
MODULE 3: Swings

Introduction

Swing contains a set of classes that provides more powerful and flexible GUI components than those of **AWT**. **Swing** provides the look and feel of modern Java GUI. Swing library is an official Java GUI tool kit released by Sun Microsystems. It is used to create graphical user interface with Java.

Swing is a set of program components for **Java** programmers that provide the ability to create graphical user interface (GUI) components, such as buttons and scroll bars, that are independent of the windowing system for specific operating system . **Swing** components are used with the **Java** Foundation Classes (JFC).

The Origins of Swing

The original Java GUI subsystem was the Abstract Window Toolkit (AWT).

AWT translates its visual components into platform-specific equivalents (peers).

Under AWT, the look and feel of a component was defined by the platform.

AWT components are referred to as **heavyweight**.

Swing was introduced in 1997 to fix the problems with AWT.

Swing offers following key features:

1. Platform Independent
 2. Customizable
 3. Extensible
 4. Configurable
 5. Lightweight
- ☐ Swing components are **lightweight** and don't rely on peers.
 - ☐ Swing supports a pluggable look and feel.
 - ☐ Swing is built on AWT.

Model-View-Controller

One component architecture is MVC - Model-View-Controller.

The **model** corresponds to the state information associated with the component.

The **view** determines how the component is displayed on the screen.

The **controller** determines how the component responds to the user.

Swing uses a modified version of MVC called "Model-Delegate". In this model the view (look) and controller (feel) are combined into a "delegate".

Because of the Model-Delegate architecture, the look and feel can be changed without affecting how the component is used in a program.

Components and Containers

A component is an independent visual control: a button, a slider, a label, ... A container holds a group of components.

In order to display a component, it must be placed in a container.

A container is also a component and can be contained in other containers.

Swing applications create a containment-hierarchy with a single top-level container.

Components

Swing components are derived from the **JComponent** class. The only exceptions are the four top-level containers: JFrame, JApplet, JWindow, and JDialog.

JComponent inherits AWT classes Container and Component.

All the Swing components are represented by classes in the javax.swing package.

All the component classes start with **J**: JLabel, JButton, JScrollbar, ...

Containers

There are two types of containers:

- 1) Top-level which do not inherit JComponent, and
- 2) Lightweight containers that do inherit JComponent.

Lightweight components are often used to organize groups of components.

Containers can contain other containers.

All the component classes start with **J**: JLabel, JButton, JScrollbar, ...

Top-level Container Panes

Each top-level component defines a collection of "panes". The top-level pane is

JRootPane.

JRootPane manages the other panes and can add a menu bar.

There are three panes in JRootPane: 1) the glass pane, 2) the content pane,
3) the layered pane.

The content pane is the container used for visual components. The content pane is an instance of JPanel.

The Swing Packages:

Swing is a very large subsystem and makes use of many packages. These are the packages used by Swing that are defined by Java SE 6.

The main package is **javax.swing**. This package must be imported into any program that uses Swing. It contains the classes that implement the basic Swing components, such as push buttons, labels, and check boxes.

Some of the Swing Packages are:

javax.swing javax.swing.border javax.swing.colorchooser javax.swing.event javax.swing.filechooser javax.swing.plaf javax.swing.plaf.basic javax.swing.plaf.metal javax.swing.plaf.multi	javax.swing.plaf.synth javax.swing.table javax.swing.text javax.swing.text.html javax.swing.text.html.parser javax.swing.text.rtf javax.swing.tree javax.swing.undo
---	--

A simple Swing Application;

There are two ways to create a frame:

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

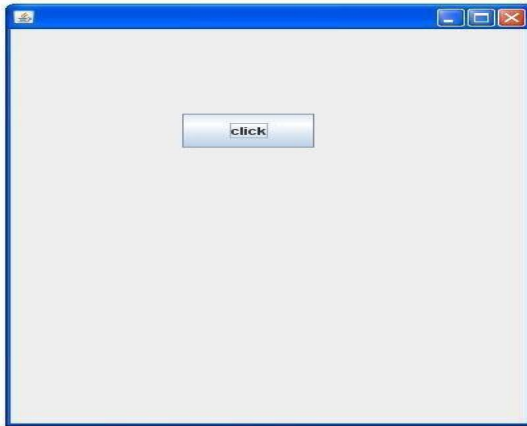
We can write the code of swing inside the main(), constructor or any other method

By creating the object of Frame class

```
import javax.swing.*;
public class FirstSwing
{
    public static void main(String[] args)
    {
        JFrame f=new JFrame(" MyApp");
        //creating instance of JFrame and title of the frame is MyApp.
        JButton b=new JButton("click");
        //creating instance of JButton and name of the button is click

        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.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }      f.setVisible(true);                  //making the frame visible
}
```

Output:



Explanation:

- ☐ The program begins by importing javax.swing. As mentioned, this package contains the components and models defined by Swing.
- ☐ For example, javax.swing defines classes that implement labels, buttons, text controls, and menus. It will be included in all programs that use Swing. Next,
- ☐ the program declares the FirstSwing class
- ☐ It begins by creating a JFrame, using this line of code:

```
JFrame f = new JFrame("My App");
```

- ☐ This creates a container called f that defines a rectangular window complete with a title bar; close, minimize, maximize, and restore buttons; and a system menu. Thus, it creates a standard, top-level window. The title of the window is passed to the constructor.

Next, the window is sized using this statement:

```
setSize(400,500);
```

- ☐ The setSize() method (which is inherited by JFrame from the AWT class Component) sets the dimensions of the window, which are specified in pixels. In this example, the width of the window is set to 400 and the height is set to 500.
- ☐ By default, when a top-level window is closed (such as when the user clicks the close box), the window is removed from the screen, but the application is not terminated.

- If want the entire application to terminate when its top-level window is closed. There are a couple of ways to achieve this. The easiest way is to call `setDefaultCloseOperation( )`, as the program does:

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

Swing by inheritance

- We can also inherit the `JFrame` class, so there is no need to create the instance of `JFrame` class explicitly.

```
import javax.swing.*;
public class MySwing extends JFrame //inheriting JFrame
{
    JFrame f;
    MySwing()
    {
        JButton b=new JButton("click");//create
        button b.setBounds(130,100,100, 40);

        add(b);//adding button on
        frame setSize(400,500);
        setLayout(null);
        setVisible(true);
    }
    public static void main(String[] args)
    {
        new MySwing();
    }
}
```

Event Handling

- The preceding example showed the basic form of a Swing program, but it left out one important part: event handling. Because JLabel does not take input from the user, it does not generate events, so no event handling was needed. However, the other Swing components do respond to user input and the events generated by those interactions need to be handled. Events can also be generated in ways not directly related to user input. For example, an event is generated when a timer goes off. Whatever the case, event handling is a large part of any Swing-based application. The event handling mechanism used by Swing is the same as that used by the AWT. This approach is called the delegation event model.
- In many cases, Swing uses the same events as does the AWT, and these events are packaged in `java.awt.event`. Events specific to Swing are stored in `javax.swing.event`. Although events are handled in Swing in the same way as they are with the AWT, it is still useful to work through a simple example. The following program handles the event generated by a Swing push button. Sample output is shown in Figure
- ```
// Handle an event in a Swing program.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class EventDemo {JLabel jlab; EventDemo() { // Create a new JFrame container.
JFrame jfrm = new JFrame("An Event Example"); // Specify FlowLayout for the
layout manager. jfrm.setLayout(new FlowLayout()); // Give the frame an initial
size.
jfrm.setSize(220,90)//Terminate the program when the user closes the application.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Make two buttons.
JButton jbtnAlpha = new JButton("Alpha");
JButton jbtnBeta = new JButton("Beta"); // Add action listener for Alpha.
jbtnAlpha.addActionListener(new ActionListener()

{ public void actionPerformed(ActionEvent ae) {

jlab.setText("Alpha was pressed.");

}

}); // Add action listener for Beta.

jbtnBeta.addActionListener(new ActionListener()

{ public void actionPerformed(ActionEvent ae)

{
```

```

jlab.setText("Beta was pressed."); } }); // Add the buttons to the content pane.
jfrm.add(jbtnAlpha);

jfrm.add(jbtnBeta); // Create a text-based label.

jlab = new JLabel("Press a button."); // Add the label to the content pane.
jfrm.add(jlab); // Display the frame.

jfrm.setVisible(true); }

public static void main(String[] args)
{
 //Create the frame on the event dispatching thread.
 SwingUtilities.invokeLater(new Runnable()
 { public void run()
 {new EventDemo(); }
 });
}

```

First, notice that the program now imports both the `java.awt` and `java.awt.event` packages. The `java.awt` package is needed because it contains the `FlowLayout` class, which supports the standard flow layout manager used to lay out components in a frame. The `java.awt.event` package is needed because it defines the `ActionListener` interface and the `ActionEvent` class. The `EventDemo` constructor begins by creating a `JFrame` called `jfrm`. It then sets the layout manager for the content pane of `jfrm` to `FlowLayout`. manager. However, for this example, `FlowLayout` is more convenient. After setting the size and default close operation, `EventDemo( )` creates two push buttons, as shown here: `JButton jbtnAlpha = new JButton("Alpha"); JButton jbtnBeta = new JButton("Beta");` The first button will contain the text "Alpha" and the second will contain the text "Beta". Swing push buttons are instances of `JButton`. `JButton` supplies several constructors. The one used here is `JButton(String msg)` The `msg` parameter specifies the string that will be displayed inside the button. When a push button is pressed, it generates an `ActionEvent`. Thus, `JButton` provides the `addActionListener( )` method, which is used to add an action listener. the `ActionListener` interface defines only one method: `actionPerformed( )`. It is shown again here for your convenience: `void actionPerformed(ActionEvent ae)` This method is called when a button is pressed. In other words, it is the event handler that is called when a button press event has occurred. Next, event listeners for the button's action events are added by the code shown here:

```

// Add action listener for Alpha.

jbtnAlpha.addActionListener(new ActionListener()

```

```

{ public void actionPerformed(ActionEvent ae)
{ jlab.setText("Alpha was pressed."); } }); // Add action listener for Beta.
jbtnBeta.addActionListener(new ActionListener()
{
public void actionPerformed(ActionEvent ae)
{ jlab.setText("Beta was pressed."); } });

```

Here, anonymous inner classes are used to provide the event handlers for the two buttons. Each time a button is pressed, the string displayed in jlab is changed to reflect which button was pressed.:

```
jbtnAlpha.addActionListener((ae) -> jlab.setText("Alpha was pressed."));
```

Introducing GUI Programming with Swing As you can see, this code is shorter. Of course, the approach you choose will be determined by the situation and your own preferences. Next, the buttons are added to the content pane of jfrm: `jfrm.add(jbtnAlpha); jfrm.add(jbtnBeta);` Finally, jlab is added to the content pane and the window is made visible. When you run the program, each time you press a button, a message is displayed in the label that indicates which button was pressed. One last point: Remember that all event handlers, such as `actionPerformed()`, are called on the event dispatching thread. Therefore, an event handler must return quickly in order to avoid slowing down the application. If your application needs to do something time consuming as the result of an event, it must use a separate thread.

## Painting in Swing

**Painting Fundamentals** Swing's approach to painting is built on the original AWT-based mechanism, but Swing's implementation offers more finely grained control. Before examining the specifics of Swing-based painting, it is useful to review the AWT-based mechanism that underlies it. The AWT class `Component` defines a method called `paint()` that is used to draw output directly to the surface of a component. For the most part, `paint()` is not called by your program. (In fact, only in the most unusual cases should it ever be called by your program.) Rather, `paint()` is called by the run-time system whenever a component must be rendered. This situation can occur for several reasons. For example, the window in which the component is displayed can be overwritten by another window and then uncovered. Or, the window might be minimized and then restored. The `paint()` method is also called when a program begins running. When writing AWT-based code, an application will override `paint()` when it needs to write output directly to the surface of the component. Because `JComponent` inherits `Component`, all Swing's lightweight components inherit the `paint()` method. However, you will

not override it to paint directly to the surface of a component. The reason is that Swing uses a bit more sophisticated approach to painting that involves three distinct methods: `paintComponent()`, `paintBorder()`, and `paintChildren()`

These methods paint the indicated portion of a component and divide the painting process into its three distinct, logical actions. In a lightweight component, the original AWT method `paint()` simply executes calls to these methods, in the order just shown. To paint to the surface of a Swing component, you will create a subclass of the component and then override its `paintComponent()` method. This is the method that paints the interior of the component. You will not normally override the other two painting methods. When overriding `paintComponent()`, the first thing you must do is call `super.paintComponent()`, so that the superclass portion of the painting process takes place. (The only time this is not required is when you are taking complete, manual control over how a component is displayed.) After that, write the output that you want to display. The `paintComponent()` method is shown here: `protected void paintComponent(Graphics g)` The parameter `g` is the graphics context to which output is written. To cause a component to be painted under program control, call `repaint()`. It works in Swing just as it does for the AWT. The `repaint()` method is defined by `Component`. Calling it causes the system to call `paint()` as soon as it is possible to do so. Because painting is a time-consuming operation, this mechanism allows the run-time system to defer painting momentarily until some higher-priority task has completed, for example. Of course, in Swing the call to `paint()` results in a call to `paintComponent()`. Therefore, to output to the surface of a component, your program will store the output until `paintComponent()` is called. Inside the overridden `paintComponent()`, you will draw the stored output.

When drawing to the surface of a component, you must be careful to restrict your output to the area that is inside the border. Although Swing automatically clips any output that will exceed the boundaries of a component, it is still possible to paint into the border, which will then get overwritten when the border is drawn. To avoid this, you must compute the paintable area of the component. This is the area defined by the current size of the component minus the space used by the border. Therefore, before you paint to a component, you must obtain the width of the border and then adjust your drawing accordingly. To obtain the border width, call `getInsets()`, shown here: `Insets getInsets()` This method is defined by `Container` and overridden by `JComponent`. It returns an `Insets` object that contains the dimensions of the border. The inset values can be obtained by using these fields: `int top`; `int bottom`; `int left`; `int right`;

These values are then used to compute the drawing area given the width and the height of the component. You can obtain the width and height of the component by calling `getWidth( )` and `getHeight( )` on the component. They are shown here: `int getWidth( ) int getHeight( )` By subtracting the value of the insets, you can compute the usable width and height of the component. A Paint Example Here is a program that puts into action the preceding discussion. It creates a class called `PaintPanel` that extends `JPanel`. The program then uses an object of that class to display lines whose endpoints have been generated randomly.

Paint lines to a panel.

```
import java.awt.*;
```

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

```
import javax.swing.*;
```

```
import java.util.*; // This class extends JPanel.
```

It overrides // the `paintComponent()` method so that random // lines are plotted in the panel. class

```
PaintPanel extends JPanel
```

```
{
```

```
 Insets ins; // holds the panel's insets
```

```
 Random rand; // used to generate random numbers // Construct a panel.
 PaintPanel()
```

```
{ // Put a border around the panel.
```

```
 setBorder(BorderFactory.createLineBorder(Color.RED, 5));
```

```
 rand = new Random();
```

```
 } // Override the paintComponent() method.
```

```
 protected void paintComponent(Graphics g)
```

```
 { // Always call the superclass method first.
```

```
 super.paintComponent(g);
```

```
 int x, y, x2, y2; // Get the height and width of the component.
```

```
 int height = getHeight();
```

```
 int width = getWidth(); // Get the insets.
```

```
 ins = getInsets(); // Draw ten lines whose endpoints are randomly generated.
 for(int i=0; i < 10; i++) { // Obtain random coordinates that define // the endpoints
 of each line.
```

```
 x = rand.nextInt(width-ins.left);
```

```

y = rand.nextInt(height-ins.bottom);
x2 = rand.nextInt(width-ins.left);
y2 = rand.nextInt(height-ins.bottom); // Draw the line.
g.drawLine(x, y, x2, y2); } } } // Demonstrate painting directly onto a panel.

class PaintDemo
{
JLabel jlab;
PaintPanel pp;
PaintDemo() { // Create a new JFrame container.

JFrame jfrm = new JFrame("Paint Demo"); // Give the frame an initial size.
jfrm.setSize(200, 150); // Terminate the program when the user closes the
application.

jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Create the panel
that will be painted.

pp = new PaintPanel(); // Add the panel to the content pane. Because the default //
border layout is used, the panel will automatically be // sized to fit the center
region. jfrm.add(pp);

Display the frame. jfrm.setVisible(true);

} public static void main(String[] args) { // Create the frame on the event
dispatching thread.

SwingUtilities.invokeLater(new Runnable()

{ public void run()

{ new PaintDemo(); } }); } }

```

Let's examine this program closely. The PaintPanel class extends JPanel. JPanel is one of Swing's lightweight containers, which means that it is a component that can be added to the content pane of a JFrame. To handle painting, PaintPanel overrides the paintComponent( ) method. This enables PaintPanel to write directly to the surface of the component when painting takes place. The size of the panel is not specified because the program uses the default border layout and the panel is added to the center. This results in the panel being sized to fill the center. If you change the size of the window, the size of the panel will be adjusted accordingly. Notice that the constructor also specifies a 5-pixel wide, red border. This is accomplished by setting the border by using the setBorder( ) method, shown here:

```
void setBorder(Border border)
```

Border is the Swing interface that encapsulates a border. You can obtain a border by calling one of the factory methods defined by the BorderFactory class. The one used in the program is createLineBorder( ),

which creates a simple line border. It is shown here: `static Border createLineBorder(Color clr, int width)` Here, `clr` specifies the color of the border and `width` specifies its width in pixels. Inside the override of `paintComponent()`, notice that it first calls `super .paintComponent()`. As explained, this is necessary to ensure that the component is properly drawn. Next, the width and height of the panel are obtained along with the insets. These values are used to ensure the lines lie within the drawing area of the panel. The drawing area is the overall width and height of a component less the border width. The computations are designed to work with differently sized `PaintPanel`s and borders. To prove this, try changing the size of the window. The lines will still all lie within the borders of the panel. The `PaintDemo` class creates a `PaintPanel` and then adds the panel to the content pane. When the application is first displayed, the overridden `paintComponent()` method is called, and the lines are drawn. Each time you resize or hide and restore the window, a new set of lines are drawn. In all cases, the lines fall within the paintable area

### **JLabel, JTextField and JPasswordField**

- **JLabel** is Swing's easiest-to-use component. It creates a label and was introduced in the preceding chapter. Here, we will look at `JLabel` a bit more closely.
- `JLabel` can be used to display text and/or an icon. It is a passive component in that it does not respond to user input. `JLabel` defines several constructors. Here are three of them:

`JLabel(Icon icon)`

`JLabel(String str)`

`JLabel(String str, Icon icon, int align)`

---

**JTextField** is the simplest Swing text component. It is also probably its most widely used text component. JTextField allows you to edit one line of text. It is derived from JTextComponent, which provides the basic functionality common to Swing text components.

Three of JTextField's constructors are shown here:

JTextField(int cols)

JTextField(String str, int cols)

JTextField(String str)

- Here, str is the string to be initially presented, and cols is the number of columns in the text field. If no string is specified, the text field is initially empty. If the number of columns is not specified, the text field is sized to fit the specified string.
  
- **JPasswordField** is a lightweight component that allows the editing of a single line of text where the view indicates something was typed, but does not show the original characters.

```
import javax.swing.*;
public class JTextFieldPgm
{

 public static void main(String[] args)
 {

 JFrame f=new JFrame("My App");

 JLabel nameLabel= new JLabel("User ID: ");
 nameLabel.setBounds(10, 20, 70, 10);
 }
}
```

---

```
JLabel passwordLabel = new JLabel("Password: "); passwordLabel.setBounds(10, 50, 70, 10);

JTextField userText = new JTextField(); userText.setBounds(80, 20, 100, 20);

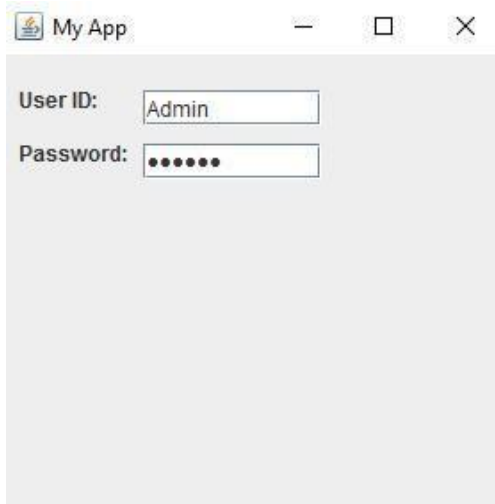
JPasswordField passwordText = new JPasswordField();
passwordText.setBounds(80, 50, 100, 20);

f.add(namelabel); f.add(
passwordLabel);
f.add(userText);
f.add(passwordText);

f.setSize(300, 300);
f.setLayout(null);
f.setVisible(true);

}

}
```



## ImageIcon with JLabel

- JLabel can be used to display text and/or an icon. It is a passive component in that it does not respond to user input. JLabel defines several constructors. Here are three of them:

JLabel(Icon icon)

JLabel(String str)

JLabel(String str, Icon icon, int align)

- Here, str and icon are the text and icon used for the label.
- The align argument specifies the horizontal alignment of the text and/or icon within the dimensions of the label. It must be one of the following values: LEFT, RIGHT, CENTER, LEADING, or TRAILING.
- These constants are defined in the Swing Constants interface, along with several others used by the Swing classes. Notice that icons are specified by objects of type Icon, which is an interface defined by Swing.
- The easiest way to obtain an icon is to use the ImageIcon class. ImageIcon implements Icon and encapsulates an image. Thus, an object of type ImageIcon can be passed as an argument to the Icon parameter of JLabel's constructor.

ImageIcon(String filename)

- It obtains the image in the file named filename. The icon and text associated with the label can be obtained by the following methods:

Icon getIcon( )

String getText( )

The icon and text associated with a label can be set by these methods: void

setIcon(Icon icon)

void setText(String str)

- Here, **icon** and **str** are the icon and text, respectively. Therefore, using setText( ) it is possible to change the text inside a label during program execution.

```
import javax.swing.*;

public class PgmImageIcon
{

 public static void main(String[] args)
 {

 JFrame jf=new JFrame("Image
 Icon"); jf.setLayout(null);

 Icon icon = new ImageIcon("a.jpg");
 JLabel label1 = new JLabel("Welocme to SVIT",icon,JLabel.RIGHT);
 label1.setBounds(20, 30,
 267, 200); jf.add(label1);

 jf.setSize(
 300,400);
 jf.setVisible(true);

 }

}
```

## The Swing Buttons:

There are four types of Swing Button

1. JButton
2. JRadioButton
3. JCheckBox
4. JComboBox

**JButton** class provides functionality of a button. A JButton is the Swing equivalent of a Button in AWT. It is used to provide an interface equivalent of a common button.

JButton class has three constructors,

**JButton**(Icon *ic*)

---

**JButton**(String *str*)

**JButton**(String *str*, Icon *ic*)

```
import javax.swing.*;

class FirstSwing
{
 public static void main(String args[])
 {
 JFrame jf=new JFrame("My App");

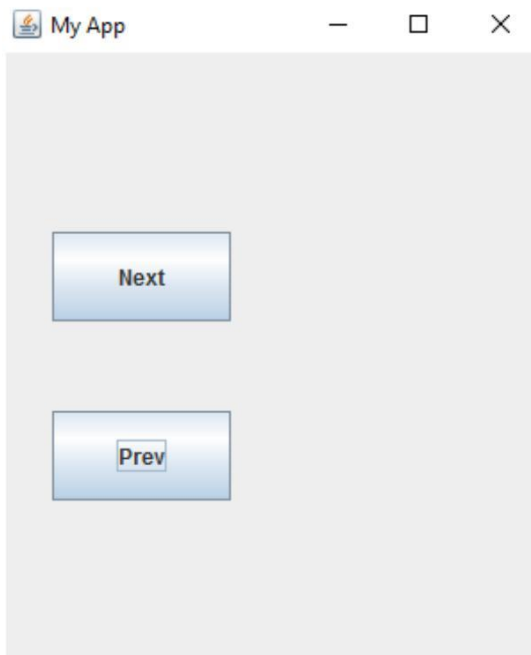
 JButton jb=new JButton("Next");
 jb.setBounds(30, 100, 100, 50);

 JButton jb1=new JButton("Prev");
 jb1.setBounds(30, 200, 100, 50);

 jf.add(jb);
 jf.add(jb1);

 jf.setSize(300, 600);
 jf.setLayout(null);
 jf.setVisible(true);

 }
}
```



A **JRadioButton** is the swing equivalent of a `RadioButton` in AWT. It is used to represent multiple option single selection elements in a form. This is performed by grouping the `JRadio` buttons using a `ButtonGroup` component. The `ButtonGroup` class can be used to group multiple buttons so that at a time only one button can be selected.

```
import javax.swing.*;

import javax.swing.*;
public class RadioButton1
{
 public static void main(String args[])
 {
 JFrame f=new JFrame("MyAppRadio");

 JRadioButton r1=new JRadioButton("Male ");

 JRadioButton r2=new JRadioButton("Female");

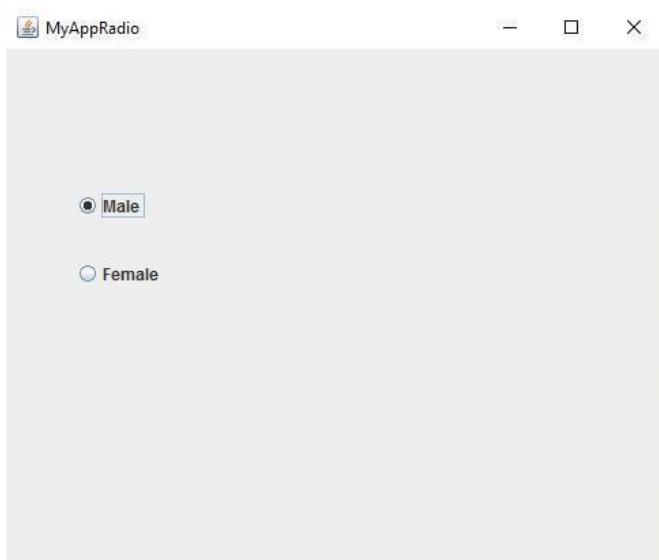
 r1.setBounds(50, 100, 70, 30);
```

```
r2.setBounds(50,150,70,30);

ButtonGroup bg=new
ButtonGroup(); bg.add(r1);
bg.add(r2);

f.add(r1)
;
f.add(r2)
;

f.setSize(500,500);
f.setLayout(null);
f.setVisible(true);
}
}
```



A **JCheckBox** is the Swing equivalent of the Checkbox component in AWT. This is sometimes called a ticker box, and is used to represent multiple option selections in a form.

```
import javax.swing.*;

class FirstSwing
{
 public static void main(String args[])
 {
 JFrame jf=new JFrame("CheckBox");

 JCheckBox jb=new JCheckBox("JAVA"); jb.setBounds(30, 100, 100, 50);

 JCheckBox jb1=new JCheckBox("Python"); jb1.setBounds(30, 200, 100, 50);

 jf.add(jb);
 jf.add(jb1);

 jf.setSize(300, 600);
 jf.setLayout(null);
 jf.setVisible(true);

 }
}
```



---

## JComboBox

The JComboBox class is used to create the combobox (drop-down list). At a time only one item can be selected from the item list.

```
import java.awt.*;

import javax.swing.*;

public class Comboexample
{
 public static void main(String[] args)
 {
 JFrame f=new JFrame("Combo demo");

 String Branch[]={"cse","ise","ec","mech"};

 JComboBox jc=new JComboBox(Branch);

 jc.setBounds(50,50,80,50);

 f.add(jc);

 f.setSize(400, 400);

 f.setLayout(null);

 f.setVisible(true);

 }
}
```

