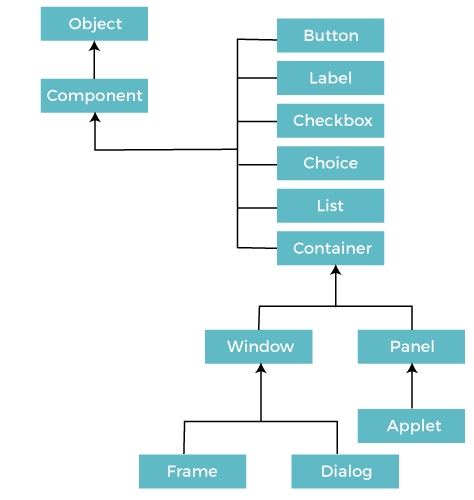
**Java AWT** (Abstract Window Toolkit) is an API to develop Graphical User Interface (GUI) or windows-based applications in Java.

Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavy weight i.e. its components are using the resources of underlying operating system (OS).

The java.awt [package](https://www.javatpoint.com/package) provides [classes](https://www.javatpoint.com/object-and-class-in-java) for AWT API such as [TextField](https://www.javatpoint.com/java-awt-textfield), [Label](https://www.javatpoint.com/java-awt-label), [TextArea](https://www.javatpoint.com/java-awt-textarea), RadioButton, [CheckBox](https://www.javatpoint.com/java-awt-checkbox), [Choice](https://www.javatpoint.com/java-awt-choice), [List](https://www.javatpoint.com/java-awt-list) etc.

The AWT tutorial will help the user to understand Java GUI programming in simple and easy steps.



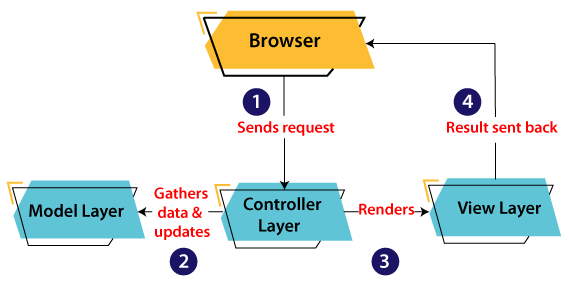
**MVC Architecture**

The Model-View-Controller (MVC) is a well-known [design pattern](https://www.javatpoint.com/design-patterns-in-java)

in the web development field. It is way to organize our code. It specifies that a program or application shall consist of data model, presentation information and control information. The MVC pattern needs all these components to be separated as different objects.

The MVC pattern architecture consists of three layers:

* **Model:** It represents the business layer of application. It is an object to carry the data that can also contain the logic to update controller if data is changed.
* **View:** It represents the presentation layer of application. It is used to visualize the data that the model contains.
* **Controller:** It works on both the model and view. It is used to manage the flow of application, i.e. data flow in the model object and to update the view whenever data is changed.



Components

All the elements like the button, text fields, scroll bars, etc. are called components. In Java AWT, there are classes for each component as shown in above diagram. In order to place every component in a particular position on a screen, we need to add them to a container.

Container

The Container is a component in AWT that can contain another components like [buttons](https://www.javatpoint.com/java-awt-button), textfields, labels etc. The classes that extends Container class are known as container such as **Frame, Dialog** and **Panel**.

It is basically a screen where the where the components are placed at their specific locations. Thus it contains and controls the layout of components.

There are four types of containers in Java AWT:

1. Window
2. Panel
3. Frame
4. Dialog

## **Useful Methods of Component Class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| public void add(Component c) | Inserts a component on this component. |
| public void setSize(int width,int height) | Sets the size (width and height) of the component. |
| public void setLayout(LayoutManager m) | Defines the layout manager for the component. |
| public void setVisible(boolean status) | Changes the visibility of the component, by default false. |

**Layout Manager**

The LayoutManagers are used to arrange components in a particular manner. The **Java LayoutManagers** facilitates us to control the positioning and size of the components in GUI forms. LayoutManager is an interface that is implemented by all the classes of layout managers.

**Flow Layout,**

The Java FlowLayout class is used to arrange the components in a line, one after another (in a flow). It is the default layout of the applet or panel.

### Fields of FlowLayout class

1. **public static final int LEFT**
2. **public static final int RIGHT**
3. **public static final int CENTER**
4. **public static final int LEADING**
5. **public static final int TRAILING**

### Constructors of FlowLayout class

1. **FlowLayout():** creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap.
2. **FlowLayout(int align):** creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.
3. **FlowLayout(int align, int hgap, int vgap):** creates a flow layout with the given alignment and the given horizontal and vertical gap.

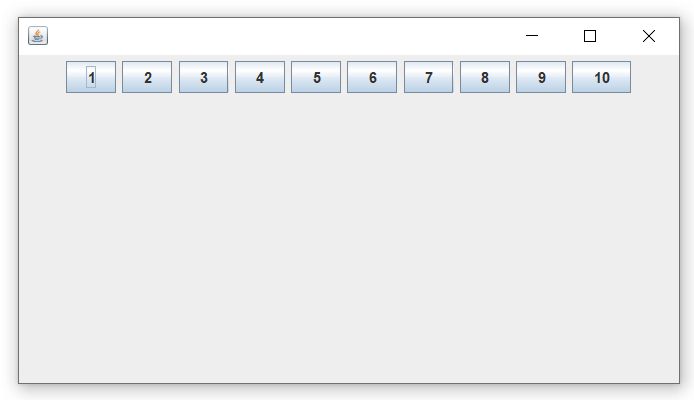
### Example of FlowLayout class: Using FlowLayout() constructor

**FileName:** FlowLayoutExample.java

1. // import statements
2. **import** java.awt.\*;
3. **import** javax.swing.\*;
5. **public** **class** FlowLayoutExample
6. {
8. JFrame frameObj;
10. // constructor
11. FlowLayoutExample()
12. {
13. // creating a frame object
14. frameObj = **new** JFrame();
16. // creating the buttons
17. JButton b1 = **new** JButton("1");
18. JButton b2 = **new** JButton("2");
19. JButton b3 = **new** JButton("3");
20. JButton b4 = **new** JButton("4");
21. JButton b5 = **new** JButton("5");
22. JButton b6 = **new** JButton("6");
23. JButton b7 = **new** JButton("7");
24. JButton b8 = **new** JButton("8");
25. JButton b9 = **new** JButton("9");
26. JButton b10 = **new** JButton("10");

29. // adding the buttons to frame
30. frameObj.add(b1); frameObj.add(b2); frameObj.add(b3); frameObj.add(b4);
31. frameObj.add(b5); frameObj.add(b6);  frameObj.add(b7);  frameObj.add(b8);
32. frameObj.add(b9);  frameObj.add(b10);
34. // parameter less constructor is used
35. // therefore, alignment is center
36. // horizontal as well as the vertical gap is 5 units.
37. frameObj.setLayout(**new** FlowLayout());
39. frameObj.setSize(300, 300);
40. frameObj.setVisible(**true**);
41. }
43. // main method
44. **public** **static** **void** main(String argvs[])
45. {
46. **new** FlowLayoutExample();
47. }
48. }

**Output:**



**Border Layout,**

The BorderLayout is used to arrange the components in five regions: north, south, east, west, and center. Each region (area) may contain one component only. It is the default layout of a frame or window. The BorderLayout provides five constants for each region:

1. **public static final int NORTH**
2. **public static final int SOUTH**
3. **public static final int EAST**
4. **public static final int WEST**
5. **public static final int CENTER**

Constructors of BorderLayout class:

* **BorderLayout():** creates a border layout but with no gaps between the components.
* **BorderLayout(int hgap, int vgap):** creates a border layout with the given horizontal and vertical gaps between the components.

Example of BorderLayout class: Using BorderLayout() constructor

**FileName:** Border.java

**import** java.awt.\*;

**import** javax.swing.\*;

**public** **class** Border

{

JFrame f;

Border()

{

    f = **new** JFrame();

     // creating buttons

    JButton b1 = **new** JButton("NORTH");; // the button will be labeled as NORH

    JButton b2 = **new** JButton("SOUTH");; // the button will be labeled as SOUH

   JButton b3 = **new** JButton("EAST");; // the button will be labeled as EAST

    JButton b4 = **new** JButton("WEST");; // the button will be labeled as WEST

    JButton b5 = **new** JButton("CENTER");; // the button will be labeled as CENTER

    f.add(b1, BorderLayout.NORTH); // b1 will be placed in the North Direction

f.add(b2, BorderLayout.SOUTH);  // b2 will be placed in the South Directi

    f.add(b3, BorderLayout.EAST);  // b2 will be placed in the East Direction

    f.add(b4, BorderLayout.WEST);  // b2 will be placed in the West Direction

    f.add(b5, BorderLayout.CENTER);  // b2 will be placed in the Center

    f.setSize(300, 300);

    f.setVisible(**true**);

}

**public** **static** **void** main(String[] args) {

**new** Border();

}

}

**Output:**

**Grid Layout,**

The Java GridLayout class is used to arrange the components in a rectangular grid. One component is displayed in each rectangle.

### Constructors of GridLayout class

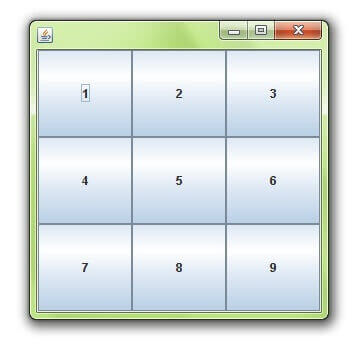
1. **GridLayout():** creates a grid layout with one column per component in a row.
2. **GridLayout(int rows, int columns):** creates a grid layout with the given rows and columns but no gaps between the components.
3. **GridLayout(int rows, int columns, int hgap, int vgap):** creates a grid layout with the given rows and columns along with given horizontal and vertical gaps.

Example of GridLayout class: Using GridLayout(int rows, int columns) Constructor

**FileName:** MyGridLayout.java

1. **import** java.awt.\*;
2. **import** javax.swing.\*;
3. **public** **class** MyGridLayout{
4. JFrame f;
5. MyGridLayout(){
6. f=**new** JFrame();
7. JButton b1=**new** JButton("1");
8. JButton b2=**new** JButton("2");
9. JButton b3=**new** JButton("3");
10. JButton b4=**new** JButton("4");
11. JButton b5=**new** JButton("5");
12. JButton b6=**new** JButton("6");
13. JButton b7=**new** JButton("7");
14. JButton b8=**new** JButton("8");
15. JButton b9=**new** JButton("9");
16. // adding buttons to the frame
17. f.add(b1); f.add(b2); f.add(b3);
18. f.add(b4); f.add(b5); f.add(b6);
19. f.add(b7); f.add(b8); f.add(b9);
21. // setting grid layout of 3 rows and 3 columns
22. f.setLayout(**new** GridLayout(3,3));
23. f.setSize(300,300);
24. f.setVisible(**true**);
25. }
26. **public** **static** **void** main(String[] args) {
27. **new** MyGridLayout();
28. }
29. }

**Output:**



**Card Layout,**

The **Java CardLayout** class manages the components in such a manner that only one component is visible at a time. It treats each component as a card that is why it is known as CardLayout.

### Constructors of CardLayout Class

1. **CardLayout():** creates a card layout with zero horizontal and vertical gap.
2. **CardLayout(int hgap, int vgap):** creates a card layout with the given horizontal and vertical gap.

### Commonly Used Methods of CardLayout Class

* **public void next(Container parent):** is used to flip to the next card of the given container.
* **public void previous(Container parent):** is used to flip to the previous card of the given container.
* **public void first(Container parent):** is used to flip to the first card of the given container.
* **public void last(Container parent):** is used to flip to the last card of the given container.
* **public void show(Container parent, String name):** is used to flip to the specified card with the given name.

### Example of CardLayout Class: Using Default Constructor

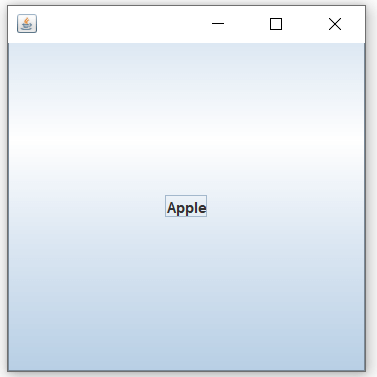
The following program uses the next() method to move to the next card of the container.

**FileName:** CardLayoutExample1.java

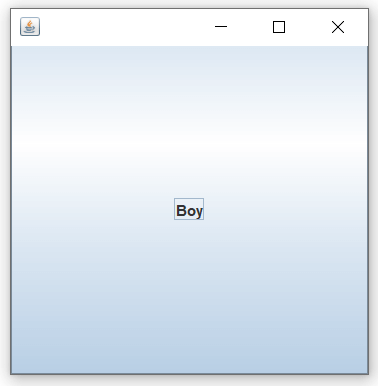
1. // import statements
2. **import** java.awt.\*;
3. **import** javax.swing.\*;
4. **import** java.awt.event.\*;
6. **public** **class** CardLayoutExample1 **extends** JFrame **implements** ActionListener
7. {
9. CardLayout crd;
11. // button variables to hold the references of buttons
12. JButton btn1, btn2, btn3;
13. Container cPane;
15. // constructor of the class
16. CardLayoutExample1()
17. {
19. cPane = getContentPane();
21. //default constructor used
22. // therefore, components will
23. // cover the whole area
24. crd = **new** CardLayout();
26. cPane.setLayout(crd);
28. // creating the buttons
29. btn1 = **new** JButton("Apple");
30. btn2 = **new** JButton("Boy");
31. btn3 = **new** JButton("Cat");
33. // adding listeners to it
34. btn1.addActionListener(**this**);
35. btn2.addActionListener(**this**);
36. btn3.addActionListener(**this**);
38. cPane.add("a", btn1); // first card is the button btn1
39. cPane.add("b", btn2); // first card is the button btn2
40. cPane.add("c", btn3);  // first card is the button btn3
42. }
43. **public** **void** actionPerformed(ActionEvent e)
44. {
45. // Upon clicking the button, the next card of the container is shown
46. // after the last card, again, the first card of the container is shown upon clicking
47. crd.next(cPane);
48. }
50. // main method
51. **public** **static** **void** main(String argvs[])
52. {
53. // creating an object of the class CardLayoutExample1
54. CardLayoutExample1 crdl = **new** CardLayoutExample1();
56. // size is 300 \* 300
57. crdl.setSize(300, 300);
58. crdl.setVisible(**true**);
59. crdl.setDefaultCloseOperation(EXIT\_ON\_CLOSE);
60. }
61. }

**Output**ASA Announces New Milestone in Mystery of Universe's Expansion Rate

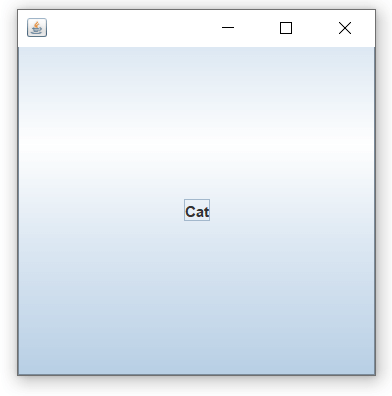
**Next**



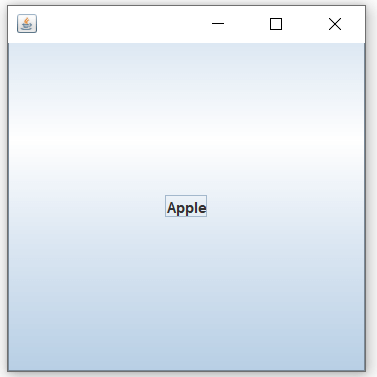
When the button named apple is clicked, we get



When the boy button is clicked, we get



Again, we reach the first card of the container if the cat button is clicked, and the cycle continues.



**Grid Bag Layout**

The Java GridBagLayout class is used to align components vertically, horizontally or along their baseline.

**GridBagLayout():** The parameterless constructor is used to create a grid bag layout manager.

1. **import** java.awt.Button;
2. **import** java.awt.GridBagConstraints;
3. **import** java.awt.GridBagLayout;
5. **import** javax.swing.\*;
6. **public** **class** GridBagLayoutExample **extends** JFrame{
7. **public** **static** **void** main(String[] args) {
8. GridBagLayoutExample a = **new** GridBagLayoutExample();
9. }
10. **public** GridBagLayoutExample() {
11. GridBagLayoutgrid = **new** GridBagLayout();
12. GridBagConstraints gbc = **new** GridBagConstraints();
13. setLayout(grid);
14. setTitle("GridBag Layout Example");
15. GridBagLayout layout = **new** GridBagLayout();
16. **this**.setLayout(layout);
17. gbc.fill = GridBagConstraints.HORIZONTAL;
18. gbc.gridx = 0;
19. gbc.gridy = 0;
20. **this**.add(**new** Button("Button One"), gbc);
21. gbc.gridx = 1;
22. gbc.gridy = 0;
23. **this**.add(**new** Button("Button two"), gbc);
24. gbc.fill = GridBagConstraints.HORIZONTAL;
25. gbc.ipady = 20;
26. gbc.gridx = 0;
27. gbc.gridy = 1;
28. **this**.add(**new** Button("Button Three"), gbc);
29. gbc.gridx = 1;
30. gbc.gridy = 1;
31. **this**.add(**new** Button("Button Four"), gbc);
32. gbc.gridx = 0;
33. gbc.gridy = 2;
34. gbc.fill = GridBagConstraints.HORIZONTAL;
35. gbc.gridwidth = 2;
36. **this**.add(**new** Button("Button Five"), gbc);
37. setSize(300, 300);
38. setPreferredSize(getSize());
39. setVisible(**true**);
40. setDefaultCloseOperation(EXIT\_ON\_CLOSE);
42. }
44. }

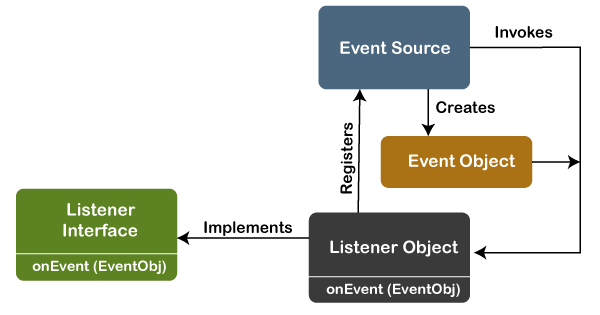
**Output:**



**Delegation Event Model in Java**

The Delegation Event model is defined to handle events in GUI [programming languages](https://www.javatpoint.com/programming-language). The [GUI](https://www.javatpoint.com/gui-full-form) stands for Graphical User Interface, where a user graphically/visually interacts with the system.

The GUI programming is inherently event-driven; whenever a user initiates an activity such as a mouse activity, clicks, scrolling, etc., each is known as an event that is mapped to a code to respond to functionality to the user. This is known as event handling.

. 

Java Program to Implement the Event Deligation Model

The below is a Java program to handle events implementing the event deligation model:

**TestApp.java:**

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** TestApp {

**public** **void** search() {

        // For searching

        System.out.println("Searching...");

    }

**public** **void** sort() {

        // for sorting

        System.out.println("Sorting....");

    }

**static** **public** **void** main(String args[]) {

       TestApp app = **new** TestApp();

       GUI gui = **new** GUI(app);

    }

}

**class** Command **implements** ActionListener  {

**static** **final** **int** SEARCH = 0;

**static** **final** **int** SORT = 1;

**int** id;

    TestApp app;

**public** Command(**int** id, TestApp app) {

**this**.id = id;

**this**.app = app;

    }

**public** **void** actionPerformed(ActionEvent e) {

**switch**(id) {

**case** SEARCH:

            app.search();

**break**;

**case** SORT:

            app.sort();

**break**;

        }

    }

}

**class** GUI {

**public** GUI(TestApp app) {

        Frame f = **new** Frame();

        f.setLayout(**new** FlowLayout());

        Command searchCmd = **new** Command(Command.SEARCH, app);

        Command sortCmd = **new** Command(Command.SORT, app);

        Button b;

        f.add(b = **new** Button("Search"));

        b.addActionListener(searchCmd);

        f.add(b = **new** Button("Sort"));

        b.addActionListener(sortCmd);

        List l;

        f.add(l = **new** List());

        l.add("Alphabetical");

        l.add("Chronological");

        l.addActionListener(sortCmd);

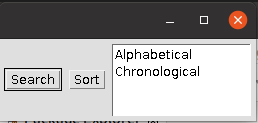
        f.pack();

        f.show();

    }

}

**Output:**



Searching...

## **Java AWT Example**

To create simple AWT example, you need a frame. There are two ways to create a GUI using Frame in AWT.

1. By extending Frame class (**inheritance**)
2. By creating the object of Frame class (**association**)

## **AWT Example by Inheritance**

Let's see a simple example of AWT where we are inheriting Frame class. Here, we are showing Button component on the Frame.

**import** java.awt.\*;

// extending Frame class to our class AWTExample1

**public** **class** AWTExample1 **extends** Frame {

   // initializing using constructor

   AWTExample1() {

      // creating a button

      Button b = **new** Button("Click Me!!");

      // setting button position on screen

      b.setBounds(30,100,80,30);

      // adding button into frame

      add(b);

      // frame size 300 width and 300 height

      setSize(300,300);

      // setting the title of Frame

      setTitle("This is our basic AWT example");

      // no layout manager

      setLayout(**null**);

      // now frame will be visible, by default it is not visible

      setVisible(**true**);

}

// main method

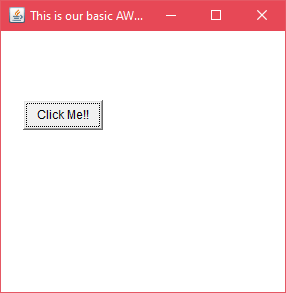
**public** **static** **void** main(String args[]) {

// creating instance of Frame class

AWTExample1 f = **new** AWTExample1();

}

}



AWT Example by Association

Let's see a simple example of AWT where we are creating instance of Frame class. Here, we are creating a TextField, Label and Button component on the Frame.

**AWTExample2.java**

// importing Java AWT class

**import** java.awt.\*;

// class AWTExample2 directly creates instance of Frame class

**class** AWTExample2

{      // initializing using constructor

    AWTExample2()

 {      // creating a Frame

       Frame f = **new** Frame();

        // creating a Label

       Label l = **new** Label("Employee id:");

        // creating a Button

       Button b = **new** Button("Submit");

        // creating a TextField

       TextField t = **new** TextField();

       // setting position of above components in the frame

       l.setBounds(20, 80, 80, 30);

       t.setBounds(20, 100, 80, 30);

       b.setBounds(100, 100, 80, 30);

        // adding components into frame

       f.add(b);

       f.add(l);

       f.add(t);

        // frame size 300 width and 300 height

       f.setSize(400,300);

        // setting the title of frame

       f.setTitle("Employee info");

       // no layout

       f.setLayout(**null**);

      // setting visibility of frame

      f.setVisible(**true**);

}

// main method

**public** **static** **void** main(String args[])

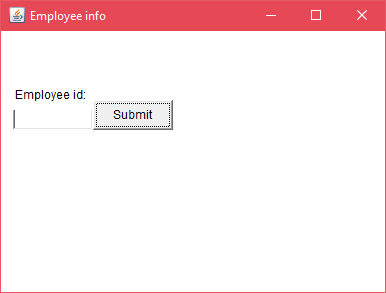
 {

// creating instance of Frame class

AWTExample2 awt\_obj = **new** AWTExample2();

}

}



# **Event and Listener (Java Event Handling)**

|  |
| --- |
| Changing the state of an object is known as an event. For example, click on button, dragging mouse etc. The java.awt.event package provides many event classes and Listener interfaces for event handling. |

## **Java Event classes and Listener interfaces**

|  |  |
| --- | --- |
| **Event Classes** | **Listener Interfaces** |
| ActionEvent | ActionListener |
| MouseEvent | MouseListener and MouseMotionListener |
| MouseWheelEvent | MouseWheelListener |
| KeyEvent | KeyListener |
| ItemEvent | ItemListener |
| TextEvent | TextListener |
| AdjustmentEvent | AdjustmentListener |
| WindowEvent | WindowListener |
| ComponentEvent | ComponentListener |
| ContainerEvent | ContainerListener |
| FocusEvent | FocusListener |

## **Steps to perform Event Handling**

Following steps are required to perform event handling:

1. Register the component with the Listener

## **Registration Methods**

For registering the component with the Listener, many classes provide the registration methods. For example:

* **Button**
  + public void addActionListener(ActionListener a){}
* **MenuItem**
  + public void addActionListener(ActionListener a){}
* **TextField**
  + public void addActionListener(ActionListener a){}
  + public void addTextListener(TextListener a){}
* **TextArea**
  + public void addTextListener(TextListener a){}
* **Checkbox**
  + public void addItemListener(ItemListener a){}
* **Choice**
  + public void addItemListener(ItemListener a){}
* **List**
  + public void addActionListener(ActionListener a){}
  + public void addItemListener(ItemListener a){}

### Java Event Handling Code

**import** java.awt.\*;

**import** java.awt.event.\*;

**class** AEvent **extends** Frame **implements** ActionListener{

TextField tf;

AEvent(){

//create components

tf=**new** TextField();

tf.setBounds(60,50,170,20);

Button b=**new** Button("click me");

b.setBounds(100,120,80,30);

//register listener

b.addActionListener(**this**);//passing current instance

//add components and set size, layout and visibility

add(b);

add(tf);

setSize(300,300);

setLayout(**null**);

setVisible(**true**);

}

**public** **void** actionPerformed(ActionEvent e){

tf.setText("Welcome");

}

**public** **static** **void** main(String args[]){

**new** AEvent();

}

}

**Handling mouse and keyboard events**

**Handling mouse**

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:

**public** **abstract** **void** mouseClicked(MouseEvent e);

**public** **abstract** **void** mouseEntered(MouseEvent e);

**public** **abstract** **void** mouseExited(MouseEvent e);

**public** **abstract** **void** mousePressed(MouseEvent e);

**public** **abstract** **void** mouseReleased(MouseEvent e);

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** MouseListenerExample **extends** Frame **implements** MouseListener{

    Label l;

    MouseListenerExample(){

        addMouseListener(**this**);

        l=**new** Label();

        l.setBounds(20,50,100,20);

        add(l);

        setSize(300,300);

        setLayout(**null**);

        setVisible(**true**);

    }

**public** **void** mouseClicked(MouseEvent e) {

        l.setText("Mouse Clicked");

    }

**public** **void** mouseEntered(MouseEvent e) {

        l.setText("Mouse Entered");

    }

**public** **void** mouseExited(MouseEvent e) {

        l.setText("Mouse Exited");

    }

**public** **void** mousePressed(MouseEvent e) {

        l.setText("Mouse Pressed");

    }

**public** **void** mouseReleased(MouseEvent e) {

        l.setText("Mouse Released");

    }

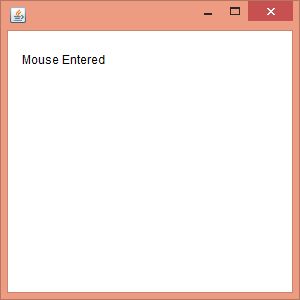
**public** **static** **void** main(String[] args) {

**new** MouseListenerExample();

}

}

Output:



**keyboard events**

The **Java KeyListener is notified whenever you change the state of key.** It is notified against KeyEvent. The KeyListener interface is found in java.awt.event package, and it has three methods.

Following is the declaration for **java.awt.event.KeyListener** interface:

1. **public** **interface** KeyListener **extends** EventListener

## **Methods of KeyListener interface**

The signature of 3 methods found in KeyListener interface are given below:

|  |  |  |
| --- | --- | --- |
| **Sr. no.** | **Method name** | **Description** |
| 1. | public abstract void keyPressed (KeyEvent e); | It is invoked when a key has been pressed. |
| 2. | public abstract void keyReleased (KeyEvent e); | It is invoked when a key has been released. |
| 3. | public abstract void keyTyped (KeyEvent e); | It is invoked when a key has been typed. |

// importing awt libraries

**import** java.awt.\*;

**import** java.awt.event.\*;

// class which inherits Frame class and implements KeyListener interface

**public** **class** KeyListenerExample **extends** Frame **implements** KeyListener {

// creating object of Label class   and TextArea class

 Label l;

    TextArea area;

// class constructor

    KeyListenerExample() {

          // creating the label

        l = **new** Label();

// setting the location of the label in frame

        l.setBounds (20, 50, 100, 20);

// creating the text area

        area = **new** TextArea();

// setting the location of text area

        area.setBounds (20, 80, 300, 300);

// adding the KeyListener to the text area

        area.addKeyListener(**this**);

// adding the label and text area to the frame

        add(l);

add(area);

// setting the size, layout and visibility of frame

        setSize (400, 400);

        setLayout (**null**);

        setVisible (**true**);

    }

// overriding the keyPressed() method of KeyListener interface where we set te text of the label when key is pressed

**public** **void** keyPressed (KeyEvent e) {

        l.setText ("Key Pressed");

    }

// overriding the keyReleased() method of KeyListener interface where we set the text of the label when key is released

**public** **void** keyReleased (KeyEvent e) {

        l.setText ("Key Released");

    }

// overriding the keyTyped() method of KeyListener interface where we set the text of the label when a key is typed

**public** **void** keyTyped (KeyEvent e) {

        l.setText ("Key Typed");

    }

  // main method

**public** **static** **void** main(String[] args) {

**new** KeyListenerExample();

    }

}

**Output:**



# **Java Adapter Classes**

Java adapter classes provide the default implementation of listener *[interfaces](https://www.javatpoint.com/interface-in-java)*

. If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces. So it saves code.

### Pros of using Adapter classes:

* It assists the unrelated classes to work combinedly.
* It provides ways to use classes in different ways.
* It increases the transparency of classes.
* It provides a way to include related patterns in the class.
* It provides a pluggable kit for developing an application.
* It increases the reusability of the class.

The adapter classes are found in **java.awt.event, java.awt.dnd** and **javax.swing.event** [packages](https://www.javatpoint.com/package)

. The Adapter classes with their corresponding listener interfaces are given below.

## **java.awt.event Adapter classes**

|  |  |
| --- | --- |
| **Adapter class** | **Listener interface** |
| WindowAdapter | [WindowListener](https://www.javatpoint.com/java-windowlistener) |
| KeyAdapter | [KeyListener](https://www.javatpoint.com/java-keylistener) |
| MouseAdapter | [MouseListener](https://www.javatpoint.com/java-mouselistener) |
| MouseMotionAdapter | [MouseMotionListener](https://www.javatpoint.com/java-mousemotionlistener) |
| FocusAdapter | FocusListener |
| ComponentAdapter | ComponentListener |
| ContainerAdapter | ContainerListener |
| HierarchyBoundsAdapter | HierarchyBoundsListener |

## **java.awt.dnd Adapter classes**

|  |  |
| --- | --- |
| **Adapter class** | **Listener interface** |
| DragSourceAdapter | DragSourceListener |
| DragTargetAdapter | DragTargetListener |

## **javax.swing.event Adapter classes**

|  |  |
| --- | --- |
| **Adapter class** | **Listener interface** |
| MouseInputAdapter | MouseInputListener |
| InternalFrameAdapter | InternalFrameListener |

## **Java WindowAdapter Example**

In the following example, we are implementing the WindowAdapter class of AWT and one its methods windowClosing() to close the frame window.

**AdapterExample.java**

51.9M

1K

Exception Handling in Java - Javatpoint

// 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

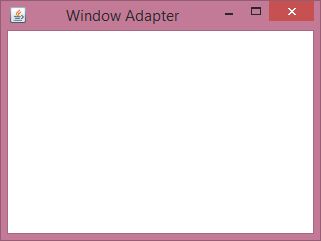
**public** **static** **void** main(String[] args) {

**new** AdapterExample();

}

}

**Output:**



## **Java MouseAdapter Example**

In the following example, we are implementing the MouseAdapter class. The MouseListener interface is added into the frame to listen the mouse event in the frame.

**MouseAdapterExample.java**

// importing the necessary libraries

**import** java.awt.\*;

**import** java.awt.event.\*;

// class which inherits the MouseAdapter class

**public** **class** MouseAdapterExample **extends** MouseAdapter {

// object of Frame class

    Frame f;

// class constructor

    MouseAdapterExample() {

// creating the frame with the title

        f = **new** Frame ("Mouse Adapter");

// adding MouseListener to the Frame

        f.addMouseListener(**this**);

   // setting the size, layout and visibility of the frame

        f.setSize (300, 300);

        f.setLayout (**null**);

        f.setVisible (**true**);

    }

// overriding the mouseClicked() method of the MouseAdapter class

**public** **void** mouseClicked (MouseEvent e) {

// creating the Graphics object and fetching them from the Frame object using getGaphics() method

Graphics g = f.getGraphics();

// setting the color of graphics object

        g.setColor (Color.BLUE);

// setting the shape of graphics object

        g.fillOval (e.getX(), e.getY(), 30, 30);

    }

  // main method

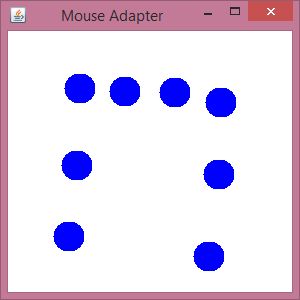
**public** **static** **void** main(String[] args) {

**new** MouseAdapterExample();

}

}

**Output:**



## **Java MouseMotionAdapter Example**

In the following example, we are implementing the MouseMotionAdapter class and its different methods to listen to the mouse motion events in the Frame window.

**MouseMotionAdapterExample.java**

// 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);

}

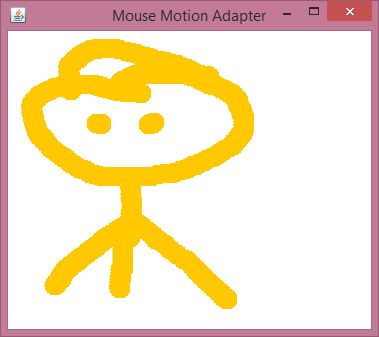
**public** **static** **void** main(String[] args) {

**new** MouseMotionAdapterExample();

}

}

**Output:**



## **Java KeyAdapter Example**

In the following example, we are implementing the KeyAdapter class and its method.

**KeyAdapterExample.java**

// importing the necessary libraries

**import** java.awt.\*;

**import** java.awt.event.\*;

// class which inherits the KeyAdapter class

**public** **class** KeyAdapterExample **extends** KeyAdapter {

// creating objects of Label, TextArea and Frame

    Label l;

    TextArea area;

    Frame f;

// class constructor

    KeyAdapterExample() {

// creating the Frame with title

        f = **new** Frame ("Key Adapter");

// creating the Label

        l = **new** Label();

// setting the location of label

l.setBounds (20, 50, 200, 20);

// creating the text area

        area = **new** TextArea();

// setting the location of text area

        area.setBounds (20, 80, 300, 300);

// adding KeyListener to text area

        area.addKeyListener(**this**);

 // adding the label and text area to frame

        f.add(l);

f.add(area);

// setting the size, layout and visibility of frame

        f.setSize (400, 400);

        f.setLayout (**null**);

        f.setVisible (**true**);

    }

// overriding the keyReleased() method

**public** **void** keyReleased (KeyEvent e) {

// creating the String object to get the text fromTextArea

        String text = area.getText();

// splitting the String into words

        String words[] = text.split ("\\s");

// setting the label text to print the number of words and characters of given string

        l.setText ("Words: " + words.length + " Characters:" + text.length());

    }

  // main method

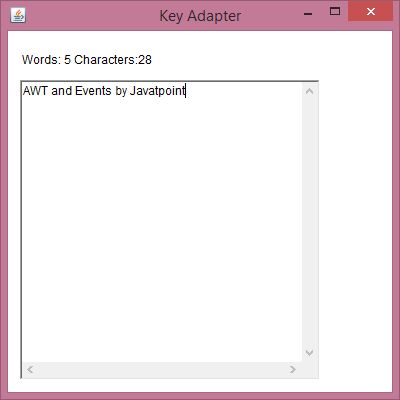
**public** **static** **void** main(String[] args) {

**new** KeyAdapterExample();

    }

}

**Output:**



**Anonymous Inner classes**

Java anonymous inner class is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain "extras" such as overloading methods of a class or interface, without having to actually subclass a class.

In simple words, a class that has no name is known as an anonymous inner class in Java. It should be used if you have to override a method of class or interface. Java Anonymous inner class can be created in two ways:

1. Class (may be abstract or concrete).
2. Interface

### Java anonymous inner class example using class

**TestAnonymousInner.java**

**abstract** **class** Person{

**abstract** **void** eat();

}

**class** TestAnonymousInner{

**public** **static** **void** main(String args[]){

  Person p=**new** Person(){

**void** eat(){System.out.println("nice fruits");}

  };

  p.eat();

 }

}

**Output**: nice fruits

## **Java anonymous inner class example using interface**

**interface** Eatable{

**void** eat();

}

**class** TestAnnonymousInner1{

**public** **static** **void** main(String args[]){

 Eatable e=**new** Eatable(){

**public** **void** eat(){System.out.println("nice fruits");}

 };

 e.eat();

 }

}

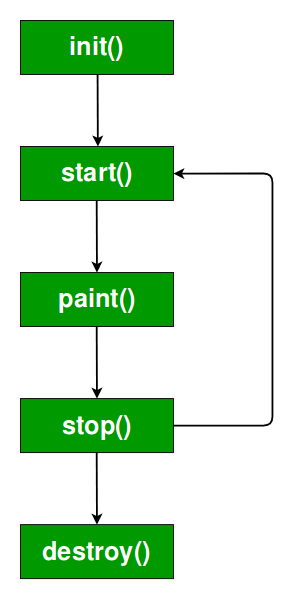
**Java Applet Basics**

An applet is a Java program that can be embedded into a web page. It runs inside the web browser and works at client side. An applet is embedded in an HTML page using the APPLET or OBJECT tag and hosted on a web server.  
Applets are used to make the website more dynamic and entertaining.

**Important points :**

1. All applets are sub-classes (either directly or indirectly) of [*java.applet.Applet*](https://docs.oracle.com/javase/7/docs/api/java/applet/Applet.html) class.
2. Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer. JDK provides a standard applet viewer tool called applet viewer.
3. In general, execution of an applet does not begin at main() method.
4. Output of an applet window is not performed by *System.out.println()*. Rather it is handled with various AWT methods, such as *drawString()*.

**Life cycle of an applet :**



It is important to understand the order in which the various methods shown in the above image are called. When an applet begins, the following methods are called, in this sequence:

1. init( )
2. start( )
3. paint( )

When an applet is terminated, the following sequence of method calls takes place:

1. stop( )
2. destroy( )

Let’s look more closely at these methods.

**1. init( ) :**The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called **only once** during the run time of your applet.

**2. start( ) :**The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Note that **init( )**is called once i.e. when the first time an applet is loaded whereas **start( )** is called each time an applet’s HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.

**3. paint( ) :**The **paint( )** method is called each time an AWT-based applet’s output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored.   
**paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )**is called.   
The **paint( )** method has one parameter of type [Graphics](https://docs.oracle.com/javase/7/docs/api/java/awt/Graphics.html). This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.   
Note: This is the only method among all the method mention above, which is parameterized. It’s prototype is   
public void paint(Graphics g)   
where g is an object reference of class Graphic.

**Creating Hello World applet :**

// A Hello World Applet

// Save file as HelloWorld.java

import java.applet.Applet;

import java.awt.Graphics;

// HelloWorld class extends Applet

public class HelloWorld extends Applet

{

public void paint(Graphics g)

{

g.drawString("Hello World", 20, 20);

}

}

**Explanation:**

1. The above java program begins with two import statements. The first import statement imports the Applet class from applet package. Every AWT-based(Abstract Window Toolkit) applet that you create must be a subclass (either directly or indirectly) of Applet class. The second statement import the [Graphics](https://docs.oracle.com/javase/7/docs/api/java/awt/Graphics.html) class from AWT package.
2. The next line in the program declares the class HelloWorld. This class must be declared as public because it will be accessed by code that is outside the program. Inside HelloWorld, **paint( )** is declared. This method is defined by the AWT and must be overridden by the applet.
3. Inside **paint( )** is a call to *drawString( )*, which is a member of the [Graphics](https://docs.oracle.com/javase/7/docs/api/java/awt/Graphics.html) class. This method outputs a string beginning at the specified X,Y location. It has the following general form:

void drawString(String message, int x, int y)

Here, message is the string to be output beginning at x,y. In a Java window, the upper-left corner is location 0,0. The call to *drawString( )*in the applet causes the message “Hello World” to be displayed beginning at location 20,20.

Notice that the applet does not have a **main( )** method. Unlike Java programs, applets do not begin execution at **main( )**. In fact, most applets don’t even have a **main( )** method. Instead, an applet begins execution when the name of its class is passed to an applet viewer or to a network browser.

**Running the HelloWorld Applet :**  
After you enter the source code for HelloWorld.java, compile in the same way that you have been compiling java programs(using *javac* command). However, running HelloWorld with the *java* command will generate an error because it is not an application.

java HelloWorld

Error: Main method not found in class HelloWorld,

please define the main method as:

public static void main(String[] args)

There are **two** standard ways in which you can run an applet :

1. Executing the applet within a Java-compatible web browser.
2. Using an applet viewer, such as the standard tool, applet-viewer. An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test your applet.

Each of these methods is described next.

**1. Using java enabled web browser :**To execute an applet in a web browser we have to write a short HTML text file that contains a tag that loads the applet. We can use APPLET or OBJECT tag for this purpose. Using APPLET, here is the HTML file that executes HelloWorld :

<applet code="HelloWorld" width=200 height=60>

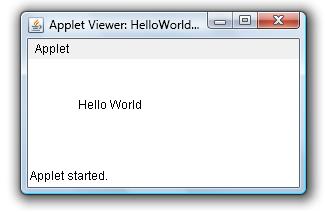
</applet>

The width and height statements specify the dimensions of the display area used by the applet. The APPLET tag contains several other options. After you create this html file, you can use it to execute the applet.

**NOTE :**Chrome and Firefox no longer supports NPAPI (technology required for Java applets). Refer [here](https://java.com/en/download/help/enable_browser.xml)

**2. Using appletviewer :**This is the easiest way to run an applet. To execute HelloWorld with an applet viewer, you may also execute the HTML file shown earlier. For example, if the preceding HTML file is saved with   
RunHelloWorld.html, then the following command line will run HelloWorld :

appletviewer RunHelloWorld.html



# **Parameter in Applet**

We can get any information from the HTML file as a parameter. For this purpose, Applet class provides a method named getParameter(). Syntax:

1. **public** String getParameter(String parameterName)

## **Example of using parameter in Applet:**

1. **import** java.applet.Applet;
2. **import** java.awt.Graphics;
4. **public** **class** UseParam **extends** Applet{
6. **public** **void** paint(Graphics g){
7. String str=getParameter("msg");
8. g.drawString(str,50, 50);
9. }
11. }

### myapplet.html

1. <html>
2. <body>
3. <applet code="UseParam.class" width="300" height="300">
4. <param name="msg" value="Welcome to applet">
5. </applet>
6. </body>
7. </html>

**Creating a Swing Applet**

|  |
| --- |
| we can use JApplet that can have all the controls of swing. The JApplet class extends the Applet class. |

## **Example of EventHandling in JApplet:**

1. **import** java.applet.\*;
2. **import** javax.swing.\*;
3. **import** java.awt.event.\*;
4. **public** **class** EventJApplet **extends** JApplet **implements** ActionListener{
5. JButton b;
6. JTextField tf;
7. **public** **void** init(){
9. tf=**new** JTextField();
10. tf.setBounds(30,40,150,20);
12. b=**new** JButton("Click");
13. b.setBounds(80,150,70,40);
15. add(b);add(tf);
16. b.addActionListener(**this**);
18. setLayout(**null**);
19. }
21. **public** **void** actionPerformed(ActionEvent e){
22. tf.setText("Welcome");
23. }
24. }

|  |
| --- |
| In the above example, we have created all the controls in init() method because it is invoked only once. |

### myapplet.html

1. <html>
2. <body>
3. <applet code="EventJApplet.class" width="300" height="300">
4. </applet>
5. </body>
6. </html>

**Java Swing**

Java swing is a part of Java Foundation Classes (JFC) that is used to create window-based applications. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

Unlike AWT, Java Swing provides platform-independent and lightweight components.

The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

### Difference between AWT and Swing

There are many differences between java awt and swing that are given below.

50.6M

988

C++ vs Java

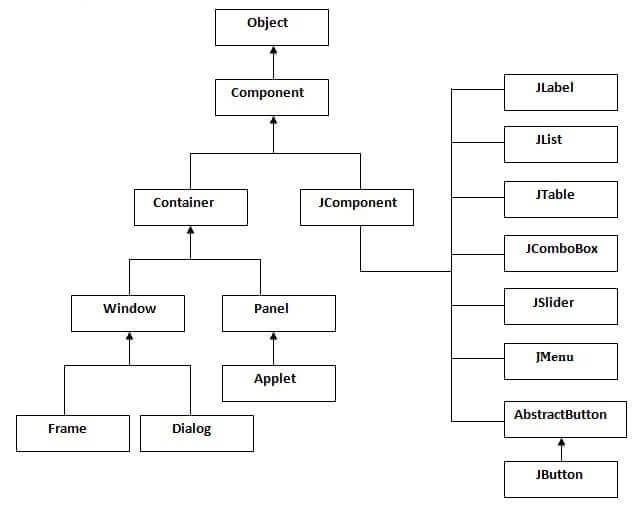
|  |  |  |
| --- | --- | --- |
| **No.** | **Java AWT** | **Java Swing** |
| 1) | AWT components are **platform-dependent**. | Java swing components are **platform-independent**. |
| 2) | AWT components are **heavyweight**. | Swing components are **lightweight**. |
| 3) | AWT **doesn't support pluggable look and feel**. | Swing **supports pluggable look and feel**. |
| 4) | AWT provides **less components** than Swing. | Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser, tabbedpane etc. |
| 5) | AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view. | Swing **follows MVC**. |

### What is JFC

The Java Foundation Classes (JFC) are a set of GUI components which simplify the development of desktop applications.

### Hierarchy of Java Swing classes

The hierarchy of java swing API is given below.



Simple Java Swing Example

Let's see a simple swing example where we are creating one button and adding it on the JFrame object inside the main() method.

*File: FirstSwingExample.java*

**import** javax.swing.\*;

**public** **class** FirstSwingExample {

**public** **static** **void** main(String[] args) {

JFrame f=**new** JFrame();//creating instance of JFrame

JButton b=**new** JButton("click");//creating instance of JButton

b.setBounds(130,100,100, 40);//x axis, y axis, width, height

f.add(b);//adding button in JFrame

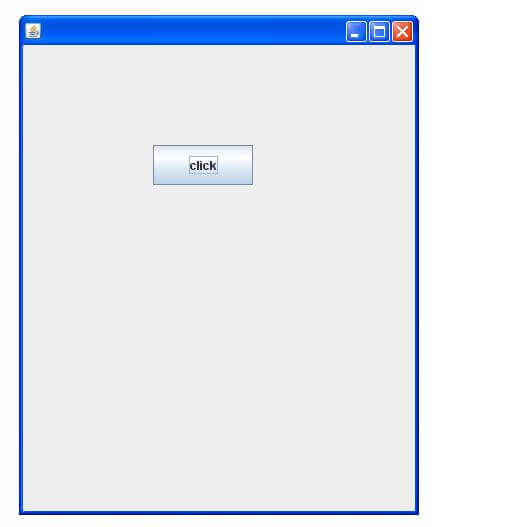
f.setSize(400,500);//400 width and 500 height

f.setLayout(**null**);//using no layout managers

f.setVisible(**true**);//making the frame visible

}

}



**Java JButton Example**

**import** java.awt.event.\*;

**import** javax.swing.\*;

**public** **class** ButtonExample {

**public** **static** **void** main(String[] args) {

    JFrame f=**new** JFrame("Button Example");

**final** JTextField tf=**new** JTextField();

    tf.setBounds(50,50, 150,20);

    JButton b=**new** JButton("Click Here");

    b.setBounds(50,100,95,30);

    b.addActionListener(**new** ActionListener(){

**public** **void** actionPerformed(ActionEvent e){

            tf.setText("Welcome to Javatpoint.");

        }

    });

    f.add(b);f.add(tf);

    f.setSize(400,400);

    f.setLayout(**null**);

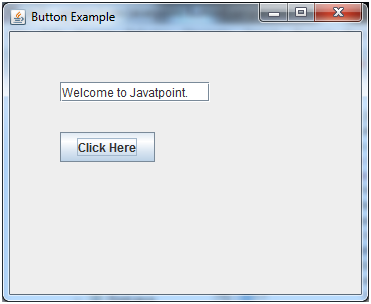
    f.setVisible(**true**);

}

}

Output:

Skip Ad



**JToggle Button**

JToggleButton is used to create toggle button, it is two-states button to switch on or off.

### Methods

|  |  |  |
| --- | --- | --- |
| **Modifier and Type** | **Method** | **Description** |
| AccessibleContext | getAccessibleContext() | It gets the AccessibleContext associated with this JToggleButton. |
| String | getUIClassID() | It returns a string that specifies the name of the l&f class that renders this component. |
| protected String | paramString() | It returns a string representation of this JToggleButton. |
| Void | updateUI() | It resets the UI property to a value from the current look and feel. |

## **JToggleButton Example**

**import** java.awt.FlowLayout;

**import** java.awt.event.ItemEvent;

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

**import** javax.swing.JFrame;

**import** javax.swing.JToggleButton;

**public** **class** JToggleButtonExample **extends** JFrame **implements** ItemListene {

**public** **static** **void** main(String[] args) {

**new** JToggleButtonExample();

    }

**private** JToggleButton button;

    JToggleButtonExample() {

        setTitle("JToggleButton with ItemListener Example");

        setLayout(**new** FlowLayout());

        setJToggleButton();

        setAction();

        setSize(200, 200);

        setVisible(**true**);

        setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

    }

**private** **void** setJToggleButton() {

        button = **new** JToggleButton("ON");

        add(button);

    }

**private** **void** setAction() {

        button.addItemListener(**this**);

    }

**public** **void** itemStateChanged(ItemEvent eve) {

**if** (button.isSelected())

            button.setText("OFF");

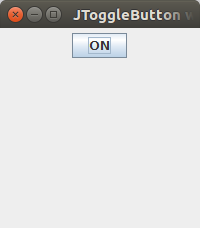
**else**

            button.setText("ON");

    }

}

Output



**JCheck Box**

The JCheckBox class is used to create a checkbox. It is used to turn an option on (true) or off (false). Clicking on a CheckBox changes its state from "on" to "off" or from "off" to "on ".It inherits [JToggleButton](https://www.javatpoint.com/java-jtogglebutton) class.

## **Java JCheckBox Example**

**import** javax.swing.\*;

**public** **class** CheckBoxExample

{

     CheckBoxExample(){

        JFrame f= **new** JFrame("CheckBox Example");

        JCheckBox checkBox1 = **new** JCheckBox("C++");

        checkBox1.setBounds(100,100, 50,50);

        JCheckBox checkBox2 = **new** JCheckBox("Java", **true**);

        checkBox2.setBounds(100,150, 50,50);

        f.add(checkBox1);

        f.add(checkBox2);

        f.setSize(400,400);

        f.setLayout(**null**);

        f.setVisible(**true**);

     }

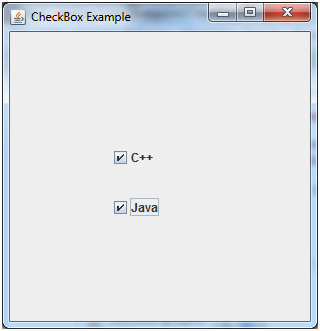
**public** **static** **void** main(String args[])

    {

**new** CheckBoxExample();

    }}

Output:



**JRadio Button**

The JRadioButton class is used to create a radio button. It is used to choose one option from multiple options. It is widely used in exam systems or quiz.

It should be added in ButtonGroup to select one radio button only.

## **Java JRadioButton Example with ActionListener**

**import** javax.swing.\*;

**import** java.awt.event.\*;

**class** RadioButtonExample **extends** JFrame **implements** ActionListener{

JRadioButton rb1,rb2;

JButton b;

RadioButtonExample(){

rb1=**new** JRadioButton("Male");

rb1.setBounds(100,50,100,30);

rb2=**new** JRadioButton("Female");

rb2.setBounds(100,100,100,30);

ButtonGroup bg=**new** ButtonGroup();

bg.add(rb1);bg.add(rb2);

b=**new** JButton("click");

b.setBounds(100,150,80,30);

b.addActionListener(**this**);

add(rb1);add(rb2);add(b);

setSize(300,300);

setLayout(**null**);

setVisible(**true**);

}

**public** **void** actionPerformed(ActionEvent e){

**if**(rb1.isSelected()){

JOptionPane.showMessageDialog(**this**,"You are Male.");

}

**if**(rb2.isSelected()){

JOptionPane.showMessageDialog(**this**,"You are Female.");

}

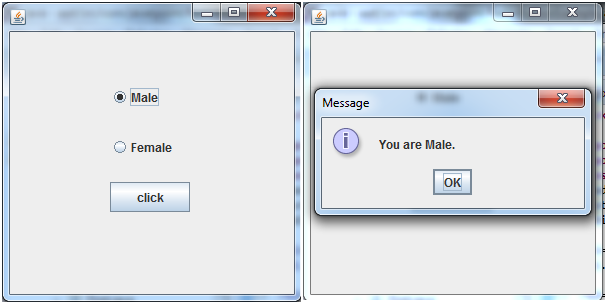
}

**public** **static** **void** main(String args[]){

**new** RadioButtonExample();

}}

Output:



**JTabbed Pane**

The JTabbedPane class is used to switch between a group of components by clicking on a tab with a given title or icon. It inherits JComponent class.

## **Java JTabbedPane Example**

**import** javax.swing.\*;

**public** **class** TabbedPaneExample {

JFrame f;

TabbedPaneExample(){

    f=**new** JFrame();

    JTextArea ta=**new** JTextArea(200,200);

    JPanel p1=**new** JPanel();

    p1.add(ta);

    JPanel p2=**new** JPanel();

    JPanel p3=**new** JPanel();

    JTabbedPane tp=**new** JTabbedPane();

    tp.setBounds(50,50,200,200);

    tp.add("main",p1);

    tp.add("visit",p2);

    tp.add("help",p3);

    f.add(tp);

    f.setSize(400,400);

    f.setLayout(**null**);

    f.setVisible(**true**);

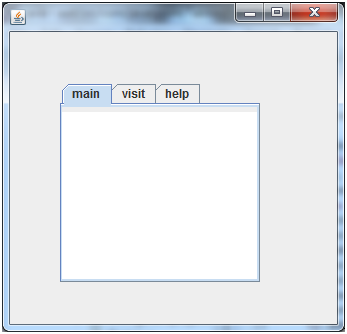
}

**public** **static** **void** main(String[] args) {

**new** TabbedPaneExample();

}}

Output:



**JScroll Pane**

A JscrollPane is used to make scrollable view of a component. When screen size is limited, we use a scroll pane to display a large component or a component whose size can change dynamically.

## **ScrollPane Example**

**import** java.awt.FlowLayout;

**import** javax.swing.JFrame;

**import** javax.swing.JScrollPane;

**import** javax.swing.JtextArea;

**public** **class** JScrollPaneExample {

**private** **static** **final** **long** serialVersionUID = 1L;

**private** **static** **void** createAndShowGUI() {

        // Create and set up the window.

**final** JFrame frame = **new** JFrame("Scroll Pane Example");

        // Display the window.

        frame.setSize(500, 500);

        frame.setVisible(**true**);

        frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

        // set flow layout for the frame

        frame.getContentPane().setLayout(**new** FlowLayout());

        JTextArea textArea = **new** JTextArea(20, 20);

        JScrollPane scrollableTextArea = **new** JScrollPane(textArea);

        scrollableTextArea.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTA\_SCROLLBAR\_ALWAYS);

        scrollableTextArea.setVerticalScrollBarPolicy(JScrollPane.VERTICAL\_SCROLLBAR\_ALWAYS);

        frame.getContentPane().add(scrollableTextArea);

    }

**public** **static** **void** main(String[] args) {

        javax.swing.SwingUtilities.invokeLater(**new** Runnable() {

**public** **void** run() {

                createAndShowGUI();

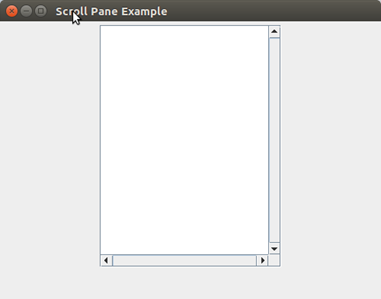
            }

        });

    }

}

Output:



**JList**

The object of JList class represents a list of text items. The list of text items can be set up so that the user can choose either one item or multiple items. It inherits JComponent class.

## **Java JList Example**

**import** javax.swing.\*;

**public** **class** ListExample

{

     ListExample(){

        JFrame f= **new** JFrame();

        DefaultListModel<String> l1 = **new** DefaultListModel<>();

          l1.addElement("Item1");

          l1.addElement("Item2");

          l1.addElement("Item3");

          l1.addElement("Item4");

          JList<String> list = **new** JList<>(l1);

          list.setBounds(100,100, 75,75);

          f.add(list);

          f.setSize(400,400);

          f.setLayout(**null**);

          f.setVisible(**true**);

     }

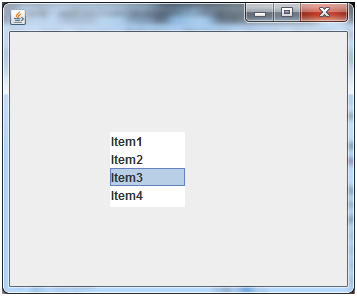
**public** **static** **void** main(String args[])

    {

**new** ListExample();

    }}

Output:



**JCombo Box**

The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a [menu](https://www.javatpoint.com/java-jmenuitem-and-jmenu)

. It inherits [JComponent](https://www.javatpoint.com/java-jcomponent)

class.

## **Java JComboBox Example**

**import** javax.swing.\*;

**public** **class** ComboBoxExample {

JFrame f;

ComboBoxExample(){

    f=**new** JFrame("ComboBox Example");

    String country[]={"India","Aus","U.S.A","England","Newzealand"};

    JComboBox cb=**new** JComboBox(country);

    cb.setBounds(50, 50,90,20);

    f.add(cb);

    f.setLayout(**null**);

    f.setSize(400,500);

    f.setVisible(**true**);

}

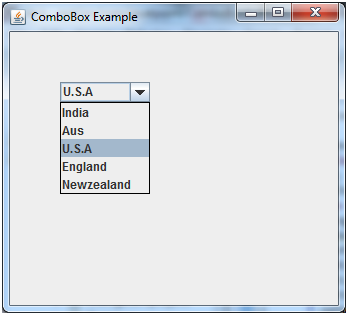
**public** **static** **void** main(String[] args) {

**new** ComboBoxExample();

}

}

Output:



**Swing Menus**

The JMenuBar class is used to display menubar on the window or frame. It may have several menus.

The object of JMenu class is a pull down menu component which is displayed from the menu bar. It inherits the JMenuItem class.

The object of JMenuItem class adds a simple labeled menu item. The items used in a menu must belong to the JMenuItem or any of its subclass.

### JMenuBar class declaration

1. **public** **class** JMenuBar **extends** JComponent **implements** MenuElement, Accessible

### JMenu class declaration

1. **public** **class** JMenu **extends** JMenuItem **implements** MenuElement, Accessible

### JMenuItem class declaration

1. **public** **class** JMenuItem **extends** AbstractButton **implements** Accessible, MenuElement

## **Java JMenuItem and JMenu Example**

**import** javax.swing.\*;

**class** MenuExample

{

          JMenu menu, submenu;

          JMenuItem i1, i2, i3, i4, i5;

          MenuExample(){

          JFrame f= **new** JFrame("Menu and MenuItem Example");

          JMenuBar mb=**new** JMenuBar();

          menu=**new** JMenu("Menu");

          submenu=**new** JMenu("Sub Menu");

          i1=**new** JMenuItem("Item 1");

          i2=**new** JMenuItem("Item 2");

          i3=**new** JMenuItem("Item 3");

          i4=**new** JMenuItem("Item 4");

          i5=**new** JMenuItem("Item 5");

          menu.add(i1); menu.add(i2); menu.add(i3);

          submenu.add(i4); submenu.add(i5);

          menu.add(submenu);

          mb.add(menu);

          f.setJMenuBar(mb);

          f.setSize(400,400);

          f.setLayout(**null**);

          f.setVisible(**true**);

}

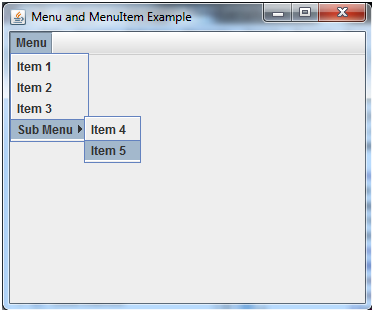
**public** **static** **void** main(String args[])

{

**new** MenuExample();

}}

Output:



**Dialogs**

# **Java JDialog**

The JDialog control represents a top level window with a border and a title used to take some form of input from the user. It inherits the Dialog class.

## **JDialog class declaration**

Let's see the declaration for javax.swing.JDialog class.

1. **public** **class** JDialog **extends** Dialog **implements** WindowConstants, Accessible, RootPaneContainer

### Commonly used Constructors:

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| JDialog() | It is used to create a modeless dialog without a title and without a specified Frame owner. |
| JDialog(Frame owner) | It is used to create a modeless dialog with specified Frame as its owner and an empty title. |
| JDialog(Frame owner, String title, boolean modal) | It is used to create a dialog with the specified title, owner Frame and modality. |

## **Java JDialog Example**

**import** javax.swing.\*;

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** DialogExample {

**private** **static** JDialog d;

    DialogExample() {

        JFrame f= **new** JFrame();

        d = **new** JDialog(f , "Dialog Example", **true**);

        d.setLayout( **new** FlowLayout() );

        JButton b = **new** JButton ("OK");

        b.addActionListener ( **new** ActionListener()

        {

**public** **void** actionPerformed( ActionEvent e )

            {

                DialogExample.d.setVisible(**false**);

            }

        });

        d.add( **new** JLabel ("Click button to continue."));

        d.add(b);

        d.setSize(300,300);

        d.setVisible(**true**);

    }

**public** **static** **void** main(String args[])

    {

**new** DialogExample();

    }

}

