

Chapter 1

Java GUI Development (AWT and SWING)

GUI: A graphical user interface (GUI) presents a user-friendly mechanism for interacting with an application. It gives an application a distinctive “look” and “feel.” Providing different applications with consistent, intuitive user interface components allows users to be somewhat familiar with an application, so that they can learn it more quickly and use it more productively. GUIs are built from GUI components. These are sometimes called controls or widgets. A GUI component is an object with which the user interacts via the mouse, the keyboard or another form of input, such as voice recognition.

Java GUI APIs: Java has two GUI packages, the original *Abstract Windows Toolkit (AWT)* and the newer *Swing*. When java was introduced, the GUI classes were bundled in a library known as **Abstract Windows Toolkit (AWT)**. It is Java’s original set of classes for building GUIs. It uses peer components of the OS; It is **heavyweight** and uses the native operating system's window routines so the visual effect is dependent on the run-time system platform. For every platform on which Java runs, the AWT components are automatically mapped to the platform-specific components through their respective agents, known as *peers*. It is not truly portable as it looks different and lays out inconsistently on different OSs. The application's GUI components display differently on each platform. In addition, AWT is adequate for many applications but it is difficult to build an attractive GUI.

Swing: is designed to solve AWT’s problems (since Java 2). It is 99% java; It is **lightweight components** as drawing of components is done in java. Swing GUI components allow you to specify a uniform look-and-feel for your application across all platforms. It also lays out consistently on all Oss. It has much bigger set of built-in components and uses AWT event handling.

Swing is built “on top of” AWT, so you need to import AWT and use a few things from it. Swing is bigger and slower. Swing is more flexible and better looking. **Swing** and **AWT** are **incompatible**. Thus, you can use either, but you can’t mix them.

Basic components/controls are practically the same in both. For example:

➤ AWT: `Button b = new Button("OK");`

➤ Swing: `JButton b = new JButton("OK");`

Swing gives **far more** options for everything (buttons with pictures on them, etc.). AWT classes are contained inside package **java.awt** while Swing classes are located in package **javax.swing**. Below is demonstration of Java GUI APIs diagrammatically.

Swing vs. AWT

- ✓ First Java GUI library was known as the Abstract Windows Toolkit (AWT).
- ✓ AWT is fine for developing simple graphical user interfaces, but not for complex GUI projects.
- ✓ A newer, more robust, and flexible library is known as Swing components.
- ✓ Swing components are less dependent on the target platform and use less of the native GUI resource.
- ✓ Swing components that don't rely on native GUI are referred to as lightweight components and AWT components are referred to as heavyweight components.
- ✓ To distinguish new Swing component classes from their older AWT counterparts, the Swing GUI component classes are named with a prefixed J.
- ✓ Although AWT components are still supported in Java, it is better to learn to how program using Swing components, because the AWT user- interface components will eventually fade away.

Difference Between AWT and Swing

Definition

AWT is a collection of GUI components (widgets) and other related services required for GUI programming in Java. Swing is a part of Java Foundation Classes (JFC) that is used to create Java-based Front end GUI applications. Hence, this explains the main difference between AWT and Swing in Java.

Type

AWT components are heavyweight while Swing components are lightweight.

Platform Dependency

Another major difference between AWT and Swing in Java is that AWT is platform dependent while Swing is platform independent.

Display

Moreover, AWT does not support a pluggable look and feel. Swing supports a pluggable look and feel. This is also an important difference between AWT and Swing in Java.

Components

Also, Swing has more advanced components than AWT.

Speed

Furthermore, execution of AWT is slower. However, Swing executes faster.

MVC

AWT does not support MVC pattern while Swing supports MVC pattern. This is another difference between AWT and Swing.

Memory Space

Moreover, AWT components require more memory space while Swing components do not require much memory space.

Package

The programmer has to import the javax.awt package to develop an AWT-based GUI. However, the programmer has to import javax.swing package to write a Swing application.

- ⌘ Text (JTextField, JTextArea)
- ⌘ Menus (JMenuBar, JMenu, JMenuItem)
- ⌘ Sliders (JSlider)
- ⌘ JComboBox (uneditable) (JComboBox)
- ⌘ List (JList)
- ⌘ **Information Display Components**
 - ⌘ JLabel
 - ⌘ Progress bars (JProgressBar)
 - ⌘ Tool tips (using JComponent's setToolTipText(s) method)
- ⌘ **Choosers**
 - ⌘ File chooser (JFileChooser)
 - ⌘ Color chooser (JColorChooser)
- ⌘ **More complex displays**
 - ⌘ Tables (JTable)
 - ⌘ Trees (JTree)

GUI helper classes: Helper classes are used to describe the properties of GUI components. It encompasses the classes for graphics context, colors, fonts and dimension.

Graphics: It is an abstract class that provides a graphical context for drawings strings, lines, and simple shapes.

Color: It deals with the colors of GUI components.

For example, we can specify background colors in components like JFrame and JPanel while we can specify colors of lines, shapes and etc.

Font: It specifies fonts for the text and drawings on GUI components. Font is available on the java.awt package.

Example: `panel1.setFont(new Font("Serif",Font.TRUETYPE_FONT + Font.ITALIC + Font.BOLD,20));`

LayoutManager: LayoutManager is an interface whose instances specify how components are arranged in a container. There are different types of layouts. FlowLayout, GridLayout and BorderLayout are three examples of the layouts. These layouts are found in the java.awt package of JDK API.

Let us see an example using one of these three layout types.

```
LayoutManager lay_out = new FlowLayout();  
JFrame Frame_based_container = new JFrame("I am a container");  
Frame_based_container.setLayout(lay_out);  
setLayout(new FlowLayout()); //default one  
setLayout(new FlowLayout(FlowLayout.LEFT));  
setLayout(new FlowLayout(FlowLayout.LEFT, 20,30));
```

It should be noted that before adding components to a certain container, the layout must be first set depending on the desired arrangement of the containers or components inside the container.

Nowadays, Java GUI focuses on Swing GUI components and Swing Containers. At the meantime, the helper classes are used from AWT.

Steps in building GUI Application

1. **Creating GUI Containers:** Create a container that could contain components or other containers. We need to create either a frame or an applet to hold the user-interface components.

Frame: It is an independent window that has decorations such as a border, a title and buttons for closing, minimizing and maximizing the window. Frame is a window that is not contained inside another window. A frame can be moved around on the screen independently of any other GUI windows. Applications with a GUI typically use **at least one frame**. Frame is the basis to contain other user interface components in Java GUI applications.

The **JFrame** class can be used to create windows. The following statement creates a window with title "My First JFrame".

```
JFrame frame = new JFrame("My First JFrame");
```

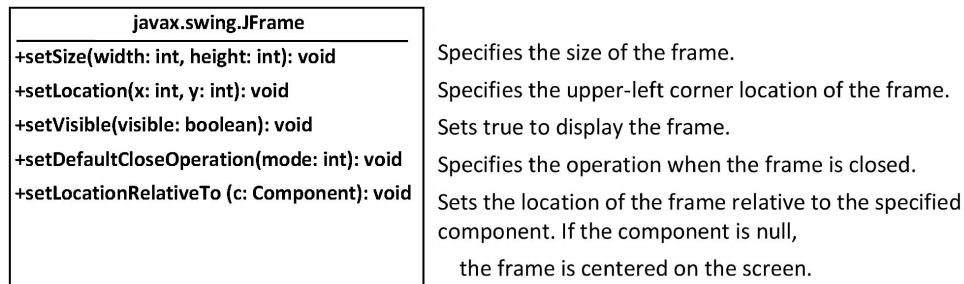


The JFrame class has the following two constructors:

JFrame(): This constructor constructs a new frame with no title and it is initially invisible.

JFrame(String title): This constructor constructs a new, initially invisible frame with specified title.

Some of the methods of JFrame class are described here in class diagram:



setBounds(int x, int y, int width,Int height) : It is another method of JFrame class that encompasses the methods setLocation(int x, int y) and setSize(int width, int height). It specifies the size of the frame and the location of the upperleft corner. This puts the upper left corner at location (x, y), where x the number of pixels from the left of the screen and y is the number from the top of the screen. height and width are as before.

public void setDefaultCloseOperation(int mode): It is a method of JFrame class that is used to specify one of several options for the close button. Every JFrame instance window has a close button. We can program how the window reacts when this close button is clicked.

Use one of the following constants to specify your choice:

JFrame.EXIT_ON_CLOSE: exit the application.

JFrame.HIDE_ON_CLOSE: Hide the frame, but keep the application running.

JFrame.DO_NOTHING_ON_CLOSE: Ignore the click.

Example: Here is a java code that creates a window/frame with the specified title, location, size, frame visibility and default close operation. Let you go through the code and understand what each line of the code means.

```
import javax.swing.*;

public class JFrameSample {

    public static void main(String[] args) {

        JFrame frame = new JFrame("First JFrame");

        frame.setSize(400, 300);
```

```

frame.setLocationRelativeTo(null);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);

```

Here is another way of writing the above code (by extending JFframe):

```

import javax.swing.JFrame;

public class Simple extends JFrame {

    public Simple() {
        setSize(300,200);
        setTitle("First JFrame");
        setLocationRelativeTo(null);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String[] args) {
        Simple simple = new Simple();
    } }

```

2. Create some more Components (such as buttons and text areas)

JButton: A button is a component that triggers an action event when clicked. A JButton component can be created using one of the following constructors.

- ∞ **JButton():** Creates a default button with no text and icon.
- ∞ **JButton(Icon icon):** Creates a button with an icon.
- ∞ **JButton(String text):** Creates a button with text.
- ∞ **JButton(String text, Icon icon):** Creates a button with text and an icon.

JButton class has different methods including the following:

- ∞ **public void setText(String text) :** Sets the button's text.
- ∞ **public String getText():** Returns the button's text.
- ∞ **public void setEnabled(boolean b) :** Enables (or disables) the button.
- ∞ **public void setSelectedIcon(Icon selectedIcon):** Sets the selected icon for the button.

∞ **public boolean isSelected()** : Returns the state of the button. True if the toggle button is selected, false if it's not.

Here two alternative examples that create a JButton component with text “Click Here”.

```
JButton button= new JButton();  
button.setText(“Click Here”);
```

OR

```
JButton button= new JButton(“Click Here”);
```

JLabel: A label is a display area for a short text(a non-editable), an image, or both.

A JLabel component can be created using one of the following constructors:

- **JLabel():** Creates a default label with no text and icon.
- **JLabel(String text):** Creates a label with text.
- **JLabel(Icon icon):** Creates a label with an icon.
- **JLabel(String text, int horizontalAlignment):** Creates a label with a text and the specified horizontal alignment.
- **JLabel(Icon icon, int horizontalAlignment):** Creates a label with an icon and the specified horizontal alignment.
- **JLabel(String text, Icon icon, int horizontalAlignment):** Creates a label with text, an icon, and the specified horizontal alignment.

JLabel component has various methods including the following:

- ∞ **public String getText():** Returns a string containing the text in the label component
- ∞ **public void setText(String):** Sets the label component to contain the string value
- ∞ **public Icon getIcon():** Returns the graphic image (icon) that the label displays.
- ∞ **public void setIcon(Icon icon):** Defines the icon this component will display. If the value of icon is null, nothing is displayed.

For example, JLabel can be used as follows.

```
// Create an image icon from image file
```

```
ImageIcon icon = new ImageIcon("image/grapes.gif");
```

```
// Create a label with text, an icon, with centered horizontal alignment
```

```
JLabel jlbl = new JLabel("Grapes", icon, SwingConstants.CENTER);
```

```
// Set label's text alignment and gap between text and icon
jlbl.setHorizontalTextPosition(SwingConstants.CENTER);
jlbl.setVerticalTextPosition(SwingConstants.BOTTOM);
jlbl.setIconTextGap(5);
```



JTextField: A **text field** is a box that contains a line of text. The user can type text into the box and the program can get it and then use it as data. The program can write the results of a calculation to a text field. Text fields are useful in that they enable the user to enter in variable data (such as a name or a description). **JTextField** is swing class for an editable text display.

JTextField components can be created using one of the following constructors:

- ∞ **JTextField():** it creates a default empty text field with number of columns set to 0.
- ∞ **JTextField(int columns):** it creates an empty text field with the specified number of columns.
- ∞ **JTextField(String text):** it creates a text field initialized with the specified text.
- ∞ **JTextField(String text, int columns):** it creates a text field initialized with the specified text and the column size.

JTextfield class has the following methods:

- ∞ **public String getText():** returns the string from the text field.
- ∞ **public void setText(String text):** puts the given string in the text field.
- ∞ **public void setEditable(boolean editable):** enables or disables the text field to be edited.
By default, editable is true.
- ∞ **public void setColumns(int):** sets the number of columns in this text field.
The length of the text field is changeable.

- ⌘ **public void select(int selectionStart, int selectionEnd):** Selects the text between the specified start and end positions.
- ⌘ **public String getSelectedText():** Returns the text value that has been highlighted in the text field.
- ⌘ **public void append(String value):** Appends the text value of the string to the already existing text in the component

Here is an example in which JTextField is used.

```
JLabel FName= new JLabel("First Name");
JTextField text = new JTextField(10);
JLabel Lname= new JLabel("Last Name");
JTextField text1 = new JTextField("Text1",10);
text.setEditable(false);
add(Fname); add(text); add(Lname); add(text1);
```

Exercise: Draw the resulting GUI of this code.

JPasswordField: Allows the editing of a single line of text where the view indicates something was typed, but does not show the original characters.

JPasswordField component can be created using one of the following constructors:

- **JPasswordField():** Constructs a new JPasswordField, with a default document, null starting text string, and 0 column width
- **JPasswordField(int columns):** Constructs a new empty JPasswordField with the specified number of columns
- **JPasswordField(String text):** Constructs a new JPasswordField initialized with the specified text.
- **JPasswordField(String text, int columns) :** Constructs a new JPasswordField initialized with the specified text and columns.

JTextArea: If you want to let the user enter multiple lines of text, you cannot use text fields unless you create several of them. The solution is to use JTextArea class, which enables the user to enter multiple lines of text.

JTextArea components can be created using one of the following constructors:

- **JTextArea(int rows, int columns):** creates a text area with the specified number of rows and columns.

- **JTextArea(String s, int rows, int columns):** creates a text area with the initial text and the number of rows and columns specified.

JCheckBox: It is a widget that has two states. On and Off. It is a box with a label. If the checkbox is checked, it is represented by a tick in a box. JCheckBox component can be created using one of the following constructors:

- **JCheckBox()** :creates an initially unselected check box button with no text, no icon
- **JCheckBox(Icon icon)** : creates an initially unselected check box with an icon
- **JCheckBox(Icon icon, boolean selected)** : creates a check box with an icon and specifies whether or not it is initially selected.
- **JCheckBox(String text)** : creates an initially unselected check box with text.
- **JCheckBox(String text,boolean selected):** creates a check box with text and specifies whether or not it is initially selected.
- **JCheckBox(String text,Icon icon)** : creates an initially unselected check box with the specified text and icon.
- **JCheckBox(String text, Icon icon, Boolean selected)** : creates a check box with text and icon, and specifies whether or not it is initially selected.

Example:

```
JCheckBox red = new JCheckBox("Red", true);
JCheckBox blue = new JCheckBox("Blue");
JCheckBox green = new JCheckBox("Green");
JCheckBox yellow = new JCheckBox("Yellow");
```



Creating Menu Driven Java Application:

We can create a menu driven application in java from which we can choose an activity.

There are three important swing components for the development of menu driven application.

These are **JMenuBar**, **JMenu** and **JMenuItem**.

JMenuBar: It is used to create the menu bar. Menu bar is created using the constructor **JMenuBar()**. For example, we may create a menu bar as follows.

```
JMenuBar jmb = new JMenuBar();
```

Once a menu bar is created, the menu bar is going to be embedded onto the container frame using the method **setJMenuBar(JMenuBar mbr)**.

JMenu: It is used to create menu or submenus. The constructor `JMenu(String name)` can be used to create a menu/submenu. For example, `JMenu fileMenu = new JMenu("File")`.

Once menus are created using `JMenu`, these menus can be added onto the menu bar using the function **`add(JMenu menu)`**. Assume we have created two menu component file and edit as follow.

```
JMenu fileMenu = new JMenu("File").
```

```
JMenu editMenu = new JMenu("Edit").
```

We can add these menu components to the menu bar we created earlier as follow

```
Jmb.add(fileMenu);
```

```
Jmb.add(editMenu)
```

JMenuItem: This component is used to add menu items under menu components. The constructor **`JMenuItem(String item)`** is used to create such items. Once the items are created the method **`add(JMenuItem item)`** can be used to add the item to a menu component.

For example, let us create two menu items under the file menu.

```
JMenuItem menuItemNew = new JMenuItem("New");
```

```
JMenuItem menuItemOpen = new JMenuItem("Open");
```

```
fileMenu.add(menuItemNew);
```

```
fileMenu.add(menuItemOpen);
```

Creating submenu under a menu: It is possible to have a menu under another menu, that is, creating a submenu under a menu. First, the submenu will be created using the **`JMenu`** constructor. Then, we will use the method **`add(JMenu menu)`** to add the submenu under a menu. Let u observe the complete code of the following example and notice how a submenu is added to a menu.

Example: consider the following example that creates the output displayed next to the code

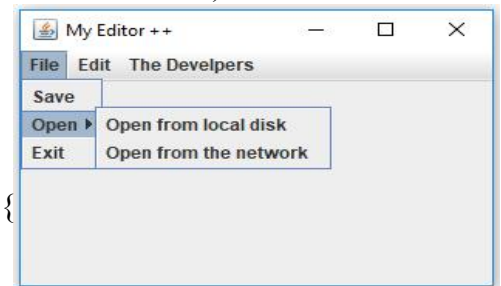
```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;

public class ClassAtMenuing {
    public static void main(String[] args) {
```

```

JFrame myEditor = new JFrame("My Editor ++");
JMenuBar myMenuBar = new JMenuBar();
myEditor.setJMenuBar(myMenuBar);
//Create the Menus
JMenu fileMenu = new JMenu("File");
JMenu editMenu = new JMenu("Edit");
JMenu developersMenu = new JMenu("The Developers");
//add menu items under File
JMenuItem itemSave = new JMenuItem("Save");
JMenuItem closeWindow = new JMenuItem("Exit");
//JMenuItem itemOpen = new JMenuItem("Open");
fileMenu.add(itemSave);
//adding Submen
JMenu openMenu = new JMenu("Open");
//items under open
JMenuItem openFromLocalDisk = new JMenuItem("Open from local disk");
JMenuItem openFromNet = new JMenuItem("Open from the network");
//add the items to the open JMenu
openMenu.add(openFromLocalDisk);
openMenu.add(openFromNet);
fileMenu.add(openMenu);
fileMenu.add(closeWindow);
closeWindow.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null, "You are closing the editor");
        myEditor.dispose();
    }
});
//add menu into the menubar
myMenuBar.add(fileMenu);
myMenuBar.add(editMenu);
myMenuBar.add(developersMenu);
myEditor.setBounds(100, 150, 400, 300);
myEditor.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
myEditor.setVisible(true);
}
}

```



Java JRadioButton

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

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

| Constructor | Description |
|--|---|
| JRadioButton() | Creates an unselected radio button with no text. |
| JRadioButton(String s) | Creates an unselected radio button with specified text. |
| JRadioButton(String s, boolean selected) | Creates a radio button with the specified text and selected status. |

Commonly used Constructors:

Commonly used Methods:

| Methods | Description |
|---|---|
| void setText(String s) | It is used to set specified text on button. |
| String getText() | It is used to return the text of the button. |
| void setEnabled(boolean b) | It is used to enable or disable the button. |
| void setIcon(Icon b) | It is used to set the specified Icon on the button. |
| Icon getIcon() | It is used to get the Icon of the button. |
| void setMnemonic(int a) | It is used to set the mnemonic on the button. |
| void addActionListener(ActionListener a) | It is used to add the action listener to this object. |

Java JRadioButton Example

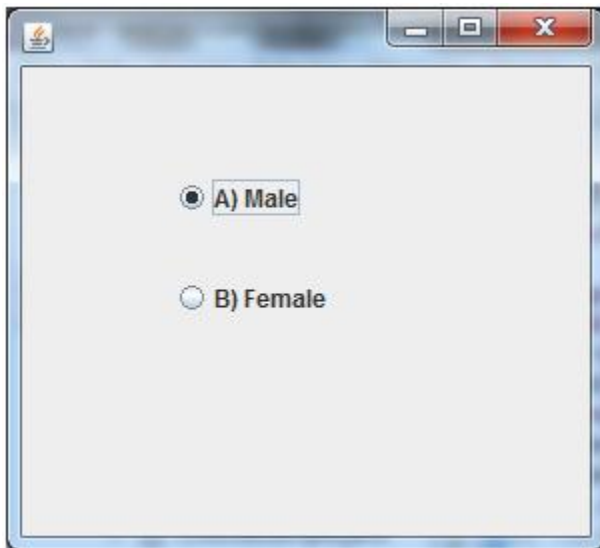
1. **import** javax.swing.*;
2. **public class** RadioButtonExample {
3. JFrame f;
4. RadioButtonExample(){

```

5. f=new JFrame();
6. JRadioButton r1=new JRadioButton("A) Male");
7. JRadioButton r2=new JRadioButton("B) Female");
8. r1.setBounds(75,50,100,30);
9. r2.setBounds(75,100,100,30);
10. ButtonGroup bg=new ButtonGroup();
11. bg.add(r1);bg.add(r2);
12. f.add(r1);f.add(r2);
13. f.setSize(300,300);
14. f.setLayout(null);
15. f.setVisible(true);
16. }
17. public static void main(String[] args) {
18.     new RadioButtonExample();
19. }
20. }

```

Output:



Java JComboBox

The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a [menu](#). It inherits [JComponent](#) class.

Commonly used Constructors:

| Constructor | Description |
|----------------------------|--|
| JComboBox() | Creates a JComboBox with a default data model. |
| JComboBox(Object[] items) | Creates a JComboBox that contains the elements in the specified array . |
| JComboBox(Vector<?> items) | Creates a JComboBox that contains the elements in the specified Vector . |

Commonly used Methods:

| Methods | Description |
|---|--|
| void addItem(Object anObject) | It is used to add an item to the item list. |
| void removeItem(Object anObject) | It is used to delete an item to the item list. |
| void removeAllItems() | It is used to remove all the items from the list. |
| void setEditable(boolean b) | It is used to determine whether the JComboBox is editable. |
| void addActionListener(ActionListener a) | It is used to add the ActionListener . |
| void addItemListener(ItemListener i) | It is used to add the ItemListener . |

Java JComboBox Example

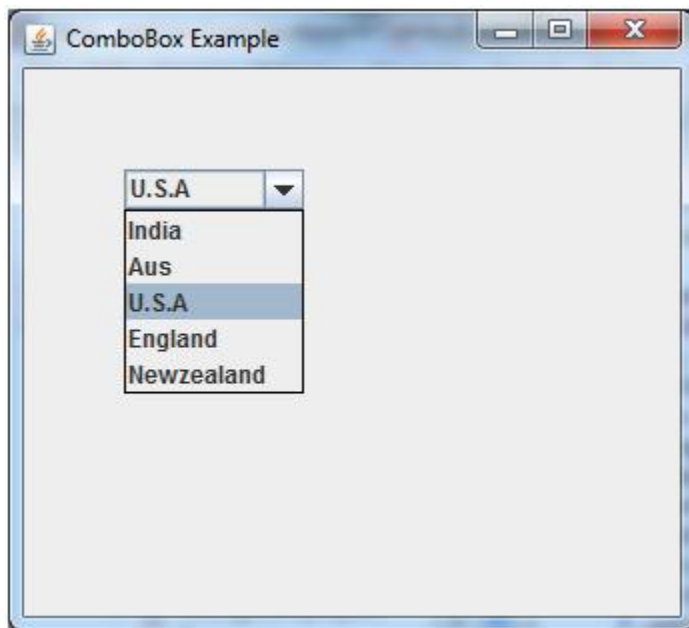
1. **import** javax.swing.*;
2. **public class** ComboBoxExample {
3. JFrame f;
4. ComboBoxExample(){
5. f=**new** JFrame("ComboBox Example");
6. String country[]={**"India", "Aus", "U.S.A", "England", "Newzealand"**};

Constructor

Description

```
7. JComboBox cb=new JComboBox(country);
8. cb.setBounds(50, 50,90,20);
9. f.add(cb);
10. f.setLayout(null);
11. f.setSize(400,500);
12. f.setVisible(true);
13. }
14. public static void main(String[] args) {
15.     new ComboBoxExample();
16. }
17. }
```

Output:



Java JTable

The JTable class is used to display data in tabular form. It is composed of rows and columns

Commonly used Constructors:

| | |
|--|--|
| JTable() | Creates a table with empty cells. |
| JTable(Object[][] rows, Object[] columns) | Creates a table with the specified data. |

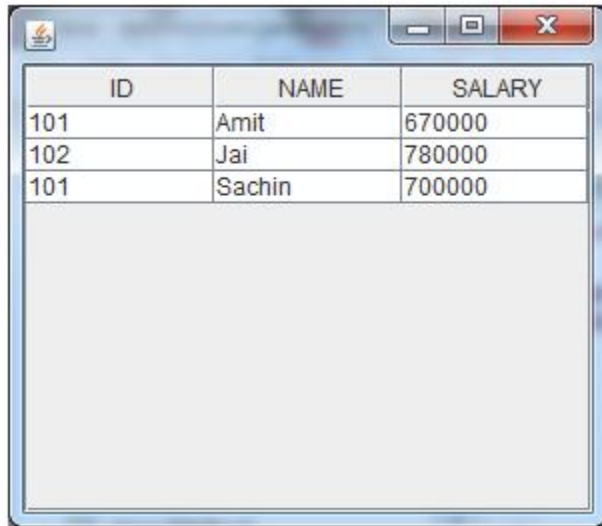
Java JTable Example

```

1. import javax.swing.*;
2. public class TableExample {
3.     JFrame f;
4.     TableExample(){
5.         f=new JFrame();
6.         String data[][]={ {"101","Amit","670000"},
7.                             {"102","Jai","780000"},
8.                             {"101","Sachin","700000"} };
9.         String column[]={ "ID","NAME","SALARY"};
10.        JTable jt=new JTable(data,column);
11.        jt.setBounds(30,40,200,300);
12.        JScrollPane sp=new JScrollPane(jt);
13.        f.add(sp);
14.        f.setSize(300,400);
15.        f.setVisible(true);
16.    }
17.    public static void main(String[] args) {
18.        new TableExample();
19.    }
20. }

```

Output:



| ID | NAME | SALARY |
|-----|--------|--------|
| 101 | Amit | 670000 |
| 102 | Jai | 780000 |
| 101 | Sachin | 700000 |

3. Add your components to your display area: choose a layout manager.

Each JFrame contains a **content pane**. A content pane is an instance of **java.awt.Container**. The GUI components such as buttons are placed in the content pane in a frame. Prior to JDK 1.5, you have to use the **getContentPane()** method in the JFrame class to return the content pane of the frame, and then invoke the content pane's **add** method to place a component into a content pane.

This was cumbersome, JDK 1.5 allows you to place the components to the content pane by invoking a frame's **add** method. This new feature is called **content pane delegation**. Strictly speaking, a component is added into the content pane of the frame. But for simplicity we say that a component is added to a frame. When a component is added to a container, a reference to the component is added to the list of components in the container.

Below are four alternatives to add a button with text "OK" into a frame instance, "frame". We assume that the frame has been instantiated somewhere.

Example 1: `JButton jbt = new JButton("OK");`

`frame.getContentPane().add(jbt);`

Example 2: `frame.getContentPane().add(new JButton("OK"));`

Example 3: `JButton jbt = new JButton("OK")`

`frame.add(jbt);`

Example 4: `frame.add(new JButton("OK"));`

Below is a full java code that adds the button to the frame (following the approach employed by example 3 above)

```

import javax.swing.*;
public class MyFrameWithComponents {
    public static void main(String[] args) {
        JFrame frame = new JFrame("MyFrameWithComponents");
        // Add a button into the frame
        JButton jbtOK = new JButton("OK");
        frame.add(jbtOK);
        frame.setSize(400, 300);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLocationRelativeTo(null); // New since JDK 1.4
    }
}

```

When you run the above program **MyFrameWithComponents**, the button is always centered in the frame and occupies the entire frame no matter how you resize it. This is because components are put in the frame by the content pane's layout manager, and the default layout manager for the content pane places the button in the **center**. Set layout Manager before adding components. In the next section, you will use several different layout managers to place components in other location as desired.

Layout Management:

Layouts tell Java **where to put components** in containers (**JPanel**, **content pane**, etc). Layout manager is created using **LayoutManager** class. Every layout manager class implements the **LayoutManager** class. Each layout manager has a different style of positioning components. If you don't specify otherwise, the container will use a default layout manager. Every **panel** (and other container) has a default layout, but it's better to set the layout explicitly for clarity.

Layout managers are set in containers using the **setLayout(LayoutManager)** method.

Example: assume we have a JFrame instance called container. If we are going to add controls inside this container and if we are interested on the flowlayout, we can set as follow.

```

LayoutManager layoutManager = new FlowLayout();
container.setLayout(layoutManager);
OR
container.setLayout(new FlowLayout());

```

Java supplies several layout managers including

FlowLayout: The Simplest layout manager (the default one) Components are placed left to right in the order in which they were added. When one row is filled, a new row is started. Components

can be right aligned, centered or left aligned by using **FlowLayout.RIGHT**, **FlowLayout.CENTER**, and **FlowLayout.LEFT** respectively.

FlowLayout has the following possible constructors:

| java.awt.FlowLayout | The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity. |
|---|--|
| -alignment: int | The alignment of this layout manager (default: CENTER). |
| -hgap: int | The horizontal gap of this layout manager (default: 5 pixels). |
| -vgap: int | The vertical gap of this layout manager (default: 5 pixels). |
| +FlowLayout() | Creates a default FlowLayout manager. |
| +FlowLayout(alignment: int) | Creates a FlowLayout manager with a specified alignment. |
| +FlowLayout(alignment: int, hgap: int, vgap: int) | Creates a FlowLayout manager with a specified alignment, horizontal gap, and vertical gap. |

GridLayout: The GridLayout manager divides the container up into a given number of rows and columns. All sections of the grid are equally sized and as large as possible.

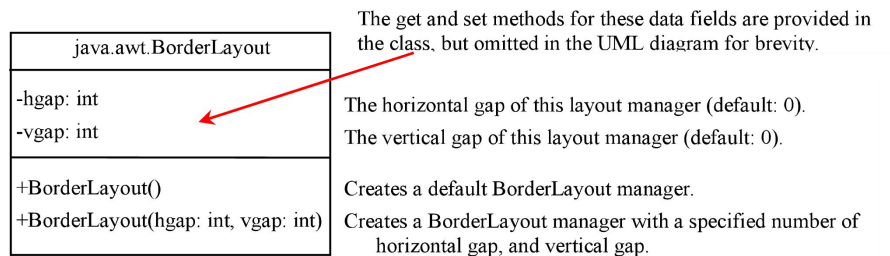
GridLayout can be set using one of the constructors depicted below:

| java.awt.GridLayout | The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity. |
|--|--|
| -rows: int | The number of rows in this layout manager (default: 1). |
| -columns: int | The number of columns in this layout manager (default: 1). |
| -hgap: int | The horizontal gap of this layout manager (default: 0). |
| -vgap: int | The vertical gap of this layout manager (default: 0). |
| +GridLayout() | Creates a default GridLayout manager. |
| +GridLayout(rows: int, columns: int) | Creates a GridLayout with a specified number of rows and columns. |
| +GridLayout(rows: int, columns: int, hgap: int, vgap: int) | Creates a GridLayout manager with a specified number of rows and columns, horizontal gap, and vertical gap. |

BorderLayout: The BorderLayout manager divides the window into five areas: East, South, West, North, and Center. At most five components can be added. If you want more components, add a Panel, then add components to it. Components are added to a BorderLayout by using **add(Component, index)** where index is a constant such as :

- BorderLayout.EAST
- BorderLayout.SOUTH
- BorderLayout.WEST
- BorderLayout.NORTH
- BorderLayout.CENTER

BorderLayout can be set using one of the constructors shown below:



4. Attach Listeners to your components:- interacting with a component causes an Event to occur. Usually, we click button, move our mouse pointer onto a text, move our mouse pointer onto a button, enter a text to a text field and etc. When a certain action is performed, some event is fired. Listeners are associated with control so that it can sense whenever an event is fired.

| Event Class | Listener Interface | Listener Methods | Example Source Object |
|--------------------|--------------------|--------------------------------|-----------------------|
| ActionEvent | ActionListener | actionPerformed(ActionEvent) | JButton, JTextField |
| MouseEvent | MouseListener | mousePressed(MouseEvent) | Component |
| | | mouseReleased(MouseEvent) | |
| | | mouseEntered(MouseEvent) | |
| | | mouseExited(MouseEvent) | |
| | | mouseClicked(MouseEvent) | |
| WindowEvent | WindowListener | windowOpened(WindowEvent) | Window |
| | | windowClosed(WindowEvent) | |
| | | windowClosing(WindowEvent) | |
| | | windowIconified(WindowEvent) | |
| | | windowDeiconified(WindowEvent) | |
| | | | |

Note that If a component can fire an event, any subclass of the component can fire the same type of event. For example, every GUI component can fire **MouseEvent**, **KeyEvent**, **FocusEvent**, and **ComponentEvent**, since **Component** is the superclass of all GUI components. Some of the packages of JDK API that we use in handling event-driven tasks include:

- java.awt.event.ActionEvent
- java.awt.event.ActionListener

- java.awt.event.MouseEvent
- java.awt.event.MouseListener
- java.awt.event.WindowEvent
- java.awt.event.WindowListener

Listeners, registrations, and handling of events:

Delegation-based model for event handling is employed where a source object fires an event, and an object interested in the event (a listener) handles the event. For an object to be a listener for an event on a source object, two things are needed. Firstly, the listener object must be an instance of the corresponding event-listener interface to ensure that the listener has the correct method for processing the event. Java provides a listener interface for every type of event. The listener interface is usually named **XListener** for **XEvent**, with the exception of **MouseMotionListener**. For example, the corresponding listener interface for **ActionEvent** is **ActionListener** and each listener for **ActionEvent** should implement the **ActionListener** interface. Secondly, the listener object must be registered by the source object. Registration methods depend on the event type. For **ActionEvent**, the method is **addActionListener**. In general, the method is named **addXListener** for **XEvent**. A source object may fire several types of events. It maintains, for each event, a list of registered listeners and notifies them by invoking the *handler* of the listener object to respond to the event

Example: Assume I have window/frame that has a label named “Name” and a text field to enter name. There is also a button of which text is “show me my input in dialog box”. Once we enter a name in the text field, we can click the button to get a dialog box that contain the name we entered in the text field. How can we implement this scenario?

Solution: There could be different solutions. But, let us see one way of setting the layout.

First, let us create the necessary containers and components with appropriate layout; then, we will add the components/containers to the appropriate containers. Here we go!

```
import java.awt.GridLayout;
import javax.swing.*;
public class Example1 extends JFrame{
    JLabel label_name;
    JTextField txt_field_name;
    JButton btn_show;
    Example1(){
        label_name= new JLabel("Name:");
        txt_field_name = new JTextField();
        btn_show = new JButton("show me my input in dialog box");

        //I want to create a JPanel that could contain the label of the name and the text field of the
        //name and, then, I will add this JPanel instance to the JFrame
    }
}
```



```

JPanel panel1 = new JPanel();
panel1.setLayout(new GridLayout(1,2));
panel1.add(label_name);
panel1.add(txt_field_name);

//let us add the panel and the button to the frame after set the layout of the frame
this.setLayout(new GridLayout(2,1));
this.add(panel1);
this.add(btn_show);

//let us set the size, location and set default close aspects of the frame and make the
window //visible
this.setSize(400,250);
this.setLocation(150,200);
this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
this.setVisible(true);
}

public static void main(String [] args){
new Example1();
}
}

```

If we run the above code, we will get the following GUI application.



But, clicking the button does not do anything as we did not associate this action on the button with the appropriate event. Thus, we are going to improve this code so that clicking the button shows the desired dialog box.

We can do this using one of the following three ways. The first is register the event listener of the source object and define the action performed at the same spot or register the event listener on the source object and define action performed in another class. This second way can, in turn,

be accomplished in two ways: using an inner class for the action definition or using another separate class. Let us see the implementation using the three approaches.

Approach 1:

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

public class Example1 extends JFrame{
    JLabel label_name;
    JTextField txt_field_name;
    JButton btn_show;
    Example1(){
        label_name= new JLabel("Name:");
        txt_field_name = new JTextField();
        btn_show = new JButton("show me my input in dialog box");
        btn_show.addActionListener(new ActionListener(){
            @Override
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(null,"You entered
                name:"+txt_field_name.getText());
            }
        });
        //I want to create a JPanel that could contain the label of the name and the text field of the //name
        and, then, I will add this JPanel instance to the JFrame

        JPanel panel1 = new JPanel();
        panel1.setLayout(new GridLayout(1,2));
        panel1.add(label_name);
        panel1.add(txt_field_name);

        //let us add the panel and the button to the frame after set the layout of the frame
        this.setLayout(new GridLayout(2,1));
        this.add(panel1); this.add(btn_show);

        //let us set the size, location and set default close aspects of the frame and make the window
        //visible
        this.setSize(400,250);
        this.setLocation(300,400);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);
    }
    public static void main(String [] args){
        //we are calling the JFrame instance
        new Example1();
    }
}
```

```
}
}
```

Approach 2:

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

public class Approach2 extends JFrame{
    JLabel label_name;
    JTextField txt_field_name;
    JButton btn_show;
    Approach2(){
        label_name= new JLabel("Name:");
        txt_field_name = new JTextField();
        btn_show = new JButton("show me my input in dialog box");
        btn_show.addActionListener(new Listenn_btn_click());
        //I want to create a JPanel that could contain the label of the name and the text field of the
        //name and, then, I will add this JPanel instance to the JFrame

        JPanel panel1 = new JPanel();
        panel1.setLayout(new GridLayout(1,2));
        panel1.add(label_name);
        panel1.add(txt_field_name);

        //let us add the panel and the button to the frame after set the layout of the frame
        this.setLayout(new GridLayout(2,1));
        this.add(panel1); this.add(btn_show);

        //let us set the size, location and set default close aspects of the frame and make the
        window //visible
        this.setSize(400,250);
        this.setLocation(300,400);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);
    }
    public static void main(String [] args){
        //we are calling the JFrame instance
        new Approach2();
    }
    //This is a class that shows the action to be performed when the clicking event is fired
    private class Listenn_btn_click implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(null, "You entered name:
             "+txt_field_name.getText());
        }
    }
}
```

```

    }
}
}

```

Approach 3:

```

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

```

```

public class Approach3 extends JFrame{
    JLabel label_name;
    static JTextField txt_field_name;
    JButton btn_show;

```

```

Approach3(){
    label_name= new JLabel("Name:");
    txt_field_name = new JTextField();
    btn_show = new JButton("show me my input in dialog box");
    btn_show.addActionListener(new Listenn_btn_click());

```

//I want to create a JPanel that could contain the label of the name and the text field of the //name and, then, I will add this JPanel instance to the JFrame

```

JPanel panel1 = new JPanel();
panel1.setLayout(new GridLayout(1,2));
panel1.add(label_name);
panel1.add(txt_field_name);

```

```

//let us add the panel and the button to the frame after set the layout of the frame
this.setLayout(new GridLayout(2,1));
this.add(panel1); this.add(btn_show);

```

//let us set the size, location and set default close aspects of the frame and make the window //visible

```

this.setSize(400,250);
this.setLocation(300,400);
this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
this.setVisible(true);
}

```

```

public static void main(String [] args){
    //we are calling the JFrame instance
    new Approach3();
}
}

```

```

class Listenn_btn_click implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null, "You entered name:
" + Approach3.txt_field_name.getText());

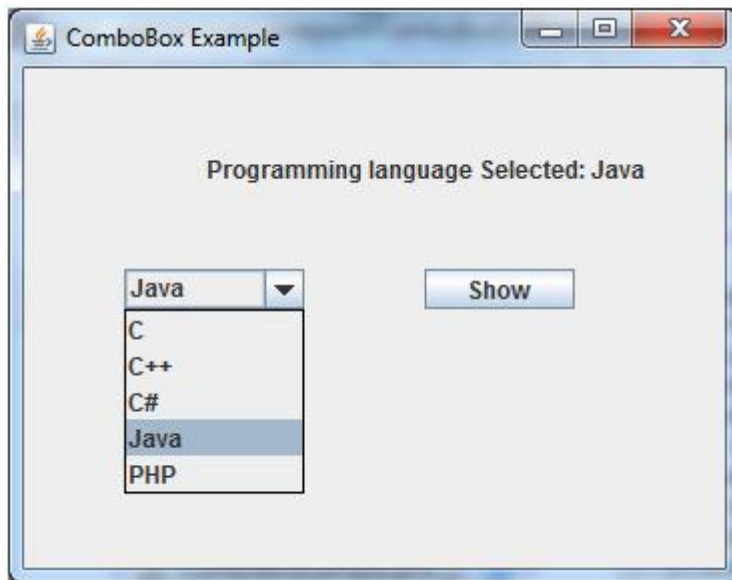
```

```
}  
}
```

Java JComboBox Example with ActionListener

```
1. import javax.swing.*;  
2. import java.awt.event.*;  
3. public class ComboBoxExample {  
4.     JFrame f;  
5.     ComboBoxExample(){  
6.         f=new JFrame("ComboBox Example");  
7.         final JLabel label = new JLabel();  
8.         label.setHorizontalAlignment(JLabel.CENTER);  
9.         label.setSize(400,100);  
10.        JButton b=new JButton("Show");  
11.        b.setBounds(200,100,75,20);  
12.        String languages[]={ "C", "C++", "C#", "Java", "PHP"};  
13.        final JComboBox cb=new JComboBox(languages);  
14.        cb.setBounds(50, 100,90,20);  
15.        f.add(cb); f.add(label); f.add(b);  
16.        f.setLayout(null);  
17.        f.setSize(350,350);  
18.        f.setVisible(true);  
19.        b.addActionListener(new ActionListener() {  
20.            public void actionPerformed(ActionEvent e) {  
21.                String data = "Programming language Selected: "  
22.                + cb.getItemAt(cb.getSelectedIndex());  
23.                label.setText(data);  
24.            }  
25.        });  
26.    }  
27.    public static void main(String[] args) {  
28.        new ComboBoxExample();  
29.    }  
30. }
```

Output:



Java JRadioButton Example with ActionListener

```

1. import javax.swing.*;
2. import java.awt.event.*;
3. class RadioButtonExample extends JFrame implements ActionListener{
4. JRadioButton rb1,rb2;
5. JButton b;
6. RadioButtonExample(){
7. rb1=new JRadioButton("Male");
8. rb1.setBounds(100,50,100,30);
9. rb2=new JRadioButton("Female");
10. rb2.setBounds(100,100,100,30);
11. ButtonGroup bg=new ButtonGroup();
12. bg.add(rb1);bg.add(rb2);
13. b=new JButton("click");
14. b.setBounds(100,150,80,30);
15. b.addActionListener(this);
16. add(rb1);add(rb2);add(b);
17. setSize(300,300);
18. setLayout(null);
19. setVisible(true);
20. }
21. public void actionPerformed(ActionEvent e){
22. if(rb1.isSelected()){
23. JOptionPane.showMessageDialog(this,"You are Male.");
24. }
25. if(rb2.isSelected()){

```

```
26. JOptionPane.showMessageDialog(this, "You are Female.");
27. }
28. }
29. public static void main(String args[]){
30. new RadioButtonExample();
31. }}
```

Output:

