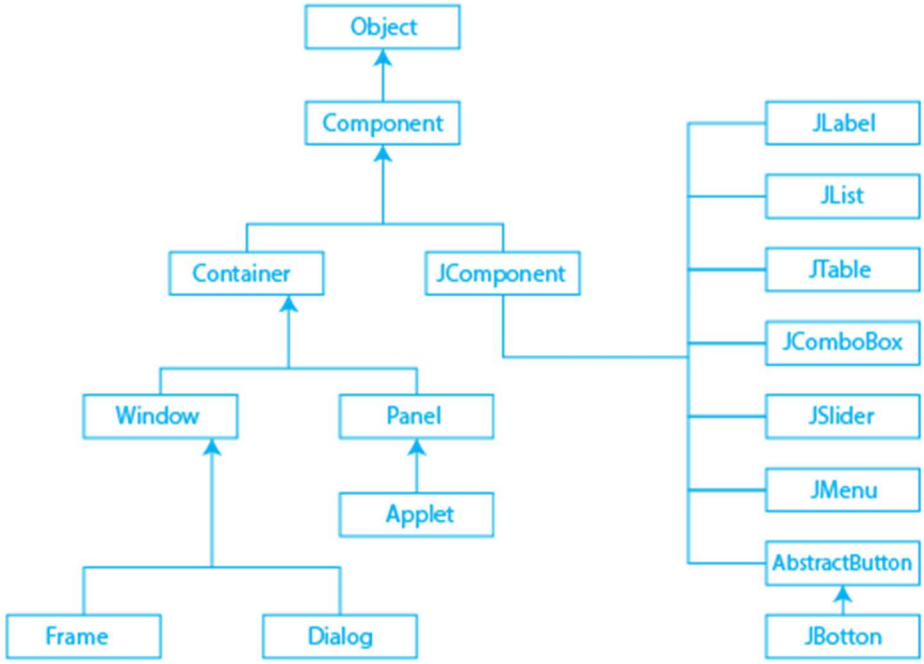
3. **Pluggable Look and Feel:** Java Swing provides a pluggable look and feels that allows developers to customize the appearance of the GUI according to the user's preferences. Developers can choose from several pre-built looks and feel themes or create their own custom themes.

4. **Layout Managers:** Java Swing provides a set of layout managers that can be used to organize the graphical components in a GUI. These layout managers enable developers to create flexible and responsive GUIs that adapt to different screen sizes and resolutions.

5. **Robust Event Handling Mechanism:** Java Swing provides a robust event handling mechanism that allows developers to handle events generated by the graphical components. Developers can register event listeners to detect and respond to user interactions with the GUI.

Java Swing Class Hierarchy

The Java Swing API hierarchy is shown below:



Java Swing Packages

Some of the commonly used packages in Java Swing are:

1. **javax.swing:** This package contains the core components of Swing, such as JButton, JLabel, JTable, JList, and many more. It also contains the classes for creating top-level containers such as JFrame and JDialog.

2. **javax.swing.event:** This package contains the classes for handling events generated by the Swing components. It includes event listener interfaces, event adapter classes, and event objects.

3. **javax.swing.border:** This package contains classes for creating borders around the Swing components. It includes the classes for creating line borders, etched borders, and titled borders.

4. **javax.swing.layout:** This package contains the classes for creating and managing layout managers in Swing. It includes the commonly used layout managers such as BorderLayout, FlowLayout, GridLayout, BoxLayout, and CardLayout.

5. **javax.swing.plaf:** This package contains the classes for the pluggable look and feels feature of Swing. It includes the classes for creating and managing the look and feel themes, and also provides the default look and feel theme for each platform.

6. **javax.swing.text:** This package contains the classes for creating and managing text components in Swing. It includes classes for creating text fields, text areas, and other text-related components.

7. **javax.swing.table:** This package contains the classes for creating and managing tables in Swing. It includes the classes for creating JTable, TableModel, TableColumn, and TableCellRenderer.

Components of Java Swing

Some of the important and common components of the Java Swing class are:

1. **JFrame:** JFrame is a top-level container that represents the main window of a GUI application. It provides a title bar, and minimizes, maximizes, and closes buttons.

2. **JPanel:** JPanel is a container that can hold other components. It is commonly used to group related components together.

3. **JButton:** JButton is a component that represents a clickable button. It is commonly used to trigger actions in a GUI application.

4. **JLabel:** JLabel is a component that displays text or an image. It is commonly used to provide information or to label other components.

5. **JTextField:** JTextField is a component that allows the user to input text. It is commonly used to get input from the user, such as a name or an address.

6. **JCheckBox:** JCheckBox is a component that represents a checkbox. It is commonly used to get a binary input from the user, such as whether or not to enable a feature.

7. **JList:** JList is a component that represents a list of elements. It is typically used to display a list of options from which the user can select one or more items.

8. **JTable:** JTable is a component that represents a data table. It is typically used to present data in a tabular fashion, such as a list of products or a list of orders.

9. **JScrollPane:** JScrollPane is a component that provides scrolling functionality to other components. It is commonly used to add scrolling to a panel or a table.

Difference between Java Swing and Java AWT

Here is a comparison of Java Swing and Java AWT:

Feature	**Java Swing**	**Java AWT**
Architecture	Platform-Independent	Platform-Dependent

Feature	Java Swing	Java AWT
Look and Feel	Pluggable look and feel	Native look and feel
Components	Richer set of components	Basic set of components
Performance	Slower due to software rendering	Faster due to native OS rendering
Event Model	More flexible and powerful	Simpler and less powerful
Thread Safety	By default, it is not thread-safe	Thread-safe by default
Customization	Highly customizable	Less customizable
Layout Managers	More layout managers are available	Fewer layout managers are available
API	Extensive API with many features	Basic API with fewer features
Graphics Support	It supports more advanced graphics	It only supports basic graphics

Feature	Java Swing	Java AWT
File Size	Size is large due to additional APIs	Size is small due to fewer APIs and classes

Advantages of Java Swing

Java Swing provides a number of advantages for developing graphical user interfaces (GUIs) in Java. Some of the key advantages of Java Swing are

1. **Platform Independence:** Swing is written entirely in Java, which makes it platform-independent. It can run on any platform that supports Java, without any modification.

2. **Look and Feel:** Java Swing provides a pluggable look and feels feature, which allows developers to customize the appearance of the components. It provides a consistent look and feels across platforms, which helps in creating a professional-looking GUI.

3. **Rich Component Set:** Java Swing provides a rich set of components, including advanced components like JTree, JTable, and JSpinner. It also provides support for multimedia components, such as audio and video.

4. **Layout Managers:** Java Swing provides a variety of layout managers, which makes it easy to arrange the components on a GUI. The layout managers help in creating GUIs that are visually appealing and easy to use.

5. **Event Handling:** Java Swing provides a powerful and flexible event handling model, which makes it easy to handle user events such as mouse clicks and keyboard presses. The event-handling model makes it easy to add interactivity to the GUI.

6. **Customizable:** Java Swing components are highly customizable, which makes it easy to create GUIs that meet the specific needs of an application. The components can be easily modified to suit the look and feel of the application.

Disadvantages of Java Swing

Some of the main disadvantages of Java Swing are:

1. **Performance:** Java Swing applications can be slower than native applications because of the overhead of running the Java Virtual Machine (JVM). This can be particularly noticeable in complex applications with large amounts of data.

2. **Look and Feel:** While Swing's pluggable look and feel feature allows for customization of the components, it can be difficult to achieve a truly native look and feel. This can make Swing applications look and feel different from other native applications on the same platform, which can be confusing for users.

3. **Learning Curve:** While Swing is easy to learn for developers who are already familiar with Java, it can be difficult for beginners who are not familiar with the language. The complex hierarchy of components and layout managers can also make it difficult to create complex GUIs.

4. **Resource Consumption:** Java Swing applications often requires a significant amount of system resources, such as memory and computing power. This can be an issue for low-end devices with limited resources or large-scale apps with a big number of users.

5. **Lack of Mobile Support:** Since Java Swing is a desktop-oriented GUI toolkit, it does not support mobile devices well. This could pose a potential challenge for developers who want to create cross-platform apps that work on both desktop and mobile platforms.

The **Model-View-Controller (MVC)** software design pattern is a method for separating concerns within a software application. In principle, the application logic, or controller, is separated from the technology used to display information to the user, or the view layer. The model is a communications vehicle between the controller and view layers.

As the name implies, the MVC pattern has three layers: The *Model* defines the business layer of the application, the *Controller* manages the flow of the application, and the *View* defines the presentation layer of the application.

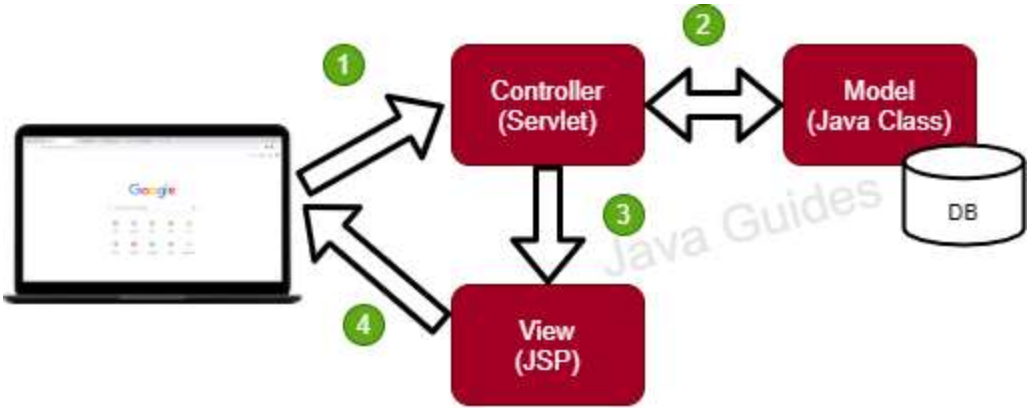
In this quick article, we'll create a small web application that implements the *Model View Controller (MVC)* design pattern, using basic Servlets and JSPs.

Get the source code of this tutorial on my **GitHub Repository**.

Key features of the MVC pattern:

1. It separates the presentation layer from the business layer
2. The **Controller** performs the action of invoking the **Model** and sending data to **View**
3. The **Model** is not even aware that it is used by some web application or a desktop application

Model-View-Controller (MVC)



1. **The Model Layer** - This is the data layer that contains the business logic of the system, and also represents the state of the application. It's independent of the presentation layer, the controller fetches the data from the Model layer and sends it to the View layer.
2. **The Controller Layer** - The controller layer acts as an interface between **View** and **Model**. It receives requests from the **View** layer and processes them, including the necessary validations.
3. **The View Layer** - This layer represents the output of the application, usually some form of UI. The presentation layer is used to display the **Model** data fetched by the **Controller**.

Java MVC Web Application using JSP and Servlet Example

To implement a web application based on the MVC design pattern, we'll create the *Employee* and *EmployeeService* classes – which will act as our **Model** layer.

EmployeeServlet class will act as a **Controller**, and for the presentation layer, we'll create an *employees.jsp* page.

Model Layer

Let's create the *Employee* and *EmployeeService* classes which will act as our **Model** layer.

Employee

```
package net.javaguides.mvc.model;


public class Employee {
   private int id;
   private String firstName;
   private String lastName;
   public Employee(int id, String firstName, String lastName) {
      super();
      this.id = id;
      this.firstName = firstName;
      this.lastName = lastName;
   }
   public int getId() {
      return id;
   }
```

```
    public void setId(int id) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

EmployeeService

```
package net.javaguides.mvc.service;

import java.util.Arrays;
import java.util.List;

import net.javaguides.mvc.model.Employee;

public class EmployeeService {

    public List < Employee > getEmployees() {
        return Arrays.asList(new Employee(1, "Ramesh", "Fadatare"), new Employee(2, "Tony", "Stark"),
            new Employee(3, "Tom", "Cruise"));
```

```
    }
}
```

Controller Layer

Now let's create our **Controller** class *EmployeeServlet*:

```
package net.javaguides.mvc.controller;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import net.javaguides.mvc.service.EmployeeService;

@WebServlet(name = "EmployeeServlet", urlPatterns = "/employees")

public class EmployeeServlet extends HttpServlet {

    private static final long serialVersionUID = 1 L;

    private EmployeeService employeeService = null;

    public void init() {
        employeeService = new EmployeeService();
    }
```

```
    private void processRequest(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        request.setAttribute("employees", employeeService.getEmployees());
        RequestDispatcher dispatcher = request.getRequestDispatcher("/WEB-INF/jsp/employees.jsp");
        dispatcher.forward(request, response);
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        processRequest(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        processRequest(request, response);
    }
}
```

View Layer

Next, let's write our presentation layer *employees.jsp*:

```
<%@page import="java.util.List"%>
<%@page import="net.javaguides.mvc.model.Employee"%>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
 pageEncoding="ISO-8859-1"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Student Record</title>
</head>
<body>
 <% List<Employee> employees = (List<Employee>)request.getAttribute("employees"); %>
 <table border="1" style="width: 50%" height="50%">
  <thead>
   <tr>
    <th>ID</th>
    <th>First Name</th>
    <th>Last Name</th>
   </tr>
  </thead>
  <tbody>
   <!--   for (Todo todo: todos) {  -->
   <% for(Employee employee : employees){ %>
   <tr>
    <td><%=employee.getId()%></td>
    <td><%=employee.getFirstName()%></td>
    <td><%=employee.getLastName()%></td>
   </tr>
   <%} %>
  </tbody>

 </table>
</body>
</html>
```

A button is a Swing component in Java that is usually used to register some action from a user. The action comes in the form of a button being clicked. To use a button in an application or as part of a graphical user interface (GUI), developers need to create an instance of the **JButton** class. **JButton** is a class that inherits from **JComponent**. Therefore, you can apply the **JComponent** features, such as layout and key bindings, on your buttons.

In this tutorial, programmers will learn how to work with buttons in Java.

Before we begin, have you ever considered taking an online course to learn Java software development? We have a great list of the Top Online Courses to Learn Java to help get you started.

How to Use the JButton Class in Java

To create a button, simply instantiate the **JButton** class in your Java code like so:

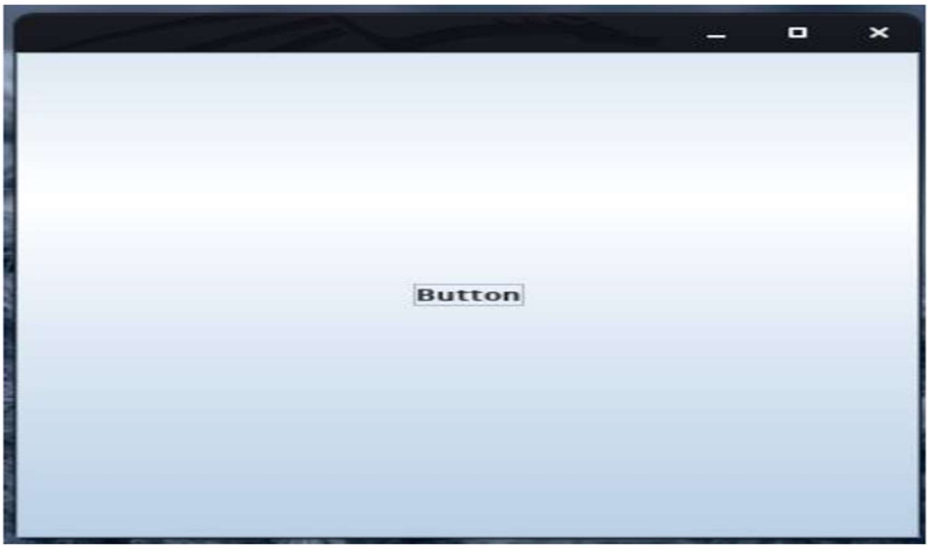
```
JButton button = new JButton("Button");
```

Programmers can supply a **string** (or icon) to the constructor of **JButton** as an identifier on the screen. Since **JButton** is a **JComponent**, you need to add it to a top level container, such as **JFrame**, **JDialog**, or **JApplet** in order for it to appear on screen.

The Java code example below uses the **JFrame** container:

```
import javax.swing.*;


class SimpleButton{


  public static void main(String args[]){


    JFrame frame = new JFrame();

    JButton button = new JButton("Button");


    frame.add(button);

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    frame.setSize(400,400);

    frame.setLocationRelativeTo(null);

    frame.setVisible(true);

  }

}
```

You should be able to see a button displayed on your screen when you run this code in your integrated development environment (IDE) or code editor:



It is important for developers to note that, when you run the code above, you may not get a similar display. This is because Swing components, by default, take on the look and feel of your application's environment.

The example code shown does not achieve anything when you click or press the button. In practice, buttons are used to perform some action when a certain event on them occurs (i.e when *pressed*). This is referred to as *listening* for an *event*. The next section discusses how to listen for button events in Java.

How to Listen for Events on Buttons in Java

There are three steps programmers need to follow in order to listen for an event on a button. First, you need to implement the **ActionListener** interface on your event handling class. You could also extend a class that implements **ActionListener** instead. Here is how that looks in Java code:

```
class EventClass implements ActionListener {

//some code here

}
```

Second, you need to add an instance of the event handler as an action listener to one or more components using the **addActionListener()** method:

```
GuiComponent.addActionListener(EventClassInstance);
```

The final step is to provide an implementation of the **actionPerformed(ActionEvent e)** method, which performs some action whenever an event is registered on a component. This method is the only method in the **ActionListener** interface and it is always called when an action is performed.

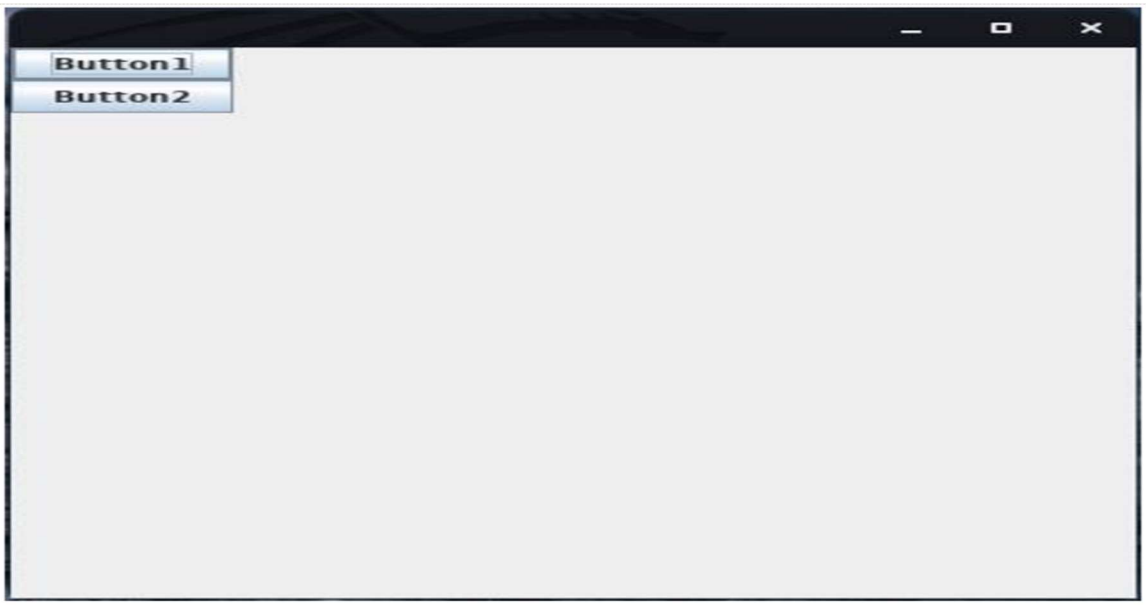
Java Code Example for Button Click Events

The Java code example below displays the number of clicks a user has so far made when they click **Button1**:

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


class ClicksCount implements ActionListener{


  int count = 0;// store number of clicks


  ClicksCount(){
     JFrame frame = new JFrame();
     JButton button1 = new JButton("Button1");
     JButton button2 = new JButton("Button2");
     button1.addActionListener(this);


     frame.setLayout(new BoxLayout(frame.getContentPane(), BoxLayout.Y_AXIS));
     frame.add(button1);
     frame.add(button2);
     frame.getRootPane().setDefaultButton(button1); // sets default button
     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     frame.setSize(450,450);
     frame.setLocationRelativeTo(null);
     frame.setVisible(true);
  }


  public void actionPerformed(ActionEvent e) {
     count++;
```

```
        System.out.println("You have clicked the ACTIVE button " + count + " times");
    }

    public static void main(String args[]){

        ClicksCount Clicks = new ClicksCount();
    }
}
```

When you compile and run the code above, you should see two buttons. If you take good notice, you will observe that **Button1** has been highlighted. This is because it has been set as the *default* button:



The *default* button is the button that initially appears to have the focus when the program is first run. When you press **Enter** on your keyboard, the program clicks this button since it was already selected by default. Pressing **Tab** will shift focus to the other button.

You can only have, at most, one default button, and you set it by calling the **setDefaultButton()** method on the root pane of a top-level container.

If you click **Button2** in this example, you will notice that there is not a message displayed. This is because no event handler has been registered to listen for events on this button. In other words, you would have to use the **addActionListener()** method with **Button2** to ensure that **actionPerformed(ActionEvent e)** is called when it is clicked.

Final Thoughts on Buttons and Events in Java

Since you are dealing with Swing components when using buttons and **JButtons**, remember to import the **javax.swing** library into your Java code. Also, in order to use an event listener, you need to add the **java.awt** library, as shown in the last code example. If you do not include these libraries, you will get a compilation error.

A Visual Guide to Layout Managers

Several AWT and Swing classes provide layout managers for general use:

- BorderLayout
- BoxLayout
- CardLayout
- FlowLayout
- GridBagLayout
- GridLayout
- GroupLayout
- SpringLayout

This section shows example GUIs that use these layout managers, and tells you where to find the how-to page for each layout manager. You can find links for running the examples in the how-to pages and in the example index.

Note: This lesson covers writing layout code by hand, which can be challenging. If you are not interested in learning all the details of layout management, you might prefer to use the GroupLayout layout manager combined with a builder tool to lay out your GUI. One such builder tool is the NetBeans IDE. Otherwise, if you want to code by hand and do not want to use GroupLayout, then GridBagLayout is recommended as the next most flexible and powerful layout manager.

If you are interested in using JavaFX to create your GUI, see Working With Layouts in JavaFX.

BorderLayout

Every content pane is initialized to use a BorderLayout. (As Using Top-Level Containers explains, the content pane is the main container in all frames, applets, and dialogs.) A BorderLayout places components in up to five areas: top, bottom, left, right, and center. All extra space is placed in the center area. Tool bars that are created using JToolBar must be created within a BorderLayout container, if you want to be able to drag and drop the bars away from their starting positions. For further details, see How to Use BorderLayout.

BoxLayout

The BoxLayout class puts components in a single row or column. It respects the components' requested maximum sizes and also lets you align components. For further details, see How to Use BoxLayout.

CardLayout

The CardLayout class lets you implement an area that contains different components at different times. A CardLayout is often controlled by a combo box, with the state of the combo box determining which panel (group of components) the CardLayout displays. An alternative to using CardLayout is using a tabbed pane, which provides similar functionality but with a pre-defined GUI. For further details, see How to Use CardLayout.

FlowLayout

FlowLayout is the default layout manager for every JPanel. It simply lays out components in a single row, starting a new row if its container is not sufficiently wide. Both panels in CardLayoutDemo, shown previously, use FlowLayout. For further details, see How to Use FlowLayout.

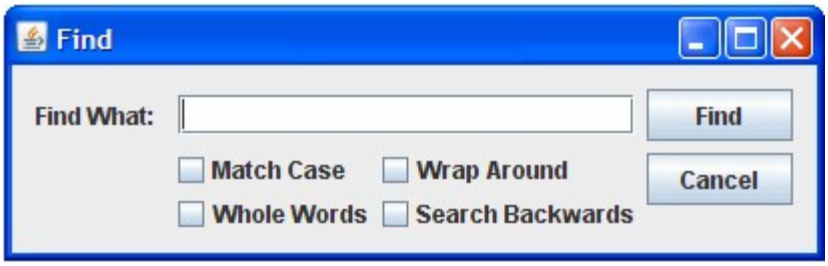
GridBagLayout

GridBagLayout is a sophisticated, flexible layout manager. It aligns components by placing them within a grid of cells, allowing components to span more than one cell. The rows in the grid can have different heights, and grid columns can have different widths. For further details, see How to Use GridBagLayout.

GridLayout

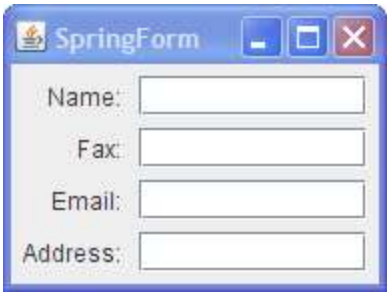
GridLayout simply makes a bunch of components equal in size and displays them in the requested number of rows and columns. For further details, see How to Use GridLayout.

GroupLayout



GroupLayout is a layout manager that was developed for use by GUI builder tools, but it can also be used manually. GroupLayout works with the horizontal and vertical layouts separately. The layout is defined for each dimension independently. Consequently, however, each component needs to be defined twice in the layout. The Find window shown above is an example of a GroupLayout. For further details, see How to Use GroupLayout.

SpringLayout



SpringLayout is a flexible layout manager designed for use by GUI builders. It lets you specify precise relationships between the edges of components under its control. For example, you might define that the left edge of one component is a certain distance (which can be dynamically calculated) from the right edge of a second component. SpringLayout lays out the children of its associated container according to a set of constraints, as shall be seen in How to Use SpringLayout.

What are Swing Components in Java?

A component is independent visual control, and Java Swing Framework contains a large set of these components, providing rich functionalities and allowing high customization. They all are derived from JComponent class. All these components are lightweight components. This class offers some standard functionality like pluggable look and feel, support for accessibility, drag and drop, layout, etc.

A container holds a group of components. It delivers a space where a component can be managed and displayed. Containers are of two types:

Top-level Containers	It inherits the Component and Container of AWT.
	We cannot contain it within other containers.
	Heavyweight.
	Example: JFrame, JDialog, JApplet
Lightweight Containers	It inherits JComponent class.
	It is a general-purpose container.
	We can use it to organize related components together.

	Example: JPanel

Top Swing components in Java are as follows

JButton

We use JButton class to create a push button on the UI. The button can include some display text or images. It yields an event when clicked and double-clicked. We can implement a JButton in the application by calling one of its constructors.

Syntax:

JButton okBtn = new JButton("Click");

This constructor returns a button with the text Click on it.

JButton homeBtn = new JButton(carIcon);

Returns a button with a car Icon on it.

JButton homeBtn2 = new JButton(carIcon, "Car");

Returns a button with the car icon and text as Car.

JLabel

We use JLabel class to render a read-only text label or images on the UI. It does not generate any event.

Syntax:

JLabel textLabel = new JLabel("This is 1st L...");

This constructor returns a label with specified text.

JLabel imgLabel = new JLabel(carIcon);

It returns a label with a car icon.

The JLabel Contains four constructors. They are as follows:

1. JLabel()
2. JLabel(String s)
3. JLabel(Icon i)
4. JLabel(String s, Icon i, int horizontalAlignment)

JTextField

The JTextField renders an editable single-line text box. Users can input non-formatted text in the box. We can initialize the text field by calling its constructor and passing an optional integer parameter. This parameter sets the box width measured by the number of columns. Also, it does not limit the number of characters that can be input into the box.

Syntax:

JTextField txtBox = new JTextField(50);

It is the most widely used text component. It has three constructors:

1. JTextField(int cols)
2. JTextField(String str, int cols)
3. JTextField(String str)

Note: cols represent the number of columns in the text field.

JCheckBox

The JCheckBox renders a check-box with a label. The check-box has two states, i.e., on and off. On selecting, the state is set to "on," and a small tick is displayed inside the box.

Syntax:

CheckBox chkBox = new JCheckBox(“Java Swing”, true);

It returns a checkbox with the label Pepperoni pizza. Notice the second parameter in the constructor. It is a boolean value that denotes the default state of the check-box. True means the check-box defaults to the "on" state.

JRadioButton

A radio button is a group of related buttons from which we can select only one. We use JRadioButton class to create a radio button in Frames and render a group of radio buttons in the UI. Users can select one choice from the group.

Syntax:

JRadioButton jrb = new JRadioButton("Easy");

JComboBox

The combo box is a combination of text fields and a drop-down list. We use JComboBox component to create a combo box in Swing.

Syntax:

JComboBox jcb = new JComboBox(name);

JTextArea

In Java, the Swing toolkit contains a JTextArea Class. It is under package javax.swing.JTextArea class. It is used for displaying multiple-line text.

Declaration:

public class JTextArea extends JTextComponent

Syntax:

JTextArea textArea_area=new JTextArea("Ninja! please write something in the text area.");

The JTextArea Contains four constructors. They are as follows:

1. JTextArea()
2. JTextArea(String s)
3. JTextArea(int row, int column)
4. JTextArea(String s, int row, int column)

JPasswordField

In Java, the Swing toolkit contains a JPasswordField Class. It is under package javax.swing.JPasswordField class. It is specifically used for the password, and we can edit them.

Declaration:

public class JPasswordField extends JTextField

Syntax:

JPasswordField password = new JPasswordField();

The JPasswordFieldContains 4 constructors. They are as follows:

1. JPasswordField()
2. JPasswordField(int columns)
3. JPasswordField(String text)
4. JPasswordField(String text, int columns)

JTable

In Java, the Swing toolkit contains a JTable Class. It is under package javax.swing.JTable class. It is used to draw a table to display data.

Syntax:

JTable table = new JTable(table_data, table_column);

The JTable contains two constructors. They are as follows:

1. JTable()
2. JTable(Object[][] rows, Object[] columns)

JList

In Java, the Swing toolkit contains a JList Class. It is under package javax.swing.JList class. It is used to represent a list of items together. We can select one or more than one items from the list.

Declaration:

public class JList extends JComponent implements Scrollable, Accessible

Syntax:

```
DefaultListModel<String> list1 = new DefaultListModel<>();
list1.addElement("Apple");
list1.addElement("Orange");
list1.addElement("Banan");
list1.addElement("Grape");
JList<String> list_1 = new JList<>(list1);
```

The JListContains 3 constructors. They are as follows:

1. JList()
2. JList(ary[] listData)
3. JList(ListModel<ary>dataModel)

JOptionPane

In Java, the Swing toolkit contains a JOptionPane Class. It is under package javax.swing.JOptionPane class. It is used for creating dialog boxes for displaying a message, confirm box, or input dialog box.

Declaration:

public class JOptionPane extends JComponent implements Accessible

Syntax:

```
JOptionPane.showMessageDialog(jframe_obj, "Good Morning, Evening & Night.");
```

The JOptionPaneContains 3 constructors. They are as following:

1. JOptionPane()
2. JOptionPane(Object message)
3. JOptionPane(Objectmessage,intmessageType)

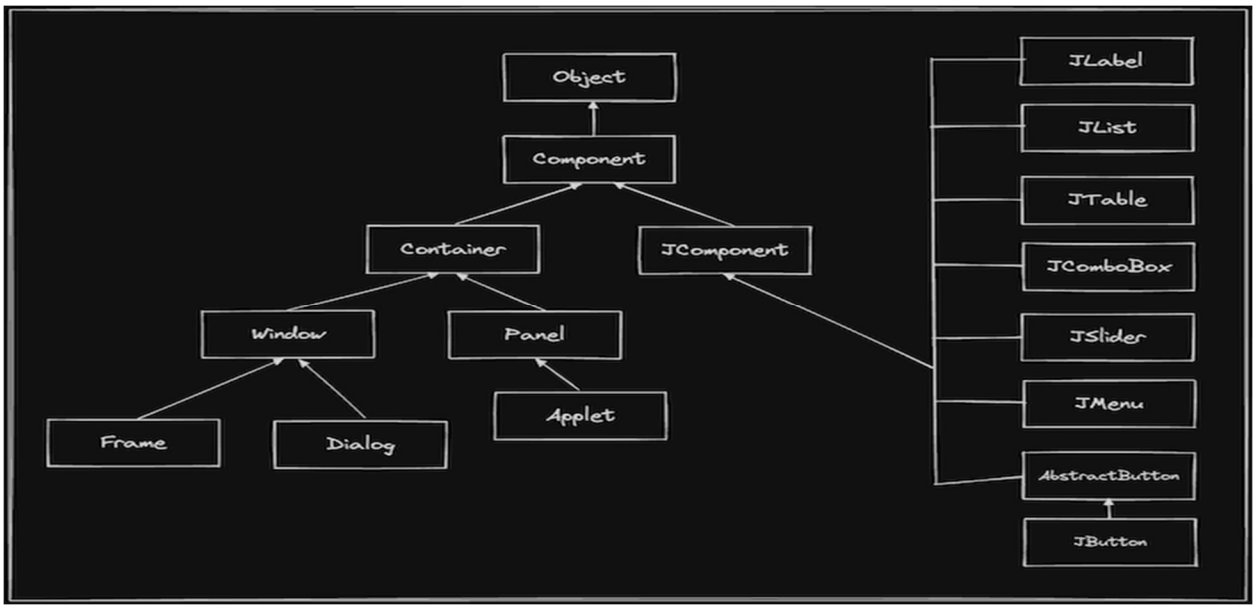
Difference between AWT and Swing

AWT (Abstract Window Toolkit) and Swing are both Java GUI (Graphical User Interface) toolkits that provide a set of classes and components to create desktop applications. The main differences between AWT and Swing are listed below:

Feature	AWT	Swing
Look and feel	Native	Cross-platform (customizable)
Performance	Faster (lightweight)	Slower (heavier)
Accessibility	Limited	More accessible
Component set	Limited	More comprehensive
Layout Managers	Limited	More comprehensive
Consistency across OS	Less consistent	More consistent
Support for advanced GUI	Limited	More advanced
2D graphics rendering	No	Yes
Event handling mechanism	Less flexible	More flexible

Hierarchy of Java Swing classes

The following diagram shows the hierarchy of Java Swing Classes:-



Commonly used Methods of Component class

The following are some commonly used methods of the component class:-

- setVisible(boolean visible): This method makes a component visible or invisible
- setEnabled(boolean enabled): This method allows you to enable or disable a component
- setSize(int width, int height): This method sets the size of a component in pixels
- setLocation(int x, int y): This method sets a component's position on its parent container
- setBounds(int x, int y, int width, int height): This method combines setSize() and setLocation() to set both the size and position of the component in one call
- repaint(): This method requests a repaint of the component. It is usually called to trigger the re-rendering of a component when its appearance needs to change
- addMouseListener(MouseListener listener): This method is used to register a MouseListener to listen for mouse events on a component

What is the difference between Swing and AWT components?

Swing and AWT are the two toolkits for building interactive Graphical User Interfaces (GUI). The critical difference between Swing and AWT in Java is that AWT is Java's conventional platform-dependent graphics and user interface widget toolkit. In contrast, Swing is a GUI widget toolkit for Java, an AWT extension.

How many components of Swing are there?

Swing is a GUI toolkit for Java that offers a wide range of components for building user interfaces. There are numerous components in Swing, including buttons, text fields, menus, tables, scroll panes, JTree and more.

How to use Swing components in Java?

To use Swing components in Java, import classes from the javax.swing package, create an instance of the component, set its properties, add it to a container, and then display the container. Add event listeners to handle user interaction with the component.

What are the 3 types of Java Swing containers?

The three types of Java Swing containers are:

1) Top-level containers such as JFrame, JDialog, and JApplet,

2) Content pane containers such as JPanel and JLayeredPane; and

3) Menu containers such as JMenu, JMenuBar, and JMenuItem.