

UNIT - 6

AWT

[ABSTARCT WINDOW TOOLKIT]

SWINGS

Introduction to AWT

AWT (*Abstract Window Toolkit*) was Java's first GUI framework, which was introduced in Java 1.0. It is used to create GUIs (*Graphical User Interfaces*). Although programmers use more advanced frameworks like Swings and JavaFX, it is important to know that they are built on top of AWT.

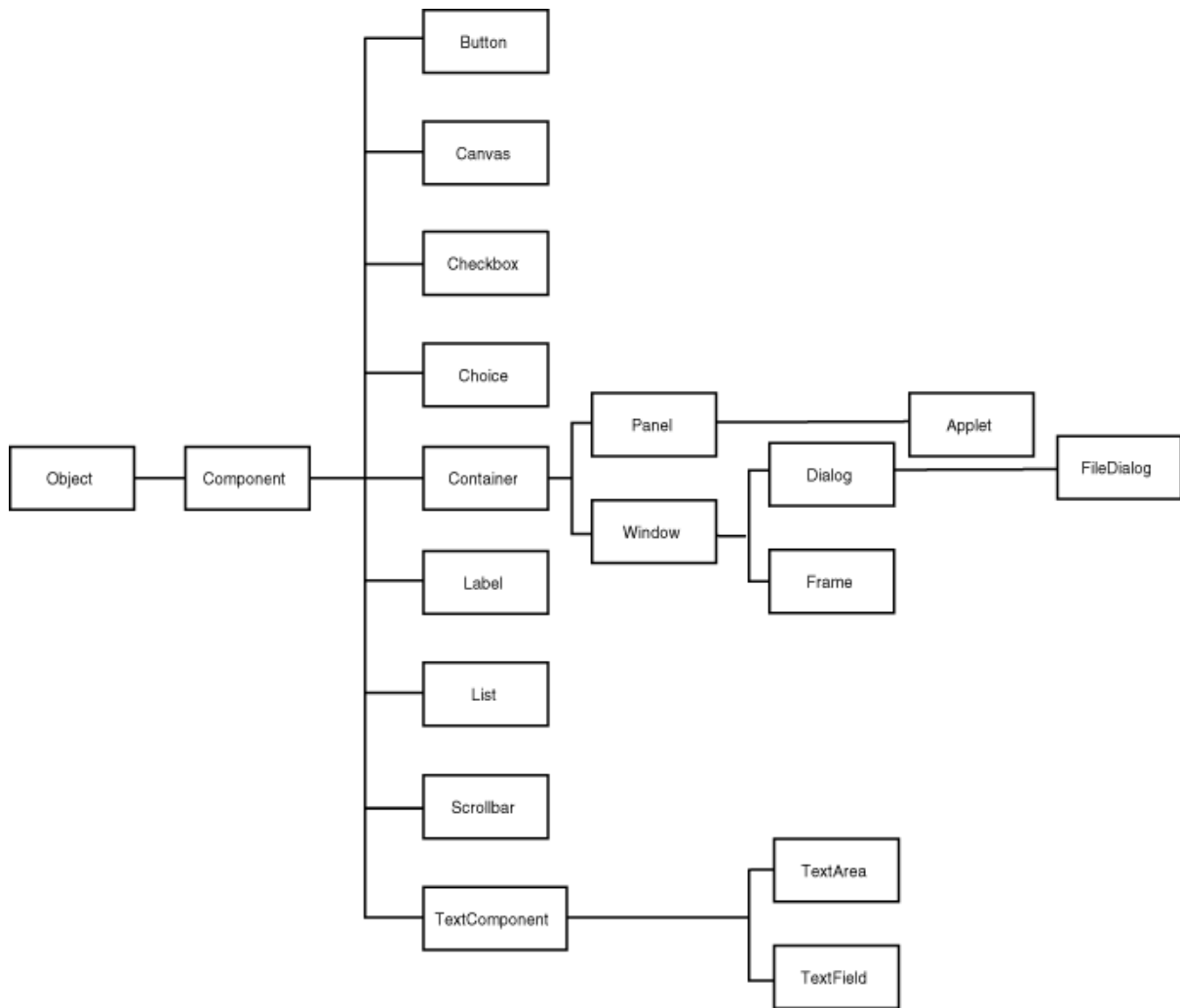
AWT Classes

The AWT framework contains many classes and interfaces using which we can create GUIs.

The AWT package is `java.awt`. Some of the frequently used AWT classes are listed below:

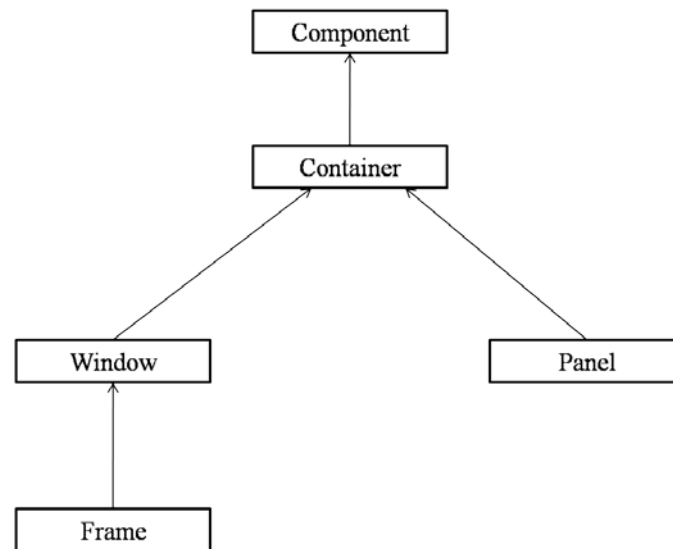
Class	Description
AWTEvent	Encapsulates AWT events and is the root class for all AWT events
BorderLayout	The border layout manager
Button	Creates a push button control
Canvas	A blank window
CardLayout	The card layout manager
Checkbox	Creates a checkbox control
CheckboxGroup	Creates a group of checkboxes (radio buttons)
Choice	Creates a drop down list
Color	Manages colors in GUI programs
Component	An abstract super class for many AWT components
Container	A sub class of container which can hold other components
Dialog	Creates a dialog window
Dimension	Specifies the dimensions (width and height) of an object
FileDialog	Creates a window from which user can select a file
FlowLayout	The flow layout manager
Frame	Creates a standard window
Graphics	Encapsulates the graphics context
GridBagLayout	The grid bag layout manager
GridLayout	The grid layout manager
Image	Encapsulates graphical images
Label	Creates a label control to display static text
List	Creates a list control
Menu	Creates a menu control
MenuBar	Creates a menu bar control
MenuItem	Create a menu item
Panel	A sub class of container which can hold other components
PopupMenu	Creates a pop-up menu
Scrollbar	Creates a scroll bar control
ScrollPane	A container that provides scroll bars for a component
TextArea	Creates a mult-line text control
TextField	Creates a single-line text control
Window	Creates a window with no frame, title bar and title

Various classes in java.awt package are arranged in a hierarchy as shown below:



Component and Container

The Component is the abstract root class for many GUI control classes. Container is a sub class of Component class. The Component and various Container classes are arranged in a hierarchy as shown below:



Component

Component is the abstract class that encapsulates all the properties of a visual component. Except for menus, most of the GUI components are inherited from the Component class.

Container

Container is a sub class of the Component class which can be used to hold other components. A Container object can hold other Containers also. A Container is responsible for laying out (positioning) the components.

Panel

Panel class is a concrete sub class of the Container class. A Panel object is a window without title bar, menu bar and border. Panel is the super class of Applet class and is capable of holding other components or containers.

Window

Window is a sub class of Container class. A Window creates a top-level container which can hold other components or containers.

Frame

Frame is a concrete sub class of Window class. The Frame encapsulates a window. Frame contains a title bar, menu bar, borders and resizable corners. To create stand alone applications in Java, we generally use Frame.

Canvas

Canvas class is derived from the Component class. A Canvas encapsulates a blank window on which we can draw.

Frame Class

The Frame class is used to create standard windows. Following are two Frame class constructors:

`Frame()`

`Frame(String title)`

Following are some of the frequently used methods of Frame class:

`void setSize(int width, int height)` – Used to specify the width and height of the frame window.

`void setSize(Dimension reference)` – Used to specify the dimensions of the frame window.

`Dimension getSize()` – Returns the dimensions of the frame window.

`void setVisible(boolean visibleFlag)` – Makes the frame window visible or non-visible based on the boolean parameter.

`void setTitle(String title)` – Used to set the title of the frame window.

Following is Java code for creating a simple frame window:

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

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

public class MyFrame extends Frame
{
    MyFrame()
    {
        setSize(600, 300);
        setTitle("My Application");
        setVisible(true);
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
            {
                System.exit(0);
            }
        });
    }

    public void paint(Graphics g)
    {
        g.drawString("This is a frame!", 40, 80);
    }

    public static void main(String[] args)
    {
        MyFrame mf = new MyFrame();
    }
}
```

```
}
```

Graphics in AWT

Graphics Class

In GUI applications, we can use Graphics class of java.awt package to create various graphics like lines, rectangles, circles, polygons etc. Let's look at some of the methods available in the Graphics class:

`void drawLine(int startX, startY, endX, endY)` – Used to draw a line between two points.

`void drawRect(int startX, int startY, int width, int height)` – Used to draw a rectangle starting from the top left corner with the given width and height. The coordinates of the top left corner of the rectangle are startX and startY.

`void fillRect(int startX, int startY, int width, int height)` – Used to draw a solid (colored) rectangle with the given parameters.

`void drawRoundRect(int startX, int startY, int width, int height, int xDiam, int yDiam)` – Used to draw a rounded rectangle whose x-diameter and y-diameter of the corners is given by xDiam and yDiam respectively.

`void fillRoundRect(int startX, int startY, int width, int height, int xDiam, int yDiam)` – Used to draw a solid (colored) rounded rectangle whose x-diameter and y-diameter of the corners is given by xDiam and yDiam respectively.

`void drawOval(int startX, int startY, int width, int height)` – Used to draw an ellipse inside an imaginary rectangle whose dimensions are specified by the given parameters. We can get a circle by giving the same value for both width and height.

`void fillOval(int startX, int startY, int width, int height)` – Used to draw a solid (colored) ellipse inside an imaginary rectangle whose dimensions are specified by the given parameters. We can get a circle by giving the same value for both width and height.

`void drawArc(int startX, int startY, int width, int height, int startAngle, int sweepAngle)` – Used to draw an arc inside an imaginary rectangle. The start angle and the sweep angle are specified by using `startAngle` and `sweepAngle` respectively.

`void fillArc(int startX, int startY, int width, int height, int startAngle, int sweepAngle)` – Used to draw a solid (colored) arc inside an imaginary rectangle. The start angle and the sweep angle are specified by using `startAngle` and `sweepAngle` respectively.

`void drawPolygon(int x[], int y[], int numPoints)` – Used to draw a polygon whose x coordinates and y coordinates of the points are specified using the x array and y array respectively. Number of points are specified using `numPoints`.

`void fillPolygon(int x[], int y[], int numPoints)` – Used to draw a solid (colored) polygon whose x coordinates and y coordinates of the points are specified using the x array and y array respectively. Number of points are specified using `numPoints`.

Following Java code demonstrates various methods from the Graphics class:

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

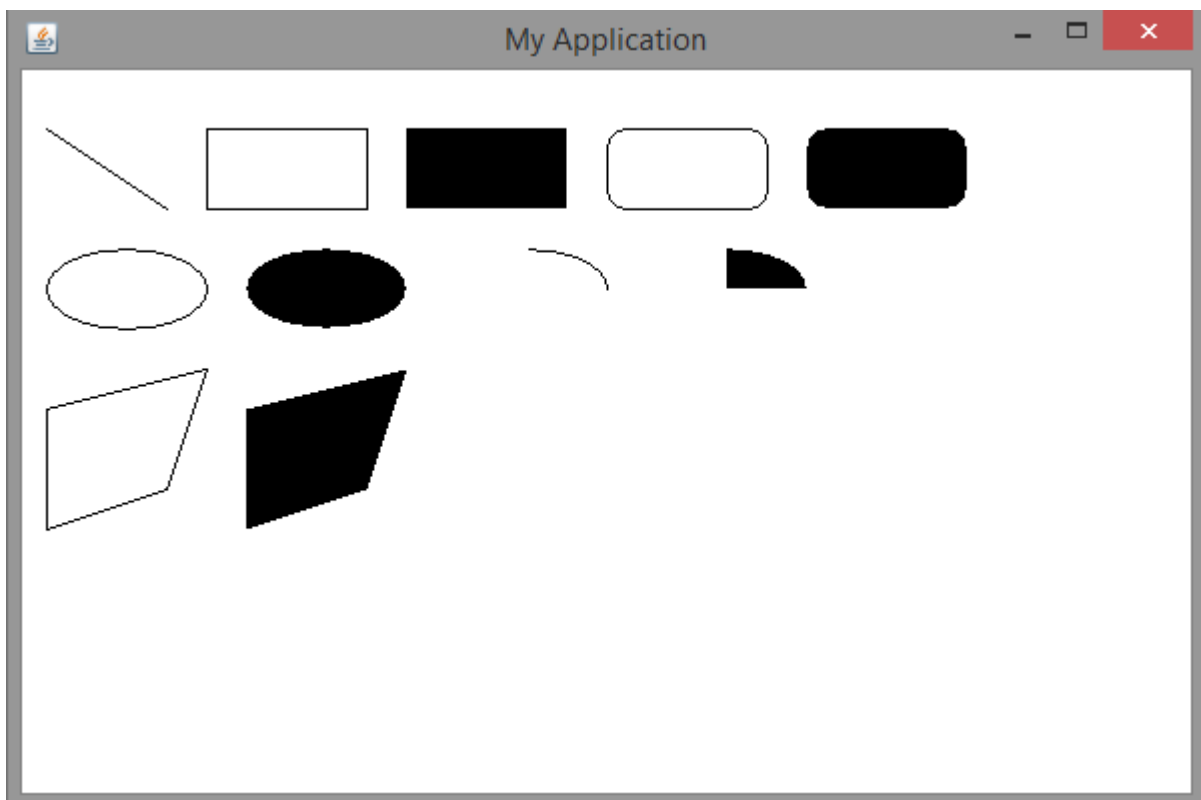
public class MyFrame extends Frame
{
```

```
MyFrame()
{
    setSize(600, 400);
    setTitle("My Application");
    setVisible(true);
    addWindowListener(new WindowAdapter()
    {
        public void windowClosing(WindowEvent we)
        {
            System.exit(0);
        }
    });
}

public void paint(Graphics g)
{
    g.drawLine(20, 60, 80, 100);
    g.drawRect(100, 60, 80, 40);
    g.fillRect(200, 60, 80, 40);
    g.drawRoundRect(300, 60, 80, 40, 20, 20);
    g.fillRoundRect(400, 60, 80, 40, 20, 20);
    g.drawOval(20, 120, 80, 40);
    g.fillOval(120, 120, 80, 40);
    g.drawArc(220, 120, 80, 40, 90, -90);
    g.fillArc(320, 120, 80, 40, 90, -90);
}
```

```
int[] x = {20, 100, 80, 20};  
int[] y = {200, 180, 240, 260};  
g.drawPolygon(x, y, 4);  
int[] fillx = {120, 200, 180, 120};  
int[] filly = {200, 180, 240, 260};  
g.fillPolygon(fillx, filly, 4);  
}  
public static void main(String[] args)  
{  
    MyFrame mf = new MyFrame();  
}  
}
```

Output of the above program is as shown below:



Color Class

To use colors in our GUI applications, AWT provides the Color class. We can create a Color object by using any of the following constructors:

```
Color(int red, int green, int blue)
```

```
Color(int rgbValue)
```

```
Color(float red, float green, float blue)
```

We can get values of red component, green component or blue component of the color by using the following methods:

```
int getRed()
```

```
int getGreen()
```

```
int getBlue()
```

We can apply or get the color of the graphic object by using the following methods:

```
void setColor(Color newColor)
```

```
Color getColor()
```

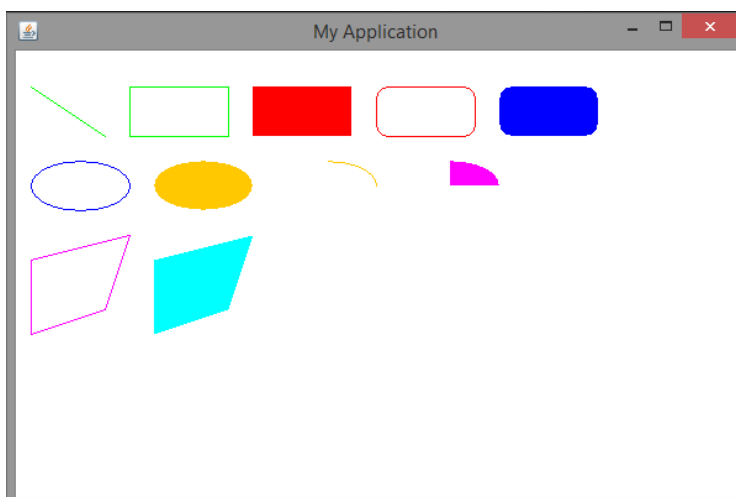
After applying colors the above Java program is as follows:

```
import java.awt.*;  
import java.awt.event.*;  
public class MyFrame extends Frame  
{  
    MyFrame()
```

```
{  
    setSize(600, 400);  
    setTitle("My Application");  
    setVisible(true);  
    addWindowListener(new WindowAdapter()  
    {  
        public void windowClosing(WindowEvent we)  
        {  
            System.exit(0);  
        }  
    }  
    );  
}  
public void paint(Graphics g)  
{  
    g.setColor(Color.GREEN);  
    g.drawLine(20, 60, 80, 100);  
    g.drawRect(100, 60, 80, 40);  
    g.setColor(Color.RED);  
    g.fillRect(200, 60, 80, 40);  
    g.drawRoundRect(300, 60, 80, 40, 20, 20);  
    g.setColor(Color.BLUE);  
    g.fillRoundRect(400, 60, 80, 40, 20, 20);  
    g.drawOval(20, 120, 80, 40);  
    g.setColor(Color.ORANGE);
```

```
g.fillOval(120, 120, 80, 40);
g.drawArc(220, 120, 80, 40, 90, -90);
g.setColor(Color.MAGENTA);
g.fillArc(320, 120, 80, 40, 90, -90);
int[] x = {20, 100, 80, 20};
int[] y = {200, 180, 240, 260};
g.drawPolygon(x, y, 4);
g.setColor(Color.CYAN);
int[] fillx = {120, 200, 180, 120};
int[] filly = {200, 180, 240, 260};
g.fillPolygon(fillx, filly, 4);
}
public static void main(String[] args)
{
    MyFrame mf = new MyFrame();
}
}
```

Output of the above program is as shown below:



AWT Controls

We can add and remove controls to a Container like Applet and Frame using the following methods available in the Container class:

Component add(Component ref)

Component remove(Component ref)

Label

A label is a GUI control which can be used to display static text. Label can be created using the Label class and its constructors which are listed below:

Label()

Label(String str)

Label(String str, int how)

The parameter how specifies the text alignment. Valid values are Label.LEFT, Label.CENTER or Label.RIGHT.

Some of the methods available in the Label class are as follows:

void setText(String str) – To set or assign text to the label.

String getText() – To retrieve the text of a label.

void setAlignment(int how) – To set the alignment of text in a label.

int getAlignment() – To get the alignment of text in a label.

Following example demonstrates working with labels in AWT:

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

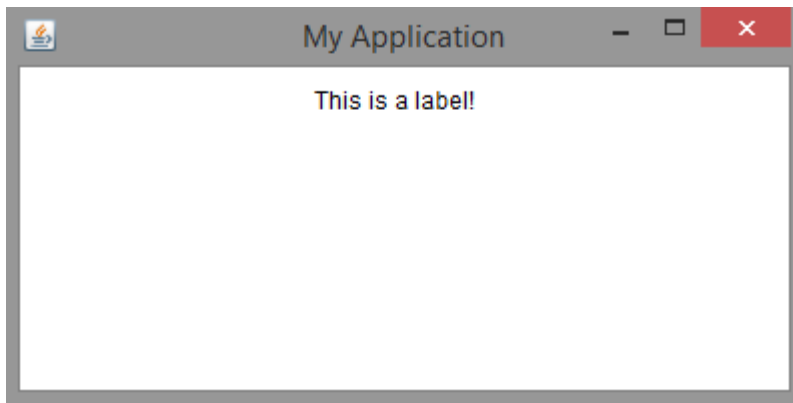
```
import java.awt.event.*;
```

```
public class MyFrame extends Frame
{
    Label myLabel;

    MyFrame()
    {
        setSize(400, 200);
        setTitle("My Application");
        setLayout(new FlowLayout());
        setVisible(true);
        myLabel = new Label("This is a label!");
        add(myLabel);
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
            {
                System.exit(0);
            }
        });
    }

    public static void main(String[] args)
    {
        MyFrame mf = new MyFrame();
    }
}
```


Output of above code is as show below:



Buttons

A push button is the frequently found GUI control. A push button or a button can be created by using the Button class and its constructors which are given below:

Button()

Button(String str)

Some of the methods available in the Button class are as follows:

`void setLabel(String str)` – To set or assign the text to be displayed on the button.

`String getLabel()` – To retrieve the text on the button.

When a button is clicked, it generates an `ActionEvent` which can be handled using the `ActionListener` interface and the event handling method is `actionPerformed()`. If there are multiple buttons we can get the label of the button which was clicked by using the method `getActionCommand()`.

Following Java code demonstrates working with buttons:

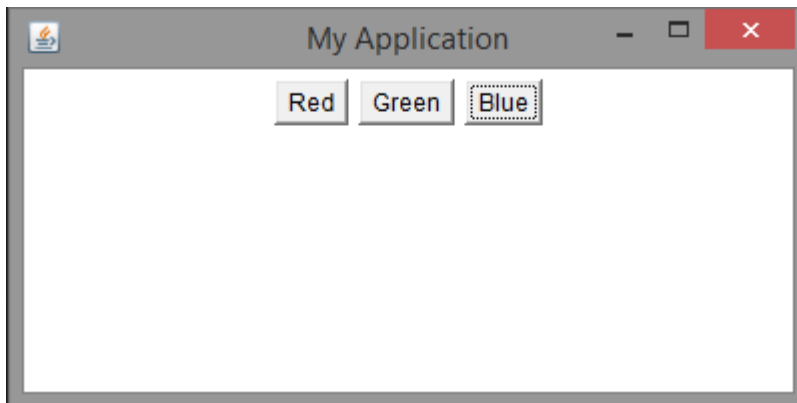
```
import java.awt.*;
```

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

```
public class MyFrame extends Frame
{
    Button b1, b2, b3;
    MyFrame()
    {
        setSize(400, 200);
        setTitle("My Application");
        setLayout(new FlowLayout());
        setVisible(true);
        b1 = new Button("Red");
        b2 = new Button("Green");
        b3 = new Button("Blue");
        add(b1);
        add(b2);
        add(b3);
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
            {
                System.exit(0);
            }
        });
    }
    public static void main(String[] args)
```

```
{  
    MyFrame mf = new MyFrame();  
}  
}
```

Output of the above code is as shown below:



Checkboxes

A checkbox control can be created using the `Checkbox` class and its following constructors:

`Checkbox()`

`Checkbox(String str)`

`Checkbox(String str, boolean on)`

`Checkbox(String str, boolean on, CheckboxGroup cbGroup)`

`Checkbox(String str, CheckboxGroup cbGroup, boolean on)`

Following are various methods available in the `Checkbox` class:

`boolean getState()` – To retrieve the state of a checkbox.

`void setState(boolean on)` – To set the state of a checkbox.

`String getLabel()` – To retrieve the text of a checkbox.

`void setLabel(String str)` – To set the text of a checkbox.

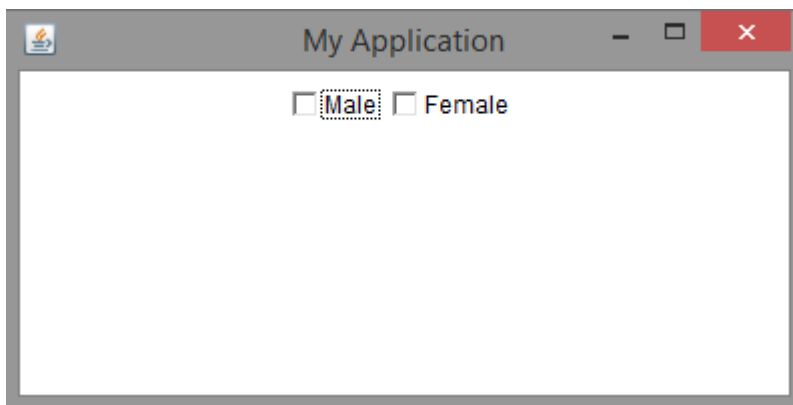
A checkbox when selected or deselected, generates an `ItemEvent` which can be handled using the `ItemListener` interface and the corresponding event handling method is `itemStateChanged()`.

Following code demonstrates working with checkboxes:

```
import java.awt.*;
import java.awt.event.*;
public class MyFrame extends Frame
{
    Checkbox c1, c2;
    MyFrame()
    {
        setSize(400, 200);
        setTitle("My Application");
        setLayout(new FlowLayout());
        setVisible(true);
        c1 = new Checkbox("Male");
        c2 = new Checkbox("Female");
        add(c1);
        add(c2);
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
            {
```

```
        System.exit(0);
    }
}
);
}
public static void main(String[] args)
{
    MyFrame mf = new MyFrame();
}
}
```

Output of the above code is shown below:



In AWT, there is no separate class for creating radio buttons. The difference between a checkbox and radio button is, a user can select one or more checkboxes. Whereas, a user can select only one radio button in a group.

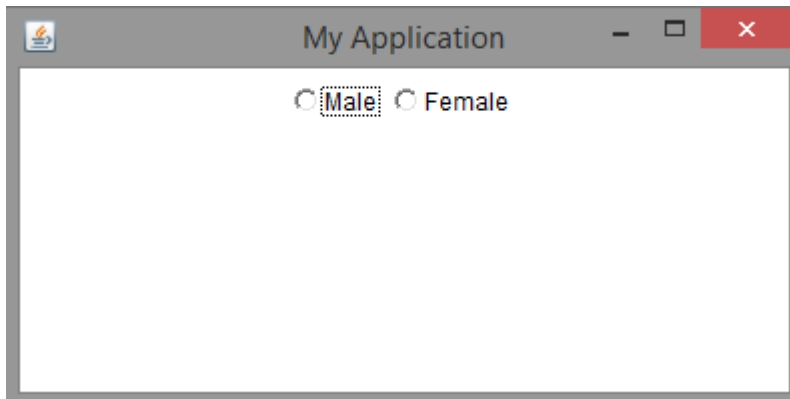
Radio buttons can be create by using `Checkbox` class and `CheckboxGroup` class as shown in the below code:

```
import java.awt.*;
import java.awt.event.*;
public class MyFrame extends Frame
```

```
{  
    Checkbox c1, c2;  
    CheckboxGroup cbg;  
    MyFrame()  
    {  
        setSize(400, 200);  
        setTitle("My Application");  
        setLayout(new FlowLayout());  
        setVisible(true);  
        cbg = new CheckboxGroup();  
        c1 = new Checkbox("Male", cbg, false);  
        c2 = new Checkbox("Female", cbg, false);  
        add(c1);  
        add(c2);  
        addWindowListener(new WindowAdapter()  
        {  
            public void windowClosing(WindowEvent we)  
            {  
                System.exit(0);  
            }  
        }  
        );  
    }  
    public static void main(String[] args)  
    {
```

```
        MyFrame mf = new MyFrame();  
    }  
}
```

Output of the above code is as shown below:



Dropdown Boxes

A drop down box or a combo box contains a list of items (strings). When a user clicks on a drop down box, it pops up a list of items from which user can select a single item.

A drop down box can be created using the Choice class. There is only one constructor in the choice class using which we can create an empty list.

Following are various methods available in Choice class:

`void add(String name)` – To add an item to the drop down list.

`String getSelectedItem()` – To retrieve the item selected by the user.

`int getSelectedItemIndex()` – To retrieve the index of the item selected by the user.

`int getItemCount()` – To retrieve the number of items in the drop down list.

`void select(int index)` – To select an item based on the given index.

`void select(String name)` – To select an item based on the given item name.

`void getItem(int index)` – To retrieve an item at the given index.

Whenever an user selects an item from the drop down box, an `ItemEvent` is generated. It can be handled using the `ItemListener` interface and the event handling method is `itemStateChanged()`.

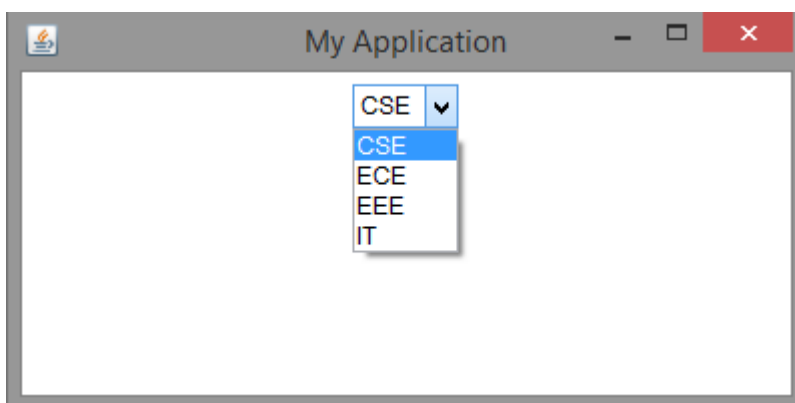
Following code demonstrates working with drop down boxes:

```
import java.awt.*;
import java.awt.event.*;
public class MyFrame extends Frame
{
    Choice myList;
    MyFrame()
    {
        setSize(400, 200);
        setTitle("My Application");
        setLayout(new FlowLayout());
        setVisible(true);
        myList = new Choice();
        myList.add("CSE");
        myList.add("ECE");
        myList.add("EEE");
        myList.add("IT");
        add(myList);
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
```



```
        {  
            System.exit(0);  
        }  
    }  
);  
}  
public static void main(String[] args)  
{  
    MyFrame mf = new MyFrame();  
}  
}
```

Output of the above code is:



List Boxes

A List box contains a list of items among which the user can select one or more items. More than one items in the list box are visible to the user. A list box can be created using the List class along with the following constructors:

List()

List(int numRows)

List(int numRows, boolean multipleSelect)

In the above constructors, `numRows` specifies the number of items to be visible to the user and `multipleSelect` specifies whether the user can select multiple items or not.

When a list item is double clicked, `ActionEvent` is generated. It can be handled with `ActionListener` and the event handling method is `actionPerformed()`. We can get the name of the item using `getActionCommand()` method.

When a list item is selected or deselected, `ItemEvent` is generated. It can be handled with `ItemListener` and the event handling method is `itemStateChanged()`. We can use `getItemSelectable()` method to obtain a reference to the object that raised this event.

Following are some of the methods available in the `List` class:

`void add(String name)` – To add an item to the list box.

`void add(String name, int index)` – To add an item at the specified index in the list box.

`String getSelectedItem()` – To get the item name which is selected by the user.

`int getSelectedItemIndex()` – To get the item index which is selected by the user.

`String[] getSelectedItemNames()` – To retrieve the selected item names by the user.

`int[] getSelectedItemIndexes()` – To retrieve the selected item indexes by the user.

`int getItemCount()` – To retrieve the number of items in the list box.

`void select(int index)` – To select an item based on the given index.

`String getItem(int index)` – To retrieve the item at the given index.

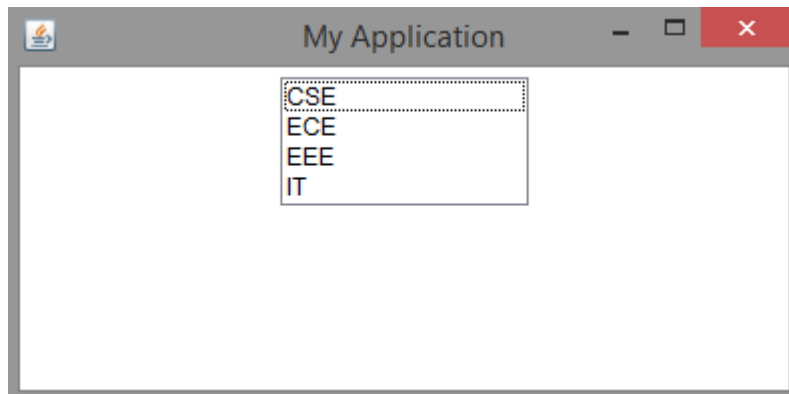
Following code demonstrates working with list boxes:

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

```
public class MyFrame extends Frame
{
    List myList;
    MyFrame()
    {
        setSize(400, 200);
        setTitle("My Application");
        setLayout(new FlowLayout());
        myList = new List();
        myList.add("CSE");
        myList.add("ECE");
        myList.add("EEE");
        myList.add("IT");
        add(myList);
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
            {
                System.exit(0);
            }
        });
        setVisible(true);
    }
    public static void main(String[] args)
```

```
{  
    MyFrame mf = new MyFrame();  
}  
}
```

Output of the above code is shown below:



Text Fields

A text field or text box is a single line text entry control which allows the user to enter a single line of text. a text field can be created using the `TextField` class along with its following constructors:

`TextField()`

`TextField(int numChars)`

`TextField(String str)`

`TextField(String str, int numChars)`

In the above constructors `numChars` specifies the width of the text field, and `str` specifies the initial text in the text field.

When an user hits 'Enter' key on the keyboard in a text field, an `ActionEvent` is generated. It can be handled using `ActionListener` and the event handling method is `actionPerformed()`.

Whenever an user modifies the text in the text field, a `TextEvent` is generated which can be handled using `TextListener` and the event handling method is `textValueChanged()`.

Following are various methods available in `TextField` class:

`String getText()` – Retrieves the text in the text field.

`void setText(String str)` – Assigns or sets text in the text field.

`String getSelectedText()` – Retrieves the selected text in the text field.

`void select(int startindex, int endindex)` – To select the text in text field from `startindex` to `endindex - 1`.

`boolean isEditable()` – To check whether the text field is editable or not.

`void setEditable(boolean canEdit)` – To make a text field editable or non-editable.

`void setEchoChar(char ch)` – To set the echo character of a text field. This is generally used for password fields.

`boolean echoCharIsSet()` – To check whether the echo character for the text field is set or not.

`char getEchoChar()` – To retrieve the current echo character.

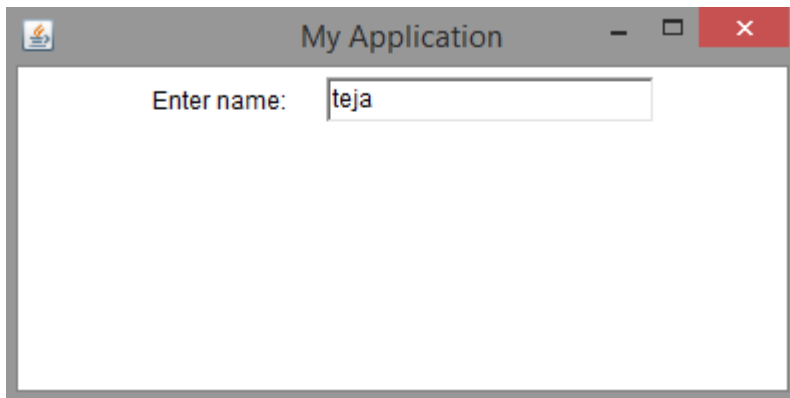
Following code demonstrates working with text fields:

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

public class MyFrame extends Frame
{
    Label myLabel;
```

```
    TextField tf;
    MyFrame()
    {
        setSize(400, 200);
        setTitle("My Application");
        setLayout(new FlowLayout());
        myLabel = new Label("Enter name: ");
        tf = new TextField(20);
        add(myLabel);
        add(tf);
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
            {
                System.exit(0);
            }
        });
        setVisible(true);
    }
    public static void main(String[] args)
    {
        MyFrame mf = new MyFrame();
    }
}
```

Output of the above code is as shown below:



Text Areas

A text area is a multi-line text entry control in which user can enter multiple lines of text. A text area can be created using the `TextArea` class along with the following constructors:

`TextArea()`

`TextArea(int numLines, int numChars)`

`TextArea(String str)`

`TextArea(String str, int numLines, int numChars)`

`TextArea(String str, int numLines, int numChars, int sBars)`

In the above constructors, `numLines` specifies the height of the text area, `numChars` specifies the width of the text area, `str` specifies the initial text in the text area and `sBars` specifies the scroll bars. Valid values of `sBars` can be any one of the following:

`SCROLLBARS_BOTH`

`SCROLLBARS_NONE`

`SCROLLBARS_HORIZONTAL_ONLY`

SCROLLBARS_VERTICAL_ONLY

Following are some of the methods available in the `TextArea` class:

`String getText()` – To retrieve the text in the text area.

`void setText(String str)` – To assign or set the text in a text area.

`String getSelectedText()` – To retrieve the selected text in a text area.

`void select(int startindex, int endindex)` – To select the text in text field from `startindex` to `endindex - 1`.

`boolean isEditable()` – To check whether the text field is editable or not.

`void setEditable(boolean canEdit)` – To make a text field editable or non-editable.

`void append(String str)` – To append the given string to the text in the text area.

`void insert(String str, int index)` – To insert the given string at the specified index.

`void replaceRange(String str, int startIndex, int endIndex)` – To replace the text from `startIndex` to `endIndex - 1` with the given string.

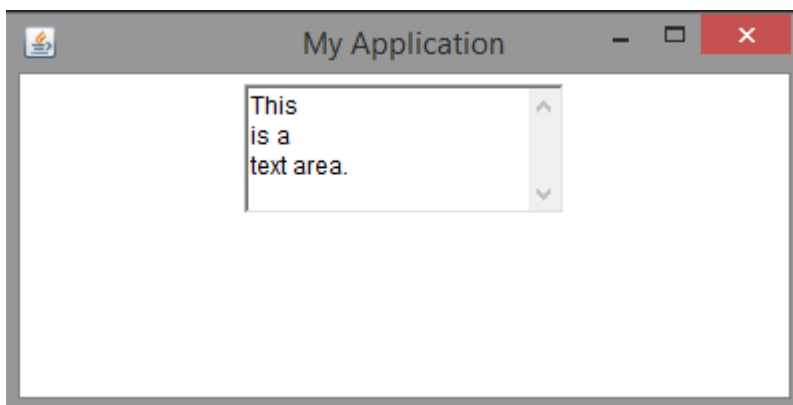
Following code demonstrates working with text areas:

```
import java.awt.*;
import java.awt.event.*;
public class MyFrame extends Frame
{
    TextArea ta;
    MyFrame()
    {
        setSize(400, 200);
        setTitle("My Application");
    }
}
```



```
        setLayout(new FlowLayout());
        ta = new TextArea(3, 20);
        add(ta);
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
            {
                System.exit(0);
            }
        });
        setVisible(true);
    }
    public static void main(String[] args)
    {
        MyFrame mf = new MyFrame();
    }
}
```

Output of the above code is shown below:



Layout Managers

A layout manager is one which automatically manages the arrangement of various of components in a container. Each container will have a default layout manager. Default layout managers for some of the container classes are given below:

Panel – Flow Layout

JPanel – Flow Layout

Applet – Flow Layout

JApplet – Border Layout

Frame – Border Layout

JFrame – Border Layout

A layout manager is an instance of that class which implements the `LayoutManager` interface. The layout manager can be set by using the `setLayout()` method whose general form is as follows:

```
void setLayout(LayoutManager layoutObj)
```

We can manually arrange the position of each component (not recommended) by passing null to `setLayout()` method and by using `setBounds()` method on each component. Different layout managers available in AWT are:

1. `FlowLayout`
2. `BorderLayout`
3. `GridLayout`
4. `CardLayout`

5. GridBagLayout

FlowLayout Manager

The flow layout manager arranges the components one after another from left-to-right and top-to-bottom manner. The flow layout manager gives some space between components.

Flow layout manager instance can be created using any one of the following constructors:

`FlowLayout()`

`FlowLayout(int how)`

`FlowLayout(int how, int hspace, int vspace)`

In the above constructors, `how` specifies the alignment, `hspace` specifies horizontal space, and `vspace` specifies vertical space. Valid values for alignment are as follows:

`FlowLayout.LEFT`

`FlowLayout.CENTER`

`FlowLayout.RIGHT`

`FlowLayout.LEADING`

`FlowLayout.TRAILING`

Following code demonstrates working with `FlowLayout`:

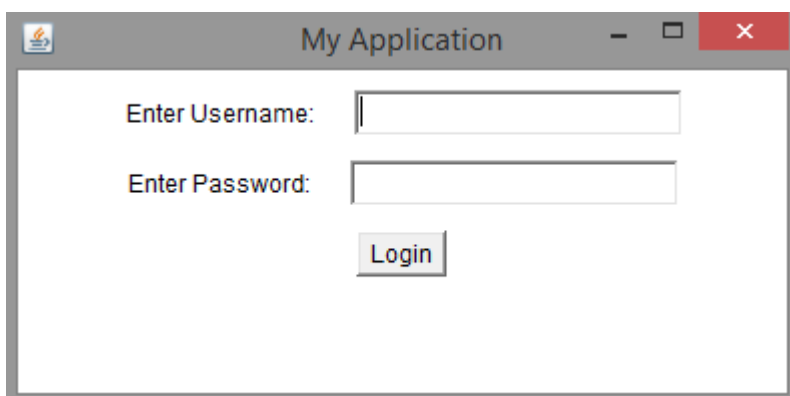
```
import java.awt.*;
import java.awt.event.*;
public class MyFrame extends Frame
{
```

```
Panel p1, p2, p3;
Label l1, l2;
TextField t1, t2;
Button b;
MyFrame()
{
    setSize(400, 200);
    setTitle("My Application");
    setLayout(new FlowLayout());
    p1 = new Panel();
    l1 = new Label("Enter Username: ");
    t1 = new TextField(20);
    p1.add(l1);
    p1.add(t1);
    p1.setPreferredSize(new Dimension(400, 30));
    p2 = new Panel();
    l2 = new Label("Enter Password: ");
    t2 = new TextField(20);
    t2.setEchoChar('*');
    p2.add(l2);
    p2.add(t2);
    p2.setPreferredSize(new Dimension(400, 30));
    p3 = new Panel();
    b = new Button("Login");
    p3.add(b);
```

```
        add(p1);
        add(p2);
        add(p3);
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
            {
                System.exit(0);
            }
        });
        setVisible(true);
    }

    public static void main(String[] args)
    {
        MyFrame mf = new MyFrame();
    }
}
```

Output of the above code is as shown below:



BorderLayout Manager

The border layout manager divides the container area into five regions namely: north, south, east, west, and center. Default region is center. You have to be careful with border layout as controls might be stacked over one another. Border layout instance can be created by using one of the below constructors:

`BorderLayout()`

`BorderLayout(int hspace, int vspace)`

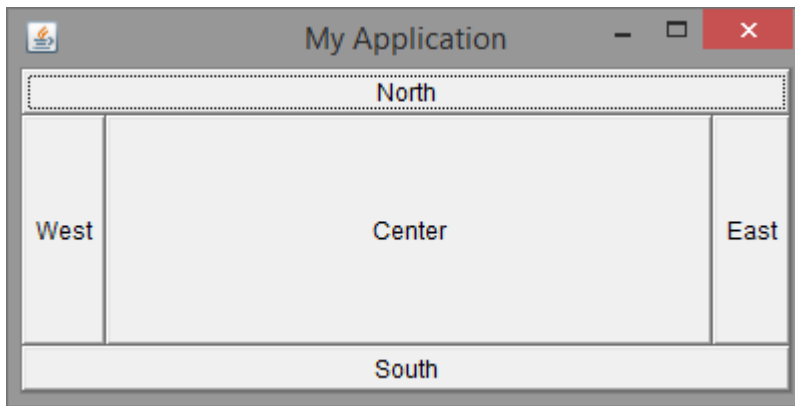
In the above constructors, *hspace* signifies horizontal space between components and *vspace* signifies vertical space between components.

Following code demonstrates working with BorderLayout:

```
import java.awt.*;
import java.awt.event.*;
public class MyFrame extends Frame
{
    Button bnorth, bsouth, beast, bwest, bcenter;
    MyFrame()
    {
        setSize(400, 200);
        setTitle("My Application");
        bnorth = new Button("North");
        bsouth = new Button("South");
        beast = new Button("East");
        bwest = new Button("West");
```

```
bcenter = new Button("Center");
add(bnorth, BorderLayout.NORTH);
add(bsouth, BorderLayout.SOUTH);
add(beast, BorderLayout.EAST);
add(bwest, BorderLayout.WEST);
add(bcenter, BorderLayout.CENTER);
addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent we)
    {
        System.exit(0);
    }
});
setVisible(true);
}
public static void main(String[] args)
{
    MyFrame mf = new MyFrame();
}
}
```

Output of the above code is as shown below:



GridLayout Manager

The grid layout manager arranges the components in a 2-dimensional grid. While creating the instance of GridLayout, we can specify the number of rows and columns in the grid. Care must be taken with the number of cells in the grid and the number of components being added to the grid. If they don't match, we might get unexpected output.

An instance of GridLayout can be created using one of the following constructors:

`GridLayout()`

`GridLayout(int numRows, int numCols)`

`GridLayout(int numRows, int numCols, int hspace, int vspace)`

In the above constructors, `numRows` and `numCols` specifies the number of rows and columns in the grid, `hspace` and `vsapce` specifies the horizontal space and vertical space between the components.

Following code demonstrates working with GridLayout:

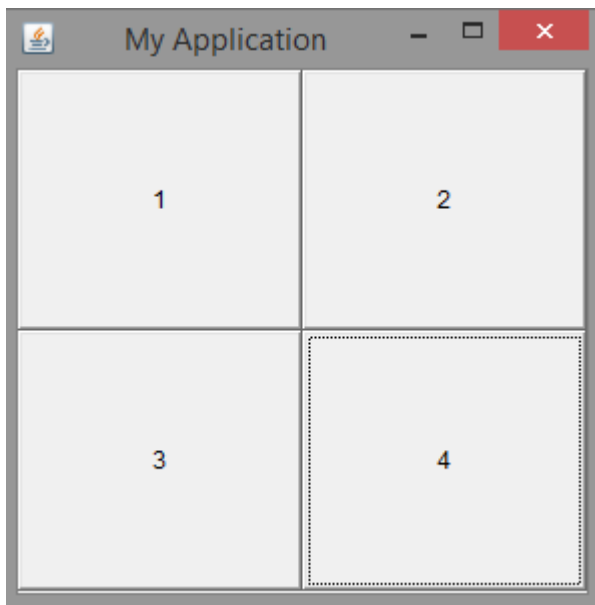
```
import java.awt.*;
import java.awt.event.*;
public class MyFrame extends Frame
```



```
{  
    Button b1, b2, b3, b4;  
    MyFrame()  
    {  
        setSize(300, 300);  
        setTitle("My Application");  
        setLayout(new GridLayout(2, 2));  
        b1 = new Button("1");  
        b2 = new Button("2");  
        b3 = new Button("3");  
        b4 = new Button("4");  
        add(b1);  
        add(b2);  
        add(b3);  
        add(b4);  
        addWindowListener(new WindowAdapter()  
        {  
            public void windowClosing(WindowEvent we)  
            {  
                System.exit(0);  
            }  
        }  
    );  
        setVisible(true);  
    }  
}
```

```
public static void main(String[] args)
{
    MyFrame mf = new MyFrame();
}
}
```

Output of the above code is as shown below:



CardLayout Manager

The card layout manager allows the user to create a deck of cards. Each card can contain different components. At any instant only one card in the deck can be displayed.

To implement card layout, we must take a panel which acts as the container for other cards. Each card in turn can be a panel which can contain different components. Components will be added to the respective cards (panels) and all the cards will be finally added to the deck (container panel).

The card layout can be instantiated using any one of the following constructors:

CardLayout()

CardLayout(int hspace, int vspace)

When adding the cards (panels) to the deck, the following `add()` method can be used:
`void add(Component panelRef, Object name)`

In the above syntax, `name` is a string which represents the name of the card (panel). After adding all the cards (panels) to the deck, we can navigate through the cards using the following methods available in `CardLayout` class:

```
void first(Container deck)
```

```
void last(Container deck)
```

```
void next(Container deck)
```

```
void previous(Container deck)
```

```
void show(Container deck, String cardName)
```

Following code demonstrates working with `CardLayout`:

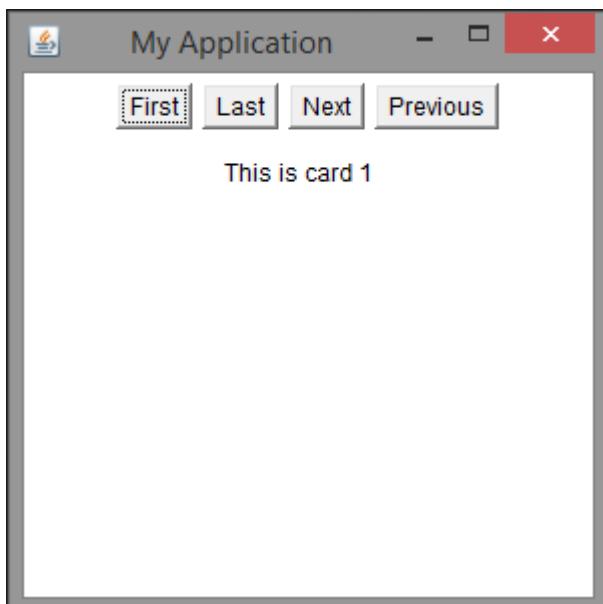
```
import java.awt.*;  
import java.awt.event.*;  
public class MyFrame extends Frame implements ActionListener  
{  
    Button first, last, next, prev;  
    Panel bpanel, deck;  
    Label l1, l2, l3;  
    Panel card1, card2, card3;  
    CardLayout cl;
```

```
MyFrame()
{
    setSize(300, 300);
    setTitle("My Application");
    first = new Button("First");
    last = new Button("Last");
    next = new Button("Next");
    prev = new Button("Previous");
    first.addActionListener(this);
    last.addActionListener(this);
    next.addActionListener(this);
    prev.addActionListener(this);
    bpanel = new Panel();
    bpanel.add(first);
    bpanel.add(last);
    bpanel.add(next);
    bpanel.add(prev);
    add(bpanel, BorderLayout.NORTH);
    cl = new CardLayout();
    l1 = new Label("This is card 1");
    l2 = new Label("This is card 2");
    l3 = new Label("This is card 3");
    card1 = new Panel();
    card2 = new Panel();
    card3 = new Panel();
```

```
card1.add(l1);
card2.add(l2);
card3.add(l3);
deck = new Panel();
deck.setLayout(cl);
deck.add(card1, "card1");
deck.add(card2, "card2");
deck.add(card3, "card3");
add(deck, BorderLayout.CENTER);
addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent we)
    {
        System.exit(0);
    }
});
setVisible(true);
}
public void actionPerformed(ActionEvent ae)
{
    if(ae.getActionCommand().equals("First"))
        cl.first(deck);
    else if(ae.getActionCommand().equals("Last"))
        cl.last(deck);
}
```

```
        else if(ae.getActionCommand().equals("Next"))
            cl.next(deck);
        else
            cl.previous(deck);
    }
    public static void main(String[] args)
    {
        MyFrame mf = new MyFrame();
    }
}
```

Output of the above code is as shown below:



GridBagLayout Manager

The grid bag layout manager can be used to create an uneven grid i.e., number of columns in each row can differ. Also the size of components within a cell can be different.

The location and size of each component are specified by a set of constraints that are contained in an object of type `GridBagConstraints`. These constraints include height, width, and placement of a component.

The general process to work with `GridBagLayout` is, first set the layout of the container to `GridBagLayout`, then set the constraints for each component using `GridBagConstraints`, and then add each component to the container.

`GridBagConstraints` class specifies the following fields which can be used to set constraints on each component:

Field	Purpose
int anchor	Specifies the location of a component within a cell. The default is <code>GridBagConstraints.CENTER</code> .
int fill	Specifies how a component is resized if the component is smaller than its cell. Valid values are <code>GridBagConstraints.NONE</code> (the default), <code>GridBagConstraints.HORIZONTAL</code> , <code>GridBagConstraints.VERTICAL</code> , <code>GridBagConstraints.BOTH</code> .
int gridheight	Specifies the height of component in terms of cells. The default is 1.
int gridwidth	Specifies the width of component in terms of cells. The default is 1.
int gridx	Specifies the X coordinate of the cell to which the component will be added. The default value is <code>GridBagConstraints.RELATIVE</code> .
int gridy	Specifies the Y coordinate of the cell to which the component will be added. The default value is <code>GridBagConstraints.RELATIVE</code> .
Insets insets	Specifies the insets. Default insets are all zero.
int ipadx	Specifies extra horizontal space that surrounds a component within a cell. The default is 0.
int ipady	Specifies extra vertical space that surrounds a component within a cell. The default is 0.
double weightx	Specifies a weight value that determines the horizontal spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window.
double weighty	Specifies a weight value that determines the vertical spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window.

Following code can be used to work with `GridBagLayout`:

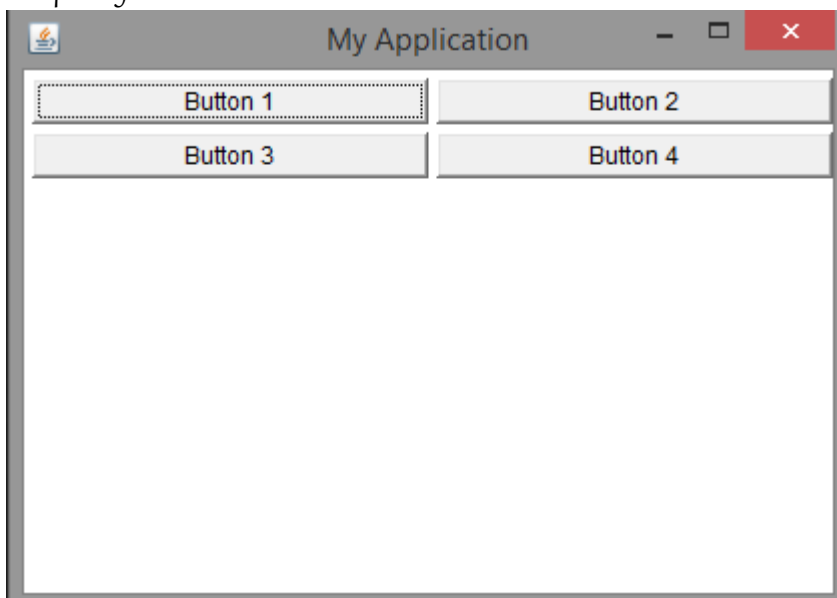
```
import java.awt.*;
import java.awt.event.*;
public class MyFrame extends Frame
```

```
{  
    Button b1, b2, b3, b4;  
    MyFrame()  
    {  
        setSize(420, 300);  
        setTitle("My Application");  
        GridBagLayout gbl = new GridBagLayout();  
        GridBagConstraints gbc = new GridBagConstraints();  
        setLayout(gbl);  
        b1 = new Button("Button 1");  
        b2 = new Button("Button 2");  
        b3 = new Button("Button 3");  
        b4 = new Button("Button 4");  
        gbc.weightx = 1.0;  
        gbc.ipadx = 200;  
        gbc.insets = new Insets(4, 4, 0, 0);  
        gbc.anchor = GridBagConstraints.NORTHWEST;  
        gbc.gridwidth = GridBagConstraints.RELATIVE;  
        gbl.setConstraints(b1, gbc);  
        gbc.gridwidth = GridBagConstraints.REMAINDER;  
        gbl.setConstraints(b2, gbc);  
        gbc.weighty = 1.0;  
        gbc.gridwidth = GridBagConstraints.RELATIVE;  
        gbl.setConstraints(b3, gbc);  
        gbc.gridwidth = GridBagConstraints.REMAINDER;  
        gbl.setConstraints(b4, gbc);  
        add(b1);  
        add(b2);  
        add(b3);  
    }  
}
```



```
        add(b4);
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
            {
                System.exit(0);
            }
        }
    );
    setVisible(true);
}
public static void main(String[] args)
{
    MyFrame mf = new MyFrame();
}
}
```

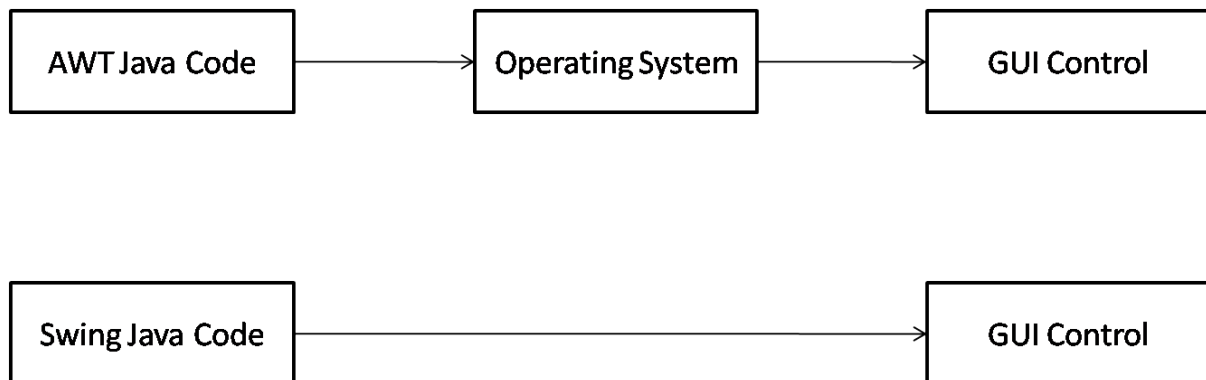
Output of the above code is as shown below:



Introduction to Swings

Although Java already provides AWT for creating GUIs, due to several limitations of AWT, they developed swings package. One limitation of AWT is, AWT takes the help of native operating system to create a GUI control like button, checkbox etc. From operating system to operating system the look and feel of the component might vary. This contradicts with Java's motto: Write Once Run Anywhere.

Java provides javax.swing package which does not need any help of the native operating system for creating GUI controls. Hence programs created using swings are portable.



Features of Swings

Swings offers the following features:

Lightweight Components: The components created using swings package doesn't need the help of native operating system. Swings is a part of Java Foundation Classes (JFC) which are purely developed in Java. Also swing components require less memory and CPU cycles than their counterpart AWT components. Hence swing components are called lightweight components.

Pluggable Look and Feel: Swings supports pluggable look and feel i.e., the appearance of a component can be separated from how the component behaves. This enables programmers to assign different look (themes) for the same component without changing its behavior.

Default look and feel of Java is metal. Java 8 also provides Nimbus look and feel. In windows operating system, windows look and feel is also available.

Differences between AWT and Swings

Following are the differences between AWT and Swings packages:

AWT	Swing
AWT components are heavyweight components	Swing components are lightweight components
AWT doesn't support pluggable look and feel	Swing supports pluggable look and feel
AWT programs are not portable	Swing programs are portable
AWT is old framework for creating GUIs	Swing is new framework for creating GUIs
AWT components require java.awt package	Swing components require javax.swing package
AWT supports limited number of GUI controls	Swing provides advanced GUI controls like Jtable, JTabbedPane etc
More code is needed to implement AWT controls functionality	Less code is needed to implement swing controls functionality
AWT doesn't follow MVC	Swing follows MVC

Model-View-Controller (MVC) Connection

MVC is an architectural pattern widely used in development of software. Every GUI control or component contains three aspects:

1. The way the component looks (appearance) when rendered on the screen.
2. The way the component reacts to the user.
3. The state information associated with the component.

MVC can be used to separate the three aspects mentioned above. By separating them, each aspect can be modified independently without any changes to the remaining aspects.

In MVC terminology, model corresponds to the state information of a component. For example, a button might contain a field that might represent if it is pressed or released. The view corresponds to the look or appearance of the component when rendered on the screen. For example, a button is displayed as a rectangle. The controller determines how the component reacts to the user events. For example, when user presses a button, a new frame can be displayed.

The controller changes the model and the model notifies the state change to the view to be updated accordingly. Java Swings uses a modified version of MVC architecture, in which view and controller are combined into single entity called `UIDelegate`. This modified architecture is called as Model-Delegate architecture or Separable Model architecture.

Components and Containers

A swing GUI contains two main elements: components and containers. A component is an individual control like a button or label. A container is an object which can hold other components and containers.

Components

Swing components are derived from the class `JComponent` (except the four top-level containers). `JComponent` supports pluggable look and feel and supports the functionality common for all the components. `JComponent` class inherits the AWT classes `Container` and `Component`.

All swing components are represented as classes and are present in `javax.swing` package. Some of the components available in the Swings package are listed below:

<code>JApplet</code>	<code>JButton</code>	<code>JCheckBox</code>	<code>JCheckBoxMenuItem</code>
<code>JColorChooser</code>	<code>JComboBox</code>	<code>JComponent</code>	<code>JDesktopPane</code>
<code>JDialog</code>	<code>JEditorPane</code>	<code>JFileChooser</code>	<code>JFormattedTextField</code>
<code>JFrame</code>	<code>JInternalFrame</code>	<code>JLabel</code>	<code>JLayer</code>
<code>JLayeredPane</code>	<code>JList</code>	<code>JMenu</code>	<code>JMenuBar</code>
<code>JMenuItem</code>	<code>JOptionPane</code>	<code>JPanel</code>	<code>JPasswordField</code>
<code>JPopupMenu</code>	<code>JProgressBar</code>	<code>JRadioButton</code>	<code>JRadioButtonMenuItem</code>
<code>JRootPane</code>	<code>JScrollBar</code>	<code>JScrollPane</code>	<code>JSeparator</code>
<code>JSlider</code>	<code>JSpinner</code>	<code>JSplitPane</code>	<code>JTabbedPane</code>
<code>JTable</code>	<code>JTextArea</code>	<code>JTextField</code>	<code>JTextPane</code>
<code>JToggleButton</code>	<code>JToolBar</code>	<code>JToolTip</code>	<code>JTree</code>
<code>JViewport</code>	<code>JWindow</code>		

Containers

Swing provides two types of containers. First are top-level containers also called heavyweight containers. These are at the top in the containment hierarchy. The top-level containers are

JFrame, JApplet, JWindow, and JDialog. All the containers inherit the AWT classes Component and Container.

Second are the lightweight containers which can be nested inside top-level containers. Lightweight containers are inherited from the class JComponent. Example of a lightweight container is JPanel.

Top-level Container Panes

Each top-level container defines a set of panes. At the top of the hierarchy is an instance of JRootPane. It is a lightweight container whose purpose is to manage other panes.

The panes that make up the JRootPane are: glass pane, layered pane, and content pane. The glass pane is a top level pane which lies above all components and covers all other panes. It can be used to manage mouse events over the entire container or to paint over any other component. Glass pane is a transparent instance of JPanel.

The layered pane is an instance of JLayeredPane which allows the components to be assigned a depth value (Z-order). The layered pane holds the content pane and the menu bar.

The content pane is the pane to which the visual components are added. The content pane is an opaque instance of JPanel.

Following are some of the swing packages:

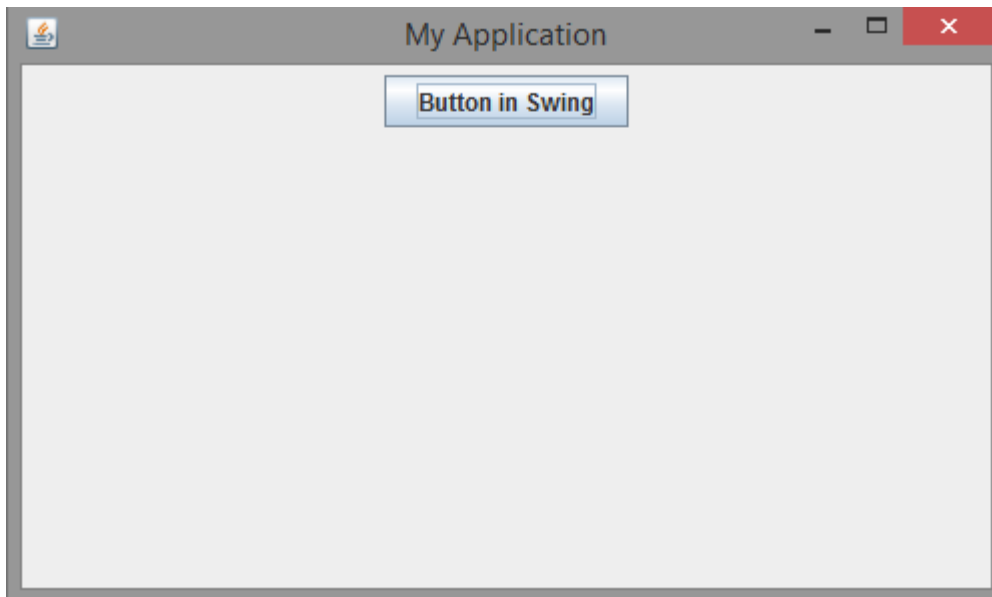
javax.swing	javax.swing.plaf.basic	javax.swing.text
javax.swing.border	javax.swing.plaf.metal	javax.swing.text.html
javax.swing.colorchooser	javax.swing.plaf.multi	javax.swing.text.html.parser
javax.swing.event	javax.swing.plaf.nimbus	javax.swing.text.rtf
javax.swing.filechooser	javax.swing.plaf.synth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

Following code demonstrates a simple GUI application using JFrame:

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

public class MyFrame extends JFrame
{
    JButton b;
    MyFrame()
    {
        setSize(500, 300);
        setTitle("My Application");
        setLayout(new FlowLayout());
        b = new JButton("Button in Swing");
        add(b);
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public static void main(String[] args)
    {
        MyFrame mf = new MyFrame();
    }
}
```

Output of the above code is as shown below:



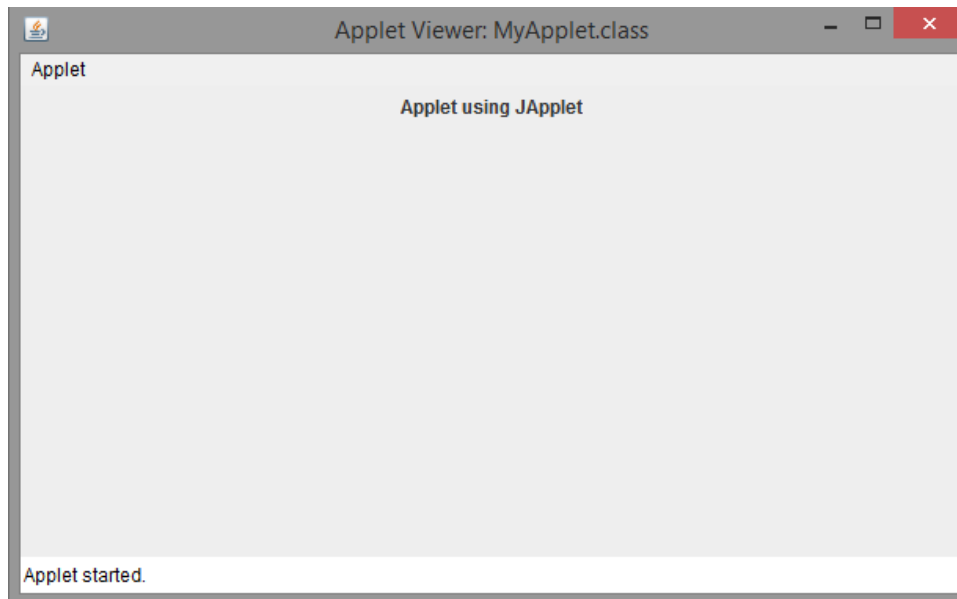
Following code demonstrates a simple GUI application using JApplet:

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

public class MyApplet extends JApplet
{
    JLabel label;

    public void init()
    {
        setSize(600,300);
        setLayout(new FlowLayout());
        label = new JLabel("Applet using JApplet");
        add(label);
    }
}
```


Output of the above code is as shown below:



Swing Controls

Labels

The `JLabel` class is used to display a label i.e., static text. A `JLabel` object can be created using any one of the following constructors:

`JLabel(Icon icon)`

`JLabel(String str)`

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

In the above constructors icon is used to specify an image to be displayed as a label. Icon is a predefined interface which is implemented by the ImageIcon class. str is used to specify the text to be displayed in the label and align is used to specify the alignment of the text.

Some of the methods available in JLabel class are as follows:

void setText(String str) – To set the text of the label

String getText() – To get the text of the label

void setIcon(Icon icon) – To display an image in the label

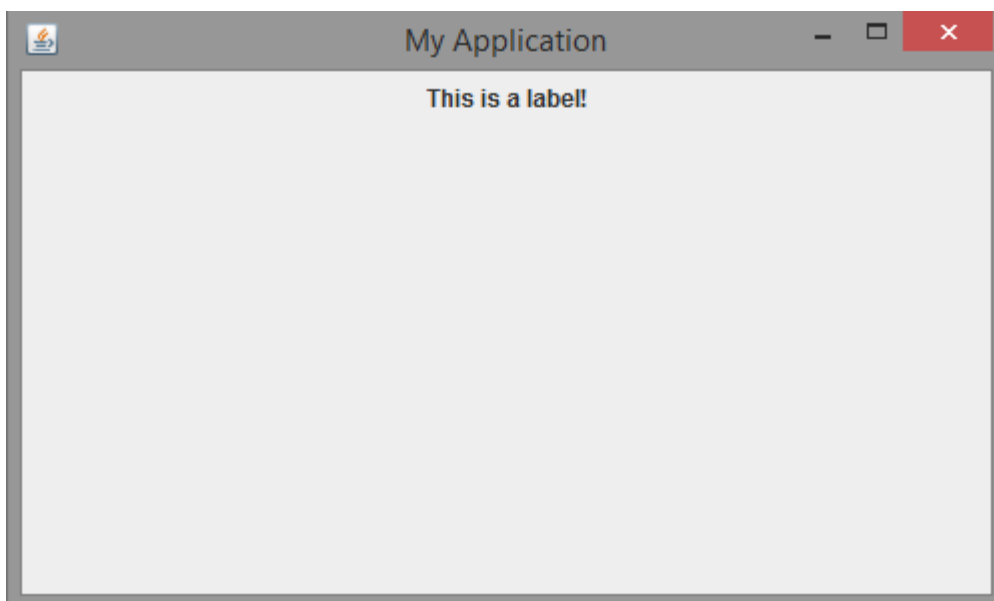
Icon getIcon() – To retrieve the image displayed in the label

Following code demonstrates working with a JLabel:

```
import java.awt.*;
import javax.swing.*;
public class MyFrame extends JFrame
{
    JLabel l;
    MyFrame()
    {
        setSize(500, 300);
        setTitle("My Application");
        setLayout(new FlowLayout());
        l = new JLabel("This is a label!");
        add(l);
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

```
    }  
    public static void main(String[] args)  
    {  
        MyFrame mf = new MyFrame();  
    }  
}
```

Output of the above code is as follows:



Buttons

The JButton class is used to display a push button. The JButton class implements the abstract class AbstractButton which provides the following methods:

`void setDisableIcon(Icon di)` – To set the icon to be displayed when button is disabled

`void setPressedIcon(Icon pi)` – To set the icon to be displayed when button is pressed

`void setSelectedIcon(Icon si)` – To set the icon to be displayed when button is selected

`void setRolloverIcon(Icon ri)` – To set the icon to be displayed when the button is rolled over

`void setText(String str)` – To set the text to be displayed on the button

`String getText()` – To retrieve the text displayed on the button

A `JButton` object can be created using any one of the following constructors:

`JButton(Icon icon)`

`JButton(String str)`

`JButton(String str, Icon icon)`

In the above constructors, `icon` specifies the image to be displayed as button and `str` specifies the text to be displayed on the button.

`JButton` objects raise `ActionEvent` when the button is clicked. It can be handled by implementing `ActionListener` interface and the event handling method is `actionPerformed()`.

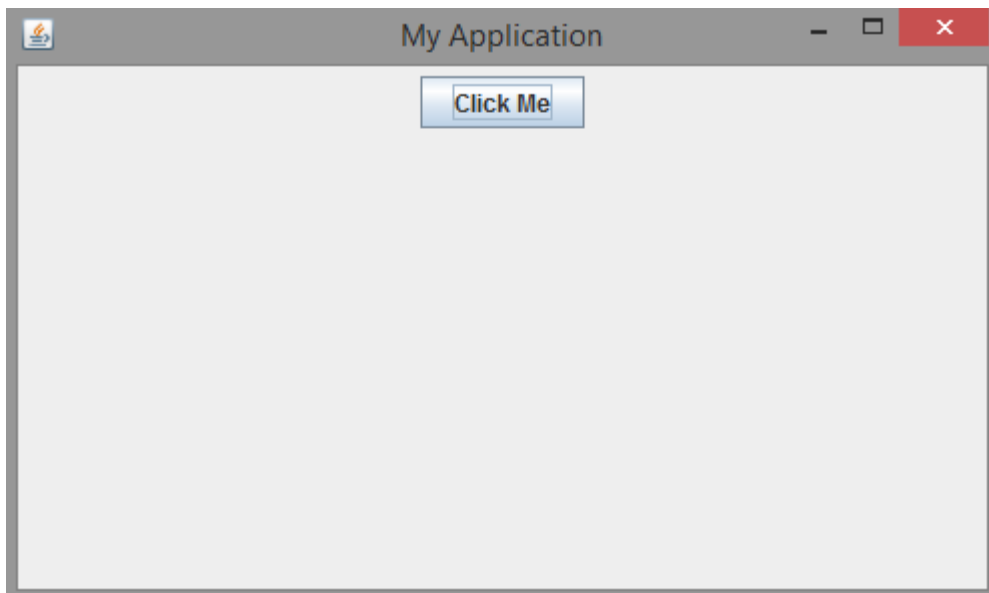
Following code demonstrates working with `JButton`:

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

public class MyFrame extends JFrame
{
    JButton b;
    MyFrame()
    {
        setSize(500, 300);
    }
}
```

```
setTitle("My Application");  
setLayout(new FlowLayout());  
b = new JButton("Click Me");  
add(b);  
setVisible(true);  
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
}  
public static void main(String[] args)  
{  
    MyFrame mf = new MyFrame();  
}  
}
```

Output of the above code is as shown below:



Toggle Buttons

The `JToggleButton` class creates a toggle button. It is different from a push button in the sense, a toggle button toggles between two states. When a user clicks on a toggle button it remains in the pushed state and again when the button is clicked it reverts back to its normal state.

The `JToggleButton` class inherits the abstract class `AbstractButton`. A `JToggleButton` object can be created using the following constructor:

`JToggleButton(String str)`

In the above constructor `str` is the text to be displayed on the button. A toggle button can raise both `ActionEvent` and `ItemEvent`.

Following code demonstrates working with `JToggleButton`:

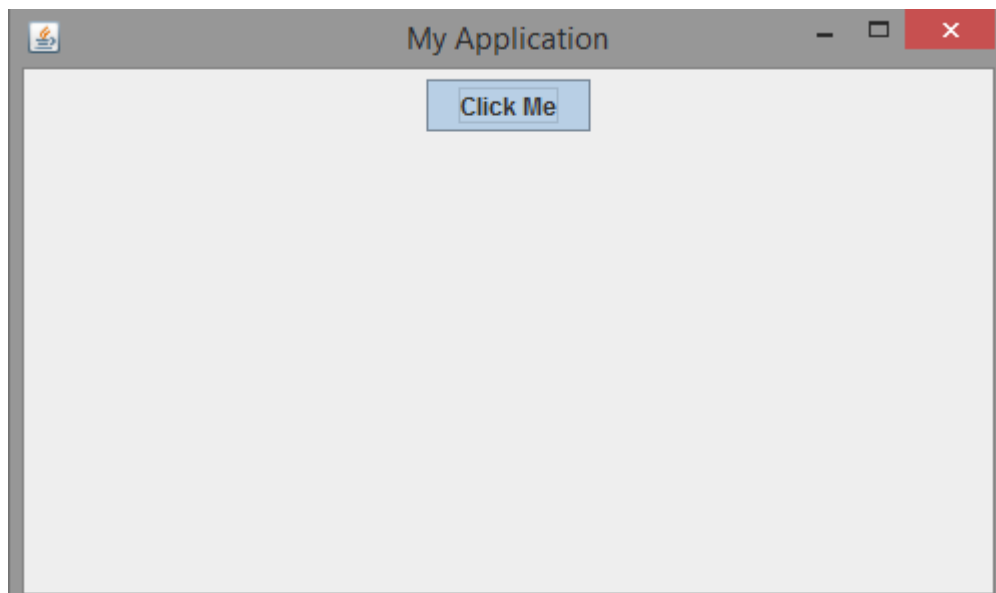
```
import java.awt.*;
import javax.swing.*;

public class MyFrame extends JFrame
{
    JToggleButton b;

    MyFrame()
    {
        setSize(500, 300);
        setTitle("My Application");
        setLayout(new FlowLayout());
        b = new JToggleButton("Click Me");
        add(b);
    }
}
```

```
        setVisible(true);  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    }  
    public static void main(String[] args)  
    {  
        MyFrame mf = new MyFrame();  
    }  
}
```

Output of the above code is as shown below:



Toggle button in the above output is clicked.

Check boxes

A check box can be created using the `JCheckBox` class which inherits the `JToggleButton` class. A `JCheckBox` object can be created using the following constructor:

```
JCheckBox(String str)
```

In the above constructor *str* specifies the string that is displayed by the side of box. A check box raises *ItemEvent* when the user selects or deselects it. This event can be handled by implementing the *ItemListener* interface and the event handling method is *itemStateChanged()*. We can use *isSelected()* method from *JCheckBox* class to know whether a check box is selected or not.

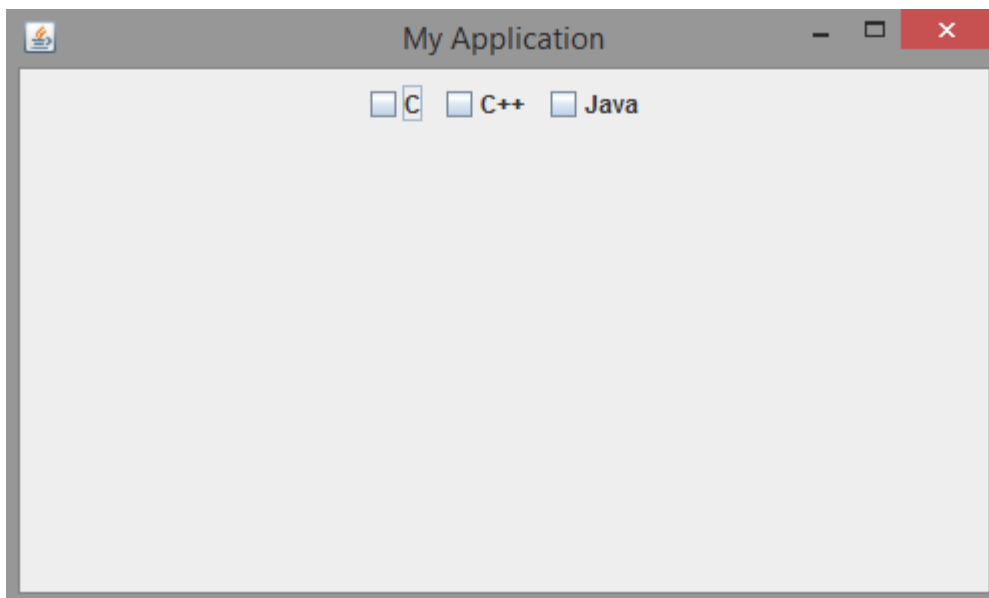
Following code demonstrates working with *JCheckBox*:

```
import java.awt.*;
import javax.swing.*;
public class MyFrame extends JFrame
{
    JCheckBox c1;
    JCheckBox c2;
    JCheckBox c3;
    MyFrame()
    {
        setSize(500, 300);
        setTitle("My Application");
        setLayout(new FlowLayout());
        c1 = new JCheckBox("C");
        c2 = new JCheckBox("C++");
        c3 = new JCheckBox("Java");
        add(c1);
        add(c2);
        add(c3);
        setVisible(true);
    }
}
```



```
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public static void main(String[] args)
    {
        MyFrame mf = new MyFrame();
    }
}
```

Output of the above code is as shown below:



Radio Buttons

Radio buttons are used to create a group of mutually exclusive buttons. User can select only one button. The `JRadioButton` class can be used to create radio buttons. The `JRadioButton` class inherits the `JToggleButton` class. A `JRadioButton` object can be created by using the following constructor:

```
JRadioButton(String str)
```

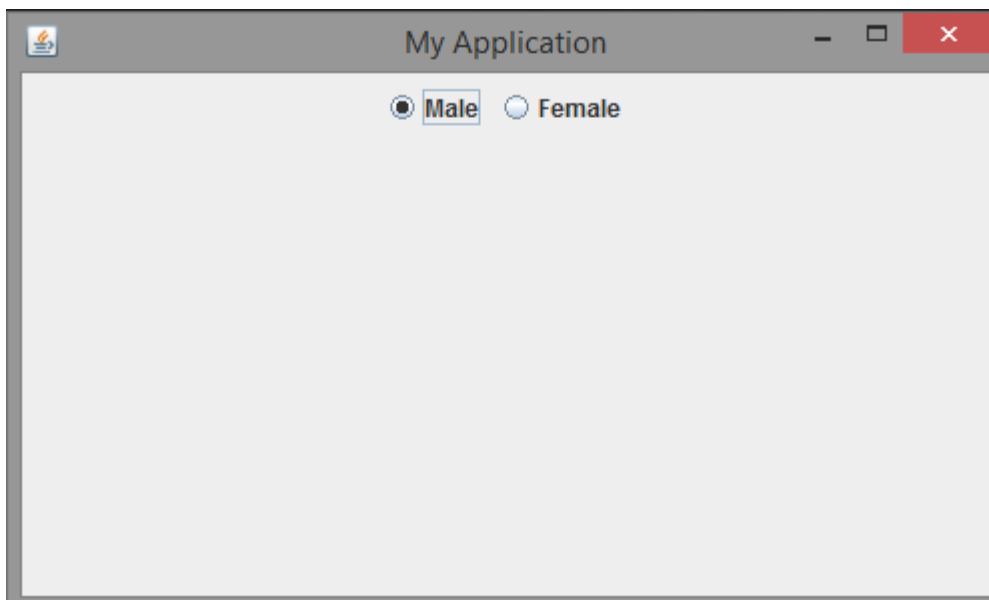
In the above constructor `str` specifies the string to be displayed beside the circle. A radio button raises action event, item event etc. Most commonly handled event is `ActionEvent`.

We must also use another class `ButtonGroup` to group the radio buttons i.e to make them mutually exclusive. Following code demonstrates working with `JRadioButton`:

```
import java.awt.*;
import javax.swing.*;
public class MyFrame extends JFrame
{
    JRadioButton r1;
    JRadioButton r2;
    ButtonGroup bg;
    MyFrame()
    {
        setSize(500, 300);
        setTitle("My Application");
        setLayout(new FlowLayout());
        r1 = new JRadioButton("Male");
        r2 = new JRadioButton("Female");
        add(r1);
        add(r2);
        bg = new ButtonGroup();
        bg.add(r1);
        bg.add(r2);
        setVisible(true);
    }
}
```

```
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public static void main(String[] args)
    {
        MyFrame mf = new MyFrame();
    }
}
```

Output of the above code is as shown below:



Text Fields

A text field is a GUI control which allows the user to enter a single line of text. A text field can be created using the class `JTextField` which inherits the class `JTextComponent`. A `JTextField` object can be created using any one of the following constructors:

`JTextField(int cols)`

`JTextField(String str)`

JTextField(String str, int cols)

In the above constructors cols specifies the size of the text field and str specifies the default string to be displayed in the text field.

A text field generates `ActionEvent` when the user hits the 'enter' key. It can also generate `CaretEvent` when the cursor position changes. `CaretEvent` is available in `javax.swing.event` package.

Following code demonstrates working with `JTextField`:

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

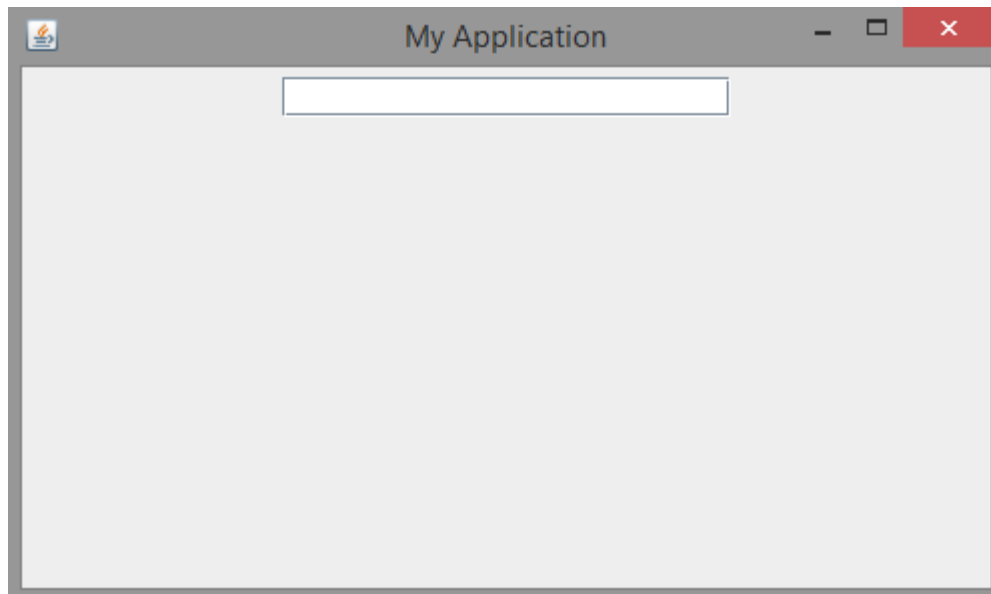
public class MyFrame extends JFrame
{
    JTextField jtf;

    MyFrame()
    {
        setSize(500, 300);
        setTitle("My Application");
        setLayout(new FlowLayout());
        jtf = new JTextField(20);
        add(jtf);
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

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

```
{  
    MyFrame mf = new MyFrame();  
}  
}
```

Output of the above code is as shown below:



Tabbed Panes

A tabbed pane is a control which can be used to manage several components in groups. Related components can be placed in a tab. Tabbed panes are frequently found in modern GUIs. A tabbed pane can be created using `JTabbedPane` class.

After creating a `JTabbedPane` object, we can add tabs by using the following method:

```
void addTab(String tabname, Component comp)
```

In the above syntax `tabname` is the name of the tab and `comp` is the reference to the component that should be added to the tab. Following code demonstrates working with `JTabbedPane`:

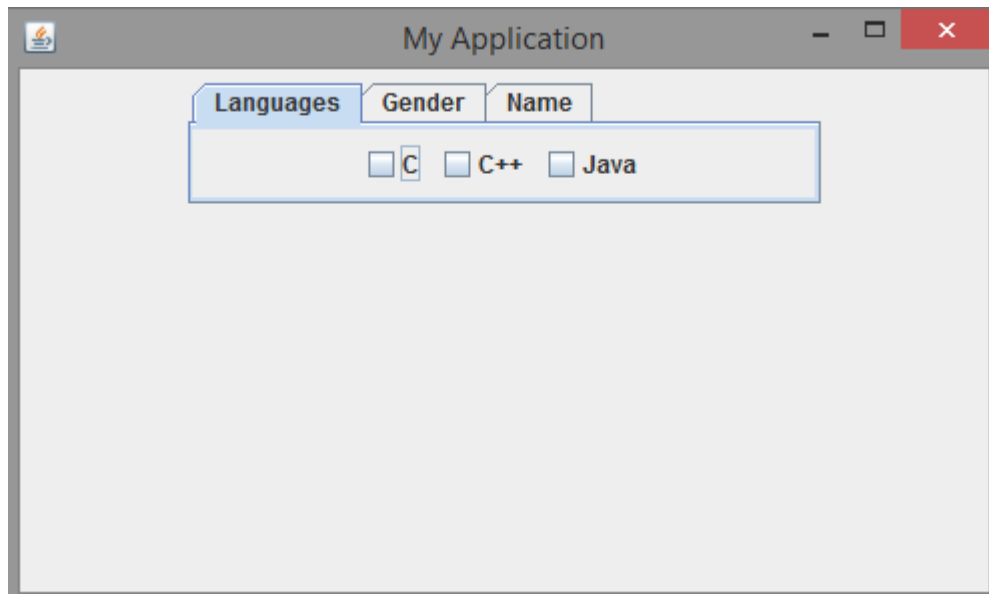
```
import java.awt.*;
import javax.swing.*;
public class MyFrame extends JFrame
{
    JCheckBox c1;
    JCheckBox c2;
    JCheckBox c3;
    JRadioButton r1;
    JRadioButton r2;
    ButtonGroup bg;
    JLabel l;
    JTextField jtf;
    JPanel langs;
    JPanel gender;
    JPanel name;
    JTabbedPane jtp;
    MyFrame()
    {
        setSize(500, 300);
        setTitle("My Application");
        setLayout(new FlowLayout());
        c1 = new JCheckBox("C");
        c2 = new JCheckBox("C++");
        c3 = new JCheckBox("Java");
        langs = new JPanel();
```

```
langs.add(c1);
langs.add(c2);
langs.add(c3);
r1 = new JRadioButton("Male");
r2 = new JRadioButton("Female");
bg = new ButtonGroup();
bg.add(r1);
bg.add(r2);
gender = new JPanel();
gender.add(r1);
gender.add(r2);
l = new JLabel("Enter Name: ");
jtf = new JTextField(20);
name = new JPanel();
name.add(l);
name.add(jtf);
jtp = new JTabbedPane();
jtp.addTab("Languages", langs);
jtp.addTab("Gender", gender);
jtp.addTab("Name", name);
add(jtp);
setVisible(true);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}

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

```
{  
    MyFrame mf = new MyFrame();  
}  
}
```

Output of the above code is as shown below:



List Boxes

A list box displays a list of items from which the user can select one or more items. A list box can be created using the class `JList`. Starting from Java 7, `JList` is defined as a generic class as shown below:

```
JList<type>
```

where `type` represents the data type of the items to be stored in the list. A `JList` object can be created using the following constructor:

```
JList(type[] items)
```


In the above constructor items is an array of the data type type. After creating a JList object, it should be wrapped inside a JScrollPane. It provides scrollbars to navigate the list of items in the JList.

The JList generates ListSelectionEvent when a user selects or deselects an item in the list. It is handled by implementing the ListSelectionListener interface which is available in javax.swing.event package and the event handling method is valueChanged().

We can set the selection mode of a JList by using setSelectionMode() whose general form is as follows:

```
void setSelectionMode(int mode)
```

In the above syntax mode can be any one of the following constants:

SINGLE_SELECTION

SINGLE_INTERVAL_SELECTION

MULTIPLE_INTERVAL_SELECTION

Following are some of the methods available in the JList class:

`int getSelectedIndex()` – Returns the index of the selected item

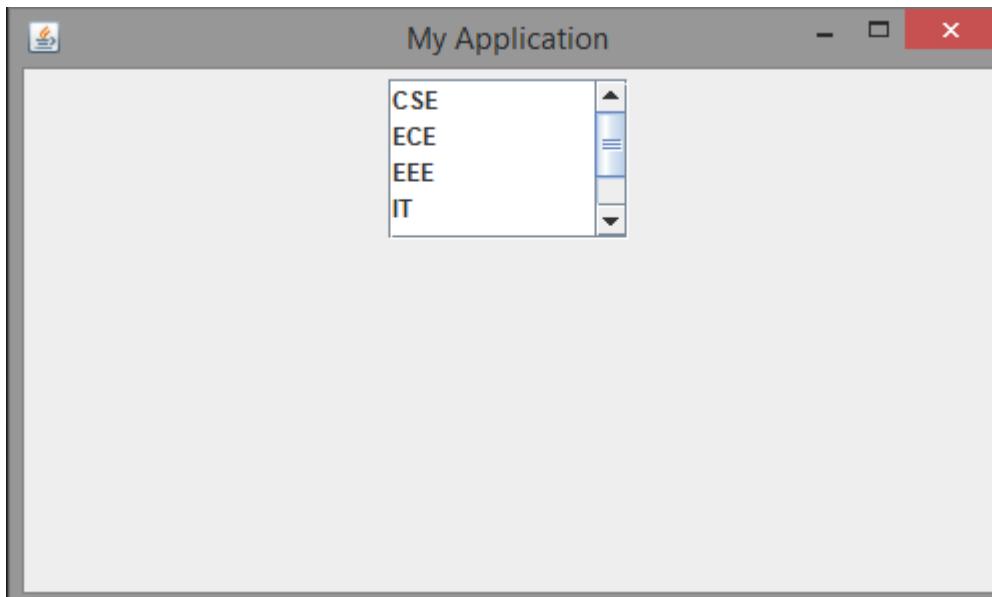
`E getSelectedValue()` – Returns the value of the item selected by the user

Following code demonstrates working with JList:

```
import java.awt.*;  
import javax.swing.*;  
public class MyFrame extends JFrame
```

```
{  
    JList<String> jl;  
    JScrollPane jsp;  
    MyFrame()  
    {  
        setSize(500, 300);  
        setTitle("My Application");  
        setLayout(new FlowLayout());  
        String[] branches = {"CSE", "ECE", "EEE", "IT", "MECH", "CIV"};  
        jl = new JList<String>(branches);  
        jsp = new JScrollPane(jl);  
        jsp.setPreferredSize(new Dimension(120, 80));  
        add(jsp);  
        setVisible(true);  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    }  
    public static void main(String[] args)  
    {  
        MyFrame mf = new MyFrame();  
    }  
}
```

Output of the above code is as shown below:



Combo Boxes

A combo box control displays a pop-up list of items from which user can select only one item. A combo box can be created using the JComboBox class. Starting from Java 7, JComboBox class is defined as a generic class as shown below:

`JComboBox<type>`

An object of JComboBox class can be created using the following constructors:

`JComboBox()`

`JComboBox(E[] items)`

Following are some of the methods available in JComboBox class:

`void addItem(E obj)` – To add an item in the combo box

`Object getSelectedItem()` – To retrieve the item selected by the user

`int getSelectedItemIndex()` – To retrieve the index of the item selected by the user

A `JComboBox` object generates action event when the user selects an item from the list. It also generates an item event when the user selects or deselects an item from the list.

Following code demonstrates working with `JComboBox`:

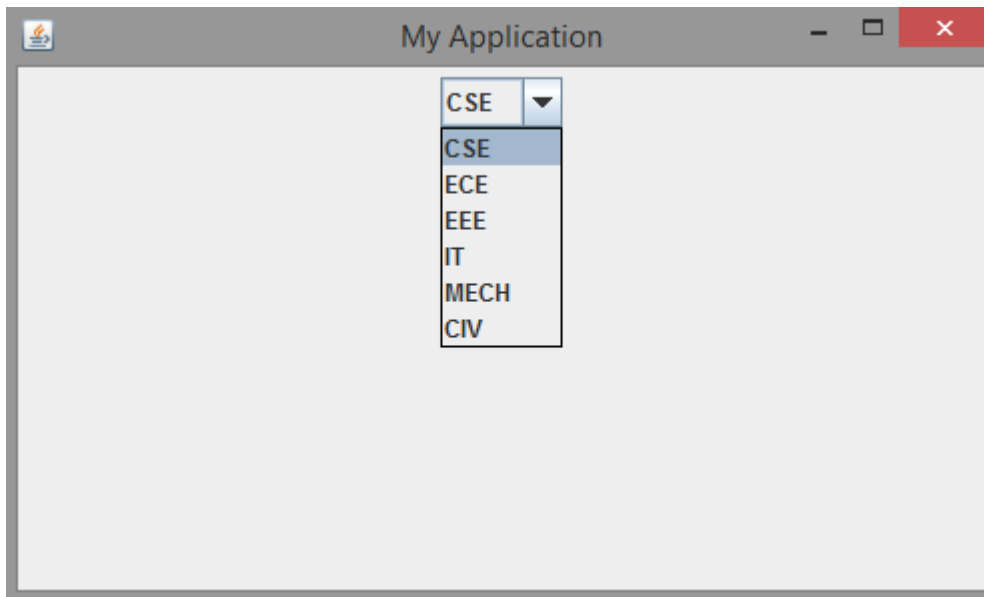
```
import java.awt.*;
import javax.swing.*;

public class MyFrame extends JFrame
{
    JComboBox<String> jc;

    MyFrame()
    {
        setSize(500, 300);
        setTitle("My Application");
        setLayout(new FlowLayout());
        String[] branches = {"CSE", "ECE", "EEE", "IT", "MECH", "CIV"};
        jc = new JComboBox<String>(branches);
        add(jc);
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public static void main(String[] args)
    {
        MyFrame mf = new MyFrame();
    }
}
```

Output of the above code is as shown below:



Tables

A table control can be used to display data in the form of rows and columns. Each column contains a heading. A table can be created using the `JTable` class. Several classes and interfaces associated with the `JTable` class are available in the `javax.swing.table` package.

A `JTable` object can be created with the following constructor:

```
JTable(Object[][] data, Object[] colHeads)
```

In the above syntax `data` is a two-dimensional array that contains the original data to be displayed in the table and `colHeads` is a one-dimensional array that contains the column headings. As `JTable` does not provide scrolling capability, it should be wrapped in `JScrollPane`.

A `JTable` generates `ListSelectionEvent` when the user selects something in a table. It generates `TableModelEvent` when the data in the table is somehow changed.

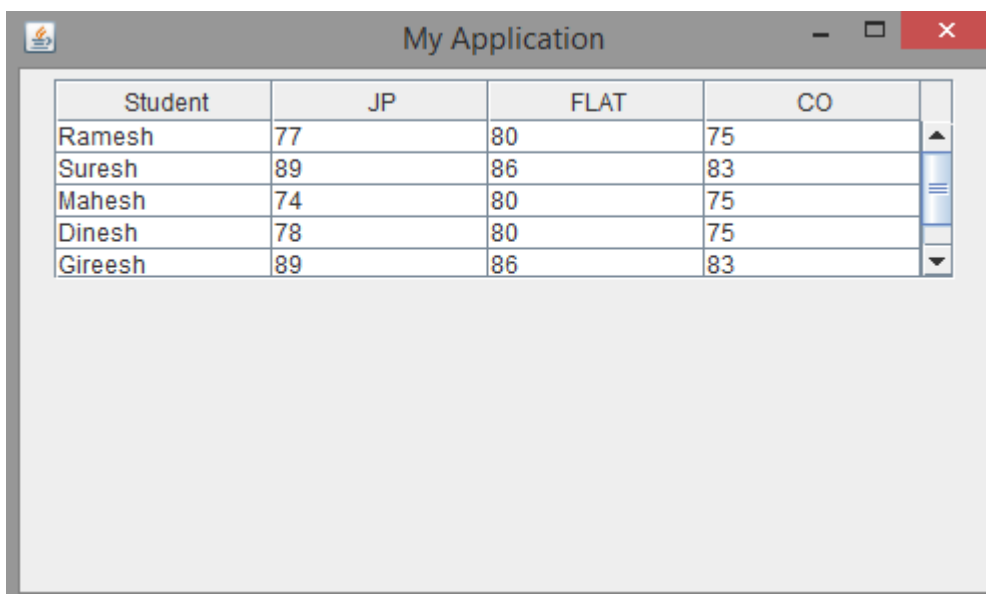
Following code demonstrates working with JTable:

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

public class MyFrame extends JFrame
{
    JTable jt;
    JScrollPane jsp;
    MyFrame()
    {
        setSize(500, 300);
        setTitle("My Application");
        setLayout(new FlowLayout());
        String[] colHeads = {"Student", "JP", "FLAT", "CO"};
        String[][] data = {
            {"Ramesh", "77", "80", "75"},
            {"Suresh", "89", "86", "83"},
            {"Mahesh", "74", "80", "75"},
            {"Dinesh", "78", "80", "75"},
            {"Gireesh", "89", "86", "83"},
            {"Paramesh", "77", "84", "72"} };
        jt = new JTable(data, colHeads);
        jsp = new JScrollPane(jt);
        jsp.setPreferredSize(new Dimension(450, 100));
        add(jsp);
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

```
}  
public static void main(String[] args)  
{  
    MyFrame mf = new MyFrame();  
}  
}
```

Output of the above code is as shown below:



The screenshot shows a window titled "My Application" with a table containing student names and their marks in three subjects: JP, FLAT, and CO. The table has five rows of data and a header row. The window also features a standard Mac OS-style title bar with a close button (red) and a maximize button (grey).

Student	JP	FLAT	CO
Ramesh	77	80	75
Suresh	89	86	83
Mahesh	74	80	75
Dinesh	78	80	75
Gireesh	89	86	83