A Complete Note

ON

Advanced Java Programming

(CSIT.412)

Fourth Year, Seventh Semester

Bachelor of Science in Computer Science & Information Technology (B.Sc. CSIT)

Far Western University

© Deepak Bhatta Kaji

Assistant Professor

Kailali Multiple Campus

Dhangadhi, Kailali

Nepal

DEEPAK BHATTA KAJI

KNIGHTEC NEPAL

A GATEWAY TO THE FUTURE

Department of Computer Science and Information Technology

Kailali Multiple Campus

Education Prepares You for the Future.

Unit 1 — 10 hrs.

AWT & Layout Management

Specific Objectives

- Understand concepts of AWT containers and controls.
- Use Containers and controls to create GUI.
- Demonstrate layout mangers and SetBound method.
- Apply graphics libraries to create graphics.
- Create menus and Menubars using AWT.

1.1. AWT Basics: AWT class Hierarchy, AWT Containers & Controls, AWT Features.

1.2. AWT Containers: Window, Frame, Panel, Dialog, Applets, Creating Frames & Panels, Creating Applets, Applet Life Cycle.

1.3. Layout Managers: Flow Layout, Grid Layout, GridBag Layout, Border Layout, Group Layout, Using SetBound method.

1.4. AWT Controls: TextField, TextArea, Button, Label, Checkbox, Checkbox Group, Choice, List, Canvas, Image.

1.5. AWT Menu: Menu Hierarchy, Menu, MenuBar, MenuItem, PopupMenu.

1.6. AWT Graphics: Graphics and Graphics2D Class, Drawing Lines, Curves, rectangles, ellipse, Changing Color & Font.

1.1. AWT Basics: AWT class Hierarchy, AWT Containers & Controls, AWT Features

Introduction to AWT:

AWT in Java stands for **Abstract Window Toolkit**, used in Java for making graphical user interfaces for Java applications. WT is a Java toolkit used for making powerful interfaces for Java applications. It provides a set of classes and methods used to develop windows, buttons, menus, dialog boxes, and other major GUI components.

AWT Basics:

Java consists of many predefined methods and frameworks, which help to make development smooth and easier. AWT in Java is a set of APIs used to make useful graphical user interface applications in Java. It is one of the older frameworks in Java. It consists of many components such as **text fields**, **password fields**, **checkboxes**, **radio buttons**, **buttons**, etc. frequently used in applications.

It can handle user input easily and integrate with the native platforms to run smoothly. This toolkit can be imported into your Java code from **java.awt** package. It contains classes required to create window-based Java programs for Java applications. AWT use resources of operating system on which it is working. Its components are heavyweight.

Important Highlights:

- ✓ AWT in Java can be imported using import using **java.awt** package.
- ✓ AWT stands for Abstract Window Toolkit for creating Graphical User Interface (GUI) for different Java applications.
- ✓ The components of AWT in Java are component, container, window, frame, dialog, canvas, panel, image, menu component, font, colors, etc.
- ✓ They also manage different events such as click, hover, and other event listeners, etc.
- ✓ They arrange different components in a proper layout with proper sizing and positioning.
- ✓ They can also handle colors and fonts which help developers to customize appearance of their Java applications easily.
- ✓ With the help of **java.awt.graphics** developers can easily draw different shapes such as Images, text and more.

- ✓ It provides containers such as panel, frames, and dialog to organize components in a proper layout.

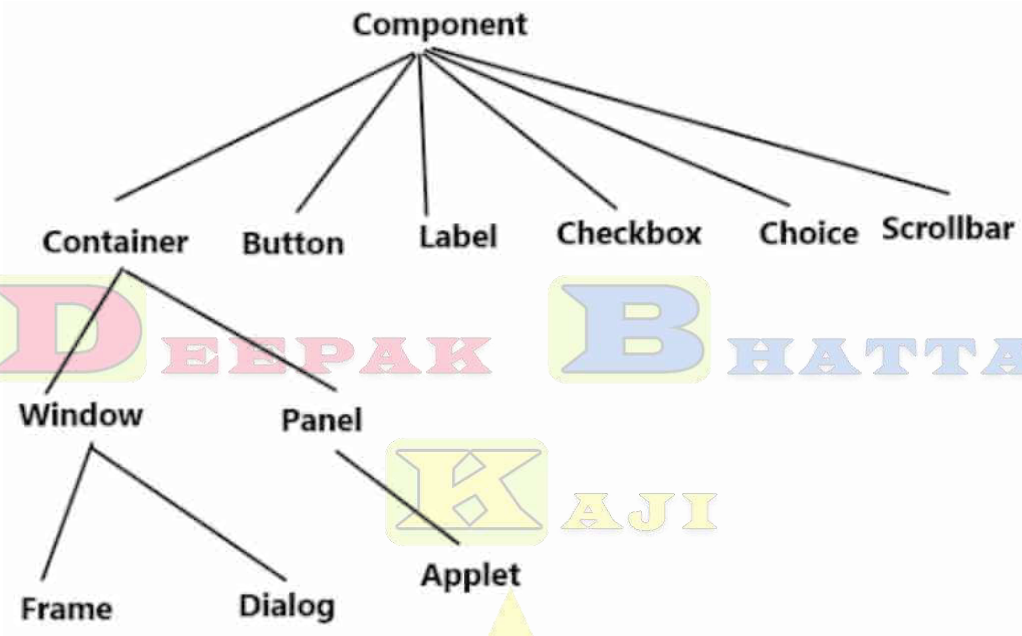
Why AWT is platform dependent?

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

In simple words, an application build on AWT would look more like a windows application when being run on Windows, however, that same application would look like a Mac application when being run on Mac Operating System.

AWT class Hierarchy:

The AWT components in Java are given proper order and preferences. Check the elements given below according to the hierarchy of AWT in java.



- **Components:**
  Components consist of Buttons, choice, list, checkbox, input, label, list, etc. which is used to make GUI elements interactive in Java.
- **Containers:**
  They are the second element in the AWT library which consist s of panel, frames and dialogues which is used to organize and give proper layout to components of AWT in Java applications.
- **Layout Managers:**
  They consist of Flow Layout, Border Layout, etc. to arrange data inside the containers in proper space.
- **Event Handling:**
  AWT can handle events such as mouse hovers, clicks, etc. to respond with a particular event listener function.
- **Graphics:**
  With the help of **java.awt.graphics** developers can create different shapes, insert images and write texts inside the components.

AWT Containers & Controls:

Containers are components that can hold other components, including controls or other containers. They provide a base to arrange other GUI components in a hierarchical structure.

1. **Frame**: A top-level window with a title and a border, used to represent main windows.
   - **Example:**

```
Frame frame = new Frame("My Window");
```

2. **Panel**: A generic container used to group components inside another container. It doesn't have a title or border.
   - **Example:**

```
Panel panel = new Panel();
```

3. **Dialog**: A pop-up window used for interaction or additional information.
   - **Example:**

```
Dialog dialog = new Dialog(frame, "My Dialog", true);
```

4. **Window**: A general-purpose top-level window, without a title or border.
   - **Example:**

```
Window window = new Window(frame);
```


Controls are the components that allow user interaction, such as buttons, text fields, and labels. These components are typically added to containers.

1. **Button**: A clickable button for user actions.
   - **Example:**

```
Button button = new Button("Click Me");
```

2. **Label**: A non-editable text field that displays a string.
   - **Example:**

```
Label label = new Label("This is a label");
```

3. **TextField**: A single-line input field where users can enter text.
   - **Example:**

```
TextField textField = new TextField(10);
```

4. **TextArea**: A multi-line text area for displaying or inputting large amounts of text.
   - **Example:**

```
TextArea textArea = new TextArea("This is a text area", 5, 20);
```

5. **Checkbox**: A box that can be checked or unchecked.
   - **Example:**

```
Checkbox checkbox = new Checkbox("Accept Terms");
```

6. **CheckboxGroup**: A group of checkboxes where only one can be selected at a time (radio buttons).
   - **Example:**

```
CheckboxGroup group = new CheckboxGroup();

Checkbox option1 = new Checkbox("Option 1", group, false);
```

7. **Choice**: A dropdown list of items where the user can select one.
   - **Example:**

```
Choice choice = new Choice();
```

8. **List**: A list of items, with options for single or multiple selection.
   - **Example:**

```
List list = new List(4, false);
```

9. **Scrollbar**: A control for adjusting numeric values using a slider.
   - **Example:**

```
Scrollbar scrollbar = new Scrollbar();
```

Will be discussed more details in coming topics…..

AWT Features:

- **User interface components:** A collection of native user interface components, such as buttons, text boxes, and menus
- **Event handling:** A robust event-handling model that allows each component to react to an event before the platform-dependent code processes it
- **Graphics and imaging tools:** Includes shape, color, and font classes
- **Layout managers:** Allows for flexible window layouts that don't depend on the screen resolution or window size
- **Data transfer classes:** Allows for cut-and-paste through the native platform clipboard
- **Containers:** Includes four types of containers: Window, Frame, Dialog, and Panel
- **Controls:** Includes a Choice control for pop-up menus, a Canvas control for drawing and receiving user input, and an Image control for representing graphical images

1.2. AWT Containers: Window, Frame, Panel, Dialog, Applets, Creating Frames & Panels, Creating Applets, Applet Life Cycle

AWT Containers:

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

Window:

A Window is the top-level, invisible container that provides a base for components like frames and dialogs. A window has no border or title bar. It serves as the base class for Frame and Dialog.

Example:

```
Window window = new Window(new Frame());
window.setSize(300, 200);
window.setVisible(true);
```

Frame:

A Frame is a top-level window with a title, border, and buttons to close, minimize, or maximize the window. It is commonly used as the main window in a GUI application.

Example:

```
import java.awt.*;

public class MyFrame extends Frame {
    public MyFrame() {
        setTitle("My Frame");
        setSize(400, 300);
        setVisible(true);
    }

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

Panel:

A Panel is a container that can hold other components, such as buttons, text fields, etc. It is a lightweight component often used inside Frame or Window to group components.

Example:

```
import java.awt.*;
public class MyPanel extends Frame {
    public MyPanel() {
        Panel panel = new Panel();
        panel.add(new Button("Click Me"));
        add(panel);
        setSize(300, 200);
        setVisible(true);
    }

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

Dialog:

A Dialog is a pop-up window that can be used to display messages, take input, or show additional options. It can be modal or non-modal. A modal dialog blocks other windows until it is closed.

Example:

```
import java.awt.*;

public class MyDialog extends Dialog {
    public MyDialog(Frame f1) {
        super(f1, "My Dialog", true);
        setSize(300, 200);
```

```
        setVisible(true);
    }

    public static void main(String args[]) {
        Frame frame = new Frame();
        new MyDialog(frame);
    }
}
```


Applets:

An Applet is a small Java program that is embedded in a web page and runs in the context of a browser or an applet viewer. Applets are part of the **java.applet** package and use the Applet class.

Example: MyApplet.java

```
import java.applet.Applet;
import java.awt.Graphics;

public class MyApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello, Applet!", 50, 50);
    }
}
```


To run an applet, you need an HTML file or an applet viewer to load it. Here's the basic HTML to embed an applet:

```
<applet code="MyApplet.class" width="300" height="200"></applet>
```

Creating Frames & Panels:

```
import java.awt.*;

public class FrameWithPanel extends Frame {
    public FrameWithPanel() {
        setTitle("Frame with Panel");
        Panel panel = new Panel();
        panel.add(new Button("OK"));
        panel.add(new Label("Enter your name:"));
        panel.add(new TextField(20));
        add(panel);
        setSize(400, 300);
        setVisible(true);
    }

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

Close the Frame in AWT:

```
frame.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent windowEvent){
            System.exit(0);
         }
      });
```

Creating Applets:

Applets are created by extending the Applet class and implementing the lifecycle methods.

NOTE: Java applets were deprecated by Java 9 in 2017. To run java applet, download older version of jdk 1.8

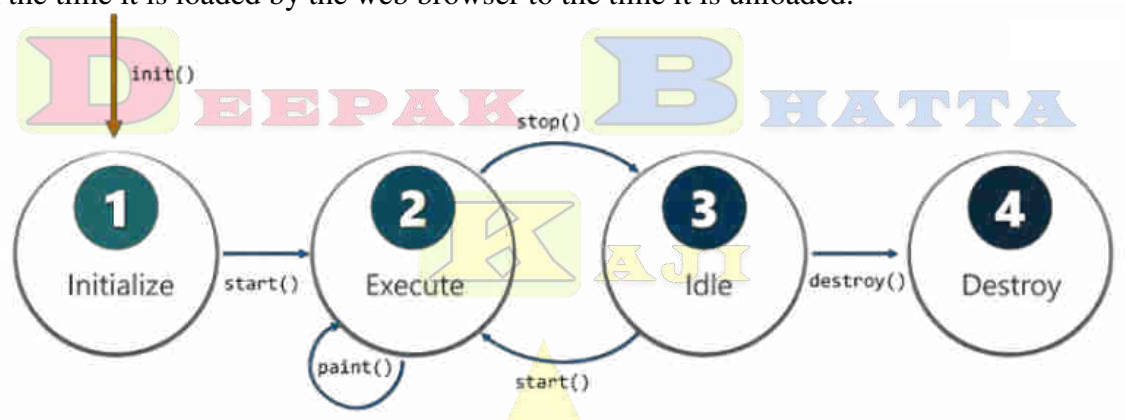
Example:

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

 * @author Deepak Bhatta Kaji
 */
public class SimpleApplet extends Applet{

    @Override

public class SimpleApplet extends Applet {
    public void init() {
        System.out.println("Applet Initialized");
    }

    public void start() {
        System.out.println("Applet Started");
    }

    public void paint(Graphics g) {
        g.drawString("Welcome to Java Applet!", 50, 50);
    }

    public void stop() {
        System.out.println("Applet Stopped");
    }

    public void destroy() {
        System.out.println("Applet Destroyed");
    }
}
```

Applet Life Cycle:

In the early days of the internet, web pages were largely static and lacked interactivity. However, with the introduction of Java Applets, web developers could add dynamic and interactive elements to web pages.

Java Applets were created as a way to run Java code within a web browser, allowing developers to create complex graphical user interfaces and animations that were not possible with HTML alone. The applet life cycle in Java is the process that an applet goes through, from the time it is loaded by the web browser to the time it is unloaded.



Methods of Applet Life Cycle

The functions involved in the Applet life cycle in Java are:

init()

This method is called when the Applet is first loaded into memory. It initializes the Applet and sets its default values. This method is called only once in the Applet life cycle.

start()

This method is called after the **init()** method. It starts the execution of the Applet and performs all necessary tasks.

paint (Graphics g)

This method is called whenever the Applet needs to be painted on the screen. It is responsible for all the necessary drawing tasks.

stop()

This method is called when the Applet is stopped. It stops the execution of the Applet and releases any resources that it was using.

destroy()

This method is called when the Applet is being removed from memory. It releases all of the resources used by the Applet.

1. **init**():
   - This method is called once when the applet is first loaded.
   - Used for one-time initialization, like setting up variables or resources.
   - **Example:**

```
public void init() {
    // Initialization code here
    System.out.println("Applet initialized.");
}
```

2. **start**():
   - This method is called after init() and each time the applet is revisited.
   - Used to start or resume the applet's execution.
   - **Example:**

```
public void start() {
    System.out.println("Applet started.");
}
```

3. **paint(Graphics g)**:
   - This method is called whenever the applet needs to repaint itself (e.g., when it first loads or when the window is resized).
   - Drawing tasks like shapes, text, or images are done here.
   - **Example:**

```
public void paint(Graphics g) {
    g.drawString("Hello Applet", 50, 50);
}
```

4. **stop**():
   - This method is called when the applet is not visible or the user navigates away from the page.
   - Used to pause any background processes or threads.
   - **Example:**

```
public void stop() {
    System.out.println("Applet stopped.");
}
```

5. **destroy**():
   - This method is called just before the applet is unloaded.
   - Used to clean up resources, such as closing files or network connections.
   - **Example:**

```
public void destroy() {
    System.out.println("Applet destroyed.");
}
```

States of Applet in Java Life Cycle:

Born or Initialization State

The first state of the Applet life cycle is initialization. This is done by calling the **init()** method. This state is invoked only once when the Applet is first loaded into memory. During this state, the Applet initializes itself and sets its default values. At this stage we can do the following actions-creating objects, initializing values, loading colors or images, and setting up the colors.

Running State

The running state is the state where the Applet is actively running and performing its tasks. It is in this state that the user interacts with the Applet. If we leave the web page containing the Applet temporarily to another page, it pushes the Applet to an idle state. When we return back to the page, this again starts the Applet running. Unlike the **init()** method, the **start()** method can be invoked more than once.

Stopped or Idle State

An Applet becomes idle when it is stopped. This can be invoked by calling the **stop()** method. In this state, the Applet stops running, and it releases any resources that it was using. Stopping occurs automatically when we leave the current web page on which the Applet was running. We can also do this by calling the **stop()** method.

Dead State

The dead state is invoked when the Applet is removed from memory. In this state, the Applet releases all of its resources, such as file handles, network sockets, and database connections. This occurs automatically by invoking the **destroy()** method. This happens when we quit the browser in which it was running. Like initialization, destroying also occurs only once.

Working of Applet Life Cycle:

The working Java Applet Life cycle is as follows:

- The Applet life cycle is managed by the Java plug-in program.
- An applet is a Java application implemented in any web browser and works on the client side. It does not contain the **main()** method as it runs in the browser. It is therefore created to be placed on an HTML page.
- The **awt.Component** class consists of the **paint()** method.
- The **applet.Applet** class consist of **start()**, **init()**, **destroy()**, **stop()** methods.
- If we want to make a class an Applet class in Java, we are required to extend the Applet.
- Whenever an applet is created, we create an instance of the existing Applet class. And therefore, we can use all the methods of that class.

Simple Applet Example:

```
 * @author Deepak Bhatta Kaji
 */
public class SimpleApplet extends Applet{

    @Override
```

```
import java.applet.Applet;
import java.awt.Graphics;

public class SimpleApplet extends Applet {

    public void init() {
        System.out.println("Applet Initialized");
    }

    public void start() {
        System.out.println("Applet Started");
    }

    public void paint(Graphics g) {
        g.drawString("Welcome to My Simple Applet!", 20, 30);
    }

    public void stop() {
        System.out.println("Applet Stopped");
    }

    public void destroy() {
        System.out.println("Applet Destroyed");
    }
```

```
}
```

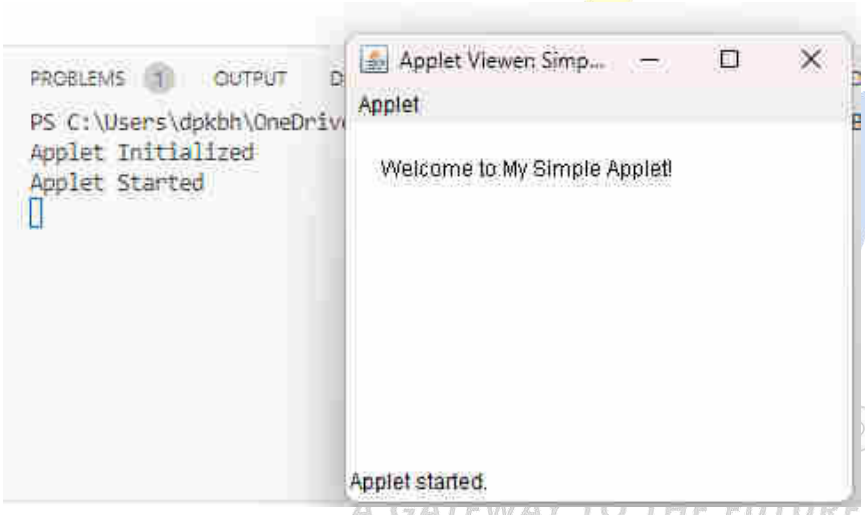
How to Run:

1. Save the applet as **SimpleApplet.java**.
2. Compile the applet using **javac SimpleApplet.java**.
3. Create an HTML file (**applet.html**) to load the applet:

```
<html>
<body>
    <applet code="SimpleApplet.class" width="300" height="150">
    </applet>
</body>
</html>
```

4. Open the HTML file in a browser or use an applet viewer to see the applet running.
   appletviewer applet.html

```
appletviewer .\applet.html
```




```
PS C:\Users\dpkbh\OneDrive - Kailali
Applet Initialized
Applet Started
Applet Stopped
Applet Destroyed
PS C:\Users\dpkbh\OneDrive - Kailali
```

Important Notes:

- **Deprecation**: Since applets are deprecated in modern Java versions, you need to use older Java versions (JDK 8 or earlier) to work with applets.

- **Browser Support**: Modern browsers no longer support applets. Running applets within browsers is outdated, so **appletviewer** is the best option to test them.

1.3. Layout Managers: Flow Layout, Grid Layout, GridBag Layout, Border Layout, Group Layout, Using SetBound method

Layout Manager:

A layout manager is an object that controls the size and position (layout) of components inside a Container object. For example, a window is a container that contains components such as buttons and labels. The layout manager in effect for the window determines how the components are sized and positioned inside the window.

A container can contain another container. For example, a window can contain a panel, which is itself a container. As you can see in the figure, the layout manager in effect for the window determines how the two panels are sized and positioned inside the window, and the layout manager in effect for each panel determines how components are sized and positioned inside the panels.

The **java.awt** package provides the following predefined layout managers that implement the **java.awt.LayoutManager** interface. Every Abstract Window Toolkit (AWT) container has a predefined layout manager as its default. It is easy to use the **container.setLayout** method to change the layout manager, and you can define your own layout manager by implementing the **java.awt.LayoutManager** interface.

List of Layout Managers:

- java.awt.BorderLayout
- java.awt.FlowLayout
- java.awt.CardLayout
- java.awt.GridLayout
- java.awt.GridBagLayout

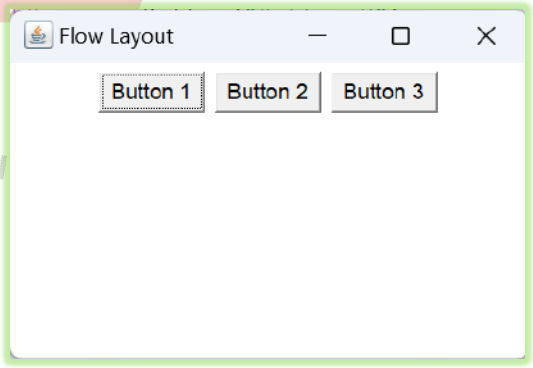
Flow Layout:

Flow Layout arranges components in a directional flow, much like text in a paragraph. By default, components are placed in a row, and when the row is filled, components flow to the next row.

The flow layout is the default layout manager for all Panel objects and applets. It places the panel components in rows according to the width of the panel and the number and size of the components. The best way to understand flow layout is to resize the demonstration window and notice how the components flow from one row to the other as you make the window wide and narrow.

Example:

```
package LayoutManagers;
import java.awt.*;
public class ExampleFlowLayout {
    Frame frame;
    Button b1, b2, b3;
    public ExampleFlowLayout() {
        frame = new Frame();
        frame.setLayout(new FlowLayout());
        b1 = new Button("Button 1");
        b2 = new Button("Button 2");
        b3 = new Button("Button 3");
        frame.add(b1);
        frame.add(b2);
        frame.add(b3);
        frame.setTitle("Flow Layout");
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        ExampleFlowLayout layout = new ExampleFlowLayout();
    }
}
```



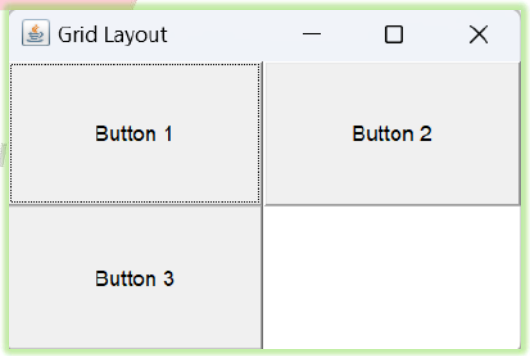
Grid Layout:

The Grid layout arranges components into a grid of rows and columns. You specify the number of rows and columns, the number of rows only and let the layout manager determine the number of columns, or the number of columns only and let the layout manager determine the number of rows. *Grid Layout places components in a grid of rows and columns, with each cell in the grid having the same size.*

The cells in the grid are equal size based on the largest component in the grid. Resize the window to see how the components are resized to fit cells as they get larger and smaller.

Example:

```
package LayoutManagers;
import java.awt.*;
public class ExampleGridLayout {
    Frame frame;
    Button b1, b2, b3;
    public ExampleGridLayout() {
        frame = new Frame();
        frame.setLayout(new GridLayout(2, 2)); // 2 rows, 2 columns
        b1 = new Button("Button 1");
        b2 = new Button("Button 2");
        b3 = new Button("Button 3");
        frame.add(b1);
        frame.add(b2);
        frame.add(b3);
        frame.setTitle("Grid Layout");
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        ExampleGridLayout gl = new ExampleGridLayout();
    }
}
```



GridBag Layout:

The Grid bag layout (like grid layout) arranges components into a grid of rows and columns, but lets you specify a number of settings to fine-tune how the components are sized and positioned within the cells. Unlike the grid layout, the rows and columns are not constrained to be a uniform size. For example, a component can be set to span multiple rows or columns, or you can change its position on the grid.

The layout manager uses the number of components in the longest row, the number of rows created, and the size of the components to determine the number and size of the cells in the grid. The set of cells a component spans can be referred to as its display area. A component's display area and the way in which it fills that display area are defined by a set of constraints that are represented by the GridBagConstraints object.

GridBag Layout is a flexible and complex layout manager that allows components to span multiple rows or columns and gives more control over component alignment and spacing.

These constraints are the following, and will be described in the section on settings below:

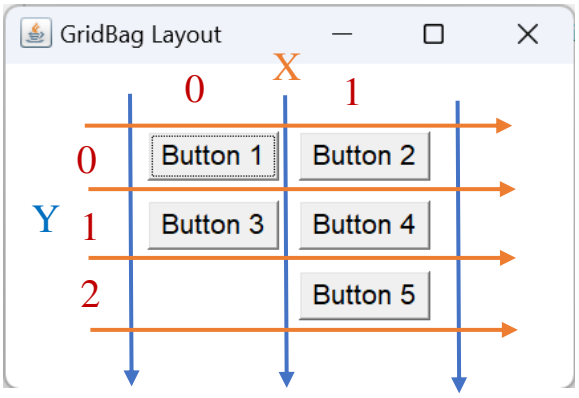
- gridx/gridy
- gridwidth/gridheight
- fill
- anchor
- ipadx/ipady
- insets

The following bulleted list outlines how to use the GridBagConstraints and GridBagLayout objects in code to achieve the look you want. This outline is for instructional purposes only and is used in the following code example. Once you understand how the objects work together, your code can, of course, do things in a different order or take another approach.

- Decide how you want the components sized and positioned.
- Create a **GridBagLayout** and **GridBagConstraints** object.
- Have the container use the GridBagLayout object.
- Customize the GridBagConstraints settings (or use defaults).
- Create **Component A**.
- Pass Component A and GridBagConstraints to the GridBagLayout object.
- Add Component A to the container. The previous bullet told the container's layout which settings to use for the component.
- Create **Component B**.
- Change the GridBagConstraints settings or leave as is.
- Pass Component B and GridBagConstraints to the GridBagLayout object.
- Add Component B to the container.

- and so on . . . .

Example:




```
package LayoutManagers;
import java.awt.*;
import java.awt.Font.*;
public class ExampleGridBagLayout {
    Frame frame;
    GridBagLayout gridbag;
    GridBagConstraints gbc;
    Button btn1, btn2, btn3, btn4, btn5;
    public ExampleGridBagLayout() {
        frame = new Frame();
        gridbag = new GridBagLayout();
        gbc = new GridBagConstraints();
        gbc.insets = new Insets(5, 5, 5, 5);
     // Add padding 5px all
        btn1 = new Button("Button 1");
        btn2 = new Button("Button 2");
        btn3 = new Button("Button 3");
        btn4 = new Button("Button 4");
        btn5 = new Button("Button 5");

        frame.setFont(new Font("Helvetica", Font.PLAIN, 14));
        frame.setBackground(Color.cyan);
        frame.setLayout(gridbag);

        gbc.gridx = 0;
        gbc.gridy = 0;
        frame.add(btn1, gbc);

        gbc.gridx = 1;
        gbc.gridy = 0;
        frame.add(btn2, gbc);
```

$X0, Y0 \rightarrow Button1$

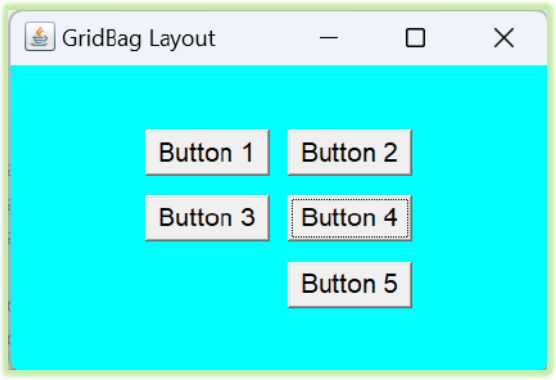
$X1, Y0 \rightarrow Button2$

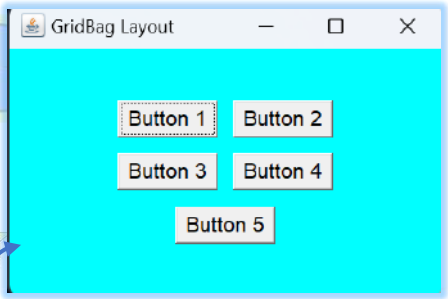
$X0, Y1 \rightarrow Button3$

$X1, Y1 \rightarrow Button4$

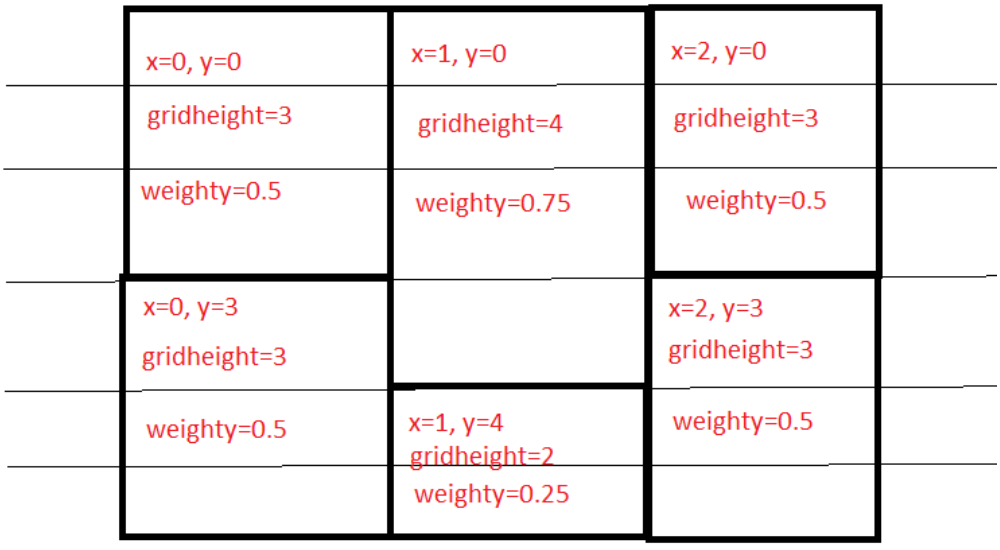
$X1, Y2 \rightarrow Button5$

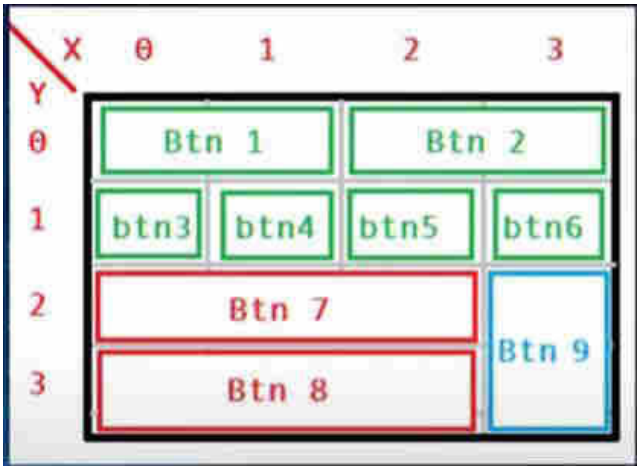
```
        gbc.gridx = 0;
        gbc.gridy = 1;
        frame.add(btn3, gbc);

        gbc.gridx = 1;
        gbc.gridy = 1;
        frame.add(btn4, gbc);

        gbc.gridx = 1;
        gbc.gridy = 2;
        frame.add(btn5, gbc);

        frame.setTitle("GridBag Layout");
        frame.setSize(300, 200);
        frame.setVisible(true);
    }

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






```
gbc.gridx = 0;
gbc.gridy = 2;
gbc.gridwidth = 2;
frame.add(btn5, gbc);
```





Button 1:	Button 2:

gbc.gridx = 0; //btn1 start at x0
gbc.gridy = 0; //btn1 starts at y0
gbc.gridwidth = 2; //span 2 columns
gbc.gridheight = 1; //set it back
	gbc.gridx = 2; //btn2 start at x2
	gbc.gridy = 0; //btn2 starts at y0
	gbc.gridwidth = 2; //span 2 columns
	gbc.gridheight = 1; //set it back

Button 3:	**Button 4:**

gbc.gridx = 0; //btn3 start at x0
gbc.gridy = 1; //btn3 starts at y1
gbc.gridwidth = 1; //set it back
gbc.gridheight = 1; //set it back
	gbc.gridx = 1; //btn4 start at x1
	gbc.gridy = 1; //btn4 starts at y1
	gbc.gridwidth = 1; //set it back
	gbc.gridheight = 1; //set it back

Button 5:	**Button 6:**

gbc.gridx = 2; //btn5 start at x2
gbc.gridy = 1; //btn5 starts at y1
gbc.gridwidth = 1; //set it back
gbc.gridheight = 1; //set it back
	gbc.gridx = 3; //btn6 start at x3
	gbc.gridy = 1; //btn6 starts at y1
	gbc.gridwidth = 1; //set it back
	gbc.gridheight = 1; //set it back

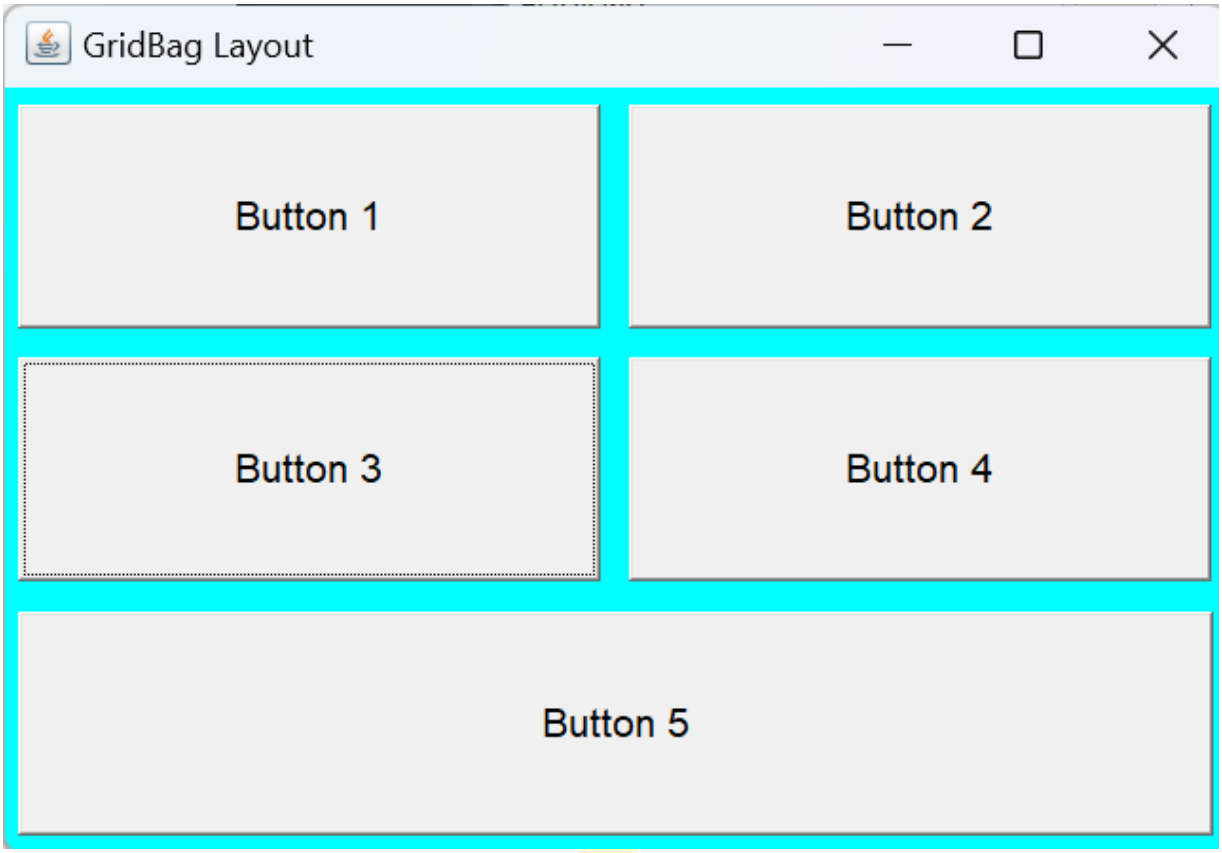
Button 7:	**Button 8:**

gbc.gridx = 0; //btn7 start at x0
gbc.gridy = 2; //btn7 starts at y2
gbc.gridwidth = 3; //span 3 columns
gbc.gridheight = 1; //set it back
	gbc.gridx = 0; //btn8 start at x0
	gbc.gridy = 3; //btn8 starts at y3
	gbc.gridwidth = 3; //span 2 columns
	gbc.gridheight = 1; //set it back

Button 9:	

gbc.gridx = 3; //btn9 start at x0
gbc.gridy = 2; //btn9 starts at y2
gbc.gridwidth = 1; //set it back
gbc.gridheight = 2; //span 2 rows	

For similar type of style then simply write it once, like btn3,4,5,6 gridwidth and gridheight are same so write it once.	



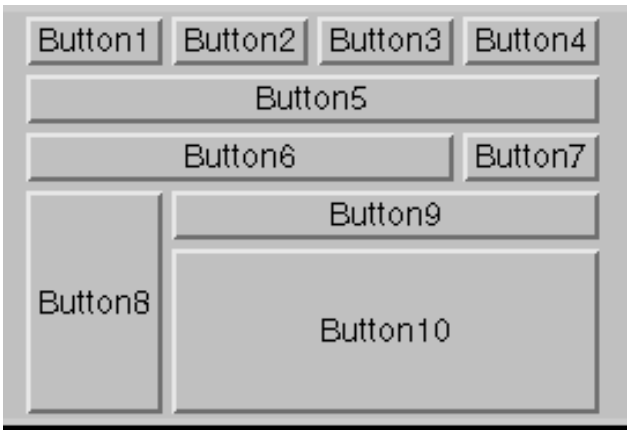
Example:

```
package LayoutManagers;
import java.awt.*;
import java.awt.Font.*;

public class ExampleGridBagLayout {
    Frame frame;
    GridBagLayout gridbag;
    GridBagConstraints gbc;
    Button b1, b2, b3, b4, b5;
    public ExampleGridBagLayout() {
        frame = new Frame();
        gridbag = new GridBagLayout();
        gbc = new GridBagConstraints();
        gbc.insets = new Insets(5, 5, 5, 5);
```

```
        b1 = new Button("Button 1");
        b2 = new Button("Button 2");
        b3 = new Button("Button 3");
        b4 = new Button("Button 4");
        b5 = new Button("Button 5");
        frame.setFont(new Font("Helvetica", Font.PLAIN, 14));
        frame.setBackground(Color.cyan);
        frame.setLayout(gridbag);
        gbc.weightx = 1;
        gbc.weighty = 1;
        gbc.fill = GridBagConstraints.BOTH;

        gbc.gridx = 0;
        gbc.gridy = 0;
//         frame.add(b1, gbc);
        gridbag.setConstraints(b1, gbc);
        frame.add(b1);

        gbc.gridx = 1;
        gbc.gridy = 0;
//         frame.add(b2, gbc);
        gridbag.setConstraints(b2, gbc);
        frame.add(b2);

        gbc.gridx = 0;
        gbc.gridy = 1;
//         frame.add(b3, gbc);
        gridbag.setConstraints(b3, gbc);
        frame.add(b3);
```

```
        gbc.gridx = 1;
        gbc.gridy = 1;
//        frame.add(b4, gbc);
        gridbag.setConstraints(b4, gbc);
        frame.add(b4);

        gbc.gridx = 0;
        gbc.gridy = 2;
        gbc.gridwidth = 2;
//        frame.add(b5, gbc);
        gridbag.setConstraints(b5, gbc);
        frame.add(b5);

        frame.setTitle("GridBag Layout");
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        ExampleGridBagLayout gbl = new ExampleGridBagLayout();
    }
}
```


Do it Yourself:



Border Layout:

The border layout is the default layout manager for all Window objects. It consists of five fixed areas: **North**, **South**, **East**, **West**, and **Center**. You do not have to put a component in every area of the border layout.

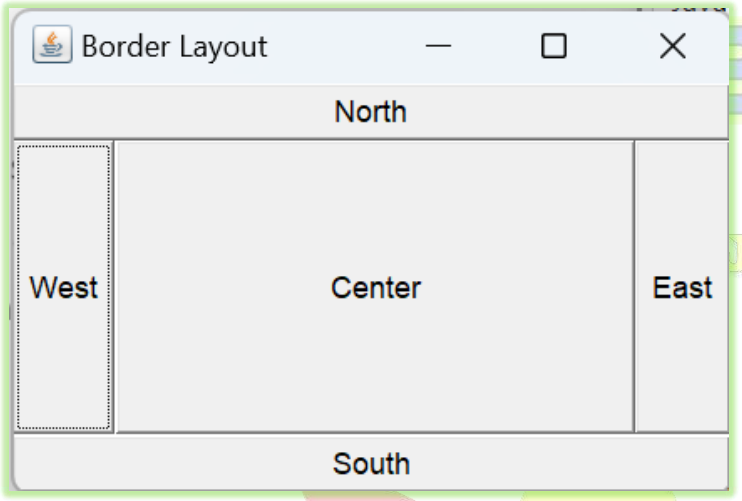
If any or all of the North, South, East, or West areas are left out, the Central area spreads into the missing area or areas. However, if the Central area is left out, the North, South, East, or West areas do not change.

You can indent the Central area by placing a label with one or more blank characters in the East or West areas.

Example:


```
package LayoutManagers;
import java.awt.*;
public class ExampleBorderLayout {
    Frame frame;
    Button btn1, btn2, btn3, btn4, btn5;

    public ExampleBorderLayout() {
        frame = new Frame();
        frame.setLayout(new BorderLayout());

        btn1 = new Button("North");
        btn2 = new Button("South");
        btn3 = new Button("East");
        btn4 = new Button("West");
        btn5 = new Button("Center");

        frame.add(btn1, BorderLayout.NORTH);
        frame.add(btn2, BorderLayout.SOUTH);
        frame.add(btn3, BorderLayout.EAST);
        frame.add(btn4, BorderLayout.WEST);
        frame.add(btn5, BorderLayout.CENTER);
```

```
        frame.setTitle("Border Layout");
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        ExampleBorderLayout bl = new ExampleBorderLayout();
    }
}
```



Group Layout:

Group Layout is typically used in **Swing** (**as opposed to AWT**) and is designed to support the hierarchical grouping of components. It's especially useful for designing forms where components are arranged based on their alignment.

Features:

- Organizes components in parallel groups or sequential groups.
- Provides finer control over alignment and sizing, typically used with GUI builders.

To change the button size use Dimension Class and setPreferredSize() method:

- `btn1.setPreferredSize(new Dimension(100,50));`

To change the icon of Frame use Image Class and setPreferredSize() method:

```
Image image = Toolkit.getDefaultToolkit().getImage("D:\\Pictures\\mypic.jpg");
frame.setIconImage(image);
```

Using SetBound method:

In Java's AWT and Swing, the **setBounds()** method is used to manually specify the position and size of a component within a container.

It takes four parameters:

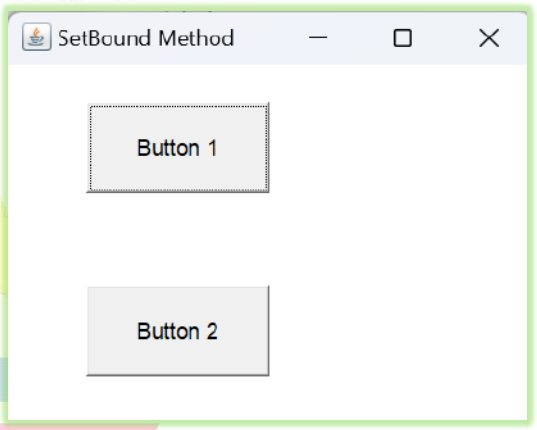
```
setBounds(int x, int y, int width, int height)
```

Where:

- **x:** The x-coordinate of the component's top-left corner relative to its parent container.
- **y:** The y-coordinate of the component's top-left corner relative to its parent container.
- **width:** The width of the component.
- **height:** The height of the component.

Using **setBounds()** effectively gives you full control over the layout, but it requires turning off the layout manager by calling **setLayout(null)** on the container. This is called **absolute positioning**, where you manually define the location and size of each component.

Example:




```
package LayoutManagers;
import java.awt.*;

public class ExampleSetBoundsMethod {
    Frame frame;
    Button btn1, btn2;
    public ExampleSetBoundsMethod()
    {
        frame = new Frame();
        btn1 = new Button("Button 1");
        btn2 = new Button("Button 2");
        frame.add(btn1);
        frame.add(btn2);
        btn1.setBounds(50, 50, 100, 50);
        btn2.setBounds(50, 150, 100, 50);
        frame.setTitle("SetBound Method");
        frame.setLayout(null);
        frame.setSize(300,300);
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        ExampleSetBoundsMethod sbm = new ExampleSetBoundsMethod();
    }
}
```

1.4. AWT Controls: TextField, TextArea, Button, Label, Checkbox, Checkbox Group, Choice, List, Canvas, Image.

Controls are components that allow a user to interact with your application in various ways.

TextField:

- A single-line text input field.
- Allows users to enter and edit a single line of text.
- **Example:** `TextField textField = new TextField();`

TextArea:

- A multi-line text input field.
- Used when you need to handle multiple lines of text input or display.
- **Example:** `TextArea textArea = new TextArea();`

Button:

- A push button that triggers an action when clicked.
- **Example:** `Button button = new Button("Click Me");`

Label:

- Displays a non-editable text or image.
- Used for displaying information.
- **Example:** `Label label = new Label("This is a label");`

Checkbox:

- Allows a user to make a choice between true/false or yes/no.
- **Example:** `Checkbox checkbox = new Checkbox("Accept Terms and Conditions");`

CheckboxGroup:

- Allows grouping of multiple checkboxes to behave like radio buttons (only one can be selected at a time).
- **Example:**

```
CheckboxGroup group = new CheckboxGroup();
Checkbox check1 = new Checkbox("Option 1", group, false);
Checkbox check2 = new Checkbox("Option 2", group, true);
```

Choice:

- A dropdown list where the user can select one item from the list.
- **Example:**

```
Choice choice = new Choice();
choice.add("Option 1");
choice.add("Option 2");
```

List:

- Displays a list of items where the user can select one or more.
- **Example:**

```
List list = new List();
list.add("Item 1");
list.add("Item 2");
```

Canvas:

- A blank rectangular area where custom drawing or painting can be done.
- Used when you need to create your own graphics.
- **Example:**

```
Canvas canvas = new Canvas();
```

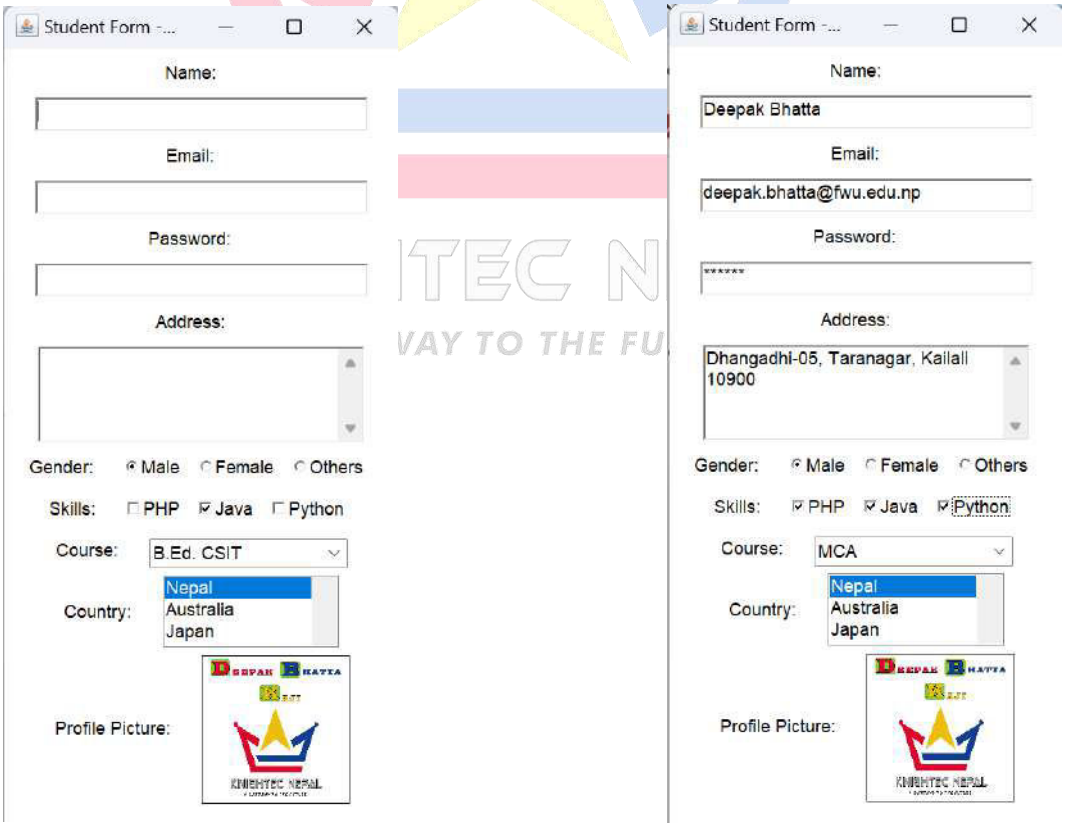
Image:

- Used to display an image.
- You typically load the image using the Toolkit class.
- **Example:**

```
Image image =
Toolkit.getDefaultToolkit().getImage("path/to/image.jpg");
```

Key Points to Remember:

1. **TextField**: For single-line input (name).
2. **TextArea**: For multi-line input (address).
3. **Checkbox and CheckboxGroup**: For selecting gender and skills.
4. **Choice**: For selecting a course from a dropdown.
5. **List**: For selecting a country.
6. **Button**: To submit the form.
7. **Canvas**: Used here to represent an image placeholder.
8. **Event Handling**: Captures form data upon clicking the submit button and prints it to the console.



```
package AWTComponentsExamples;
import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class StudentForm extends Frame {

    Label lblName, lblEmail, lblPassword, lblAddress, lblGender,
lblSkills, lblImage, lblCourse, lblCountry;
    TextField txtName, txtEmail, txtPassword;
    TextArea txtAddress;
    Checkbox ckboxPHP, ckboxJava, ckboxPython;
    CheckboxGroup ckboxGender;
    Checkbox ckboxMale, ckboxFemale, ckboxOthers;
    Choice chCourse;
    List listCountry;
    Button btnSubmit, btnClear;
    Canvas canvas;

    public StudentForm() {
        setLayout(new FlowLayout());

        lblName = new Label("Name: ");
        add(lblName);
        txtName = new TextField(20);
        add(txtName);

        lblEmail = new Label("Email: ");
        add(lblEmail);
        txtEmail = new TextField(20);
        add(txtEmail);
```

```
lblPassword = new Label("Password: ");
add(lblPassword);
txtPassword = new TextField(20);
add(txtPassword);
txtPassword.setEchoChar('*');

lblAddress = new Label("Address: ");
add(lblAddress);
txtAddress = new TextArea(3, 20);
add(txtAddress);

lblGender = new Label("Gender: ");
add(lblGender);

ckboxGender = new CheckboxGroup();
ckboxMale = new Checkbox("Male", ckboxGender, true);
ckboxFemale = new Checkbox("Female", ckboxGender, false);
ckboxOthers = new Checkbox("Others", ckboxGender, false);
add(ckboxMale);
add(ckboxFemale);
add(ckboxOthers);

lblSkills = new Label("Skills: ");
add(lblSkills);

ckboxPHP = new Checkbox("PHP");
ckboxJava = new Checkbox("Java", true);
ckboxPython = new Checkbox("Python");
add(ckboxPHP);
add(ckboxJava);
add(ckboxPython);
```

```
lblCourse = new Label("Course: ");
add(lblCourse);

chCourse = new Choice();
chCourse.setSize(10, 50);
chCourse.add("--- Select Course ---");
chCourse.add("B.SC. CSIT");
chCourse.add("B.Ed. CSIT");
chCourse.select(2);
chCourse.add("BE Computer");
chCourse.add("BCA");
chCourse.add("BIT");
chCourse.add("BIM");
chCourse.add("MCA");
add(chCourse);

lblCountry = new Label("Country: ");
add(lblCountry);

listCountry = new List(3, true);
listCountry.add("USA");
listCountry.add("Russia");
listCountry.add("China");
listCountry.add("India");
listCountry.add("Nepal");
listCountry.select(4);
listCountry.add("Australia");
listCountry.add("Japan");
add(listCountry);
```

```
        lblImage = new Label("Profile Picture: ");
        add(lblImage);
        Image image = Toolkit.getDefaultToolkit().getImage("H:\\One
Drive KMC\\OneDrive - Kailali Multiple Campus\\logo.png");

        canvas = new Canvas() {
            public void paint(Graphics g) {
                g.drawRect(0, 0, 100, 100);
                g.drawImage(image, 0, 0, 100, 100, this);
            }
        };
        canvas.setSize(100, 100);
        add(canvas);

        setTitle("Student Form - AWT JAVA");
        setSize(280, 600);
        setVisible(true);

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });
    }

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

1.5. AWT Menu: Menu Hierarchy, Menu, MenuBar, MenuItem, PopupMenu

In Java AWT, menus allow users to interact with the application through a structured hierarchical list of commands.

1. Menu Hierarchy

- In Java AWT, the menu structure is hierarchical. A MenuBar holds multiple Menu objects, each of which can contain MenuItem objects. You can also nest menus to create submenus.

2. MenuBar

- The MenuBar is the bar typically located at the top of a window that contains multiple Menu objects.
- Each Frame can have one MenuBar, and this is where menus like "File," "Edit," etc., are placed.
- **Example:**

```
MenuBar menuBar = new MenuBar();
setMenuBar(menuBar);  // Add it to the frame
```

3. Menu

- A Menu is a dropdown list that contains multiple MenuItem objects or submenus.
- **Example:**

```
Menu fileMenu = new Menu("File");
menuBar.add(fileMenu);  // Add the menu to the menu bar
```

4. MenuItem

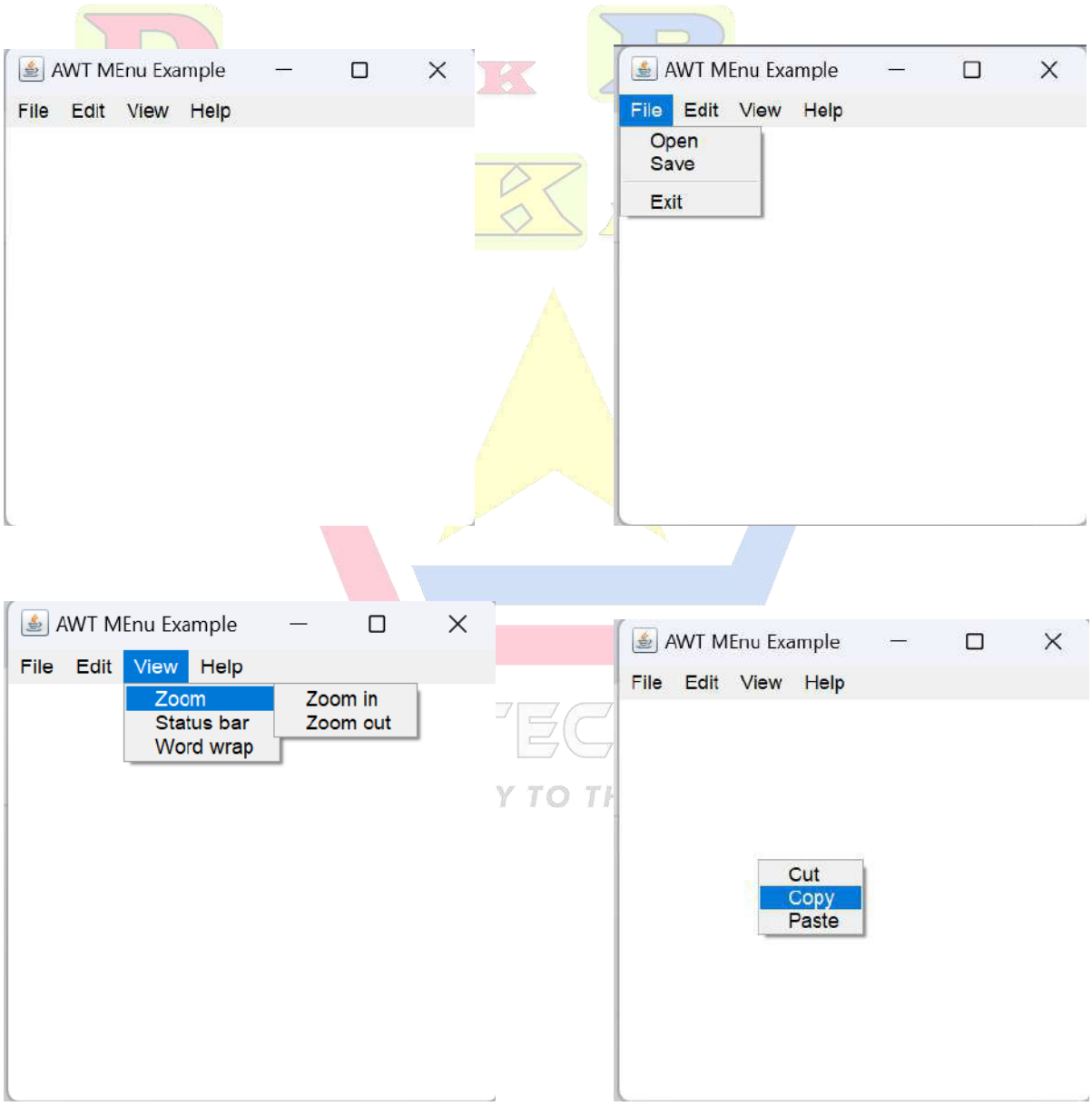
- A MenuItem represents an individual item inside a Menu. Clicking on it triggers an action.
- **Example:**

```
MenuItem openItem = new MenuItem("Open");
fileMenu.add(openItem);  // Add the menu item to the File menu
```

5. PopupMenu

- A PopupMenu is a context-sensitive menu that appears on right-click (or another trigger).
- It is typically used for right-click options within the window.
- **Example:**

```
PopupMenu popupMenu = new PopupMenu();
add(popupMenu);   // Add it to the component where it will be displayed
```



```
package AWTComponentsExamples;

import java.awt.*;

import java.awt.event.*;

public class AWTMenuExample extends Frame {

    MenuBar menuBar;

    Menu menuFile, menuEdit, menuView, menuZoom, menuHelp;

    MenuItem itemOpen, itemSave, itemExit, itemZoomIn, itemZoomOut,
itemStatusBar, itemWordWrap;

    PopupMenu popupMenu;

    public AWTMenuExample() {

        menuBar = new MenuBar();

        setMenuBar(menuBar);

        menuFile = new Menu("File");

        menuEdit = new Menu("Edit");

        menuView = new Menu("View");

        menuHelp = new Menu("Help");

        menuBar.add(menuFile);

        menuBar.add(menuEdit);

        menuBar.add(menuView);

        menuBar.add(menuHelp);

        itemOpen = new MenuItem("Open");

        itemSave = new MenuItem("Save");

        itemExit = new MenuItem("Exit");

        menuFile.add(itemOpen);

        menuFile.add(itemSave);

        menuFile.addSeparator();
```

```
        menuFile.add(itemExit);

        menuZoom = new Menu("Zoom");
        itemZoomIn = new MenuItem("Zoom in");
        itemZoomOut = new MenuItem("Zoom out");
        itemStatusBar = new MenuItem("Status bar");
        itemWordWrap = new MenuItem("Word wrap");

        menuView.add(menuZoom);
        menuView.add(itemStatusBar);
        menuView.add(itemWordWrap);
        menuZoom.add(itemZoomIn);
        menuZoom.add(itemZoomOut);

        popupMenu = new PopupMenu();

        MenuItem cut = new MenuItem("Cut");
        MenuItem copy = new MenuItem("Copy");
        MenuItem paste = new MenuItem("Paste");

        popupMenu.add(cut);
        popupMenu.add(copy);
        popupMenu.add(paste);
        add(popupMenu);

        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                if (e.isPopupTrigger()) {
                    popupMenu.show(e.getComponent(), e.getX(),
e.getY());
                }
            }
```

```
            public void mouseReleased(MouseEvent e) {
                if (e.isPopupTrigger()) {
                    popupMenu.show(e.getComponent(), e.getX(),
e.getY());
                }
            }
        });

        setTitle("AWT MEnu Example");
        setSize(300, 300);
        setVisible(true);

        addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }

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

1.6. AWT Graphics: Graphics and Graphics2D Class, Drawing Lines, Curves, rectangles, ellipse, Changing Color & Font

In Java AWT, the Graphics and Graphics2D classes are essential for drawing shapes, lines, and text on a GUI. The Graphics class is the base class for all drawing operations, while Graphics2D (a subclass of Graphics) provides more sophisticated control over geometry, coordinate transformations, color management, and text layout.

- **Graphics Class**: Provides basic methods for drawing shapes, text, and images on a component.
- **Graphics2D Class**: Extends Graphics and adds more advanced features, like rendering hints, strokes, and transformations.

Drawing Lines:

- Use g2.drawLine(x1, y1, x2, y2) to draw a line from point (x1, y1) to (x2, y2).
- **Example:**

```
g2.drawLine(50, 100, 300, 100);
```

Drawing Curves:

- QuadCurve2D and CubicCurve2D are used for drawing quadratic and cubic Bezier curves respectively.
- **Example:**

```
g2.draw(new QuadCurve2D.Float(550, 450, 650, 350, 750, 450));
```

Drawing Rectangles:

- Use g2.drawRect(x, y, width, height) to draw a rectangle outline.
- Use g2.fillRect(x, y, width, height) to draw a filled rectangle.
- **Example:**

```
g2.drawOval(50, 300, 200, 100);  // Outline
g2.fillOval(300, 300, 200, 100); // Filled
```

Drawing Ellipse:

- Use g2.drawOval(x, y, width, height) to draw an ellipse outline.
- Use g2.fillOval(x, y, width, height) to draw a filled ellipse.
- **Example:**

```
g2.drawOval(50, 300, 200, 100);  // Outline
g2.fillOval(300, 300, 200, 100); // Filled
```

Changing Color:

- Use g2.setColor(Color.RED) (or any other color) to set the color for drawing shapes.
- You can switch between different colors for each shape drawn.
- **Example:**

```
g2.setColor(Color.BLUE);
```

Changing Font:

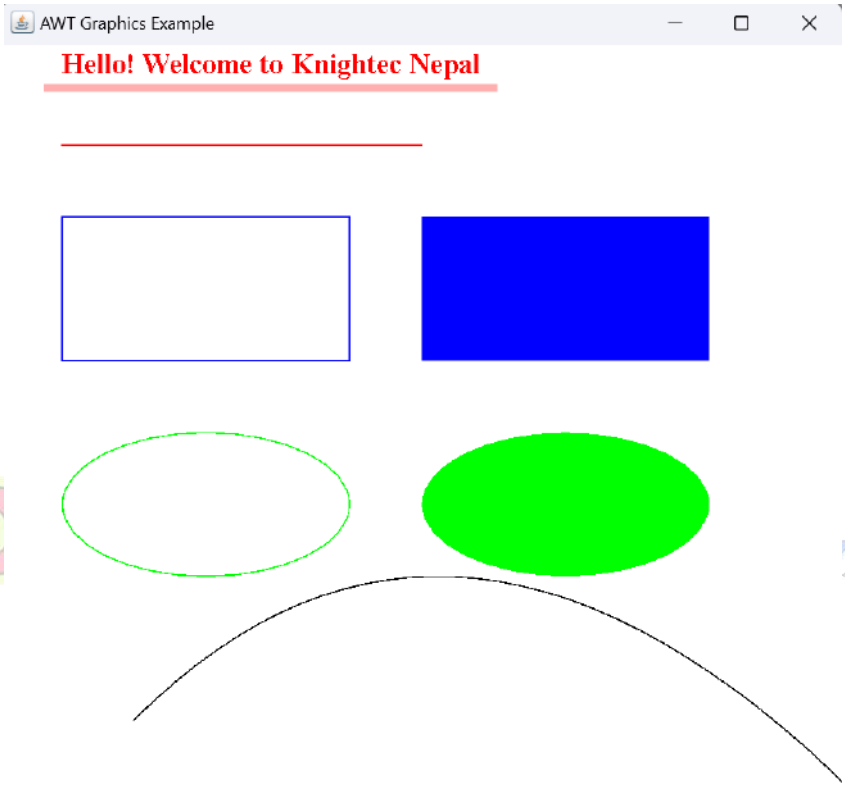
- Use g2.setFont(new Font("FontName", FontStyle, FontSize)) to set a specific font for drawing text.
- **Example:**

```
g2.setFont(new Font("Serif", Font.BOLD, 24));
g2.drawString("AWT Graphics Example", x, y) // draws the text at a specified location.
```

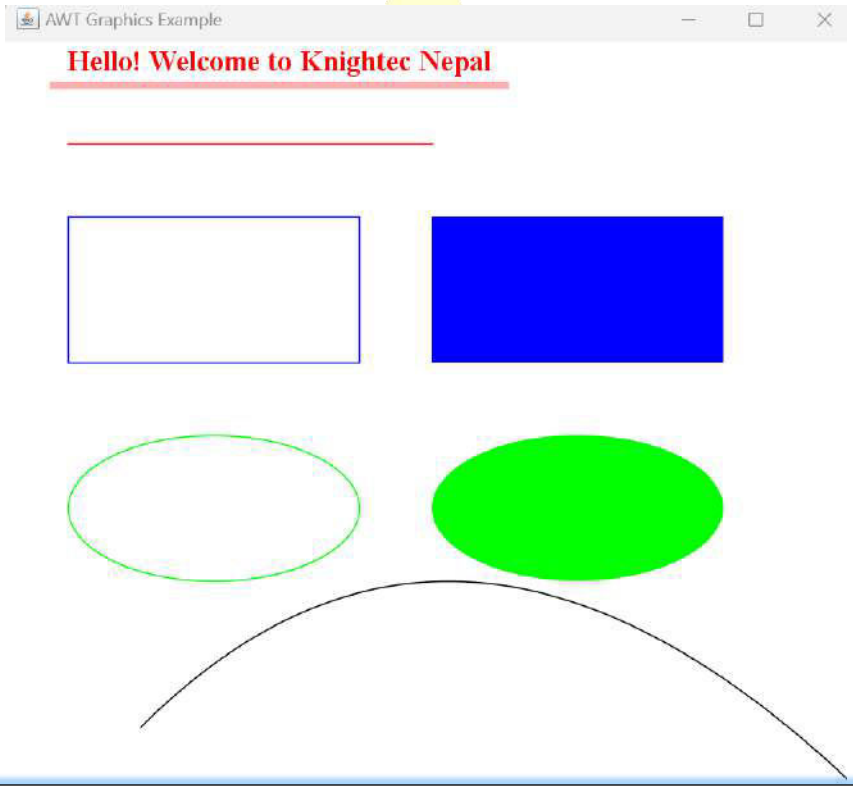
Changing Stroke (Line Thickness):

- Use g2.setStroke(new BasicStroke(thickness)) to change the thickness of lines or shapes.
- **Example:**

```
g2.setStroke(new BasicStroke(5)); // 5-pixel thick line
g2.drawLine(50, 550, 300, 550);   // Drawing a thicker line
```



Output: Without using Rendering



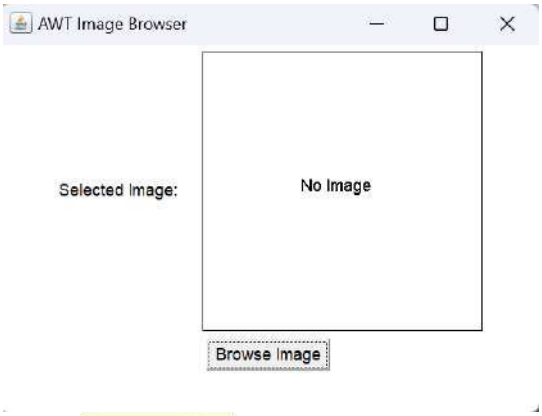

```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
```

Output: Using Rendering

```
package AWTComponentsExamples;
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.QuadCurve2D;

public class DrawingGraphicsExample extends Frame {

    public DrawingGraphicsExample() {
        setTitle("AWT Graphics Example");
        setSize(600, 600);
        setVisible(true);

        addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }

    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;
        //g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
        g2.setColor(Color.RED);
        g2.drawLine(50, 100, 300, 100);

        g2.setColor(Color.BLUE);
        g2.drawRect(50, 150, 200, 100);
        g2.fillRect(300, 150, 200, 100);
```

```
        g2.setColor(Color.GREEN);
        g2.drawOval(50, 300, 200, 100);
        g2.fillOval(300, 300, 200, 100);

        g2.setColor(Color.BLACK);
        g2.draw(new QuadCurve2D.Float(100, 500, 400, 200, 800, 800));

        g2.setColor(Color.RED);
        g2.setFont(new Font("Serif", Font.BOLD, 20));
        g2.drawString("Hello! Welcome to Knightec Nepal", 50, 50);

        g2.setStroke(new BasicStroke(5));
        g2.setColor(Color.PINK);
        g2.drawLine(40, 60, 350, 60);

    }

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

}
```

Extra Dose - Image Browser:




```
package AWTComponentsExamples;

import java.awt.*;
import java.awt.event.*;
import java.io.File;

public class AWTImageBrowse extends Frame {

    Label imageLabel;
    Button browseButton;
    Canvas canvas;
    Image selectedImage;

    public AWTImageBrowse() {
        // Set layout for the frame
        setLayout(new FlowLayout());

        // Label for Image
        imageLabel = new Label("Selected Image:");
        add(imageLabel);
```

```
        // Canvas to display the image
        canvas = new Canvas() {
            public void paint(Graphics g) {
                if (selectedImage != null) {
                    g.drawImage(selectedImage, 0, 0, 200, 200, this);
// Draw the selected image
                } else {
                 g.drawRect(0, 0, 200, 200); // Placeholder rectangle
                 g.drawString("No Image", 70, 100);
                }
            }
        };
        canvas.setSize(200, 200);
        add(canvas);

        // Button to Browse image
        browseButton = new Button("Browse Image");
        add(browseButton);

        // Action listener for Browse button
        browseButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                FileDialog fileDialog = new
FileDialog(AWTImageBrowse.this, "Select an Image", FileDialog.LOAD);
                fileDialog.setVisible(true);

                // Get the selected file's path
                String directory = fileDialog.getDirectory();
                String fileName = fileDialog.getFile();
```

```
                if (directory != null && fileName != null) {
                    String filePath = directory + fileName;
                    selectedImage =
Toolkit.getDefaultToolkit().getImage(filePath);
                    repaint(); // Repaint the canvas to display the
new image
                }
            }
        });

        // Set frame properties
        setTitle("AWT Image Browser");
        setSize(400, 300);
        setVisible(true);

        // Close the frame on window close
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });
    }

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


THE END

Unit 2 8 hrs.

GUI with Swing

Specific Objectives

- Compare Swing with AWT and understand differences.
- Use Swing library to create GUI.
- with different controls and menus.
- Demonstrate the use of advanced swing components.
- Demonstrate the use of dialog boxes and internal frames.
- Understand the use of different component organizers.

2.1. Swing Basics: Swing Hierarchy, Swing Features, AWT vs Swing.

2.2. Text Input: Text Fields, Password Fields, Text Areas, Scroll Pane, Label and Labelling Components.

2.3. Choice Components: Check Boxes, Radio Buttons, Borders, Combo Boxes, Sliders.

2.4. Menus: Menu Building, Icons in Menu Items, Check box and Radio Buttons in Menu Items, Pop-up Menus, Keyboard Mnemonics and Accelerators, Enabling and Disabling menu Items, Toolbars, Tooltips.

2.5. Dialog Boxes: Option Dialogs, Creating Dialogs, Data Exchange, File Choosers, Color Choosers.

2.6. Components Organizers: Split Panes, Tabbed Panes, Desktop Panes and Internal Frames, Cascading and Tiling.

2.7. Advance Swing Components: List, Trees, Tables, Progress Bars.

2.1. Swing Basics: Swing Hierarchy, Swing Features, AWT vs Swing.

Introduction to Swing:

Java Swing is a powerful toolkit for creating **Graphical User Interfaces** (GUIs) in Java. It provides a wide range of components, such as buttons, text fields, and menus, that can be used to create attractive and functional GUIs. Swing is platform-independent, which means that your *GUIs will look the same on any platform, including Windows, Mac OS X, and Linux*.

What is Java Swing?

Java Swing is a set of graphical user interface (GUI) components that are part of the Java platform. Swing components are built on top of the Abstract Window Toolkit (AWT), but they provide a number of advantages over AWT components. For example, *Swing components are more lightweight and efficient, and they are platform-independent.*

Java Swing is included in the Java Development Kit (JDK). If you already have the JDK installed not to worry, just go for your first GUI application using Swing.

Components and Containers

A Swing GUI consists of two key items: ***components*** and ***containers***. However, this distinction is mostly conceptual because all containers are also components. The difference between the two is found in their intended purpose: As the term is commonly used, a component is an independent visual control, such as a push button or slider. A container holds a group of components.

Thus, a container is a special type of component that is designed to hold other components. Furthermore, in order for a component to be displayed, it must be held within a container.

Thus, all Swing GUIs will have at least one container. Because containers are components, a container can also hold other containers. This enables Swing to define what is called a containment hierarchy, at the top of which must be a *top-level container*.

Components

In general, Swing components are derived from the **JComponent** class. (The only exceptions to this are the four top-level containers, described in the next section.) **JComponent** provides the functionality that is common to all components. For example, **JComponent** supports the pluggable look and feel. **JComponent** inherits the AWT classes Container and Component. Thus, a Swing component is built on and compatible with an AWT component. All of Swing's components are represented by classes defined within the package **javax.swing**.

The following table shows the class names for Swing components (including those used as containers).

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

Notice that all component classes begin with the letter **J**. For example, the class for a label is **JLabel**; the class for a push button is **JButton**; and the class for a scroll bar is **JScrollBar**.

Containers

Swing defines two types of containers. The first are top-level containers: **JFrame**, **JApplet**, **JWindow**, and **JDialog**. These containers do not inherit **JComponent**. They do, however, inherit the AWT classes **Component** and **Container**. Unlike Swing's other components, which are lightweight, the top-level containers are heavyweight. This makes the top-level containers a special case in the Swing component library. As the name implies, a top-level container must be at the top of a containment hierarchy. A top-level container is not contained

within any other container. Furthermore, every containment hierarchy must begin with a top-level container.

The one most commonly used for applications is **JFrame**. The one used for applets is **JApplet**. The second type of containers supported by Swing are lightweight containers. Lightweight containers do inherit **JComponent**.

An example of a lightweight container is **JPanel**, which is a general-purpose container. Lightweight containers are often used to organize and manage groups of related components because a lightweight container can be contained within another container.

Thus, you can use lightweight containers such as **JPanel** to create subgroups of related controls that are contained within an outer container.

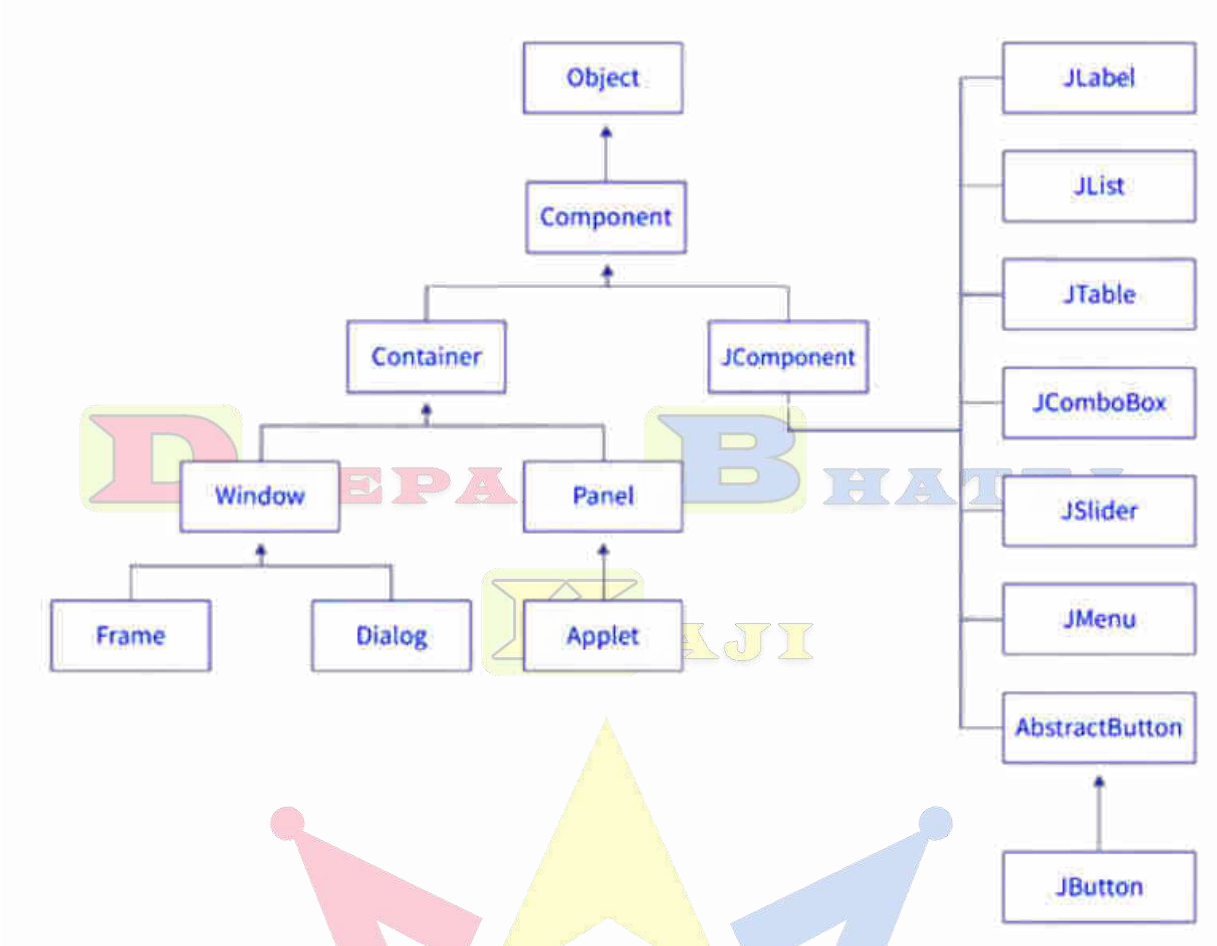
The Swing Packages

Swing is a very large subsystem and makes use of many packages. At the time of this writing, these are the packages defined by Swing.

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

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

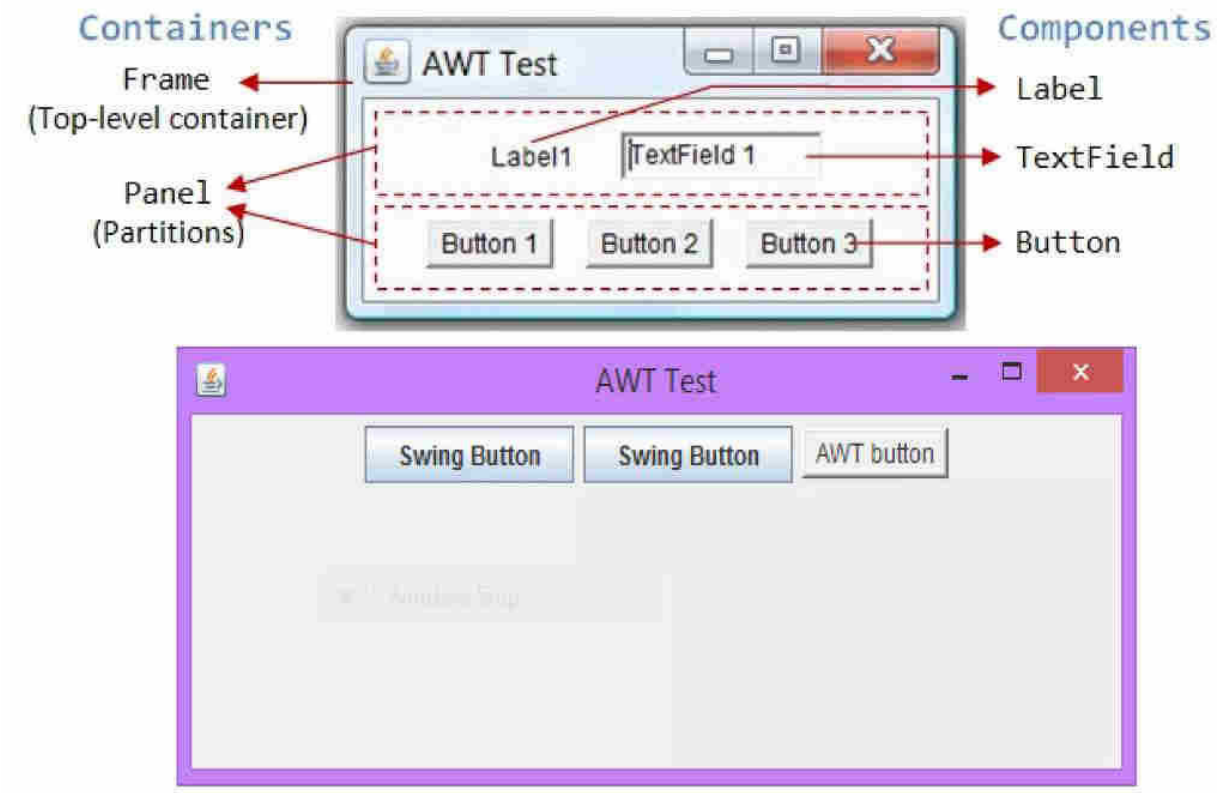
Swing Hierarchy:



- **java.lang.Object**
  - The root class for all Java classes.
- **java.awt.Component**
  - The superclass of all AWT and Swing components. This class defines the properties of a visual element (like size, color, position).
- **java.awt.Container**
  - A subclass of Component that can hold other components (e.g., JPanel, JFrame).
- **javax.swing.JComponent**
  - The base class for all Swing components that inherit visual behavior and customization options.

Swing Features:

- ✓ **Lightweight Components** – Swing components are independent of the operating system.
- ✓ **Pluggable Look and Feel** – You can change the application's appearance without changing the code.
- ✓ **Platform Independence** – Swing runs on any system with a Java Virtual Machine (JVM).
- ✓ **Rich Set of Components** – Provides buttons, labels, text fields, tables, etc.
- ✓ **Event Handling** – Uses a powerful system for managing user actions.
- ✓ **Customizable Components** – You can modify the appearance and behavior of components.
- ✓ **MVC Architecture** – Separates data, view, and control logic.
- ✓ **Double Buffering** – Reduces flickering and improves GUI performance.
- ✓ **Custom Painting** – Allows you to create custom graphics in components.
- ✓ **Thread-Safety** – Ensures safe UI updates via the Event Dispatch Thread.
- ✓ **Drag and Drop Support** – Enables moving data between components or applications.
- ✓ **Internationalization** – Supports building applications for multiple languages.



AWT vs Swing:

AWT	Swing
In Java, awt is an API that is used for developing GUI (Graphical user interface) applications.	Swing on another hand, in Java, is used for creating various applications including the web.
Java AWT components are heavily weighted.	Java swing components are light-weighted.
Java AWT has fewer functionalities as compared to that of swing.	Swing has greater functionalities as compared to awt in Java.
The code execution time in Java AWT is more.	The execution time of code in Java swing is less compared to that of awt.
In Java AWT, the components are platform-dependent.	The swing components are platform-independent.
MVC (Model-View-Controller) is not supported by Java awt.	MVC (Model-View-Controller) is supported by Java swing.
In Java, awt provides less powerful components.	Swing provide more powerful components.
The awt components are provided by the package java.awt	The swing components are provided by javax.swing package
There is abundant features in awt that are developed by developers and act as a thin layer between development and operating system.	Swing has a higher level of in-built components for developers which facilitates them to write less code.
The awt in Java has slower performance as compared to swing.	Swing is faster than that of awt.
AWT Components are dependent on the operating system.	Swing components are completely scripted in Java. Its components are not dependent on operating system.
Java AWT stands for Abstract Window Toolkit.	Java Swing is mainly referred to as Java Foundation Classes (JFC).

2.2. Text Input: Text Fields, Password Fields, Text Areas, Scroll Pane, Label and Labelling Components.

Text Fields:

- A single-line text input field that allows users to enter and edit text.
- **Example:**

```
JTextField textField = new JTextField(20); // 20 columns wide
```

Password Fields:

- Similar to JTextField but hides the input for secure text entry, often used for passwords.
- **Example:**

```
JPasswordField passwordField = new JPasswordField(20);
```

Text Areas:

- A multi-line area for displaying and editing large amounts of text.
- **Example:**

```
JTextArea textArea = new JTextArea(5, 20); // 5 rows and 20 columns
```

Scroll Pane:

- A container that provides scroll bars for components that have too much content to fit in the visible area, often used with JTextArea or other large components.
- **Example:**

```
JScrollPane scrollPane = new JScrollPane(textArea); // adds scrolling to textArea
```

Labels:

- A non-editable text component used to display a label or description, typically for other components like text fields or buttons.
- **Example:**

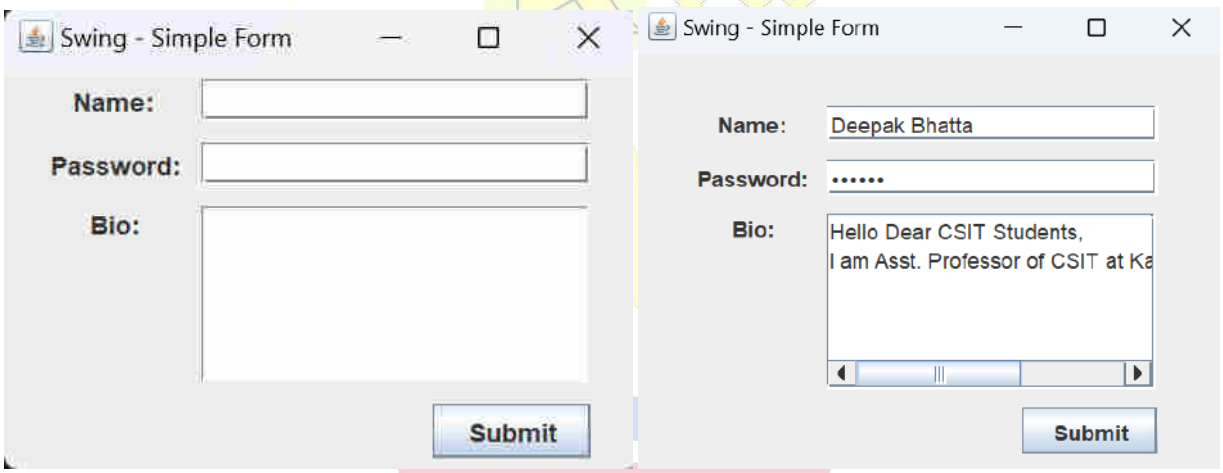
```
JLabel label = new JLabel("Enter your name:");
```

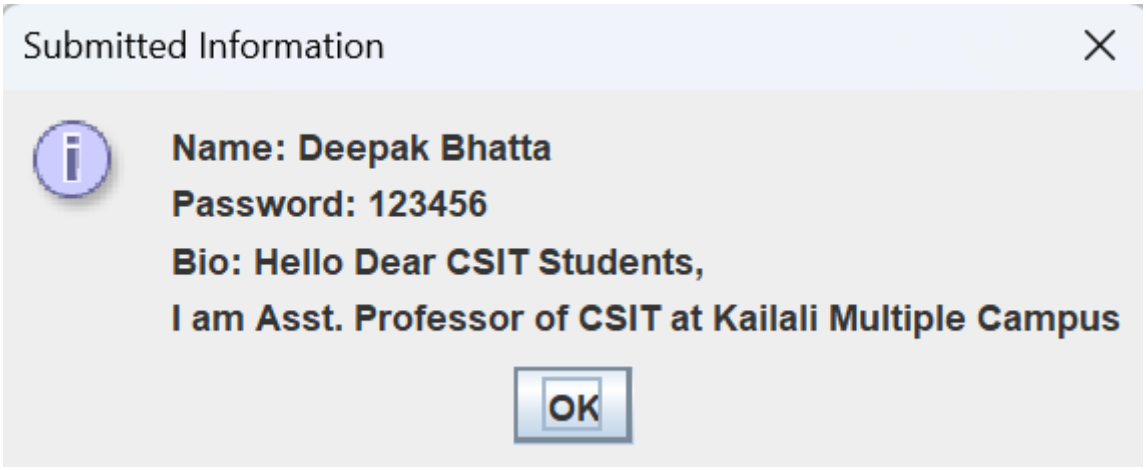
Labelling Components:

- Components like **JLabel** are often used for labeling input fields and buttons to provide a user-friendly interface.
- **setLabelFor()** method can be used to associate a label with a specific component for better accessibility.
- **Example:**

```
JLabel nameLabel = new JLabel("Name:");
nameLabel.setLabelFor(textField);
```


Example:






```
package SwingComponentsExamples;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class SimpleFormTextFields extends JFrame {

    public SimpleFormTextFields() {
        // Create frame with title
        super("Swing - Simple Form");

        // Set the layout
        setLayout(new GridBagLayout());
        GridBagConstraints gbc = new GridBagConstraints();
        gbc.insets = new Insets(5, 5, 5, 5); // Padding for components

        // Create components
        JLabel nameLabel = new JLabel("Name:");
        JTextField nameField = new JTextField(20);

        JLabel passwordLabel = new JLabel("Password:");
        JPasswordField passwordField = new JPasswordField(20);

        JLabel bioLabel = new JLabel("Bio:");
```

```
        JTextArea bioArea = new JTextArea(5, 20);

        JScrollPane bioScrollPane = new JScrollPane(bioArea,
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);

        JButton submitButton = new JButton("Submit");

  // Add components to the frame

        // Name Label
        gbc.gridx = 0;
        gbc.gridy = 0;
        add(nameLabel, gbc);

       // Name Field
        gbc.gridx = 1;
        gbc.gridy = 0;
        add(nameField, gbc);

       // Password Label
        gbc.gridx = 0;
        gbc.gridy = 1;
        add(passwordLabel, gbc);

       // Password Field
        gbc.gridx = 1;
        gbc.gridy = 1;
        add(passwordField, gbc);

       // Bio Label
        gbc.gridx = 0;
        gbc.gridy = 2;
        gbc.anchor = GridBagConstraints.NORTH;
        add(bioLabel, gbc);
```

```
        // Bio Text Area with Scroll Pane
         gbc.gridx = 1;
         gbc.gridy = 2;
         add(bioScrollPane, gbc);

        // Submit Button
         gbc.gridx = 1;
         gbc.gridy = 3;
         gbc.anchor = GridBagConstraints.EAST; // Align button to the right
         add(submitButton, gbc);
        // Action Listener for Submit Button
         submitButton.addActionListener(new ActionListener() {
             @Override
             public void actionPerformed(ActionEvent e) {
                 String name = nameField.getText();
                 String password = new String(passwordField.getPassword());
                 String bio = bioArea.getText();

// Display the input
 JOptionPane.showMessageDialog(SimpleFormTextFields.this, "Name: " + name +
"\nPassword: " + password + "\nBio: " + bio, "Submitted Information",
JOptionPane.INFORMATION_MESSAGE);
             }
         });

        // Frame settings
         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         setSize(400, 400);
//         pack(); // Adjusts the window to fit components
         setLocationRelativeTo(null); // Centers the window on the screen
         setVisible(true);
     }

     public static void main(String[] args) {
// Run the application
SimpleFormTextFields simpleFormTextFields = new SimpleFormTextFields();
```

```
    }
}
```

Components Used for above Application:

- **JTextField** and **JPasswordField** are used for single-line input fields (*for name and password*).
- **JTextArea** inside a **JScrollPane** allows multi-line text input for the bio section.
- **JLabel** components are used for labeling each input field.
- A **JButton** triggers an action that displays the entered data in a **JOptionPane** message dialog when the "Submit" button is clicked.

2.3. Choice Components: Check Boxes, Radio Buttons, Borders, Combo Boxes, Sliders.

Check Boxes:

- A checkbox allows users to select or deselect an option, typically for multiple-choice selections.
- **Example:**

```
JCheckBox checkBox = new JCheckBox("Accept Terms and Conditions");
```

Radio Buttons:

- Radio buttons allow users to select one option from a group of options. They are grouped using a ButtonGroup, so only one button in the group can be selected at a time.
- **Example:**

```
JRadioButton option1 = new JRadioButton("Option 1");
JRadioButton option2 = new JRadioButton("Option 2");
ButtonGroup group = new ButtonGroup();
group.add(option1);
group.add(option2);
```

Borders:

- Borders are used to visually group or emphasize components. There are different types of borders like LineBorder, EtchedBorder, TitledBorder, etc.
- **Example:**

```
JPanel panel = new JPanel();
panel.setBorder(BorderFactory.createTitledBorder("Personal Information"));
```

Combo Boxes:

- A combo box provides a drop-down list of items from which the user can select one. It can be editable, allowing the user to type a value or select from the list.
- **Example:**

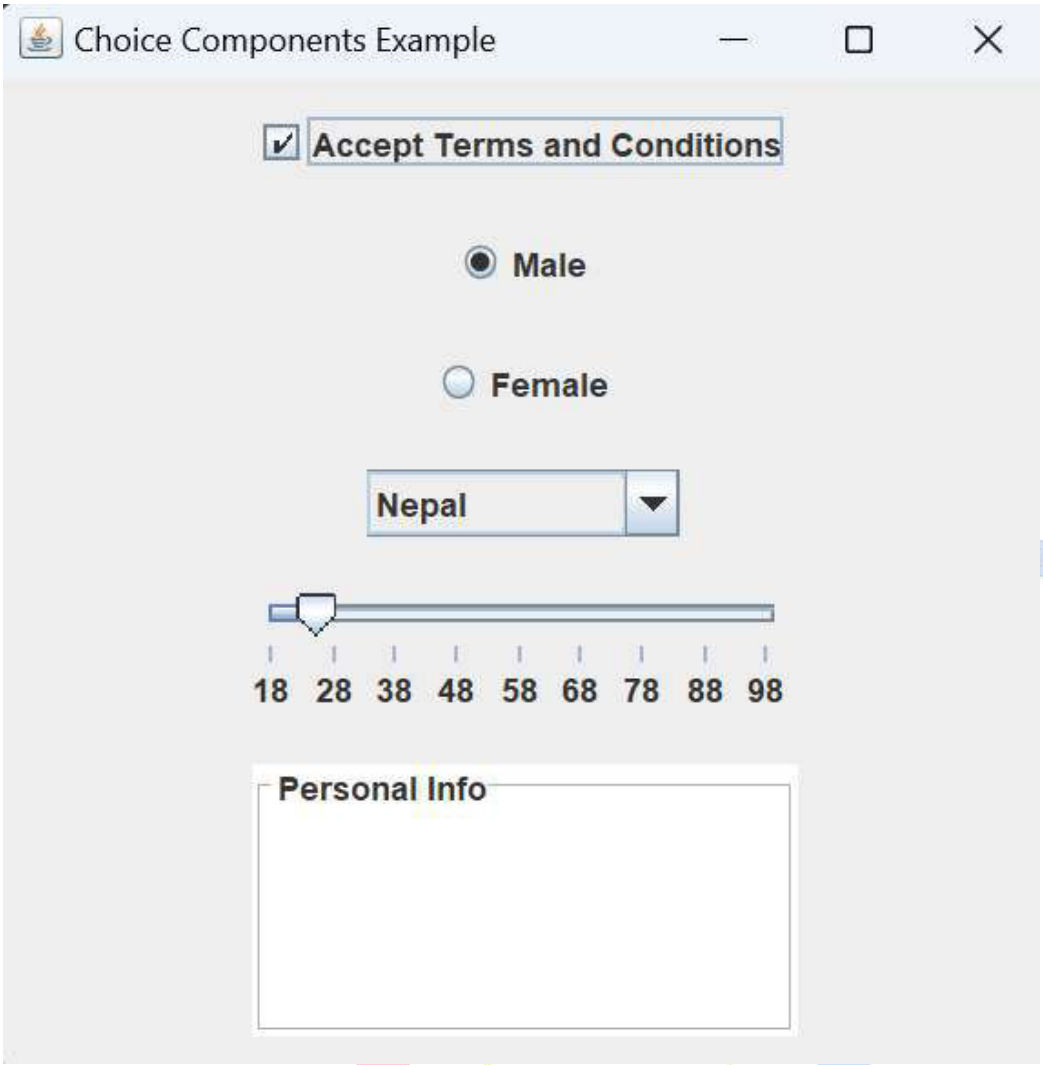
```
String[] items = { "Item 1", "Item 2", "Item 3" };
JComboBox<String> comboBox = new JComboBox<>(items);
```

Sliders:

- A slider allows users to select a value from a continuous range by sliding a knob. You can set minimum, maximum, and initial values.
- **Example:**

```
JSlider slider = new JSlider(JSlider.HORIZONTAL, 0, 100, 50);
// Min 0, Max 100, Initial 50
```

Example:



```
package SwingComponentsExamples;

import javax.swing.*;
import javax.swing.border.TitledBorder;
import java.awt.*;

public class SimpleFormChoice extends JFrame {

    public SimpleFormChoice() {
        // Create frame with title
        setTitle("Choice Components Example");

        // Set the layout
        setLayout(new GridBagLayout());
        GridBagConstraints gbc = new GridBagConstraints();
        gbc.insets = new Insets(10, 10, 10, 10); // Padding

        // Check Box
        JCheckBox checkBox = new JCheckBox("Accept Terms and Conditions",
true);
        gbc.gridx = 0;
        gbc.gridy = 0;
        add(checkBox, gbc);

        // Radio Buttons and Group
        JRadioButton maleButton = new JRadioButton("Male", true);
        JRadioButton femaleButton = new JRadioButton("Female");
        ButtonGroup genderGroup = new ButtonGroup();
        genderGroup.add(maleButton);
        genderGroup.add(femaleButton);

        gbc.gridx = 0;
        gbc.gridy = 1;
        add(maleButton, gbc);
```

```
        gbc.gridx = 0;
        gbc.gridy = 2;
        add(femaleButton, gbc);

       // Combo Box
        String[] countryList = {"Select Country", "USA", "Canada", "Nepal",
"India"};
        JComboBox<String> countryComboBox = new JComboBox<>(countryList);
        countryComboBox.setSelectedItem("Nepal");
        gbc.gridx = 0;
        gbc.gridy = 3;
        add(countryComboBox, gbc);

       // Slider
        JSlider ageSlider = new JSlider(JSlider.HORIZONTAL, 18, 100, 25);
// Min 18, Max 100, Initial 25
        ageSlider.setMajorTickSpacing(10);
        ageSlider.setPaintTicks(true);
        ageSlider.setPaintLabels(true);

        gbc.gridx = 0;
        gbc.gridy = 4;
        add(ageSlider, gbc);

       // Panel with Border
        JPanel infoPanel = new JPanel();
        infoPanel.setBackground(Color.WHITE);
        infoPanel.setPreferredSize(new Dimension(200, 100));

infoPanel.setBorder(BorderFactory.createTitledBorder(BorderFactory.createEt
chedBorder(), "Personal Info", TitledBorder.LEFT, TitledBorder.TOP));

        gbc.gridx = 0;
        gbc.gridy = 5;
        add(infoPanel, gbc);
```

```
        // Frame settings
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(400, 400);
        setLocationRelativeTo(null); // Center the window on the screen
        setVisible(true);
    }

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


Components Used for above Application:

- **Check Box**: A checkbox is added to let users select or deselect an option.
- **Radio Buttons**: Two radio buttons for gender selection are added and grouped using a ButtonGroup.
- **Combo Box**: A combo box with a list of countries allows the user to select one.
- **Slider**: A slider that lets users select an age between 18 and 100 is added, with ticks and labels.
- **Border**: A titled border is used around a JPanel for grouping "Personal Info."

2.4. Menus: Menu Building, Icons in Menu Items, Check box and Radio Buttons in Menu Items, Pop-up Menus, Keyboard Mnemonics and Accelerators, Enabling and Disabling menu Items, Toolbars, Tooltips.

Menus:

Menus are essential components of Java Swing applications, providing users with a structured way to access various functionalities.

- ✓ Menus can be nested to create hierarchical structures.
- ✓ Menu items can have accelerators (keyboard shortcuts) associated with them.
- ✓ Menu items can have icons associated with them.
- ✓ Menu items can have tooltips to provide additional information.
- ✓ Menu items can be disabled or enabled based on certain conditions.

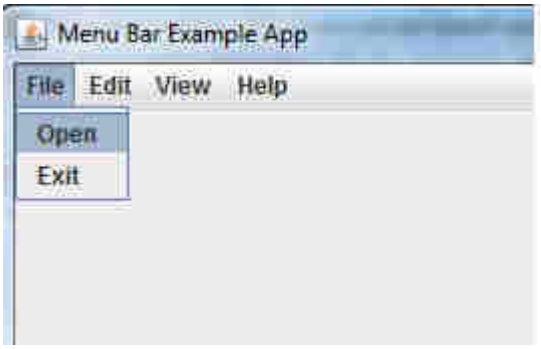
Swing offers two main types of menus: **MenuBar** and **PopupMenu**.

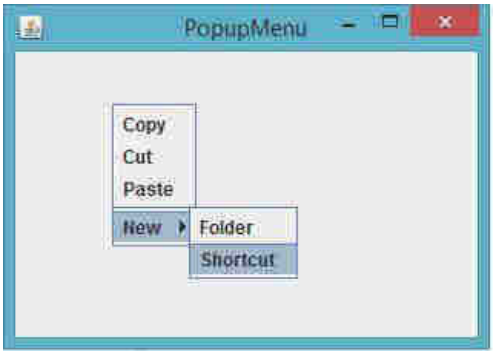
MenuBar

- A horizontal bar placed at the top of the application window.
- Contains one or more **JMenu** objects, which are like dropdown menus.
- Each JMenu can have **JMenuItem** objects, which represent individual menu items.
- Can be added to a JFrame or JApplet using the setJMenuBar() method.

PopupMenu

- A context-sensitive menu that appears when the user right-clicks on a component.
- Can be associated with a component using the setComponentPopupMenu() method.
- Works similarly to a MenuBar in terms of JMenu and JMenuItem objects.





Menu Building:

You can create a menu bar with multiple menus and add items to those menus.

Example:

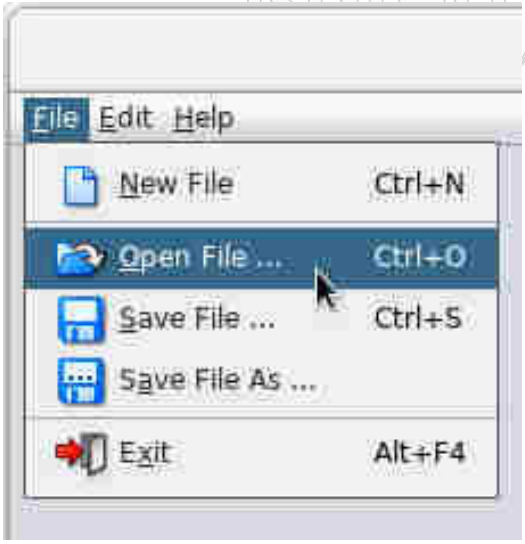
```
JMenuBar menuBar = new JMenuBar();
JMenu fileMenu = new JMenu("File");
JMenuItem openItem = new JMenuItem("Open");
JMenuItem exitItem = new JMenuItem("Exit");

// Add items to the menu
fileMenu.add(openItem);
fileMenu.add(exitItem);

// Add the menu to the menu bar
menuBar.add(fileMenu);

// Set the menu bar to the frame
setJMenuBar(menuBar);
```

Icons in Menu Items:

You can add icons to **JMenuItem** for visual clarity.

Example:

```
JMenuItem saveItem = new JMenuItem("Save", new ImageIcon("save_icon.png"));
```



Check box and Radio Buttons in Menu Items:

You can create checkboxes and radio buttons inside a menu using **JCheckBoxMenuItem** and **JRadioButtonMenuItem**.

Example:

```
// Check Box Menu Item
JCheckBoxMenuItem showStatusBar = new JCheckBoxMenuItem("Show Status Bar",
true);

// Radio Button Menu Item
JRadioButtonMenuItem option1 = new JRadioButtonMenuItem("Option 1");
JRadioButtonMenuItem option2 = new JRadioButtonMenuItem("Option 2");
ButtonGroup group = new ButtonGroup();
group.add(option1);
group.add(option2);
```

Pop-up Menus:

A pop-up menu is a context-sensitive menu that appears when you right-click.

Example:

```
JPopupMenu popupMenu = new JPopupMenu();
JMenuItem cutItem = new JMenuItem("Cut");
JMenuItem copyItem = new JMenuItem("Copy");
JMenuItem pasteItem = new JMenuItem("Paste");

popupMenu.add(cutItem);
popupMenu.add(copyItem);
popupMenu.add(pasteItem);

// Add the popup menu to a component
JTextField textField = new JTextField();
textField.setComponentPopupMenu(popupMenu);
```

Keyboard Mnemonics and Accelerators:

Mnemonics: Mnemonics allow the user to open a menu or select an item using the keyboard (**Alt + Key**).

Example:

```
fileMenu.setMnemonic('F'); // Alt + F to open the File menu
openItem.setMnemonic('O'); // Alt + O to select the Open menu item
```

Accelerators: Accelerators allow you to assign a key combination (like **Ctrl+S**) for quick access to a menu item.

Example:

```
saveItem.setAccelerator(KeyStroke.getKeyStroke('S',Toolkit.getDefaultToolkit()
.getMenuShortcutKeyMaskEx()));
```

Enabling and Disabling menu Items:

You can enable or disable menu items based on certain conditions.

Example:

```
openItem.setEnabled(false); // Disable the Open item
```

Toolbars:

A toolbar provides a set of frequently used actions in the form of buttons.

Example:

```
JToolBar toolBar = new JToolBar();
JButton newButton = new JButton(new ImageIcon("new_icon.png"));
JButton openButton = new JButton(new ImageIcon("open_icon.png"));
toolBar.add(newButton);
toolBar.add(openButton);
// Add the toolbar to the frame
add(toolBar, BorderLayout.NORTH);
```

```
Image scaledImg = img.getScaledInstance(32, 32, Image.SCALE_SMOOTH);
ImageIcon scaledIcon = new ImageIcon(scaledImg);
```

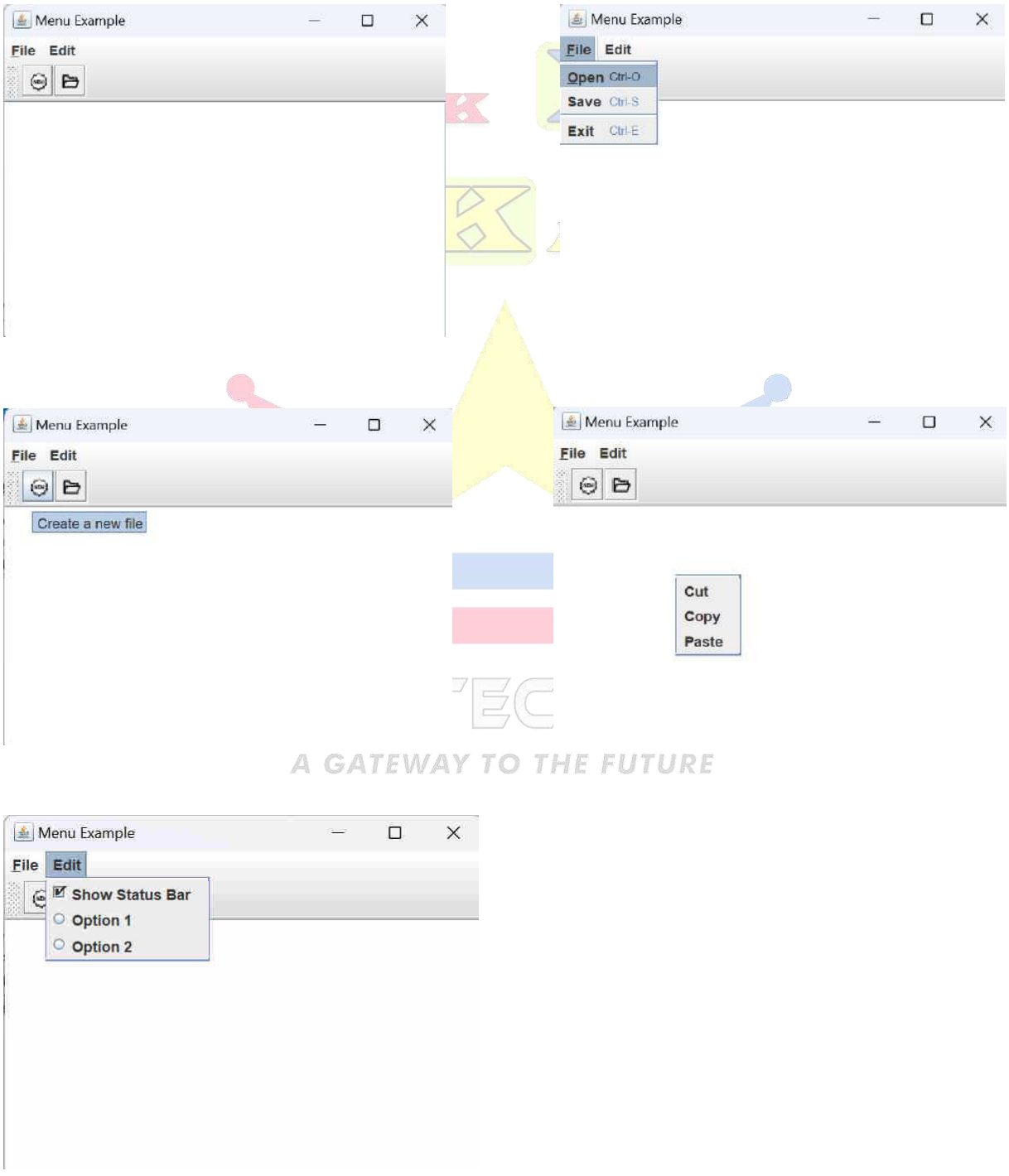
Tooltips:

You can add tooltips to menu items or toolbar buttons to provide hints to the user.

Example:

```
newButton.setToolTipText("Create a new file");
saveItem.setToolTipText("Save the current file");
```

Swing – Menu Example:



```
package SwingComponentsExamples;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MenuExamples extends JFrame {

    public MenuExamples() {
        // Set up frame
        setTitle("Menu Example");
        setSize(400, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);

        // Menu Bar
        JMenuBar menuBar = new JMenuBar();

        // File Menu
        JMenu fileMenu = new JMenu("File");
        fileMenu.setMnemonic('F'); // Alt+F for File menu

        JMenuItem openItem = new JMenuItem("Open", new
ImageIcon("open_icon.png"));
        openItem.setMnemonic('O'); // Alt+O for Open
        openItem.setAccelerator(KeyStroke.getKeyStroke('O',
Toolkit.getDefaultToolkit().getMenuShortcutKeyMaskEx()));

        JMenuItem saveItem = new JMenuItem("Save", new
ImageIcon("save_icon.png"));
        saveItem.setAccelerator(KeyStroke.getKeyStroke('S',
Toolkit.getDefaultToolkit().getMenuShortcutKeyMaskEx()));
```

```
        JMenuItem exitItem = new JMenuItem("Exit");
        exitItem.setAccelerator(KeyStroke.getKeyStroke('E',
Toolkit.getDefaultToolkit().getMenuShortcutKeyMaskEx()));

        // Add items to the file menu
        fileMenu.add(openItem);
        fileMenu.add(saveItem);
        fileMenu.addSeparator(); // Add a separator line
        fileMenu.add(exitItem);

        // Edit Menu with Check Box and Radio Button
        JMenu editMenu = new JMenu("Edit");
        JCheckBoxMenuItem showStatusBar = new JCheckBoxMenuItem("Show Status
Bar", true);
        editMenu.add(showStatusBar);

        // Radio Buttons in a Button Group
        JRadioButtonMenuItem option1 = new JRadioButtonMenuItem("Option 1");
        JRadioButtonMenuItem option2 = new JRadioButtonMenuItem("Option 2");
        ButtonGroup group = new ButtonGroup();
        group.add(option1);
        group.add(option2);
        editMenu.add(option1);
        editMenu.add(option2);

        // Add menus to the menu bar
        menuBar.add(fileMenu);
        menuBar.add(editMenu);

        // Set the menu bar to the frame
        setJMenuBar(menuBar);

        // Tool Bar
        JToolBar toolBar = new JToolBar();
```

```
//        JButton newButton = new JButton(new ImageIcon("H:\\One Drive
KMC\\OneDrive - Kailali Multiple
Campus\\Documents\\NetBeansProjects\\Advanced_Java\\src\\main\\java\\SwingComp
onentsExamples\\images\\new-48.png"));

//        newButton.setToolTipText("Create a new file");

//        JButton openButton = new JButton(new ImageIcon("H:\\One Drive
KMC\\OneDrive - Kailali Multiple
Campus\\Documents\\NetBeansProjects\\Advanced_Java\\src\\main\\java\\SwingComp
onentsExamples\\images\\open-48.png"));

//        openButton.setToolTipText("Open a file");


        ImageIcon new48 = new ImageIcon("H:\\One Drive KMC\\OneDrive - Kailali
Multiple
Campus\\Documents\\NetBeansProjects\\Advanced_Java\\src\\main\\java\\SwingComp
onentsExamples\\images\\new-48.png");
        Image new_img = new48.getImage();


        Image scaledImg_new16 = new_img.getScaledInstance(16, 16,
Image.SCALE_SMOOTH);
        ImageIcon scaled_new16 = new ImageIcon(scaledImg_new16);
        JButton btn_new16 = new JButton(scaled_new16);
        btn_new16.setToolTipText("Create a new file");
        toolBar.add(btn_new16);


        ImageIcon open48 = new ImageIcon("H:\\One Drive KMC\\OneDrive -
Kailali Multiple
Campus\\Documents\\NetBeansProjects\\Advanced_Java\\src\\main\\java\\SwingComp
onentsExamples\\images\\open-48.png");
        Image open_img = open48.getImage();


        Image scaledImg_open16 = open_img.getScaledInstance(16, 16,
Image.SCALE_SMOOTH);
        ImageIcon scaled_open16 = new ImageIcon(scaledImg_open16);
        JButton btn_open16 = new JButton(scaled_open16);
        btn_open16.setToolTipText("Open a file");
        toolBar.add(btn_open16);
```

```
        // Add toolbar to the frame
        add(toolBar, BorderLayout.NORTH);

        // Pop-up Menu
        JPopupMenu popupMenu = new JPopupMenu();
        JMenuItem cutItem = new JMenuItem("Cut");
        JMenuItem copyItem = new JMenuItem("Copy");
        JMenuItem pasteItem = new JMenuItem("Paste");
        popupMenu.add(cutItem);
        popupMenu.add(copyItem);
        popupMenu.add(pasteItem);

        // Text Field with Pop-up Menu
        JTextField textField = new JTextField(20);
        textField.setComponentPopupMenu(popupMenu);
        add(textField, BorderLayout.CENTER);

        // Button to exit
        exitItem.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });
    }

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

Components Used for above Application:

- **Menu Bar:** Contains "File" and "Edit" menus with items like Open, Save, Exit, and status options.
- **Icons:** Menu items have icons (e.g., "open_icon.png").
- **Checkboxes and Radio Buttons:** JCheckBoxMenuItem and JRadioButtonMenuItem are used in the Edit menu.
- **Pop-up Menu:** Right-clicking on the text field shows a pop-up menu (Cut, Copy, Paste).
- **Mnemonics and Accelerators:** Mnemonics allow keyboard access (**Alt + F** for File), and accelerators allow shortcuts (**Ctrl + O** for Open).
- **Tool Bar:** Toolbar buttons for quick actions.
- **Tooltips:** Buttons and menu items have tooltips for better usability.

2.5. Dialog Boxes: Option Dialogs, Creating Dialogs, Data Exchange, File Choosers, Color Choosers.

Dialog Boxes:

Dialog boxes are windows that appear on top of the main application window to prompt the user for input, provide information, or confirm actions. Swing provides several types of dialog boxes, each with its own purpose and appearance:

JOptionPane

- A versatile class for creating various types of dialog boxes.
- Offers methods like *showMessageDialog()*, *showConfirmDialog()*, *showInputDialog()*, and *showOptionDialog()* for different use cases.
- Can be used to display simple messages, ask for user confirmation, obtain input, or present custom options.

JDialog

- A more customizable dialog box that can be created from scratch.
- Provides greater control over layout, components, and behavior.
- Often used for complex dialogs with custom components or logic.

Option Dialogs:

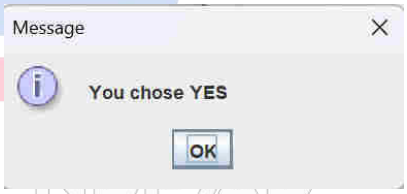
JOptionPane is a simple way to create common dialog boxes like message dialogs, confirm dialogs, and input dialogs.

Example: Message Dialog

```
JOptionPane.showMessageDialog(null, "This is a message dialog.");
```

Example: Confirm Dialog

```
int result = JOptionPane.showConfirmDialog(null, "Do you want to continue?",
"Confirmation", JOptionPane.YES_NO_OPTION);
if (result == JOptionPane.YES_OPTION) {
    System.out.println("User chose YES");
}
```

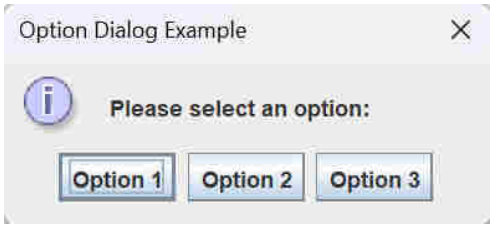


Example: Input Dialog

```
String input = JOptionPane.showInputDialog(null, "Enter your name:");
System.out.println("User input: " + input);
```

Example: Option Dialog

```
package SwingComponentsExamples;

import javax.swing.*;

public class OptionDialogExample {
    public static void main(String[] args) {
        // Define the custom options
        String[] options = {"Option 1", "Option 2", "Option 3"};

        // Display the option dialog
        int choice = JOptionPane.showOptionDialog(null,
                "Please select an option:", // Message to display
                "Option Dialog Example",    // Title of the dialog
                JOptionPane.DEFAULT_OPTION, // Dialog type
                JOptionPane.INFORMATION_MESSAGE, // Message type
                null,                       // Icon (null for no icon)
                options,                    // Options array
                options[0]);                // Default option (pre-selected)

        // Handle the user's choice
        if (choice >= 0 && choice < options.length) {
            JOptionPane.showMessageDialog(null,
                    "You selected: " + options[choice]);
        } else {
            JOptionPane.showMessageDialog(null,
                    "No option selected.");
        }
    }
}
```



Creating Dialogs:

JDialog allows you to create a custom dialog box with more control over its components and behavior.

Example: Custom Dialog

```
JDialog dialog = new JDialog();
dialog.setTitle("Custom Dialog");
dialog.setSize(300, 150);
dialog.setLayout(new FlowLayout());

JLabel label = new JLabel("This is a custom dialog.");
JButton closeButton = new JButton("Close");
closeButton.addActionListener(e -> dialog.dispose());

dialog.add(label);
dialog.add(closeButton);
dialog.setModal(true); // Make the dialog modal (blocks other windows)
dialog.setLocationRelativeTo(null); // Center the dialog
dialog.setVisible(true);
```

Data Exchange:

Dialogs are often used to retrieve input from the user. After getting the input, it can be used in the main application.

Example: Input Dialog to get a value

```
String input = JOptionPane.showInputDialog(null, "Enter your age:");
if (input != null && !input.isEmpty()) {
    int age = Integer.parseInt(input);
    System.out.println("User's age: " + age);
}
```

File Choosers:

JFileChooser allows users to select files or directories from the file system.

Example: Opening a File

```
JFileChooser fileChooser = new JFileChooser();
int result = fileChooser.showOpenDialog(null);
if (result == JFileChooser.APPROVE_OPTION) {
    File selectedFile = fileChooser.getSelectedFile();
    System.out.println("Selected file: " + selectedFile.getAbsolutePath());
}
```

Example: Saving a File

```
JFileChooser fileChooser = new JFileChooser();
int result = fileChooser.showSaveDialog(null);
if (result == JFileChooser.APPROVE_OPTION) {
    File fileToSave = fileChooser.getSelectedFile();
    System.out.println("File to save: " + fileToSave.getAbsolutePath());
}
```

Color Choosers:

JColorChooser allows users to pick a color from a palette.

Example: Choosing a Color

```
Color selectedColor = JColorChooser.showDialog(null, "Pick a Color",
Color.RED);
if (selectedColor != null) {
    System.out.println("Selected color: " + selectedColor.toString());
}
```


```
package SwingComponentsExamples;

import javax.swing.*;
import java.awt.*;
import java.io.File;

public class DialogExample extends JFrame {

    public DialogExample() {
        // Set up frame
        setVisible(true);
        setTitle("Dialog Example");
        setSize(400, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());

        // Button to show message dialog
        JButton messageButton = new JButton("Show Message");
        messageButton.addActionListener(e ->
JOptionPane.showMessageDialog(null, "This is a message dialog"));

        // Button to show confirm dialog
        JButton confirmButton = new JButton("Show Confirm");

        confirmButton.addActionListener(e -> {
            int result = JOptionPane.showConfirmDialog(null, "Do you want to
continue?", "Confirm", JOptionPane.YES_NO_OPTION);
```

```
            if (result == JOptionPane.YES_OPTION) {
                JOptionPane.showMessageDialog(null, "You chose YES");
            } else {
                JOptionPane.showMessageDialog(null, "You chose NO");
            }
        });

        // Button to show input dialog
        JButton inputButton = new JButton("Show Input");
        inputButton.addActionListener(e -> {
            String input = JOptionPane.showInputDialog(null, "Enter your
name:");
            if (input != null && !input.isEmpty()) {
                JOptionPane.showMessageDialog(null, "Hello, " + input);
            }
        });

        // Button to show file chooser
        JButton fileButton = new JButton("Choose File");
        fileButton.addActionListener(e -> {
            JFileChooser fileChooser = new JFileChooser();

            int result = fileChooser.showOpenDialog(null);

            if (result == JFileChooser.APPROVE_OPTION) {
                File selectedFile = fileChooser.getSelectedFile();
                JOptionPane.showMessageDialog(null, "Selected file: " +
selectedFile.getAbsolutePath());
            }
        });
```

```
    // Button to show color chooser
        JButton colorButton = new JButton("Choose Color");

        colorButton.addActionListener(e -> {
            Color color = JColorChooser.showDialog(null, "Pick a Color",
Color.BLUE);
            if (color != null) {
                JOptionPane.showMessageDialog(null, "You chose: " +
color.toString());
            }
        });

        // Add buttons to the frame
        add(messageButton);
        add(confirmButton);
        add(inputButton);
        add(fileButton);
        add(colorButton);
    }

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


Components Used for above Application:

- **Message Dialog:** Shows a simple message to the user.
- **Confirm Dialog:** Asks the user for confirmation (Yes/No).
- **Input Dialog:** Prompts the user for input.
- **File Chooser:** Lets the user select a file from their file system.
- **Color Chooser:** Allows the user to pick a color from a palette.

2.6. Components Organizers: Split Panes, Tabbed Panes, Desktop Panes and Internal Frames, Cascading and Tiling.

Components Organizers:

In Java Swing, components like **Split Panes**, **Tabbed Panes**, **Desktop Panes**, and **Internal Frames** are useful for organizing and structuring GUI applications.

Split Panes:

JSplitPane allows you to divide the area into two resizable sections, either horizontally or vertically. You can add two components into the split pane, and users can adjust the divider to resize them.

Example: Horizontal Split Pane

```
JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT);
splitPane.setLeftComponent(new JButton("Left Component"));
splitPane.setRightComponent(new JButton("Right Component"));
splitPane.setDividerLocation(150); // Initial divider position
```

Example: Vertical Split Pane

```
JSplitPane verticalSplitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT);
verticalSplitPane.setTopComponent(new JLabel("Top Component"));
verticalSplitPane.setBottomComponent(new JLabel("Bottom Component"));
verticalSplitPane.setDividerLocation(100); // Initial divider position
```

Tabbed Panes:

JTabbedPane allows multiple panels (tabs) in the same container, where each tab represents a different panel. Users can switch between them by clicking on the tabs.

Example: Tabbed Pane

```
JTabbedPane tabbedPane = new JTabbedPane();


JPanel panel1 = new JPanel();
panel1.add(new JLabel("This is the first tab"));


JPanel panel2 = new JPanel();
panel2.add(new JLabel("This is the second tab"));


tabbedPane.addTab("Tab 1", panel1);
tabbedPane.addTab("Tab 2", panel2);
```

You can also add icons to the tabs and set tooltips for them using the methods **addTab(String, Icon, Component)** and **setToolTipTextAt(int, String)**.

Desktop Panes and Internal Frames:

- **JDesktopPane** is a container that allows you to create multiple internal frames (child windows) inside the main window.
- **JInternalFrame** represents a frame that can be moved, resized, maximized, minimized, and closed within the desktop pane, similar to windows in a multi-window desktop environment.

Example: Desktop Pane with Internal Frames

```
JDesktopPane desktopPane = new JDesktopPane();


JInternalFrame internalFrame1 = new JInternalFrame("Internal Frame 1", true,
true, true, true);
internalFrame1.setSize(200, 150);
internalFrame1.setVisible(true);
```

```
JInternalFrame internalFrame2 = new JInternalFrame("Internal Frame 2", true,
true, true, true);

internalFrame2.setSize(200, 150);

internalFrame2.setVisible(true);


desktopPane.add(internalFrame1);

desktopPane.add(internalFrame2);
```

Cascading and Tiling:

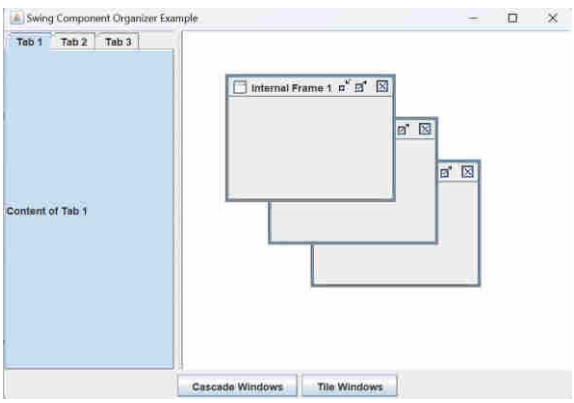
- **Cascading**: Arranging internal frames so that they overlap slightly, similar to cascading windows in an operating system.
- **Tiling**: Arranging internal frames side by side or in a grid so that they don't overlap.

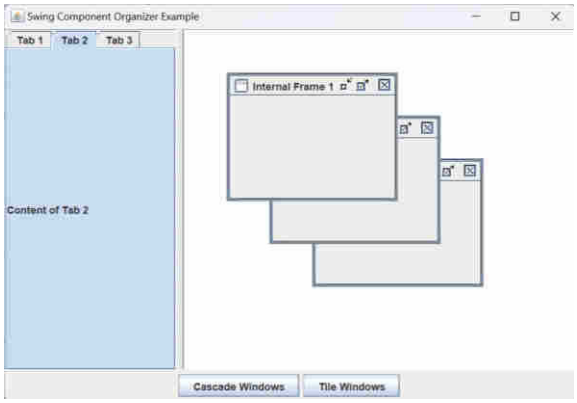
Example: Cascading Internal Frames

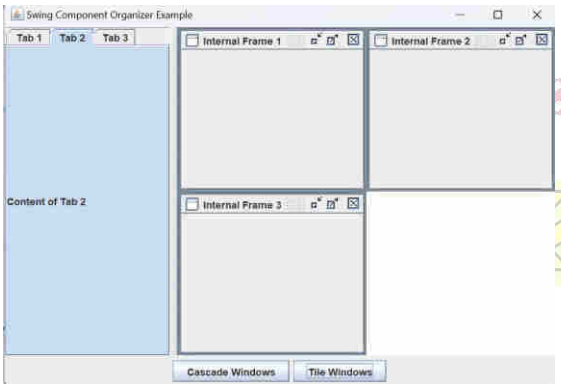
```
int offsetX = 30;

int offsetY = 30;

for (int i = 0; i < desktopPane.getAllFrames().length; i++) {

    JInternalFrame frame = desktopPane.getAllFrames()[i];

    frame.setLocation(i * offsetX, i * offsetY);

}
```

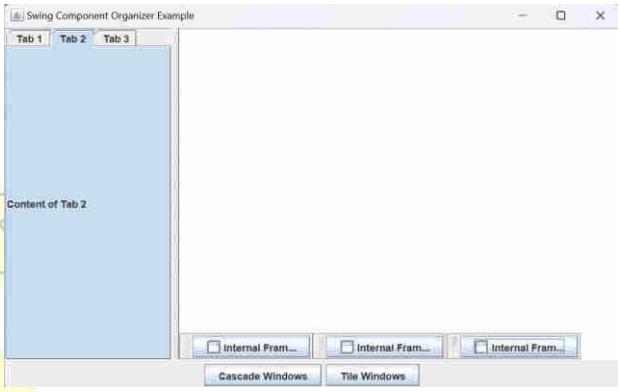
Example: Tiling Internal Frames

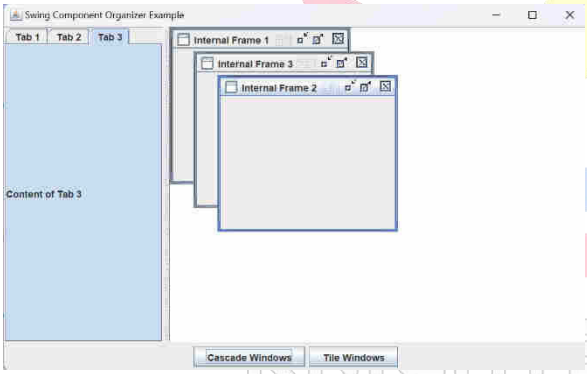
```
JInternalFrame[] frames = desktopPane.getAllFrames();

int columns = (int) Math.ceil(Math.sqrt(frames.length));

int rows = (int) Math.ceil((double) frames.length / columns);

int width = desktopPane.getWidth() / columns;

int height = desktopPane.getHeight() / rows;


for (int i = 0; i < frames.length; i++) {

    frames[i].setBounds((i % columns) * width, (i / columns) * height, width,
height);

}
```

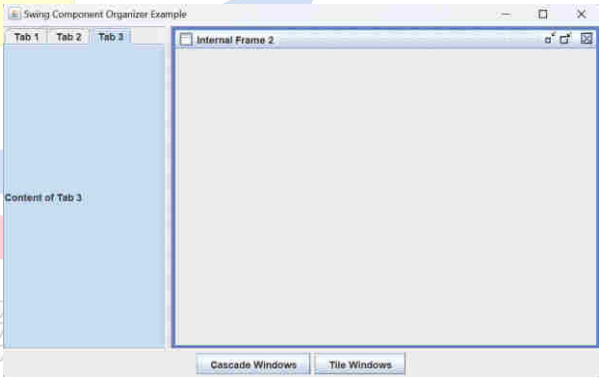












Components Used for above Application:

- **Split Panes (JSplitPane)**: Divides the window into two resizable sections.
- **Tabbed Panes (JTabbedPane)**: Organizes multiple panels in tabs, allowing users to switch between them.
- **Desktop Panes (JDesktopPane)**: Creates a multi-window interface within a single window, where each internal window can behave like a frame (move, resize, close, etc.).
- **Internal Frames (JInternalFrame)**: Frames inside a **JDesktopPane** that can be cascaded or tiled.

- **Cascading and Tiling**: Techniques for arranging internal frames inside a desktop pane.

```
package SwingComponentsExamples;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class OrganizerExample extends JFrame {
    JDesktopPane desktopPane;

    public OrganizerExample() {
        // Set up frame
        setVisible(true);
        setTitle("Swing Component Organizer Example");
        setExtendedState(JFrame.MAXIMIZED_BOTH); //Full Screen
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);

        // Create the main panel with a tabbed pane inside a split pane
        JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT);
        splitPane.setDividerLocation(200);

        // Tabbed Pane on the Left
        JTabbedPane tabbedPane = new JTabbedPane();
        tabbedPane.addTab("Tab 1", new JLabel("Content of Tab 1"));
        tabbedPane.addTab("Tab 2", new JLabel("Content of Tab 2"));
        tabbedPane.addTab("Tab 3", new JLabel("Content of Tab 3"));

        // Split pane: Left side with tabbed pane
        splitPane.setLeftComponent(tabbedPane);

        // Desktop Pane on the right (for internal frames)
```

```
        desktopPane = new JDesktopPane();
        splitPane.setRightComponent(desktopPane);

        // Add split pane to the frame
        add(splitPane, BorderLayout.CENTER);

        // Control Panel for cascading and tiling buttons
        JPanel controlPanel = new JPanel();
        JButton cascadeButton = new JButton("Cascade Windows");
        JButton tileButton = new JButton("Tile Windows");

        // Add buttons to the control panel
        controlPanel.add(cascadeButton);
        controlPanel.add(tileButton);

        // Add control panel to the bottom of the frame
        add(controlPanel, BorderLayout.SOUTH);

        // Add action listeners for the buttons
        cascadeButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                cascadeFrames();
            }
        });

        tileButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                tileFrames();
            }
        });

        // Add some internal frames to the desktop pane
        createInternalFrame("Internal Frame 1", 50, 50);
```

```
        createInternalFrame("Internal Frame 2", 100, 100);
        createInternalFrame("Internal Frame 3", 150, 150);
    }
    // Method to create internal frames
    private void createInternalFrame(String title, int x, int y) {
        JInternalFrame internalFrame = new JInternalFrame(title, true, true,
true, true);
        internalFrame.setSize(200, 150);
        internalFrame.setLocation(x, y);
        internalFrame.setVisible(true);
        desktopPane.add(internalFrame);
    }

    // Method to cascade internal frames
    private void cascadeFrames() {
        JInternalFrame[] frames = desktopPane.getAllFrames();
        int offsetX = 30;
        int offsetY = 30;

        for (int i = 0; i < frames.length; i++) {
            frames[i].setLocation(i * offsetX, i * offsetY);
            frames[i].moveToFront();
        }
    }

    // Method to tile internal frames
    private void tileFrames() {
        JInternalFrame[] frames = desktopPane.getAllFrames();
        int numFrames = frames.length;

        if (numFrames == 0) {
            return;
        }
```

```
        int rows = (int) Math.ceil(Math.sqrt(numFrames));
        int cols = (int) Math.ceil((double) numFrames / rows);

        int frameWidth = desktopPane.getWidth() / cols;
        int frameHeight = desktopPane.getHeight() / rows;

        for (int i = 0; i < numFrames; i++) {
            frames[i].setBounds((i % cols) * frameWidth, (i / cols) *
frameHeight, frameWidth, frameHeight);
        }
    }

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


Summary of above Application

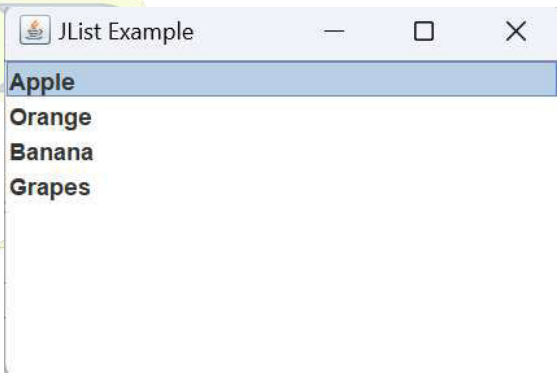
1. **JSplitPane**:
   - A JSplitPane is used to divide the window into two parts:
     - **Left side**: Contains a JTabbedPane with three tabs.
     - **Right side**: Contains a JDesktopPane, which allows adding internal frames.
2. **JTabbedPane**:
   - Contains three tabs, each displaying different content. You can switch between tabs by clicking on them.
3. **JDesktopPane** and **JInternalFrame**:
   - A JDesktopPane is used to manage multiple JInternalFrame objects.
   - Each internal frame acts like a window inside the main application. The user can move, resize, minimize, or close each internal frame.
4. **Cascading and Tiling Internal Frames**:
   - **Cascading**: The internal frames are arranged diagonally with an offset so they overlap slightly.
   - **Tiling**: The internal frames are arranged in a grid without overlap.

5. **Buttons**:
   - **Cascade Windows**: When clicked, the internal frames are arranged in a cascading style.
   - **Tile Windows**: When clicked, the internal frames are arranged in a tiled format.

2.7. Advance Swing Components: List, Trees, Tables, Progress Bars.

List:

JList is used to display a list of items that users can select from. It supports single and multiple selection modes.



Example:

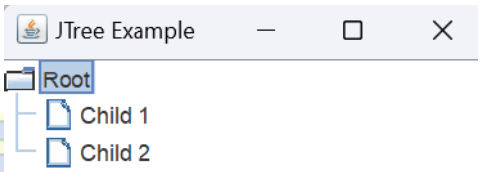
```
import javax.swing.*;
import java.awt.*;

public class ListExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JList Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 200);

        // List data
        String[] fruits = {"Apple", "Orange", "Banana", "Grapes"};

        // Create the list
        JList<String> list = new JList<>(fruits);
        list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION); // Single selection

        // Add the list to a scroll pane
        JScrollPane scrollPane = new JScrollPane(list);

        frame.add(scrollPane, BorderLayout.CENTER);
        frame.setVisible(true);
```

```
    }
}
```

Trees:

JTree is used to represent hierarchical data, such as file systems or organizational structures.

Example:



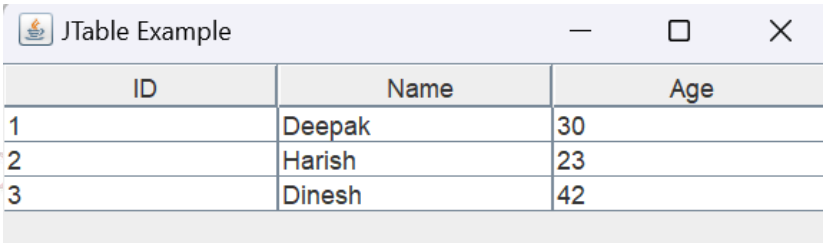

```
import javax.swing.*;
import javax.swing.tree.DefaultMutableTreeNode;
import java.awt.*;

public class TreeExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JTree Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 300);

        // Create the root node
        DefaultMutableTreeNode root = new DefaultMutableTreeNode("Root");

        // Create child nodes
        DefaultMutableTreeNode child1 = new DefaultMutableTreeNode("Child 1");
        DefaultMutableTreeNode child2 = new DefaultMutableTreeNode("Child 2");

        // Add child nodes to root
        root.add(child1);
        root.add(child2);

        // Create the tree by passing in the root node
        JTree tree = new JTree(root);

        // Add the tree to a scroll pane
        JScrollPane scrollPane = new JScrollPane(tree);
```

```
        frame.add(scrollPane, BorderLayout.CENTER);
        frame.setVisible(true);
    }
}
```

Tables:

JTable is used for displaying and editing tabular data in a grid of rows and columns.

Example:



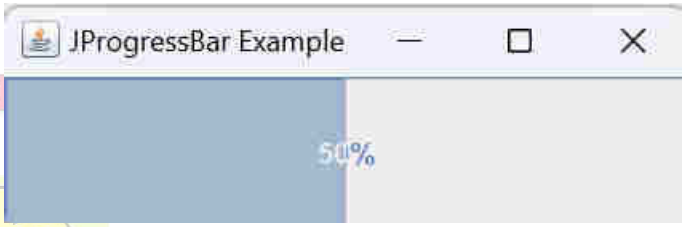

```
import javax.swing.*;
import java.awt.*;

public class TableExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JTable Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);

        // Table column names
        String[] columnNames = {"ID", "Name", "Age"};

        // Table data
        Object[][] data = {
            {1, "Deepak", 30},
            {2, "Harish", 23},
            {3, "Dinesh", 42}
        };

        // Create the table
        JTable table = new JTable(data, columnNames);

        // Add the table to a scroll pane
        JScrollPane scrollPane = new JScrollPane(table);
```

```
        frame.add(scrollPane, BorderLayout.CENTER);
        frame.setVisible(true);
    }
}
```

Progress Bars:

JProgressBar is used to show the progress of a long-running task.

Example:

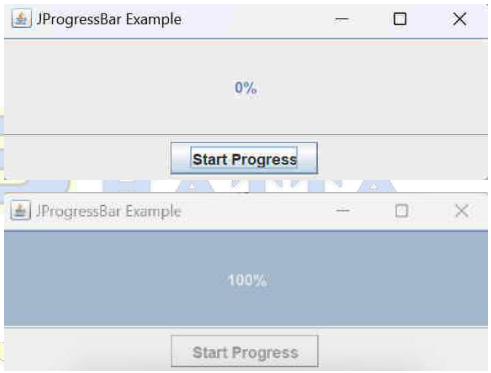



```
package AdvanceSwingComponents;

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

public class ProgressBarExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JProgressBar Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 100);

        // Create a progress bar
        JProgressBar progressBar = new JProgressBar(0, 100);
        progressBar.setValue(50);  // Set current progress (e.g., 50%)
        progressBar.setStringPainted(true);  // Show percentage

        frame.add(progressBar, BorderLayout.CENTER);
        frame.setVisible(true);
    }
}
```

- **JList**: Displays a scrollable list of items, allowing users to select one or more options.
- **JTree**: Represents hierarchical data with expandable and collapsible nodes.

- **JTable**: Displays tabular data in rows and columns, supporting sorting, editing, and custom rendering.
- **JProgressBar**: Visualizes the progress of tasks like file downloads or computations.

Extra on Progress Bar:

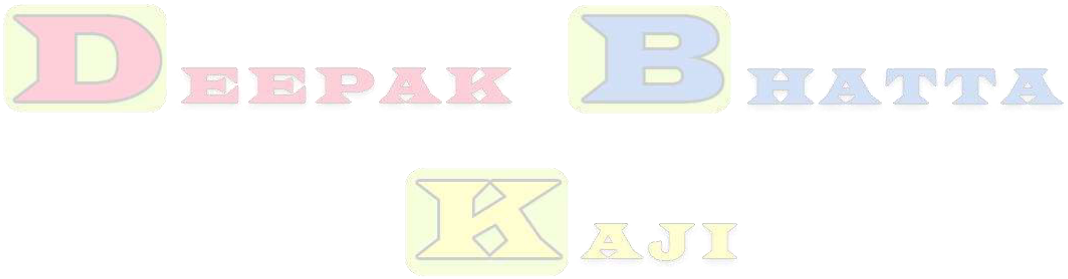



```
package AdvanceSwingComponents;

import javax.swing.*;
import java.awt.*;
import java.util.concurrent.ExecutionException;

public class RunningProgressBarExample {

    public static void main(String[] args) {
        // Create the frame
        JFrame frame = new JFrame("JProgressBar Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 150);
        frame.setLayout(new BorderLayout());

        // Create a progress bar
        JProgressBar progressBar = new JProgressBar(0, 100);
        progressBar.setStringPainted(true);  // Show the percentage

        // Add progress bar to the frame
        frame.add(progressBar, BorderLayout.CENTER);

        // Create a panel for the button
        JPanel panel = new JPanel();
        JButton startButton = new JButton("Start Progress");
        panel.add(startButton);

        // Add panel to the frame
```

```
        frame.add(panel, BorderLayout.SOUTH);

        // Button click starts the progress bar
        startButton.addActionListener(e -> {
            // Disable button while task runs
            startButton.setEnabled(false);

            // Create a SwingWorker to handle the progress in the background
            SwingWorker<Void, Void> worker = new SwingWorker<Void, Void>() {
                @Override
                protected Void doInBackground() throws InterruptedException {
                    // Simulate task progress from 0 to 100%
                    for (int i = 0; i <= 100; i++) {
                        Thread.sleep(50);  // Simulate work with delay
                        progressBar.setValue(i);  // Update progress bar
                    }
                    return null;
                }
                @Override
                protected void done() {
                    // When done, show a success message and re-enable the button
                    try {
                        get();
                        JOptionPane.showMessageDialog(frame, "Task Completed
Successfully!", "Success", JOptionPane.INFORMATION_MESSAGE);
                    } catch (InterruptedException | ExecutionException ex) {
                        ex.printStackTrace();
                    }
                    startButton.setEnabled(true);  // Re-enable button
                }
            };
            // Start the background task
            worker.execute();
        });
```

```
        // Display the frame
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }
}
```

THE END



Unit 3 — 6 hrs.

Event Handling

Specific Objectives

- Understand event handling models.
- Demonstrate the use of listeners and adapters.
- Write programs to handle different types of events.

3.1. Introduction: Standard Event Handling, Using Delegated Class, Using Action Commands, Listener Interfaces, Adapter Classes.

3.2. Handling Events: Action Events, Key Events, Focus Events, Window Event, Mouse Event, Item Event.

3.1. Introduction: Standard Event Handling, Using Delegated Class, Using Action Commands, Listener Interfaces, Adapter Classes.

Introduction Event:

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

Standard Event Handling:

Event handling in Java is a key concept for making applications interactive. It refers to the mechanism by which events generated by user actions (such as clicking a button or moving the mouse) are handled in a Java application.

Java provides a powerful event-handling model with support for:

1. **Event Sources**: Objects that generate events.
2. **Event Listeners**: Objects that "listen" for events and define responses.

Key components:

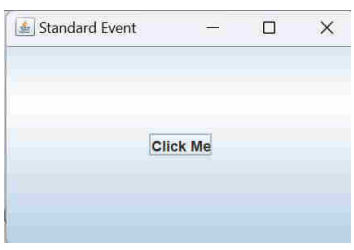
- **Event Object**: Encapsulates the information about the event.
- **Event Source**: The component (e.g., button, text field) where the event is triggered.
- **Event Listener**: Defines the code to execute when the event occurs.

In standard event handling:

- **Create an event listener:** Implement an appropriate interface (e.g., *ActionListener, MouseListener)* for the desired event type.
- **Register the event listener:** Attach the listener to the event source (e.g., a *button*) using a method like *addActionListener()*.
- **Implement the event handler:** Write the method that will be executed when the event occurs.

Example:

```
package EventHandling;

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

public class StandardEventHandling extends JFrame implements ActionListener {

    JButton button;

    public StandardEventHandling() {
        button = new JButton("Click Me");
        button.addActionListener(this); // Registering listener to the button
        add(button);

        setSize(300, 200);
        setTitle("Standard Event");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked!");
    }

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



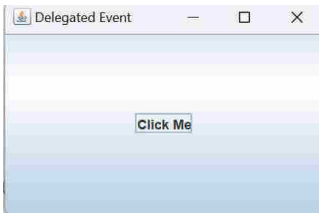

```
--- exec:3.1.0:exec (default-cli) @ Advanced_Java ---
Button clicked!
```

Using Delegated Class:

Delegate event handling is used to a separate a class for better organization and reusability. Instead of implementing the listener interface in the main class, you can create a separate class dedicated to handling the event.

Example:

```
package EventHandling;

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

class ButtonHandler implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked from delegated class!");
    }
}
public class DelegatedEventExample extends JFrame {
    JButton button;

    public DelegatedEventExample() {
        button = new JButton("Click Me");
        button.addActionListener(new ButtonHandler()); // Attaching listener
from another class
        add(button);

        setSize(300, 200);
        setTitle("Delegated Event");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }
    public static void main(String[] args) {
        DelegatedEventExample delegatedEventExample = new
DelegatedEventExample();
    }
}
```




```
--- exec:3.1.0:exec (default-cli) @ Advanced_Java ---
Button clicked from delegated class!
```

Using Action Commands:

Action commands allow you to differentiate between multiple sources of events. For example, you can set different action commands on buttons and handle them accordingly in the listener.

Example:

```
package EventHandling;

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

public class ActionCommandExample extends JFrame implements ActionListener {

    JButton button1, button2;

    public ActionCommandExample() {
        button1 = new JButton("Button 1");
        button2 = new JButton("Button 2");

        button1.setActionCommand("B1"); // Setting action command
        button2.setActionCommand("B2");

        button1.addActionListener(this);
        button2.addActionListener(this);

        add(button1, BorderLayout.NORTH);
        add(button2, BorderLayout.SOUTH);

        setSize(300, 200);
        setTitle("Action Command Event");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        String command = e.getActionCommand(); // Retrieving the action command
```

```
        if (command.equals("B1")) {
            System.out.println("Button 1 clicked!");
        } else if (command.equals("B2")) {
            System.out.println("Button 2 clicked!");
        }
    }

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


```
--- exec:3.1.0:exec (default-cli) @ Advanced_Java ---
Button 1 clicked!
Button 2 clicked!
```

Listener Interfaces:

Java provides different listener interfaces to handle various types of events, such as:

- **ActionListener**: For handling action events (e.g., button clicks).
- **MouseListener**: For handling mouse events (e.g., clicks, entering, exiting).
- **KeyListener**: For handling keyboard events.

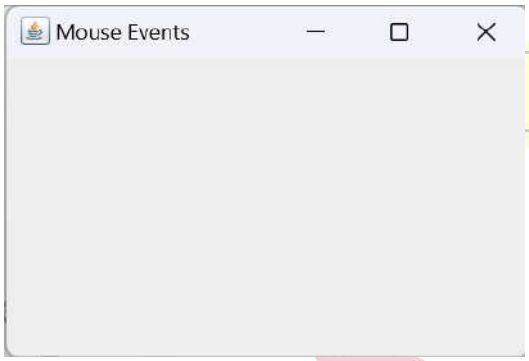
Each listener interface contains a set of methods corresponding to different event types. For example, MouseListener has the following methods:

- mouseClicked()
- mouseEntered()
- mouseExited()
- mousePressed()
- mouseReleased()

Example for handling mouse events:

```
package EventHandling;

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

public class MouseListenerExample extends JFrame implements MouseListener {

    public MouseListenerExample() {
        addMouseListener(this);
        setSize(300, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    @Override
    public void mouseClicked(MouseEvent e) {
        System.out.println("Mouse Clicked!");
    }

    @Override
    public void mouseEntered(MouseEvent e) {
        System.out.println("Mouse Entered!");
    }

    @Override
    public void mouseExited(MouseEvent e) {
        System.out.println("Mouse Exited!");
    }

    @Override
    public void mousePressed(MouseEvent e) {
        System.out.println("Mouse Pressed!");
    }
```

```
    @Override
    public void mouseReleased(MouseEvent e) {
        System.out.println("Mouse Released!");
    }

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

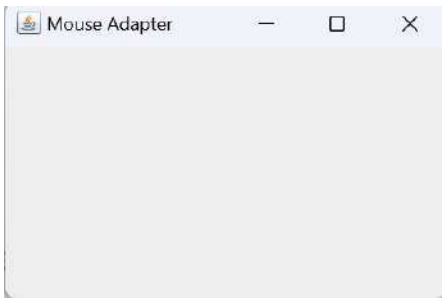



```
Mouse Entered!
Mouse Pressed!
Mouse Released!
Mouse Pressed!
Mouse Released!
Mouse Clicked!
Mouse Exited!
```

Adapter Classes:

Adapter classes provide empty implementations of listener interfaces so that you can override only the methods you're interested in, without having to implement all the methods. This is useful when a listener interface has many methods, and you only want to use a few.

For example, using MouseAdapter:




```
--- exec:3.1.0:exec (default-cli) @ Advanced_Java ---
Mouse Clicked using adapter!
```

```
package EventHandling;

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

public class MouseAdapterExample extends JFrame {

    public MouseAdapterExample() {
        addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                System.out.println("Mouse Clicked using adapter!");
            }
        });

        setSize(300, 200);
        setTitle("Mouse Adapter");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

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


- For **simple applications** or learning purposes, **Standard Event Handling** is easy and intuitive.
- For **modular applications** where you need to separate concerns, using **Delegated Class** is better as it improves reusability and clarity.
- If you're handling multiple events from various sources (like many buttons), **Action Commands** provide a neat solution to avoid creating multiple listeners.
- When dealing with complex event types (like mouse or keyboard events), **Listener Interfaces** provide precise control, though **Adapter Classes** are useful to avoid unnecessary method implementations.

3.2. Handling Events: Action Events, Key Events, Focus Events, Window Event, Mouse Event, Item Event.

Handling Events:

In Java, events are actions or occurrences detected by the program, often generated by user interactions with the graphical user interface (GUI). Different types of events are triggered by different components, and Java provides specific event-handling mechanisms for each.

Action Events:

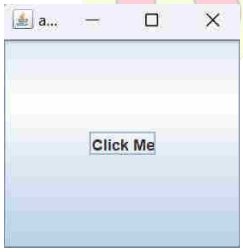
Action events are typically triggered when a user interacts with components like buttons, menu items, or text fields (by pressing Enter). The **ActionEvent** class encapsulates these events, and the **ActionListener** interface is used to handle them.

- **Event Source**: Button, MenuItem, TextField.
- **Listener Interface**: ActionListener.
- **Method to Implement**: actionPerformed(ActionEvent e).

Example:

```
package EventHandling.HandlingEvents;

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

public class ActionEventExample extends JFrame implements ActionListener {
    JButton button;

    public ActionEventExample() {
        button = new JButton("Click Me");
        button.addActionListener(this); // Register ActionListener
        add(button);
        setSize(200, 200);
        setTitle("action Event");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }
```

```
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked!");
    }

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

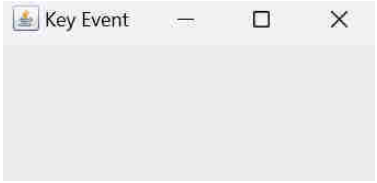



```
--- exec:3.1.0:exec (default-cli) @ Advanced_Java ---
Button clicked!
```

Key Events:

Key events are generated when the user interacts with the keyboard. These events allow you to capture when keys are pressed, released, or typed. The **KeyEvent** class encapsulates these events, and the **KeyListener** interface is used to handle them.

- **Event Source**: Keyboard input.
- **Listener Interface**: KeyListener.
- **Methods to Implement**:
  - ✓ keyPressed(KeyEvent e)
  - ✓ keyReleased(KeyEvent e)
  - ✓ keyTyped(KeyEvent e)




```
--- exec:3.1.0:exec (default-cli) @ Advanced_Java ---
Key Pressed: k
Key Released: k
Key Pressed: a
Key Released: a
Key Pressed: j
Key Released: j
Key Pressed: i
Key Released: i
```

Example:

```
package EventHandling.HandlingEvents;

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

public class KeyEventExample extends JFrame implements KeyListener {

    public KeyEventExample() {
        addKeyListener(this);  // Register KeyListener
        setTitle("Key Event");
        setSize(200, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    @Override
    public void keyPressed(KeyEvent e) {
        System.out.println("Key Pressed: " + e.getKeyChar());
    }

    @Override
    public void keyReleased(KeyEvent e) {
        System.out.println("Key Released: " + e.getKeyChar());
    }

    @Override
    public void keyTyped(KeyEvent e) {
    }

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

Focus Events:

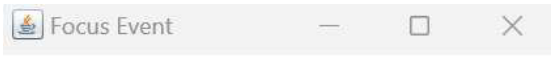
Focus events occur when a component gains or loses keyboard focus. These events are encapsulated in the **FocusEvent** class, and the **FocusListener** interface is used to handle them. They are often used for validating user input or highlighting the focused component.

- **Event Source**: Components like text fields, buttons, etc.
- **Listener Interface**: FocusListener.
- **Methods to Implement**:
  - ✓ focusGained(FocusEvent e)
  - ✓ focusLost(FocusEvent e)

Example:


```
package EventHandling.HandlingEvents;

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

public class FocusEventExample extends JFrame implements FocusListener {

    JTextField textField;

    public FocusEventExample() {
        textField = new JTextField("Click to focus", 20);
        textField.addFocusListener(this);  // Register FocusListener
        add(textField);
        setSize(300, 100);
        setTitle("Focus Event");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }
```

```
    @Override
    public void focusGained(FocusEvent e) {
        System.out.println("Focus Gained");
    }

    @Override
    public void focusLost(FocusEvent e) {
        System.out.println("Focus Lost");
    }

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



Click to focus

```
--- exec:3.1.0:exec (default-cli) @ Advanced_Java ---
Focus Gained
Focus Lost
Focus Gained
Focus Lost
Focus Gained
Focus Lost
Focus Gained
```

Window Event:

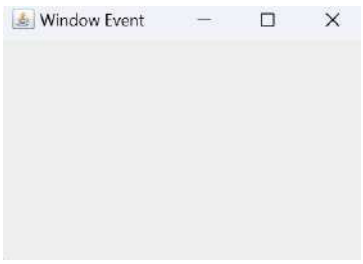
Window events are triggered when a window is opened, closed, activated, deactivated, or resized. These events are encapsulated in the **WindowEvent** class, and the **WindowListener** interface handles them.

- **Event Source**: JFrame, JDialog, etc.
- **Listener Interface**: WindowListener.
- **Methods to Implement**:
  - ✓ windowOpened(WindowEvent e)
  - ✓ windowClosing(WindowEvent e)
  - ✓ windowClosed(WindowEvent e)
  - ✓ windowIconified(WindowEvent e)
  - ✓ windowDeiconified(WindowEvent e)
  - ✓ windowActivated(WindowEvent e)
  - ✓ windowDeactivated(WindowEvent e)

Example:

```
package EventHandling.HandlingEvents;
import javax.swing.*;
import java.awt.event.*;

public class WindowEventExample extends JFrame implements WindowListener {

    public WindowEventExample() {
        addWindowListener(this);  // Register WindowListener
        setSize(200, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    @Override
    public void windowOpened(WindowEvent e) {
        System.out.println("Window Opened");
    }
```

```
    @Override
    public void windowClosing(WindowEvent e) {
        System.out.println("Window Closing");
    }

    @Override
    public void windowClosed(WindowEvent e) {
    }

    @Override
    public void windowIconified(WindowEvent e) {
    }

    @Override
    public void windowDeiconified(WindowEvent e) {
    }

    @Override
    public void windowActivated(WindowEvent e) {
    }

    @Override
    public void windowDeactivated(WindowEvent e) {
    }

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




```
--- exec:3.1.0:exec (default-cli) @ Advanced_Java ---
Window Opened
Window Closing
```

Mouse Event:

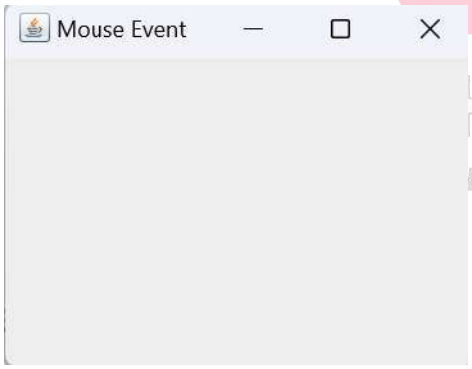
Mouse events are generated when the user interacts with the mouse, such as clicking, pressing, or moving the mouse. The **MouseEvent** class encapsulates these events, and both the **MouseListener** and **MouseMotionListener** interfaces can be used to handle them.

- **Event Source**: Components that detect mouse actions, such as panels, buttons, etc.
- **Listener Interfaces**:
  - ✓ MouseListener: Detects mouse clicks, presses, releases, enters, exits.
  - ✓ MouseMotionListener: Detects mouse movements and drags.
- **Methods to Implement**:
  - ✓ mouseClicked(MouseEvent e)
  - ✓ mousePressed(MouseEvent e)
  - ✓ mouseReleased(MouseEvent e)
  - ✓ mouseEntered(MouseEvent e)
  - ✓ mouseExited(MouseEvent e)

Example:

```
package EventHandling.HandlingEvents;

import javax.swing.*;
import java.awt.event.*;
public class MouseEventExample extends JFrame implements MouseListener {

    public MouseEventExample() {
        addMouseListener(this);  // Register MouseListener
        setTitle("Mouse Event");
        setSize(200, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    @Override
    public void mouseClicked(MouseEvent e) {
        System.out.println("Mouse Clicked at (" + e.getX() + ", " + e.getY() +
")");
    }
```

```
    @Override
    public void mousePressed(MouseEvent e) {
    }

    @Override
    public void mouseReleased(MouseEvent e) {
    }

    @Override
    public void mouseEntered(MouseEvent e) {
    }

    @Override
    public void mouseExited(MouseEvent e) {
    }

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



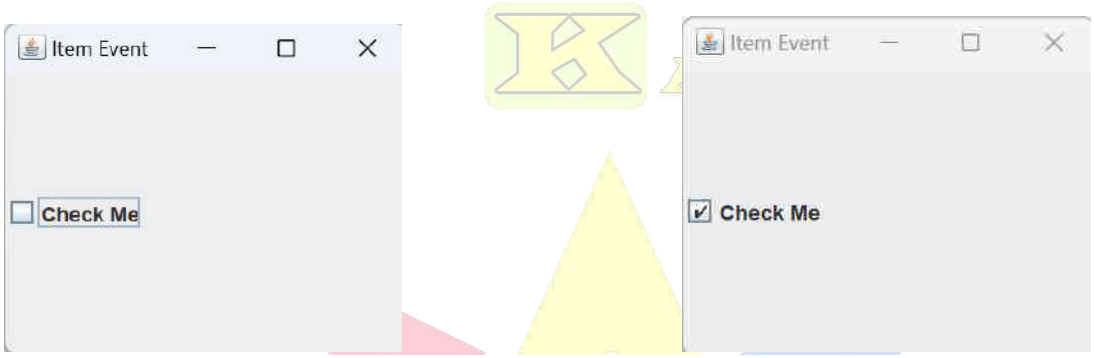

```
--- exec:3.1.0:exec (default-cli) @ Advanced_Java ---
Mouse Clicked at (143, 87)
```

Item Event:

Item events are triggered when the state of an item changes in components like checkboxes, radio buttons, or combo boxes. The **ItemEvent** class encapsulates these events, and the **ItemListener** interface handles them.

- **Event Source**: Components with selectable items, like checkboxes, radio buttons, combo boxes.
- **Listener Interface**: ItemListener.
- **Method to Implement**: itemStateChanged(ItemEvent e).

Example:




```
--- exec:3.1.0:exec (default-cli) @ Advanced_Java ---
Checkbox selected

--- exec:3.1.0:exec (default-cli) @ Advanced_Java ---
Checkbox selected
Checkbox deselected
```



```
package EventHandling.HandlingEvents;

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

public class ItemEventExample extends JFrame implements ItemListener {

    JCheckBox checkBox;
```

```
public ItemEventExample() {
        checkBox = new JCheckBox("Check Me");
        checkBox.addItemListener(this);  // Register ItemListener
        add(checkBox);
        setTitle("Item Event");
        setSize(200, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    @Override
    public void itemStateChanged(ItemEvent e) {
        if (e.getStateChange() == ItemEvent.SELECTED) {
            System.out.println("Checkbox selected");
        } else {
            System.out.println("Checkbox deselected");
        }
    }

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


THE END

Unit 4 — 6 hrs.

Java Database Connectivity

Specific Objectives

- Understand JDBC architecture and driver types.
- Explain different steps used in connecting with databases.
- Demonstrate used of different types of statements.
- Create programs to executes.
- DDL and DML statement.

4.1. Design of JDBC: JDBC Architectures, Drivers & Jar Files, Driver Types, Steps for Connecting to JDBC.

4.2. Executing SQL Statements: Managing Connections, Statements, Result Set, SQL Exceptions, Populating Database.

4.3. Query Execution: Prepared Statements, Reading and Writing LOBs, SQL Escapes, Multiple Results, Scrollable Result Sets, Updateable Result Sets, Row Sets and Cached Row Sets, Transactions.

4.1. Design of JDBC: JDBC Architectures, Drivers & Jar Files, Driver Types, Steps for Connecting to JDBC.

Introduction JDBC:

Java Database Connectivity (JDBC) is an application programming interface (API) for the Java programming language which defines how a client may access a database. It is a Java-based data access technology used for Java database connectivity.

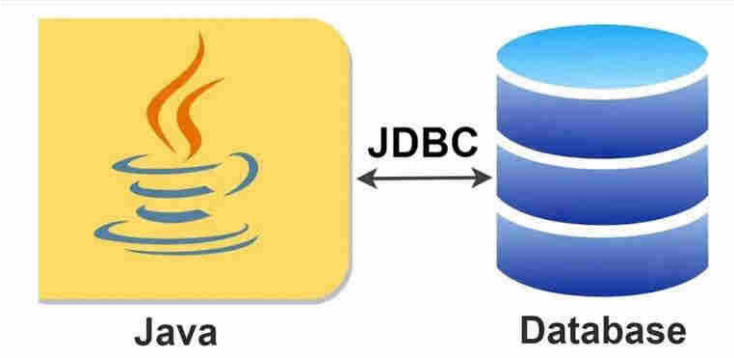
It is part of the Java Standard Edition platform, from Oracle Corporation. It provides methods to query and update data in a database, and is oriented toward relational databases. A JDBC-to-ODBC bridge enables connections to any ODBC-accessible data source in the Java virtual machine (JVM) host environment.

Sun Microsystems released JDBC as part of Java Development Kit (JDK) 1.1 on **February 19, 1997**. Since then, it has been part of the Java Platform, Standard Edition (Java SE).

The JDBC classes are contained in the Java package **java.sql** and **javax.sql**.

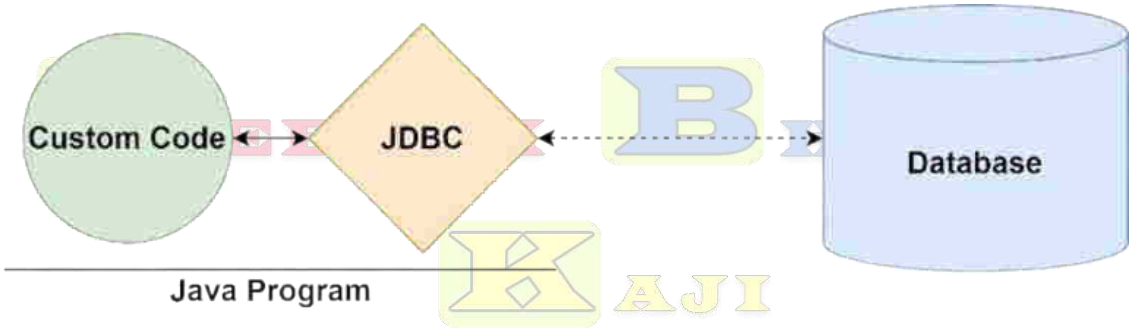
JDBC versions

JDBC version	Java version	Release Type	Release date
1.1	JDK 1.1	Main	1997-02-19.[1]
3.0	J2SE 1.4	Main	2002-05-09
4.0	Java SE 6	Main	2006-12-11
4.1	Java SE 7	Maintenance	2011-10-13
4.2	Java SE 8	Maintenance	2014-03-04
4.3	Java SE 9	Maintenance	2017-09-21

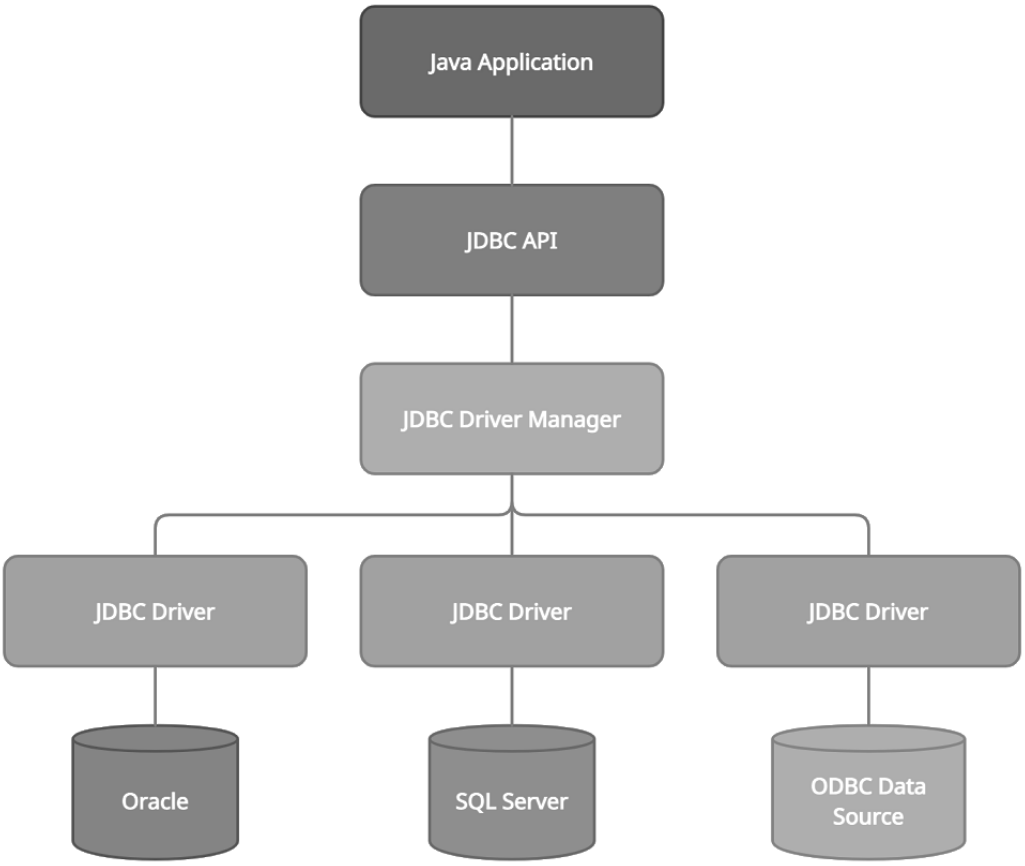


Design of JDBC:

Design of JDBC defines the components of JDBC, which is used for connecting to the database. The **design of JDBC (Java Database Connectivity)** focuses on creating a standard API for connecting Java applications to databases. JDBC simplifies interaction with databases by providing methods for executing SQL queries, retrieving results, and managing database connections. JDBC follows a layered architecture to separate database connectivity logic from the application logic.



Components of JDBC



JDBC has four major components that are used for the interaction with the database.

1. JDBC API
2. JDBC Test Suite
3. JDBC Driver Manger
4. JDBC ODBC Bridge Driver

Note: *Since Java 8, the JDBC-ODBC drivers have been removed. Oracle suggests using drivers provided by the vendor of the database.*

1. JDBC API:

The **JDBC API (Application Programming Interface)** is the core of JDBC, providing standard interfaces and classes to interact with databases. It enables Java applications to:

- Connect to databases.
- Send SQL queries.
- Retrieve query results.
- Perform transaction management.
- Handle errors using SQL exceptions.

Key classes and interfaces within the JDBC API:

- **DriverManager:** Manages database drivers and establishes connections.
- **Connection:** Represents a session with a database.
- **Statement/PreparedStatement:** Used to execute SQL queries.
- **ResultSet:** Stores the result of SQL queries.
- **SQLException:** Handles errors during database interaction.

The API abstracts database-specific details, making it easier to work with different databases using the same interface.

2. JDBC Test Suite:

The **JDBC Test Suite** is used for testing the compatibility of a JDBC driver with the JDBC API. It ensures that:

- JDBC drivers correctly implement the standard JDBC interfaces and classes.
- Applications using JDBC drivers can interact consistently with the database, regardless of which driver is in use.

It helps database vendors and developers verify that their JDBC driver implementations meet the specifications set by the Java community.

3. JDBC Driver Manager:

The **JDBC DriverManager** is responsible for managing multiple JDBC drivers. It serves as the communication channel between the Java application and the database drivers. Key roles include:

- **Loading drivers:** It dynamically loads the correct JDBC driver when the application tries to connect to a database.
- **Connection management:** It provides the connection to the database by selecting the appropriate driver based on the connection URL.

When the **DriverManager.getConnection()** method is called, the **DriverManager** checks the list of registered drivers and attempts to connect to the database.

4. JDBC-ODBC Bridge Driver:

The **JDBC-ODBC Bridge Driver** is a Type 1 JDBC driver that allows JDBC applications to use ODBC drivers to interact with databases. ***ODBC (Open Database Connectivity) is an older standard*** used to connect to various databases.

- The JDBC-ODBC bridge translates JDBC API calls into ODBC calls and communicates with the database via the ODBC driver.
- **Pros:** Allows JDBC to connect to databases that only have ODBC drivers.
- **Cons:** Slower performance due to the additional ODBC layer. It also requires an ODBC driver installation on the client machine.

JDBC Architectures:

JDBC is a Java-based API used to connect and interact with relational databases. It provides a standard interface for connecting Java applications to databases.

JDBC architecture consists of two layers:

1. **Application Layer:** Java application that interacts with the database.
2. **JDBC API Layer:** Acts as a bridge between the application and the database. It contains classes and methods to send SQL queries, retrieve data, and manage database connections.

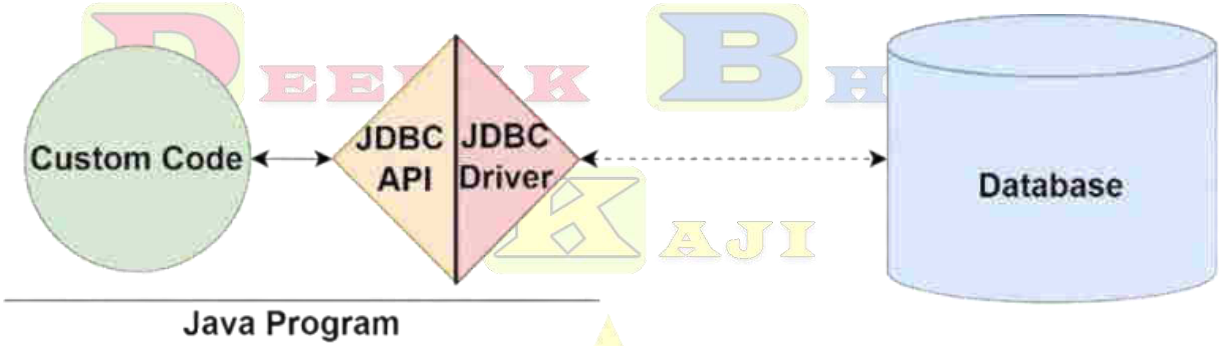


Fig: JDBC's architecture consists of the JDBC API and JDBC drivers.

JDBC API Layer Components:

A few of the crucial interfaces and classes defined in the JDBC API are the following:

- **Driver Manager:** Manages a list of database drivers, and selects the correct driver to connect to a specific database.
- **JDBC Driver:** Provides the implementation of the JDBC interface for a specific database.
- **Connection:** Represents a session with a database.
- **Statement/PreparedStatement:** Used to execute SQL queries.
- **ResultSet:** Holds the result of a query.
- **SQLException:** Used to handle any errors that occur during database interaction.

There are two architectures of JDBC:

1. Two-Tier Architecture (Client-Server Architecture):

In the **two-tier architecture**, the Java application directly interacts with the database. It is also called **client-server architecture** because the application (client) communicates directly with the database server without any intermediary layer.

Key Features of Two-Tier Architecture:

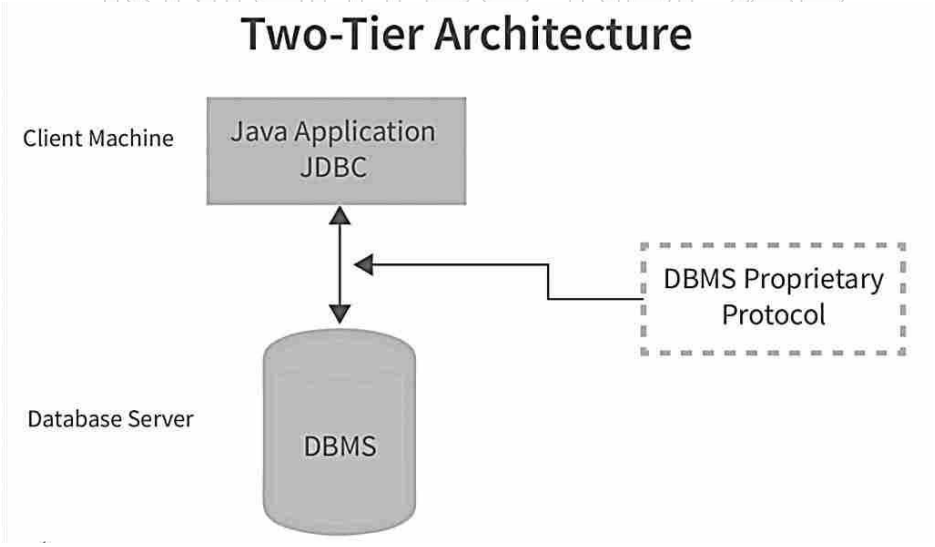
- **Client-Server Model:** The client (Java application) establishes a connection with the database server using JDBC.
- **Direct Communication:** The application sends SQL queries to the database server and retrieves the results without involving any middleware.
- **Simpler Design:** This architecture is simple and suitable for small to medium-sized applications where direct database access is sufficient.

How It Works:

- The Java application uses JDBC to connect to the database.
- Queries are executed directly on the database.
- The database returns the results to the client.

Example:

A desktop application (*like a payroll system*) directly connects to a MySQL database to fetch employee data.



2. Three-Tier Architecture (Client-Application Server-Database Architecture):

In the **three-tier architecture**, an additional middle layer, called an **application server** or **middleware**, is introduced between the client and the database. This architecture is more suitable for large-scale, distributed applications.

Key Features of Three-Tier Architecture:

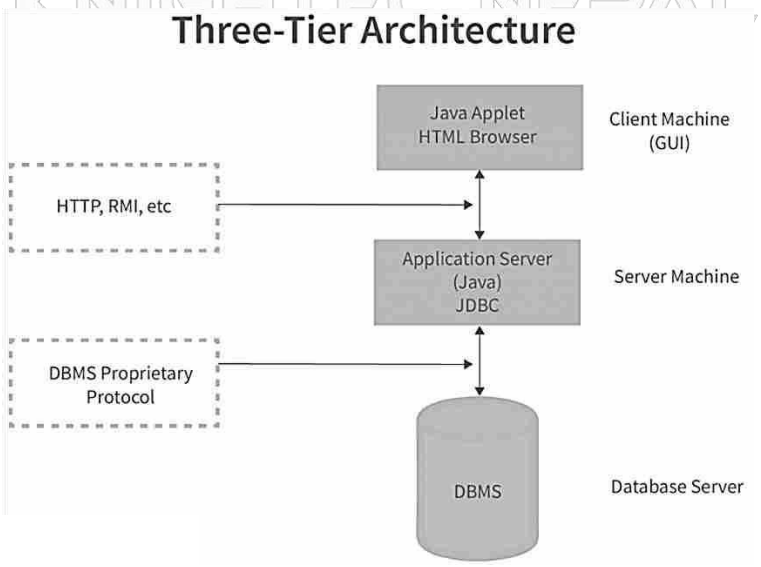
- **Client-Application Server-Database Model:** The client interacts with the database through an application server that manages the logic and database interactions.
- **Separation of Concerns:** The application server handles business logic, while the database server handles data storage.
- **Middle Layer:** The application server receives client requests, processes them, and communicates with the database on behalf of the client.

How It Works:

- The client sends requests to the application server.
- The application server processes the business logic and interacts with the database using JDBC.
- The database server returns results to the application server, which are then passed back to the client.

Example:

A web application where the client (**browser**) sends HTTP requests to a web server (like **Tomcat**), which processes the business logic and interacts with a database (such as **Oracle**) using JDBC.



Drivers & Jar Files:

Drivers are software components that act as intermediaries between a Java application and a specific hardware device or software system. They provide the necessary interface and protocols for the application to interact with the device or system effectively.

Drivers enable Java applications to utilize the functionalities of various hardware devices, such as printers, scanners, databases, and network interfaces. They handle the complexities of device-specific communication and translate the application's requests into the appropriate commands for the device.

Types:

- **JDBC Drivers:** Connect Java applications to databases, allowing them to execute SQL queries and retrieve data.
- **Network Drivers:** Facilitate communication between Java applications and network devices like routers and switches.
- **Printer Drivers:** Enable Java applications to print documents to various printers.
- **Scanner Drivers:** Allow Java applications to scan documents and images.

JAR (**J**va **AR**chive) files are compressed archives that contain multiple Java files and resources. They are commonly used to package Java applications and libraries for distribution and deployment.

- **Packaging:** JAR files bundle all the necessary files for a Java application, including class files, resource files (images, text files, etc.), and metadata.
- **Distribution:** JAR files can be easily distributed and deployed on different systems.
- **Execution:** JAR files can be executed directly using the java -jar command, making it convenient for running Java applications.

Common JDBC Jar Files:

- **MySQL:** mysql-connector-java-x.x.x.jar
- **Oracle:** ojdbc8.jar
- **PostgreSQL:** postgresql-x.x.x.jar
- **SQL Server:** mssql-jdbc-x.x.x.jar

These JAR files contain the necessary classes to connect to the respective databases using JDBC.

Driver Types:

There are four types of JDBC drivers, categorized based on how they communicate with the database:

1. Type 1: JDBC-ODBC Bridge Driver:

- Uses ODBC (Open Database Connectivity) to interact with the database.
- Works with ODBC drivers to translate Java calls to database-specific calls.
- **Pros:** Easy to use and universal.
- **Cons:** Slow due to the overhead of ODBC. Requires ODBC installation.

2. Type 2: Native-API Driver:

- Uses native database client libraries to communicate with the database.
- **Pros:** Faster than Type 1, no need for ODBC.
- **Cons:** Database-specific; requires database client installation.

3. Type 3: Network Protocol Driver (Middleware Driver):

- Uses a middleware server to communicate with the database.
- **Pros:** No need for database-specific client-side libraries.
- **Cons:** Requires additional middleware setup.

4. Type 4: Thin Driver (Pure Java Driver):

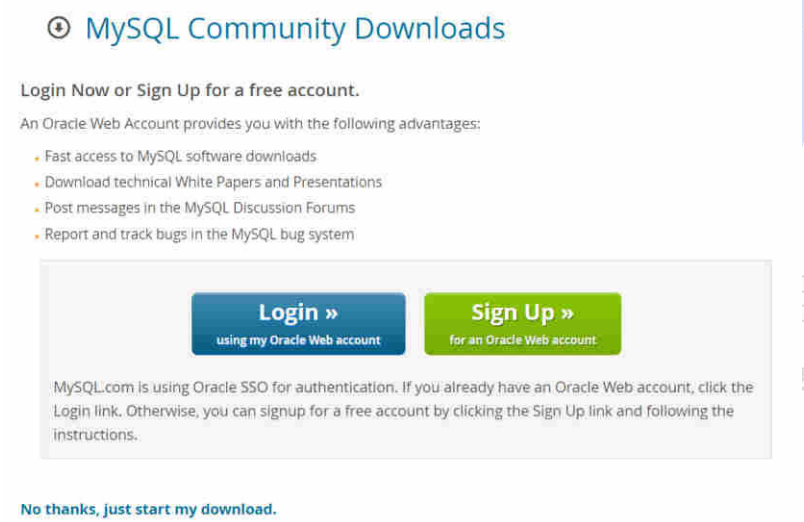
- Written entirely in Java and communicates directly with the database server.
- **Pros:** Platform-independent, no need for database-specific libraries.
- **Cons:** Database-specific; requires the correct version for different databases.

Steps for Database Connectivity:

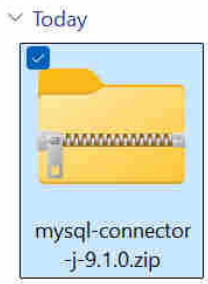
1. **Go to this link and Select Operating System that you are using.**
   https://dev.mysql.com/downloads/connector/j/

2. Select Platform Independent, even if you are using Windows Operating System.

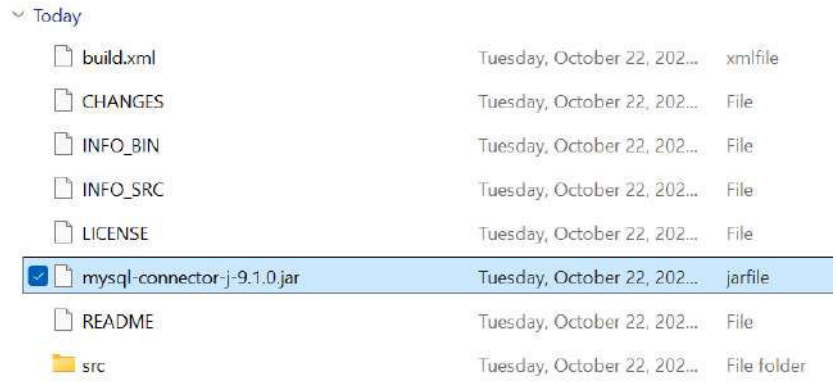
3. Don't waste your time to Login or Sign Up, Click on **No thanks, just start my download.**



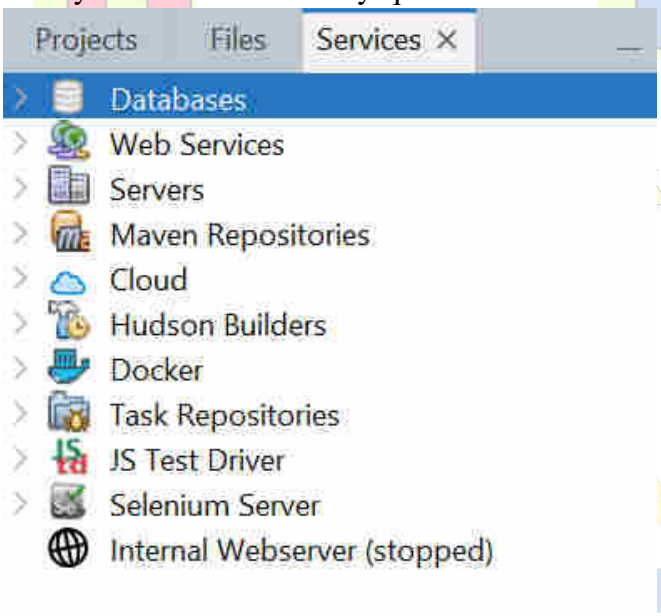
4. Now, your download process will start and after completed you can see the zip file.



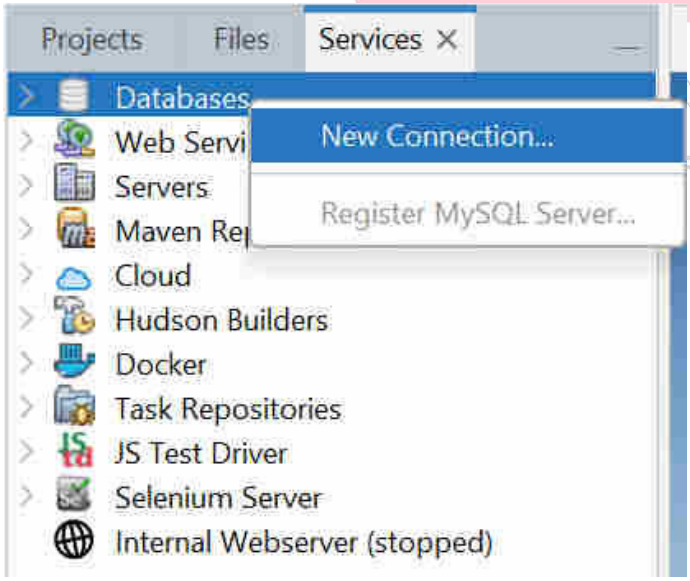
5. Extract the zip file that you just downloaded; inside the folder you can see **mysql-connector-j-9.1.0.jar** in my case **9.1.0** is my version (2024).



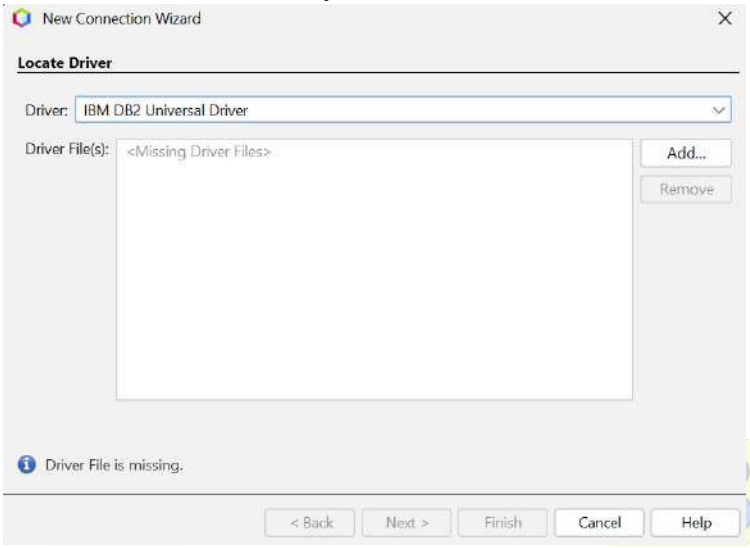
6. Now you need to add the mysql-connector into the Services.



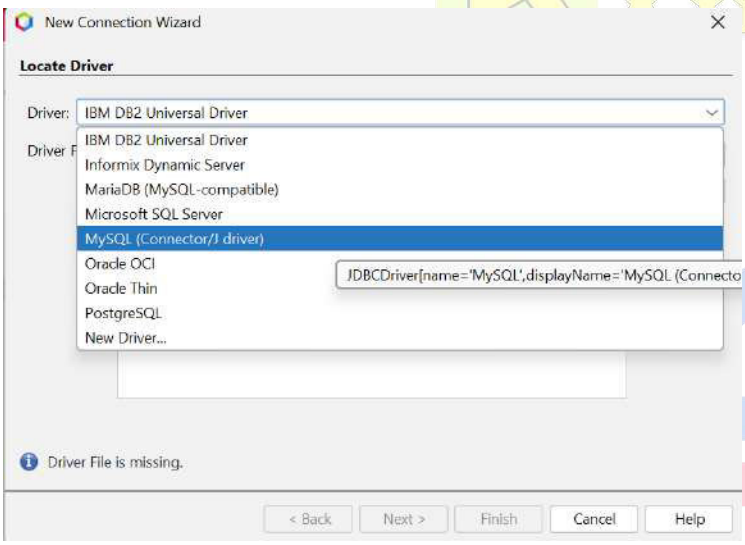
7. Right click on the **Databases ➔ click on New Connection…**



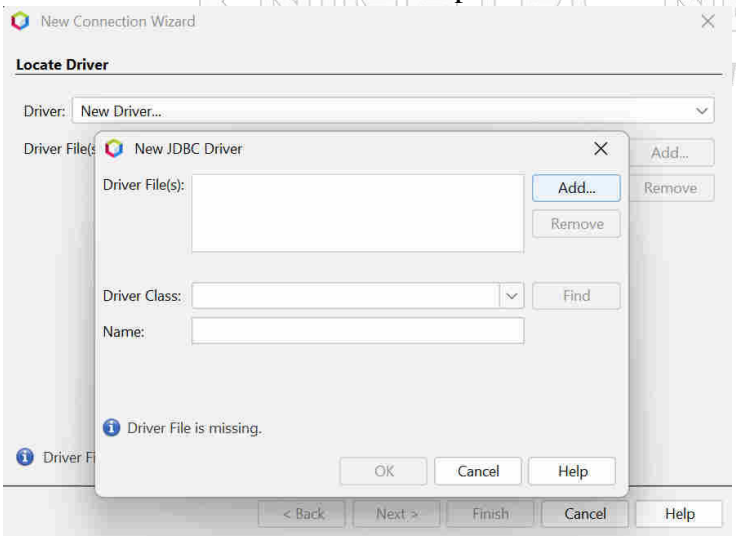
8. Locate the Driver which you want to use we will use **MYSQL** database so search for it.



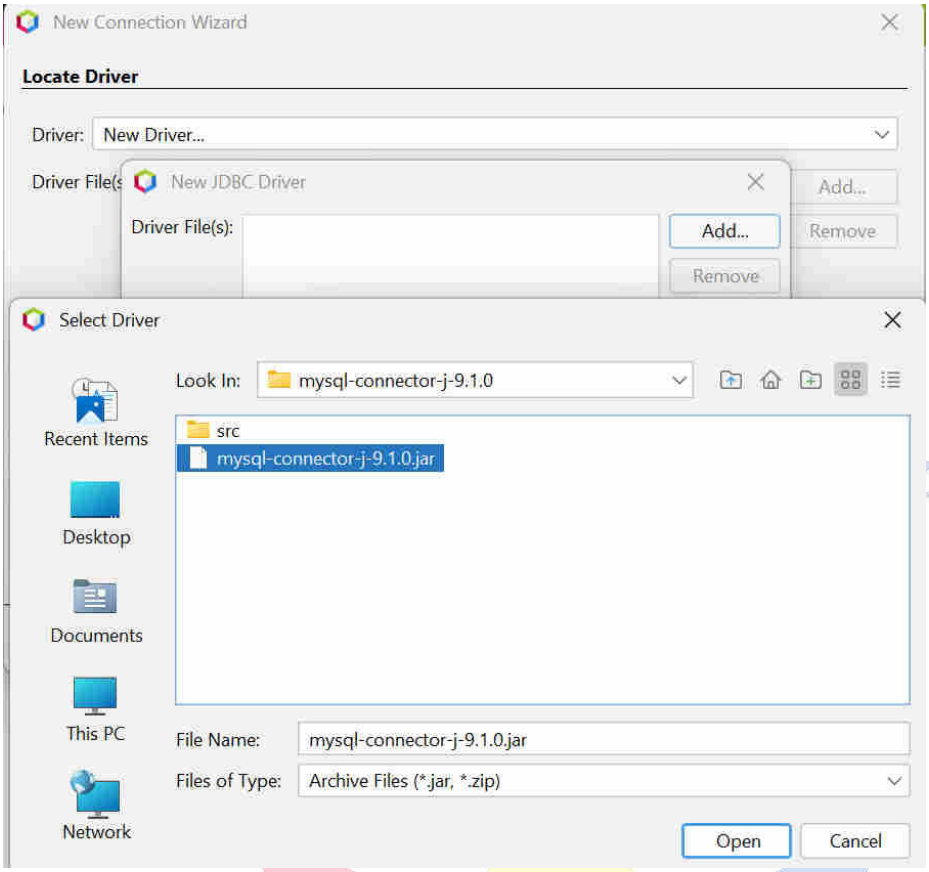
9. Here in my case, I already have **MYSQL**. If MYSQL (Connector/J driver) is mentioned in the list then select it.



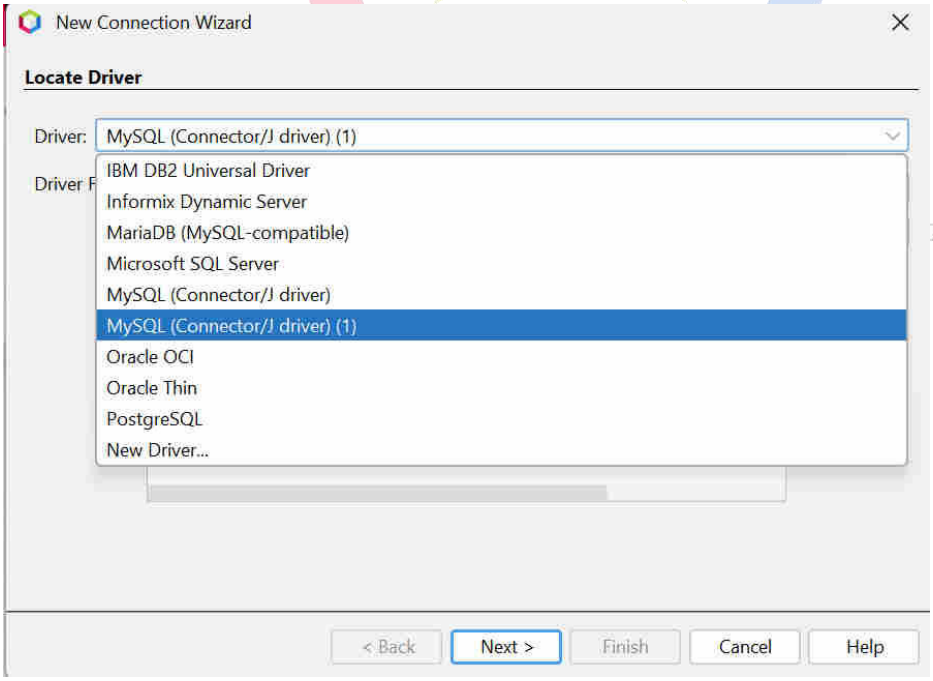
10. If not listed select **New Driver…** option and click on the **Add…** button.



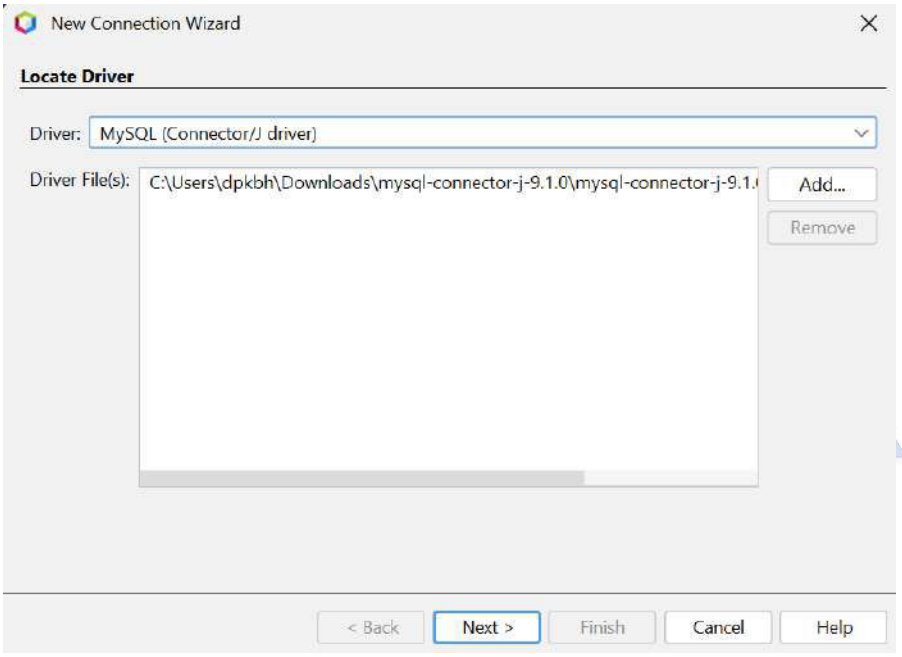
11. Search for the **mysql-connector-j-9.1.0.jar** file select it and click open.



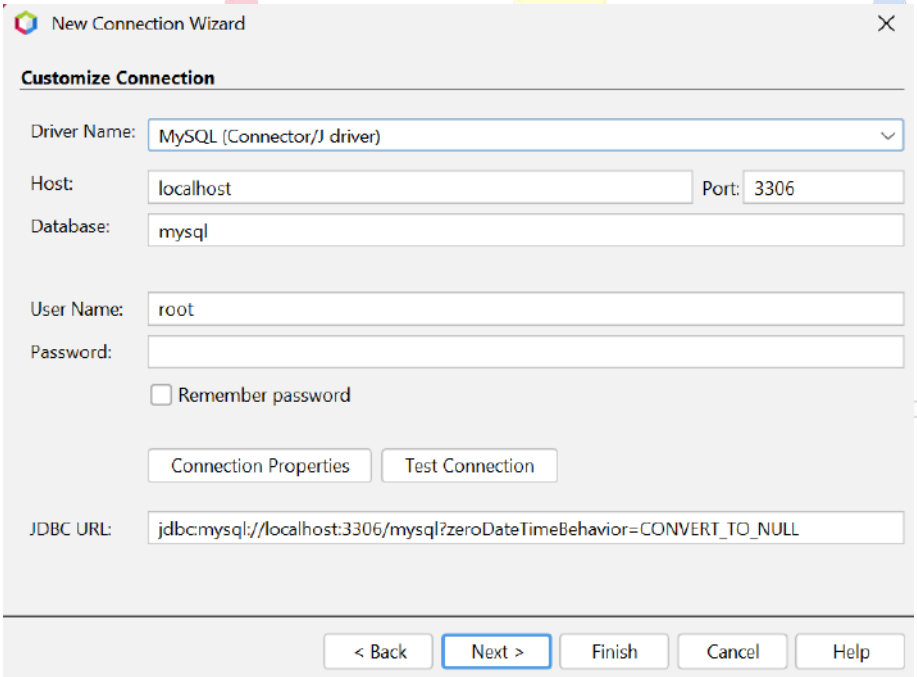
12. Now you will see the **MYSQL** driver in the **Driver** list.



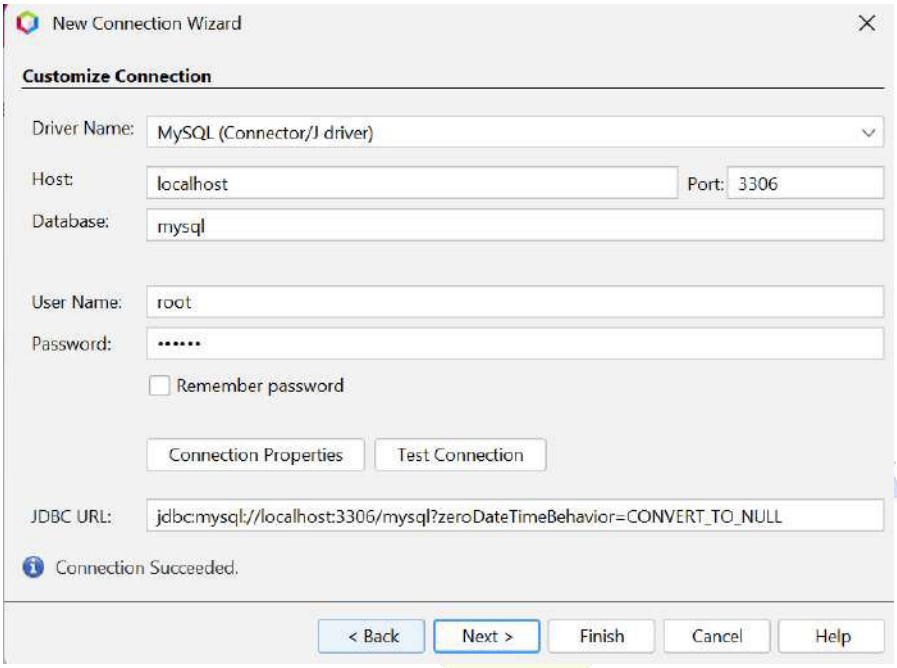
13. Now Select the **MYSQL (Connector/J driver)** from the list. If Driver File(s) is not listed then click on the Add… button and select the same file. If all good then click on the **Next.**



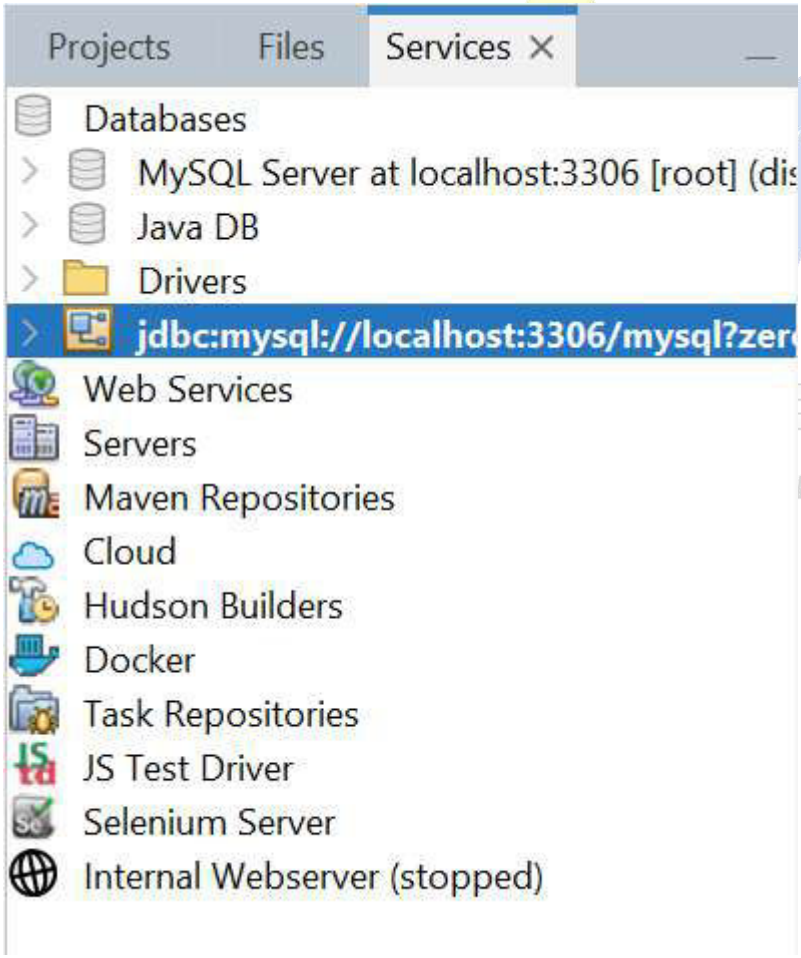
14. Now you will see the below **Customize Connection** Wizard.



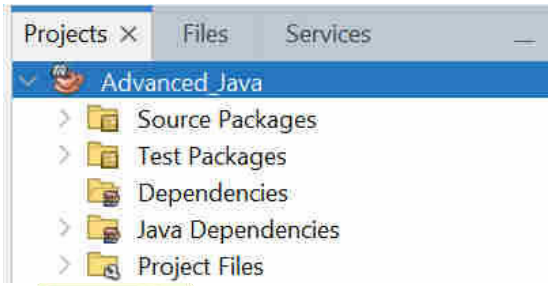
15. Set up the necessary things. Click on **Next**.



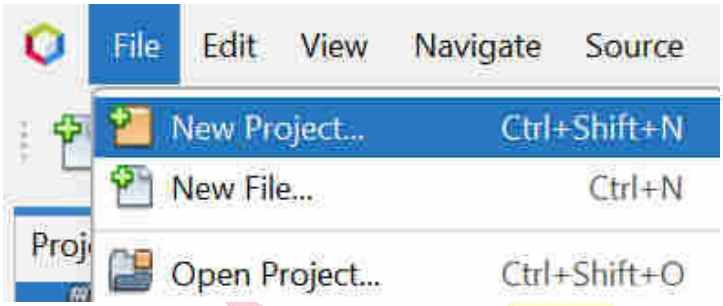
16. Now you can see the **jdbc:mysql://localhost:3306/** in the Services.



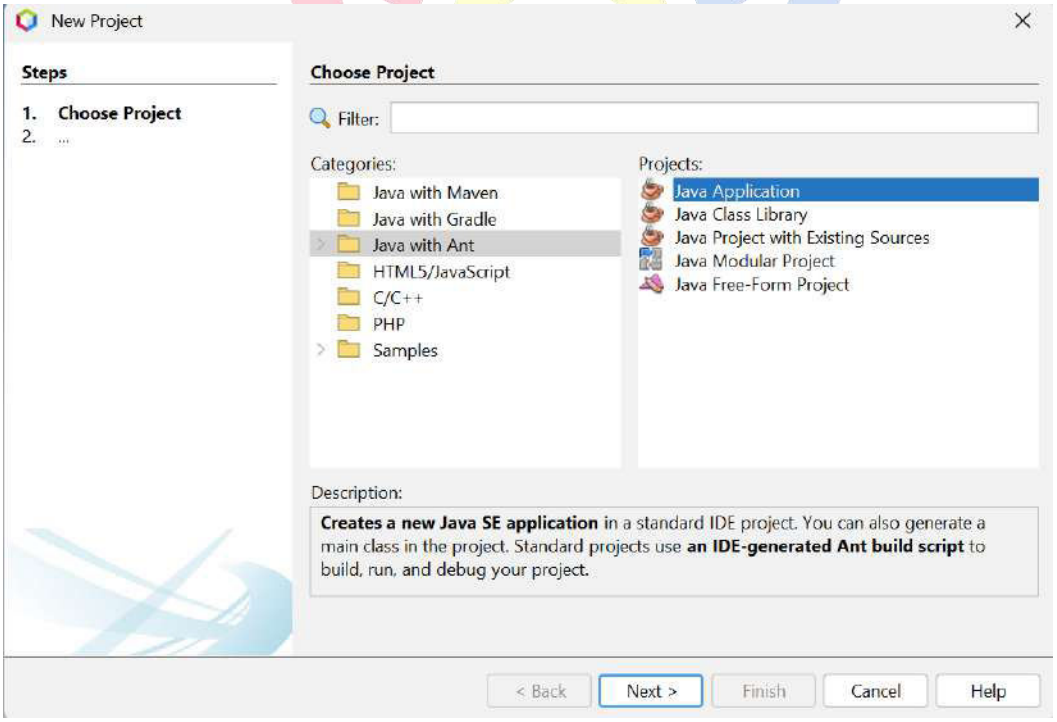
17. Now the thing is we need a **Libraries** folder in our Projects. This Libraries folder can only add the jar files. See below we have **Advanced_Java** Project folder is there but observe it we didn't see any Libraries folder. If you have observed carefully in the previous topic example, we have chosen **Java with Maven** while creating the project.



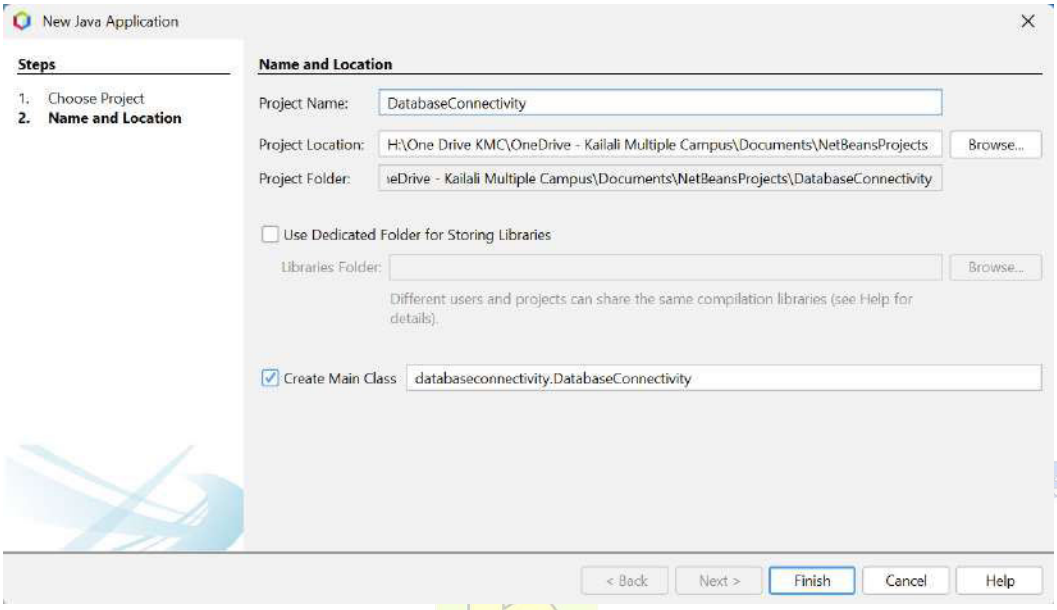
18. Now Go to **File ➔ New Project**.



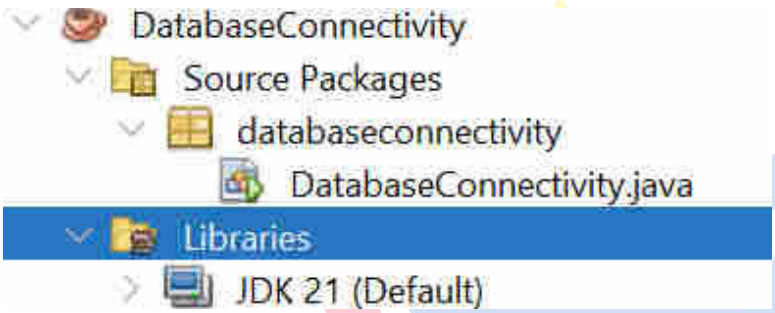
19. Choose the Project **Java with Ant ➔ Java Application**.



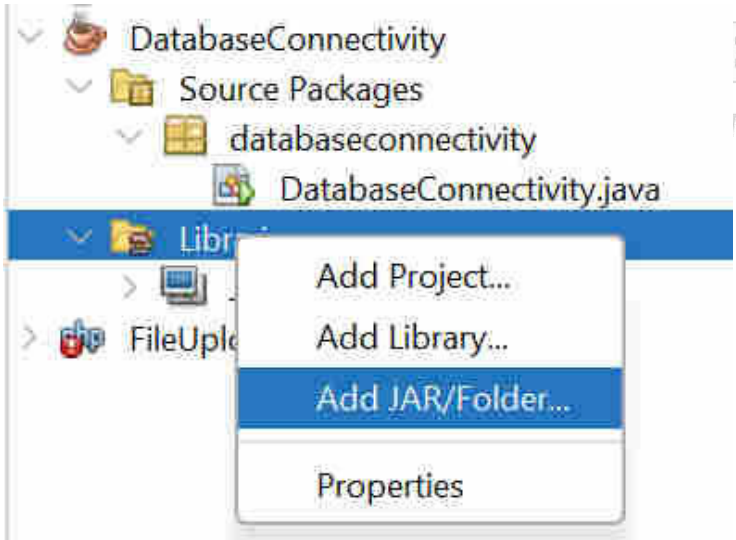
20. Provide the Project Name and Location.



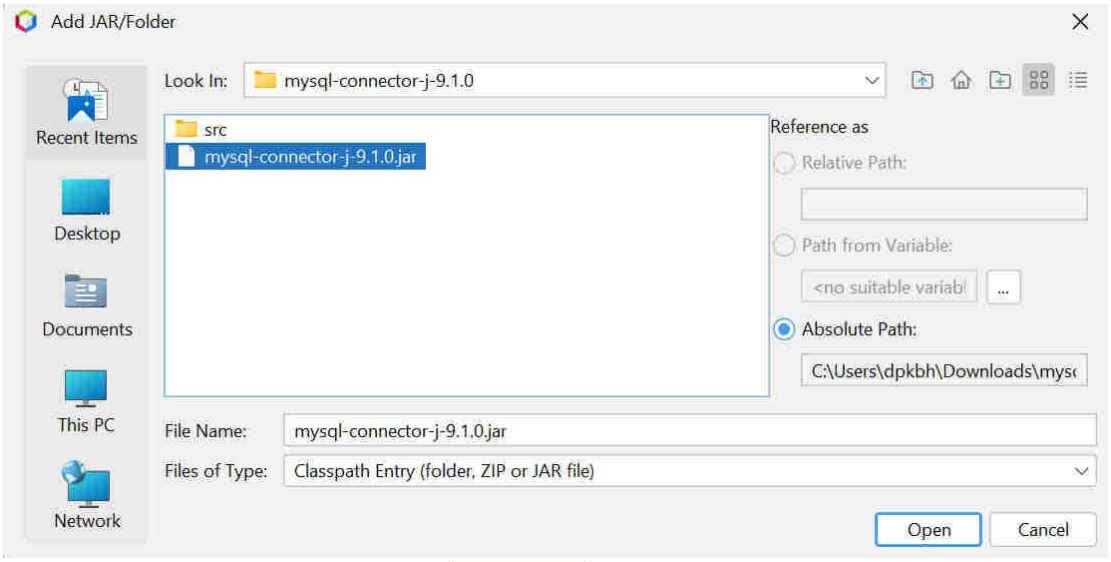
21. See, now you can see the **Libraries** folder.



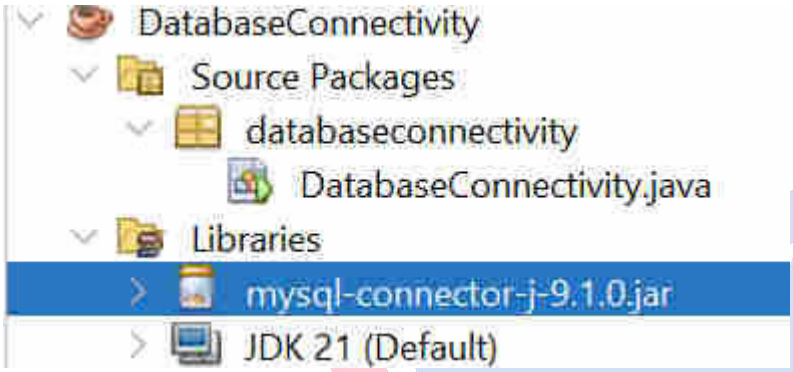
22. Right click on **Libraries** folder and click **Add JAR/Folder…**



23. Search for the **mysql-connector-j-9.1.0.jar** file and click **Open**.



24. Finally, we have added **mysql-connector-j-9.1.0.jar** file successfully.



25. Now time to test the database connection.

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| kmc_college        |
| mahabaudha         |
| mysql              |
| nodejs_database    |
| performance_schema |
| sakila             |
| sys                |
| wordpress          |
| world              |
+--------------------+
10 rows in set (0.00 sec)
```

Database Connection in JAVA

```
package databaseconnectivity;


import java.sql.*;


public class DatabaseConnectivity {


    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            Connection conn = DriverManager.getConnection("jdbc:mysql://
localhost:3306/kmc_college", "root", "bhatta");
            System.out.println("kmc_college Database Connected");
        } catch (ClassNotFoundException ex) {
            System.out.println("ERROR: Class not Found!" + ex);
        } catch (SQLException ex) {
            System.out.println("ERROR: Problem in SQL!" + ex);
        }
    }
}
```

Output:

```
run:
kmc_college Database Connected
BUILD SUCCESSFUL (total time: 0 seconds)
```

Steps for Connecting to JDBC:

1. Load the JDBC Driver:

Use *Class.forName("driver class name")* to load the driver, or use the ***java.sql.DriverManager*** class which automatically loads drivers since Java 6.

```
Class.forName("com.mysql.cj.jdbc.Driver");  // MySQL example
```

2. Establish a Connection:

Use **DriverManager.getConnection()** to create a connection to the database.

```
Connection conn = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/dbname", "username", "password");
```

3. Create a Statement:

Create a Statement or **PreparedStatement** object to send SQL queries.

```
Statement stmt = conn.createStatement();
```

4. Execute SQL Queries:

Use **executeQuery()** for retrieving data (SELECT statements) or **executeUpdate()** for modifying data (INSERT, UPDATE, DELETE).

```
ResultSet rs = stmt.executeQuery("SELECT * FROM users");
```

5. Process Results:

Iterate over the **ResultSet** to access the data retrieved from the database.

```
while (rs.next()) {
    System.out.println(rs.getString("name"));
}
```

6. Close the Connection:

Close the ResultSet, Statement, and Connection to free resources.

```
rs.close();
stmt.close();
conn.close();
```

4.2. Executing SQL Statements: Managing Connections, Statements, Result Set, SQL Exceptions, Populating Database.

Executing SQL Statements:

To interact with databases in a Java application, JDBC provides classes and interfaces for managing connections, executing SQL statements, handling result sets, managing SQL exceptions, and populating databases.

Managing Connections:

A **connection** to the database is the first step in interacting with it. JDBC provides the Connection interface for managing this session.

Steps to Manage a Connection:

- **Load the JDBC Driver:** Load the appropriate JDBC driver for the database.

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

- **Establish the Connection:** Use the **DriverManager.getConnection()** method to connect to the database. You need the database URL, username, and password.

```
Connection conn = DriverManager.getConnection(
"jdbc:mysql://localhost:3306/mydatabase", "username", "password");
```

- **Close the Connection:** After all database interactions are completed, you must close the connection to free resources.

```
conn.close();
```

Managing Statements:

The Statement interface is used to execute SQL queries. There are three types of statements in JDBC:

- **Statement:** Used for executing simple SQL queries without parameters.
- **PreparedStatement:** Precompiled SQL statement, used for executing parameterized queries. Improves performance and security (prevents SQL injection).
- **CallableStatement:** Used to execute stored procedures in the database.

Steps to Manage Statements:

- **Create a Statement:**

```
Statement stmt = conn.createStatement();
```

- **Execute SQL Queries:**
  - For **data retrieval** (SELECT queries), use **executeQuery**():

```
ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
```

  - For **data modification** (INSERT, UPDATE, DELETE), use **executeUpdate**():

```
int rowsAffected = stmt.executeUpdate("INSERT INTO employees (name, age) VALUES ('Deepak', 30)");
```

Example:

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM employees");

while (rs.next()) {
    System.out.println(rs.getString("name") + ", " + rs.getInt("age"));
}
stmt.close();
```

Managing Result Set:

A ResultSet object is a table of data representing a database result set, which is usually generated by executing a statement that queries the database. A ResultSet object can be created through any object that implements the Statement interface, including PreparedStatement, CallableStatement, and RowSet.

We access the data in a ResultSet object through a cursor. This cursor is a pointer that points to one row of data in the ResultSet. Initially, the cursor is positioned before the first row. The method **ResultSet.next()** moves the cursor to the next row. This method returns false if the cursor is positioned after the last row. This method repeatedly calls the ResultSet.next() method with a while loop to iterate through all the data in the ResultSet.

The ResultSet interface holds the result of a SELECT query. It acts as a cursor that can iterate over the rows of the result set.

Steps to Manage ResultSets:

- **Move the Cursor:** By default, the cursor points to the first row of the result set. Use next() to move the cursor to the next row.
- **Retrieve Data:** Use getter methods (e.g., getString(), getInt(), etc.) to retrieve data from the current row.
- **Close the ResultSet:** After processing the data, close the ResultSet to free up resources.

Example:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM employees");

while (rs.next()) {
    int id = rs.getInt("id");
    String name = rs.getString("name");
    int age = rs.getInt("age");

    System.out.println("ID: " + id + ", Name: " + name + ", Age: " + age);
}
rs.close();
```

SQL Exceptions:

When interacting with databases, errors may occur. JDBC provides the SQLException class to handle errors that happen during database operations.

Key Points About SQLException:

- **Error Code:** Use getErrorCode() to retrieve the database-specific error code.
- **SQL State:** getSQLState() gives the X/Open SQL standard code for the error.
- **Error Message:** getMessage() provides the detailed error message.
- **Stack Trace:** printStackTrace() prints the error stack trace to identify where the error occurred.

Example:

```
try {
    Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:
3306/DatabaseName", "UserName", "Password");
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT * FROM table_Name");
} catch (SQLException e) {
    System.out.println("Error Code: " + e.getErrorCode());
    System.out.println("SQL State: " + e.getSQLState());
    System.out.println("Error Message: " + e.getMessage());
    e.printStackTrace();
}
```

Populating Database:

You can populate the database using SQL INSERT statements. These statements can be executed using either **Statement** or **PreparedStatement**.

Example Using PreparedStatement:

```
String insertSQL = "INSERT INTO employees (name, age, department) VALUES
(?, ?, ?)";
PreparedStatement pstmt = conn.prepareStatement(insertSQL);
pstmt.setString(1, "Deepak Bhatta");
pstmt.setInt(2, 30);
pstmt.setString(3, "Asst. Professor");
```

```
int rowsAffected = pstmt.executeUpdate();
System.out.println(rowsAffected + " row(s) inserted.");

pstmt.close();
```

Example Using Batch Insertion:

Batch insertion is useful when inserting large amounts of data.

```
String insertSQL = "INSERT INTO employees (name, age) VALUES (?, ?)";
PreparedStatement pstmt = conn.prepareStatement(insertSQL);

pstmt.setString(1, "Deepak");
pstmt.setInt(2, 30);
pstmt.addBatch();

pstmt.setString(1, "Billi");
pstmt.setInt(2, 22);
pstmt.addBatch();

int[] affectedRows = pstmt.executeBatch();
System.out.println(affectedRows.length + " rows inserted.");

pstmt.close();
```

Note:

If you are refering old books, Notes and old Gen codes then you might have seen that there is a use of **sun.jdbc.odbc.JdbcOdbcDriver**.

The **sun.jdbc.odbc.JdbcOdbcDriver** is not supported in Java 8 or later, and is not available on non-Windows JVMs. The JDBC-ODBC bridge driver was **removed from Java 8** and later releases.

- **What it is**: The **sun.jdbc.odbc.JdbcOdbcDriver** is a JDBC driver that translates JDBC operations into ODBC operations. It is also known as the JDBC-ODBC bridge.
- **How it works**: The JDBC-ODBC bridge driver converts JDBC method calls into ODBC function calls.
- **When it was used**: The JDBC-ODBC bridge driver was used in **Java 5, 6, and 7**.
- **Why it was removed**: The JDBC-ODBC bridge driver was removed because ODBC is not appropriate for direct use from the Java programming language.

CRUD Operations in Database:

The operations such as insertion, creation, deletion, updating and selection of the tabular data are known as CRUD operation in database. Following are the CRUD operations with the appropriate examples;

CREATE OPERATION:

Example: Program that CREATE a database in MYSQL through java.

```
package databaseconnectivity;
import java.sql.*;
public class CreateDatabase {

    public static void main(String[] args) {
        Connection conn = null;
        Statement stat = null;

        try {
           // Load the MySQL JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");

           // Establish a connection to MySQL server (without specifying a database)
            String url = "jdbc:mysql://localhost:3306/"; // No specific database
            String user = "root";
            String password = "bhatta";
            conn = DriverManager.getConnection(url, user, password);

           // Create a statement object
            stat = conn.createStatement();

           // Create database query
            String createDatabaseQuery = "CREATE DATABASE IF NOT EXISTS nast_college";

           // Execute the create database query
            stat.executeUpdate(createDatabaseQuery);
```

```
            System.out.println("Database 'kmc_college' created
successfully.");
        } catch (ClassNotFoundException e) {
            System.out.println("ERROR: JDBC Driver not found: " + e);
        } catch (SQLException e) {
            System.out.println("ERROR: SQL Exception: " + e);
        } finally {
           // Close the connection and statement to free resources
            try {
                if (stat != null) {
                    stat.close();
                }
                if (conn != null) {
                    conn.close();
                }
            } catch (SQLException e) {
                System.out.println("ERROR: Closing resources: " + e);
            }
        }
    }
}
```

```
run:
Database 'kmc_college' created successfully.
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| kmc_college        |
| mahabaudha         |
| mysql              |
| nodejs_database    |
| performance_schema |
| sakila             |
| sys                |
| wordpress          |
| world              |
+--------------------+
10 rows in set (0.00 sec)
```

CREATE OPERATION:

Example: Program that CREATE a student table in database through java.

```
package databaseconnectivity;

import java.sql.*;

public class CreateTable {

    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;

        try {
            // Load the MySQL JDBC driver
             Class.forName("com.mysql.cj.jdbc.Driver");

            // Establish a connection to the database
             String url = "jdbc:mysql://localhost:3306/kmc_college";
             String user = "root";
             String password = "bhatta";
             conn = DriverManager.getConnection(url, user, password);

            // Create a statement object
             stmt = conn.createStatement();

            // Create table query
             String createTableQuery = "CREATE TABLE tbl_students ("
                    + "id INT NOT NULL AUTO_INCREMENT PRIMARY KEY, "
                    + "firstName VARCHAR(30) NOT NULL, "
                    + "middleName VARCHAR(20) NULL, "
                    + "lastName VARCHAR(30) NOT NULL, "
                    + "email VARCHAR(50) NOT NULL, "
                    + "password VARCHAR(50) NOT NULL, "
                    + "isActive BIT NULL DEFAULT 1 "
                    + ")";
```

```
            // Execute the create table query
            stmt.executeUpdate(createTableQuery);
            System.out.println("Table 'tbl_students' created successfully.");

        } catch (ClassNotFoundException e) {
            System.out.println("ERROR: JDBC Driver not found: " + e);
        } catch (SQLException e) {
            System.out.println("ERROR: SQL Exception: " + e);
        } finally {
            // Close the connection and statement to free resources
            try {
                if (stmt != null) {
                    stmt.close();
                }
                if (conn != null) {
                    conn.close();
                }
            } catch (SQLException e) {
                System.out.println("ERROR: Closing resources: " + e);
            }
        }
    }
}
```



```
run:
Table 'tbl_students' created successfully.
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
mysql> describe tbl_students;
+------------+-------------+------+-----+---------+-------------------+
| Field      | Type        | Null | Key | Default | Extra             |
+------------+-------------+------+-----+---------+-------------------+
| id         | int         | NO   | PRI | NULL    | auto_increment    |
| firstName  | varchar(30) | NO   |     | NULL    |                   |
| middleName | varchar(20) | YES  |     | NULL    |                   |
| lastName   | varchar(30) | NO   |     | NULL    |                   |
| email      | varchar(50) | NO   |     | NULL    |                   |
| password   | varchar(50) | NO   |     | NULL    |                   |
| isActive   | bit(1)      | YES  |     | 1       | DEFAULT_GENERATED |
+------------+-------------+------+-----+---------+-------------------+
7 rows in set (0.00 sec)
```

```
mysql> use kmc_college
Database changed
mysql> show tables;
+----------------------+
| Tables_in_kmc_college |
+----------------------+
| tbl_students          |
+----------------------+
1 row in set (0.00 sec)
```

INSERT OPERATION:

Example1: Program that INSERT a data in the student table through java.

```
package databaseconnectivity;
import java.sql.*;
public class DatabaseConnectivity {

    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;

        try {
           // Load the MySQL JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");

           // Establish a connection to the database
            String url = "jdbc:mysql://localhost:3306/kmc_college";
            String user = "root";
            String password = "bhatta";
            conn = DriverManager.getConnection(url, user, password);

           // Create SQL insert query
String str = "INSERT INTO tbl_students (firstName, middleName, lastName,
email, password) " +
"VALUES ('Billi', '', 'Pant', 'ghugu_billa@gmail.com', '123')";

           // Create a statement object
            stmt = conn.createStatement();

           // Execute the query
            stat.executeUpdate(str);
            System.out.println("Data inserted successfully");
```

```
        } catch (ClassNotFoundException e) {
            System.out.println("ERROR: JDBC Driver not found: " + e);
        } catch (SQLException e) {
            System.out.println("ERROR: SQL Exception: " + e);
        } finally {
           // Close the connection and statement to free resources
            try {
                if (stmt != null) {
                    stmt.close();
                }
                if (conn != null) {
                    conn.close();
                }
            } catch (SQLException e) {
                System.out.println("ERROR: Closing resources: " + e);
            }
        }
    }
}
```

Before Executing the Code:

```
mysql> SELECT * FROM tbl_students;
+----+-----------+------------+----------+-------------------------+----------+--------------------+
| id | firstName | middleName | lastName | email                   | password | isActive           |
+----+-----------+------------+----------+-------------------------+----------+--------------------+
|  1 | Deepak    |            | Bhatta   | dpkbhatta2051@gmail.com | 12345    | 0x01               |
+----+-----------+------------+----------+-------------------------+----------+--------------------+
1 row in set (0.00 sec)
```

After Executing the Code:

```
run:
Data inserted successfully
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
mysql> SELECT * FROM tbl_students;
+----+-----------+------------+----------+-------------------------+----------+--------------------+
| id | firstName | middleName | lastName | email                   | password | isActive           |
+----+-----------+------------+----------+-------------------------+----------+--------------------+
|  1 | Deepak    |            | Bhatta   | dpkbhatta2051@gmail.com | 12345    | 0x01               |
|  3 | Billi     |            | Pant     | ghugu_billa@gmail.com   | 123      | 0x01               |
+----+-----------+------------+----------+-------------------------+----------+--------------------+
2 rows in set (0.00 sec)
```

Example2: Program that INSERT multiple data in the student table through java.

```
package databaseconnectivity;
import java.sql.*;
public class InsertMultipleData {

    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;
        try {
            // Load the MySQL JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Establish a connection to the database
            String url = "jdbc:mysql://localhost:3306/kmc_college";
            String user = "root";
            String password = "bhatta";
            conn = DriverManager.getConnection(url, user, password);

            // Create a statement object
            stmt = conn.createStatement();

            // Add multiple insert statements to the batch
            stmt.addBatch("INSERT INTO tbl_students (firstName, middleName,
lastName, email, password) " +
"VALUES ('Harish', '', 'Bhatta', 'harry2001@gmail.com', '123456')");

            stmt.addBatch("INSERT INTO tbl_students (firstName, middleName,
lastName, email, password) " +
"VALUES ('Dinesh', 'Chandra', 'Bhatta', 'dns_bhatta@gmail.com', '12345')");

            stmt.addBatch("INSERT INTO tbl_students (firstName, middleName,
lastName, email, password) " +
"VALUES ('Pushpa', '', 'Bhatta', 'pusu07@gmail.com', '2145')");

            stmt.addBatch("INSERT INTO tbl_students (firstName, middleName,
lastName, email, password) " +
"VALUES ('Hemanti', '', 'Bhatta', 'hemu20@gmail.com', '147')");
```

```
            // Execute all the insert queries as a batch
             int[] results = stmt.executeBatch();

            // Check the results
             System.out.println("Inserted rows: " + results.length);

        } catch (ClassNotFoundException e) {
             System.out.println("ERROR: JDBC Driver not found: " + e);
        } catch (SQLException e) {
             System.out.println("ERROR: SQL Exception: " + e);
        } finally {
            // Close the connection and statement to free resources
             try {
                 if (stmt != null) {
                     stmt.close();
                 }
                 if (conn != null) {
                     conn.close();
                 }
             } catch (SQLException e) {
                 System.out.println("ERROR: Closing resources: " + e);
             }
        }
    }
}
```


After Executing the Code:

```
run:
Inserted rows: 4
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
mysql> SELECT * FROM tbl_students;
+----+-----------+------------+----------+------------------------+----------+--------------------+
| id | firstName | middleName | lastName | email                  | password | isActive           |
+----+-----------+------------+----------+------------------------+----------+--------------------+
|  1 | Deepak    |            | Bhatta   | dpkbhatta2051@gmail.com | 12345    | 0x01               |
|  3 | Billi     |            | Pant     | ghugu_billa@gmail.com  | 123      | 0x01               |
|  4 | Harish    |            | Bhatta   | harry2001@gmail.com    | 123456   | 0x01               |
|  5 | Dinesh    | Chandra    | Bhatta   | dns_bhatta@gmail.com   | 12345    | 0x01               |
|  6 | Pushpa    |            | Bhatta   | pusu07@gmail.com       | 2145     | 0x01               |
|  7 | Hemanti   |            | Bhatta   | hemu20@gmail.com       | 147      | 0x01               |
+----+-----------+------------+----------+------------------------+----------+--------------------+
6 rows in set (0.00 sec)
```

UPDATE OPERATION:

Example: Program that is used to MODIFY the record in the table of SQL database.

```
package databaseconnectivity;

import java.sql.*;

public class UpdateData {

    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;
        try {
            // Load the MySQL JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Establish a connection to the database
            String url = "jdbc:mysql://localhost:3306/kmc_college";
            String user = "root";
            String password = "bhatta";
            conn = DriverManager.getConnection(url, user, password);

            // Create a statement object
            stmt = conn.createStatement();

            // Update operation query
            String updateQuery = "UPDATE tbl_students "
                    + "SET lastName = 'Bhatta Kaji', password = 'csit_123'
"
                    + "WHERE id = 1";

            // Execute the update
            int rowsAffected = stat.executeUpdate(updateQuery);

            // Check if any rows were updated
            if (rowsAffected > 0) {
                System.out.println("Update successful. Rows affected: " +
rowsAffected);
```

```
            } else {
                System.out.println("No matching records found to update.");
            }
        } catch (ClassNotFoundException e) {
            System.out.println("ERROR: JDBC Driver not found: " + e);
        } catch (SQLException e) {
            System.out.println("ERROR: SQL Exception: " + e);
        } finally {
            // Close the connection and statement to free resources
            try {
                if (stmt != null) {
                    stmt.close();
                }
                if (conn != null) {
                    conn.close();
                }
            } catch (SQLException e) {
                System.out.println("ERROR: Closing resources: " + e);
            }
        }
    }
}
```


After Executing the Code:

```
run:
Update successful. Rows affected: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
mysql> SELECT * FROM tbl_students;
+----+-----------+------------+-------------+-------------------------+----------+--------------------+
| id | firstName | middleName | lastName    | email                   | password | isActive           |
+----+-----------+------------+-------------+-------------------------+----------+--------------------+
|  1 | Deepak    |            | Bhatta Kaji | dpkbhatta2051@gmail.com | csit_123 | 0x01               |
|  3 | Billi     |            | Pant        | ghugu_billa@gmail.com   | 123      | 0x01               |
|  4 | Harish    |            | Bhatta      | harry2001@gmail.com     | 123456   | 0x01               |
|  5 | Dinesh    | Chandra    | Bhatta      | dns_bhatta@gmail.com    | 12345    | 0x01               |
|  6 | Pushpa    |            | Bhatta      | pusu07@gmail.com        | 2145     | 0x01               |
|  7 | Hemanti   |            | Bhatta      | hemu20@gmail.com        | 147      | 0x01               |
+----+-----------+------------+-------------+-------------------------+----------+--------------------+
6 rows in set (0.00 sec)
```

DELETE OPERATION:

Example: Program that is used to DELETE record from the table of SQL database.

```
package databaseconnectivity;

import java.sql.*;

public class DeleteData {

    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;

        try {
            // Load the MySQL JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Establish a connection to the database
            String url = "jdbc:mysql://localhost:3306/kmc_college";
            String user = "root";
            String password = "bhatta";
            conn = DriverManager.getConnection(url, user, password);

            // Create a statement object
            stmt = conn.createStatement();

            // Delete operation query
            String deleteQuery = "DELETE FROM tbl_students WHERE
            firstName = 'Kalu' AND lastname = 'Billa'";

            // Execute the delete operation
            int rowsAffected = stmt.executeUpdate(deleteQuery);

            // Check if any rows were deleted
            if (rowsAffected > 0) {
                System.out.println("Delete successful. Rows affected: " +
rowsAffected);
            } else {
```

```
                System.out.println("No matching records found to delete.");
            }
        } catch (ClassNotFoundException e) {
            System.out.println("ERROR: JDBC Driver not found: " + e);
        } catch (SQLException e) {
            System.out.println("ERROR: SQL Exception: " + e);
        } finally {
           // Close the connection and statement to free resources
            try {
                if (stmt != null) { stmt.close(); }
                if (conn != null) { conn.close(); }
            } catch (SQLException e) {
                System.out.println("ERROR: Closing resources: " + e);
            }
        }
    }
}
```

Before Executing the Code:

```
mysql> SELECT * FROM tbl_students;
+----+-----------+------------+-------------+-------------------------+----------+--------------------+
| id | firstName | middleName | lastName    | email                   | password | isActive           |
+----+-----------+------------+-------------+-------------------------+----------+--------------------+
|  1 | Deepak    |            | Bhatta Kaji | dpkbhatta2051@gmail.com | csit_123 | 0x01               |
|  3 | Billi     |            | Pant        | ghugu_billa@gmail.com   | 123      | 0x01               |
|  4 | Harish    |            | Bhatta      | harry2001@gmail.com     | 123456   | 0x01               |
|  5 | Dinesh    | Chandra    | Bhatta      | dns_bhatta@gmail.com    | 12345    | 0x01               |
|  6 | Pushpa    |            | Bhatta      | pusu07@gmail.com        | 2145     | 0x01               |
|  7 | Hemanti   |            | Bhatta      | hemu20@gmail.com        | 147      | 0x01               |
|  9 | Kalu      |            | Billa       | kalubilla@gmail.com     | 12345    | 0x01               |
+----+-----------+------------+-------------+-------------------------+----------+--------------------+
7 rows in set (0.00 sec)
```

After Executing the Code:

```
run:
Delete successful. Rows affected: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
mysql> SELECT * FROM tbl_students;
+----+-----------+------------+-------------+-------------------------+----------+--------------------+
| id | firstName | middleName | lastName    | email                   | password | isActive           |
+----+-----------+------------+-------------+-------------------------+----------+--------------------+
|  1 | Deepak    |            | Bhatta Kaji | dpkbhatta2051@gmail.com | csit_123 | 0x01               |
|  3 | Billi     |            | Pant        | ghugu_billa@gmail.com   | 123      | 0x01               |
|  4 | Harish    |            | Bhatta      | harry2001@gmail.com     | 123456   | 0x01               |
|  5 | Dinesh    | Chandra    | Bhatta      | dns_bhatta@gmail.com    | 12345    | 0x01               |
|  6 | Pushpa    |            | Bhatta      | pusu07@gmail.com        | 2145     | 0x01               |
|  7 | Hemanti   |            | Bhatta      | hemu20@gmail.com        | 147      | 0x01               |
+----+-----------+------------+-------------+-------------------------+----------+--------------------+
6 rows in set (0.00 sec)
```

4.3. Query Execution: Prepared Statements, Reading and Writing LOBs, SQL Escapes, Multiple Results, Scrollable Result Sets, Updateable Result Sets, Row Sets and Cached Row Sets, Transactions.

Query Execution:

To **run SQL queries** in **Java**, you need a Connection object. See the previous topic.

Prepared Statements:

The PreparedStatement is derived from the more general class, Statement. The PreparedStatement interface extends the Statement interface which gives us added functionality over a generic Statement object. This statement gives us the flexibility of supplying arguments dynamically. It is more efficient to use the PreparedStatement because the SQL statement that is sent gets pre-compiled in the DBMS.

We can also use PreparedStatement to safely provide values to the SQL parameters, through a range of setter methods (i.e. setInt (int, int), setString (int, String), etc.). We can execute the same query repeatedly with different parameter values. The PreparedStatement interface accepts input parameters at runtime i.e. the PreparedStatement object only uses the IN parameter. Just as we close a Statement object, we should also close the PreparedStatement object.

Example:

```
String query =
"INSERT INTO tbl_students (id, firstName, middleName, lastName, email)
VALUES (?, ?, ?, ?, ?)";
PreparedStatement pstmt = conn.prepareStatement(query);
  pstmt.setInt(1, 15);
  pstmt.setString(2, "Deepak");
  pstmt.setString(3, "");
  pstmt.setString(4, "Bhatta");
  pstmt.setString(5, "dpkbhatta2051.in@gmail.com");
  pstmt.execute();
```

- The five question marks (?) in the the preceding SQL statement's last line are placeholders for values that will be passed as part of the query to the database.
- Before executing a PreparedStatement, the program must specify the parameter values by using the Prepared- Statement interface's set methods.
- For the preceding query, parameters are int and strings that can be set with Prepared-Statement method **setInt()** and **setString()**.
- Method **setInt**'s and **setString**'s *first argument represents the parameter number* being set, and the *second argument is that parameter's value*.
- Parameter numbers are counted from 1, starting with the first question mark (?).

Reading and Writing LOBs:

LOBs include data types such as **BLOB** (**Binary Large Object**) and **CLOB** (**Character Large Object**), used for handling large data like images or text.

For Example, for LOB's:

- TINYBLOB ≈ 255 bytes,
- BLOB ≈ 64KB,
- MEDIUMBLOB ≈ 16MB and
- LONGBLOB ≈ 4GB

Use Alter table command as per your Space requirements:

```
ALTER TABLE 'TABLE_NAME' MODIFY 'FIELD_NAME' MEDIUMBLOB;
```


First Create a Table for storing file

```
CREATE TABLE files (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100),
    file_data MEDIUMBLOB
);
```

```
mysql> describe files;
+-----------+--------------+------+-----+---------+----------------+
| Field     | Type         | Null | Key | Default | Extra          |
+-----------+--------------+------+-----+---------+----------------+
| id        | int          | NO   | PRI | NULL    | auto_increment |
| name      | varchar(100) | YES  |     | NULL    |                |
| file_data | mediumblob   | YES  |     | NULL    |                |
+-----------+--------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)
```

Example: Writing a BLOB to the Database

```
package BlobExamples;
import java.io.*;
import java.sql.*;
public class WriteBlob {
    public static void main(String[] args) {
        Connection conn = null;
        PreparedStatement stmt = null;
        FileInputStream fis = null;
        String jdbcURL = "jdbc:mysql://localhost:3306/kmc_college";
        String username = "root";
        String password = "bhatta";

        String filePath = "H:\\One Drive KMC\\OneDrive - Kailali Multiple Campus\\Desktop\\My Image DBK.png";

        String sql = "INSERT INTO files (name, file_data) VALUES (?, ?)";
        try {
            conn = DriverManager.getConnection(jdbcURL, username, password);
            stmt = conn.prepareStatement(sql);
            fis = new FileInputStream(filePath);

            stmt.setString(1, "Sample Image");
            stmt.setBinaryStream(2, fis, fis.available());

            int rowsInserted = stmt.executeUpdate();
            if (rowsInserted > 0) {
                System.out.println("A new file was inserted successfully!");
            }
        } catch (SQLException | IOException ex) {
            ex.printStackTrace();
        }
```


```
run:
A new file was inserted successfully!
BUILD SUCCESSFUL (total time: 0 seconds)
```

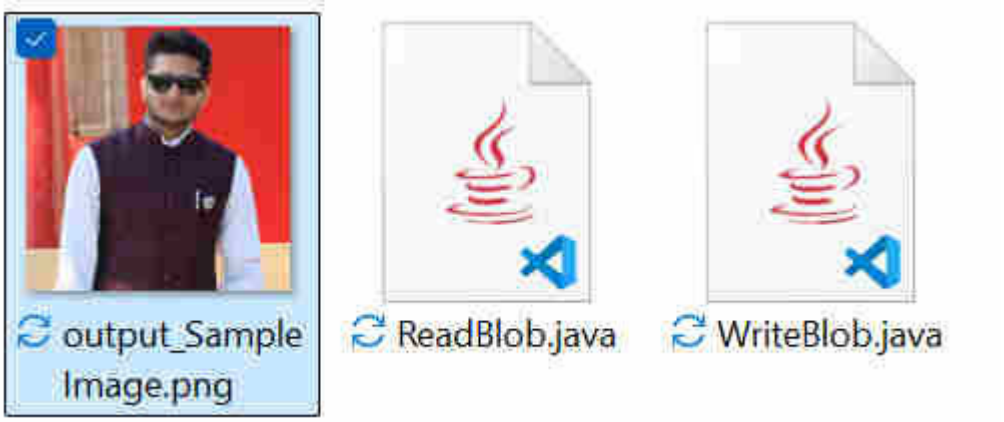
```
    }
}
```

Example: Reading a BLOB from the Database

```
package BlobExamples;

import java.io.*;

import java.sql.*;

public class ReadBlob {

    public static void main(String[] args) {

        Connection conn = null;
        PreparedStatement stmt = null;
        InputStream inputStream;
        FileOutputStream outputStream;

        String jdbcURL = "jdbc:mysql://localhost:3306/kmc_college";
        String username = "root";
        String password = "bhatta";

        String sql = "SELECT name, file_data FROM files WHERE id=?";

        try {
            conn     =     DriverManager.getConnection(jdbcURL,     username,
password);
            stmt = conn.prepareStatement(sql);

            stmt.setInt(1, 1); // Specify the ID of the file to retrieve

            ResultSet result = stmt.executeQuery();
            if (result.next()) {
                String fileName = result.getString("name");
                inputStream = result.getBinaryStream("file_data");

              // Specify the path to save the file
```

```
                outputStream = new FileOutputStream("H:\\One Drive KMC\\OneDrive - Kailali Multiple Campus\\Documents\\NetBeansProjects\\DatabaseConnectivity\\src\\BlobExamples\\output_" + fileName + ".png");
                byte[] buffer = new byte[1024];
                int bytesRead = -1;

                // Read the binary stream and write to the file
                while ((bytesRead = inputStream.read(buffer)) != -1) {
                    outputStream.write(buffer, 0, bytesRead);
                }
                outputStream.close();
                inputStream.close();

                System.out.println("File saved: output_" + fileName);
            }
        } catch (SQLException | IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

```
run:
File saved: output_Sample Image
BUILD SUCCESSFUL (total time: 0 seconds)
```



SQL Escapes:

SQL escapes allow special syntax that can be translated into database-specific syntax. Examples include handling dates, functions, and calling stored procedures.

Example:

```
String query = "SELECT * FROM tbl_students WHERE registration_date = {d '2024-10-22'}";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(query);
```

Multiple Results:

You can retrieve multiple result sets when executing stored procedures or complex queries.

Example:

```
String query = "{CALL getMultipleResults()}"; // Assume stored procedure returns multiple result sets
CallableStatement cstmt = conn.prepareCall(query);
boolean hasMoreResults = cstmt.execute();
while (hasMoreResults) {
    ResultSet rs = cstmt.getResultSet();
    while (rs.next()) {
        // Process each ResultSet
    }
    hasMoreResults = cstmt.getMoreResults();
}
```

Scrollable Result Sets:

Scrollable result sets allow navigation through the rows in any direction (forward, backward, random).

A **scrollable result set** allows you to move the cursor both forward and backward through the data in a ResultSet. This is useful when you need more flexibility in navigating through query results. By default, ResultSet in JDBC is **forward-only**, meaning you can only move from the first row to the last.

To create a scrollable result set, you need to specify the type of ResultSet when creating the Statement object. There are two primary types for scrollable result sets:

- **TYPE_SCROLL_INSENSITIVE:** The result set can be scrolled, and it does not reflect changes made by others to the database after the result set was created.
- **TYPE_SCROLL_SENSITIVE:** The result set can be scrolled, and it reflects changes made by others after it was created.

Example:

```
package ResultSetExamples;
import java.sql.*;
public class ScrollableResultSet {
    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;

        String jdbcURL = "jdbc:mysql://localhost:3306/kmc_college";
        String username = "root";
        String password = "bhatta";

        try {
            conn     =     DriverManager.getConnection(jdbcURL,     username,
password);
            // Create a Statement with scrollable ResultSet
```

```
            stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY); // Execute query and get a scrollable ResultSet

            rs = stmt.executeQuery("SELECT id, firstName, lastName FROM
tbl_students");

           // Move cursor to the last row

            if (rs.last()) {

                System.out.println("Last row data: " + rs.getInt("id") + ",
" + rs.getString("firstName") + ", " + rs.getString("lastName"));

            }

           // Move cursor to the first row

            if (rs.first()) {

                System.out.println("First row data: " + rs.getInt("id") +
", " + rs.getString("firstName") + ", " + rs.getString("lastName"));

            }

           // Move cursor to the 5th row

            if (rs.absolute(5)) {

                System.out.println("5th row data: " + rs.getInt("id") + ",
" + rs.getString("firstName") + ", " + rs.getString("lastName"));

            }

           // Move cursor backward (previous row) in this case 5th row Previous

            if (rs.previous()) {

                System.out.println("Previous row data: " + rs.getInt("id")
+ ", " + rs.getString("firstName") + ", " + rs.getString("lastName"));

            }

        } catch (SQLException ex) {

            ex.printStackTrace();

        }

    }
}
```

```
run:
Last row data: 7, Hemanti, Bhatta
First row data: 1, Deepak, Bhatta Kaji
5th row data: 6, Pushpa, Bhatta
Previous row data: 5, Dinesh, Bhatta
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
mysql> SELECT * FROM tbl_students;
+----+-----------+------------+------------+-------------------------+----------+--------------------+
| id | firstName | middleName | lastName   | email                   | password | isActive           |
+----+-----------+------------+------------+-------------------------+----------+--------------------+
|  1 | Deepak    |            | Bhatta Kaji | dpkbhatta2051@gmail.com | csit_123 | 0x01               |
|  3 | Billi     |            | Pant       | ghugu_billa@gmail.com   | 123      | 0x01               |
|  4 | Harish    |            | Bhatta     | harry2001@gmail.com     | 123456   | 0x01               |
|  5 | Dinesh    | Chandra    | Bhatta     | dns_bhatta@gmail.com    | 12345    | 0x01               |
|  6 | Pushpa    |            | Bhatta     | pusu07@gmail.com        | 2145     | 0x01               |
|  7 | Hemanti   |            | Bhatta     | hemu20@gmail.com        | 147      | 0x01               |
+----+-----------+------------+------------+-------------------------+----------+--------------------+
6 rows in set (0.00 sec)
```

Updateable Result Sets:

Updatable result sets allow updates to rows fetched by the result set directly.

Example:

```
package ResultSetExamples;
import java.sql.*;
public class UpdatableResultSet {
    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;

        String jdbcURL = "jdbc:mysql://localhost:3306/kmc_college";
        String username = "root";
        String password = "bhatta";
        try {
            conn = DriverManager.getConnection(jdbcURL, username, password);
            // Create a Statement with updatable ResultSet
             stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE); // Execute query and get an updatable ResultSet
             rs = stmt.executeQuery("SELECT id, firstName, lastName FROM
tbl_students");
             if (rs.first()) {   // Update the last name of the first row
                 System.out.println("Before update: " + rs.getInt("id") + ",
" + rs.getString("firstName") + ", " + rs.getString("lastName"));

               rs.updateString("lastName", "Bhatta"); // Modify the last name
                 rs.updateRow();   // Commit the change to the database
                 System.out.println("After update: " + rs.getInt("id") + ",
" + rs.getString("firstName") + ", " + rs.getString("lastName"));
             }
        } catch (SQLException ex) {
            ex.printStackTrace();
        }
```

```
run:
Before update: 1, Deepak, Bhatta Kaji
After update: 1, Deepak, Bhatta
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
    }
}
```

Row Sets and Cached Row Sets:

RowSets extend the **ResultSet** interface and can operate in a disconnected mode, while **CachedRowSet** is a type of **RowSet** that stores its data in memory, making it very useful in disconnected environments.

Row Sets:

- Uses a **JdbcRowSet**, which is connected, meaning the database connection remains open while processing the rows.
- You set up connection details directly with the **rowSet.setUrl()**, **setUsername()**, and **setPassword()** methods.
- The **execute()** method is used to run the query and fetch the data.
- The data is processed while the connection is active.

Example:

```
package QueryExecution;
import javax.sql.rowset.JdbcRowSet;
import javax.sql.rowset.RowSetProvider;
import java.sql.SQLException;

public class RowSetExample {

    public static void main(String[] args) {
        try {
            // Create JdbcRowSet instance using RowSetProvider
             JdbcRowSet rowSet =
RowSetProvider.newFactory().createJdbcRowSet();

            // Set connection details
             rowSet.setUrl("jdbc:mysql://localhost:3306/kmc_college");
             rowSet.setUsername("root");
             rowSet.setPassword("bhatta");
```

```
            // Set the query to be executed
            rowSet.setCommand("SELECT * FROM tbl_students");

           // Execute the query
            rowSet.execute();
System.out.println("Id \tFirst Name \tMiddle Name \tLast Name \tEmail
\t\t\t\tPassword");

            // Process the result set
            while (rowSet.next()) {
                System.out.println(
                        rowSet.getInt("id") + "\t"
                        + rowSet.getString("firstName") + "\t\t"
                        + rowSet.getString("middleName") + "\t\t"
                        + rowSet.getString("lastName") + "\t\t"
                        + rowSet.getString("email") + "\t\t"
                        + rowSet.getString("password") + "\t"
                );
            }
           // Closing the RowSet
            rowSet.close();

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

```
run:
Id      First Name      Middle Name     Last Name       Email                           Password
1       Deepak                          Bhatta          dpkbhatta2051@gmail.com                 csit_123
3       Billi                           Pant            ghugu_billa@gmail.com           123
4       Harish                          Bhatta          harry2001@gmail.com             123456
5       Dinesh          Chandra         Bhatta          dns_bhatta@gmail.com            12345
6       Pushpa                          Bhatta          pusu07@gmail.com                2145
7       Hemanti                         Bhatta          hemu20@gmail.com                147
BUILD SUCCESSFUL (total time: 0 seconds)
```

CachedRowSet:

- Uses a **CachedRowSet**, which operates in a disconnected mode, allowing the data to be cached locally.
- You set the connection details and execute the query using **cachedRowSet.setCommand()** and **execute()**.
- You can modify the data locally and then synchronize it back to the database with **acceptChanges()**.
- The connection does not need to stay open while reading or modifying the data, making it useful in applications with intermittent connections.

Example:

```
package QueryExecution;

import com.mysql.cj.jdbc.MysqlDataSource;
import java.sql.Connection;
import javax.sql.rowset.CachedRowSet;
import javax.sql.rowset.RowSetProvider;
import java.sql.SQLException;

public class CachedRowSetExample {

    public static void main(String[] args) {
        Connection conn = null;
        try {
           // Manually create a database connection
            MysqlDataSource dataSource = new MysqlDataSource();
            dataSource.setUrl("jdbc:mysql://localhost:3306/kmc_college");
            dataSource.setUser("root");
            dataSource.setPassword("bhatta");
```

```
            // Get a connection from the DataSource
             conn = (Connection) dataSource.getConnection();

            // Disable auto-commit for this connection
             conn.setAutoCommit(false);
            // Create CachedRowSet instance using RowSetProvider
             CachedRowSet cachedRowSet =
RowSetProvider.newFactory().createCachedRowSet();

            // Set connection details
             cachedRowSet.setUrl("jdbc:mysql://localhost:3306/kmc_college");
             cachedRowSet.setUsername("root");
             cachedRowSet.setPassword("bhatta");

            // Set the query to retrieve data
             cachedRowSet.setCommand("SELECT * FROM tbl_students");

            // Execute the query and fetch data into CachedRowSet
             cachedRowSet.execute();
             System.out.println("Id\tFirst Name\tMiddle Name\tLast
Name\tEmail\tPassword");

            // Process the CachedRowSet data (in a disconnected mode)
             while (cachedRowSet.next()) {
                 System.out.println(
                         cachedRowSet.getInt("id") + "\t"
                         + cachedRowSet.getString("firstName") + "\t\t"
                         + cachedRowSet.getString("middleName") + "\t\t"
                         + cachedRowSet.getString("lastName") + "\t\t"
                         + cachedRowSet.getString("email") + "\t\t"
                         + cachedRowSet.getString("password") + "\t"
                 );
             }
         // Modifying data in CachedRowSet
          cachedRowSet.absolute(1); // Go to the first row
          cachedRowSet.updateString("lastName", "Bhatta");
```

```
            cachedRowSet.updateRow(); // Save changes

            // Reconnect and synchronize changes to the database
            cachedRowSet.acceptChanges(conn); // Pass the connection for updating

              // Commit changes manually
               conn.commit();
           } catch (SQLException e) {
               e.printStackTrace();
           }
       }
}
```


Transactions:

JDBC allows transactions, which let you execute multiple statements as a single unit of work. You can **COMMIT** or **ROLLBACK** based on success or failure.

Example:

```
try {
    conn.setAutoCommit(false); // Disable auto-commit
    // Execute first update
    String query1 = "UPDATE tbl_students SET email = dpkbhatta2051.in@gmail.com' WHERE id = 1";
    Statement stmt = conn.createStatement();
    stmt.executeUpdate(query1);

    // Execute second update
    String query2 = "UPDATE tbl_students SET email = billi@gmail.com' WHERE id = 2";
    stmt.executeUpdate(query2);
    // Commit transaction
    conn.commit();
} catch (SQLException e) {
    conn.rollback(); // Rollback if any exception occurs
    e.printStackTrace();
} finally {
```

```
    conn.setAutoCommit(true); // Restore auto-commit
}
```

THE END

Unit 5 4 hrs.

Network Programming

Specific Objectives

- Understand concepts of ports, IP address, and Protocols.
- Implement TCP/UDP servers and clients.
- Perform different operations with URLs.

5.1. Networking Basics: Transmission control Protocol (TCP), User Datagram Protocol (UDP), Ports, IP Address Network Classes in JDK.

5.2. Working with URLS: Connecting to URLS, Reading Directly from URLS, Inet Address Class.

5.3. Sockets: TCP Sockets, UDP Sockets, Serving Multiple Clients, Half Close, Interruptible Sockets, Sending Email.

5.1. Networking Basics: Transmission control Protocol (TCP), User Datagram Protocol (UDP), Ports, IP Address Network Classes in JDK.

Networking Basics:

Networking is the practice of connecting computers and other devices to share resources, exchange data, and communicate. The Internet, one of the most well-known networks, is a global system of interconnected computer networks. Networking is crucial for many applications such as file sharing, email, and web browsing, and it plays a foundational role in distributed systems and cloud computing.

Basic Components of a Network:

- **Nodes**: Any device connected to a network (e.g., computers, servers, routers).
- **Switch**: A device that connects multiple devices within a LAN and directs data to the correct destination.
- **Router**: A device that connects different networks, such as a LAN to the Internet.
- **Hub**: A basic device that broadcasts data to all devices in a network, without considering the destination.
- **Modem**: Converts data between digital form (used by computers) and analog form (used by telephone lines or cable systems).

Networking Protocols:

Protocols are rules that govern communication between devices on a network. They ensure the reliable and secure transfer of data.

- **Transmission Control Protocol/Internet Protocol (TCP/IP)**: The fundamental suite of protocols that governs the Internet and most modern networks. TCP ensures reliable transmission, while IP handles addressing and routing.
- **User Datagram Protocol (UDP)**: A faster but less reliable protocol than TCP. It's often used in real-time applications like video streaming or online gaming.
- **Hypertext Transfer Protocol (HTTP/HTTPS)**: The protocol used for web browsing. HTTPS is the secure version of HTTP.
- **File Transfer Protocol (FTP)**: Used to transfer files over a network.
- **Simple Mail Transfer Protocol (SMTP)**: Used to send emails.

Transmission Control Protocol (TCP):

- **TCP** is a connection-oriented protocol that ensures reliable, ordered, and error-checked delivery of data between applications.
- It establishes a connection before transmitting data, making sure packets arrive in the correct order.

In Java, the **java.net** package provides classes to handle TCP communication:

- **ServerSocket**: Used by the server to listen for incoming connections.
- **Socket**: Used by both client and server to send and receive data once the connection is established.

Example of a simple TCP server and client in Java:

Server:

```
ServerSocket serverSocket = new ServerSocket(8080); // Server listening on port 8080
Socket socket = serverSocket.accept(); // Accept incoming connection
```

Client:

```
Socket socket = new Socket("localhost", 8080); // Connect to the server on port 8080
```

Implementing TCP/IP based Server and Client

TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to-point, stream-based connections between hosts on the Internet. There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients.

The following steps occur when establishing a TCP connection between two computers using sockets:

1. The server initializes a **ServerSocket** object, denoting in which port number communication is to start.
2. The server invokes the **accept()** method of the **ServerSocket** class. This method waits until a client connects to the server on the given port.
3. After the server is waiting, a client instantiates a Socket object, specifying the **server name** and **port number** to connect to.
4. The constructor of the Socket class attempts to connect the client to the specified server and port number. If communication is established, the client now has a Socket object capable of communicating with the server.
5. On the server side, the **accept()** method returns a reference to a new socket on the server that is connected to the client's socket.

After the connections are established, communication can occur using **I/O streams**. Each socket has both an **OutputStream** and an **InputStream**.

The client's OutputStream is connected to the server's InputStream, and the client's InputStream is connected to the server's OutputStream.

TCP is a two-way communication protocol, so data can be sent across both streams at the same time. There are following useful classes providing complete set of methods to implement sockets.

TCP/IP Server Socket

The TCP/IP server socket is used to communicate with the client machine. The steps to get a server up and running are shown below:

1. Create a server socket and name the socket
2. Wait for a request to connect, a new client socket is created here
3. Read data sent from client and send data back to client
4. Close client socket
5. Loop back if not told to exit
6. Close server socket

TCP Server Program in Java:

The server program creates a **ServerSocket** to listen on a specified port, and it accepts connections from clients. Once a connection is accepted, the server can read and respond to the client's messages.

Example:

```
package networkprogramming;
import java.io.*;
import java.net.*;
public class TCPServerExample {

    public static void main(String[] args) {
        try {
           // Create a server socket that listens on port 8080
            ServerSocket serverSocket = new ServerSocket(8080);
            System.out.println("Server is listening on port 8080");

           // Wait for a client connection
            Socket socket = serverSocket.accept();
            System.out.println("Client connected");

           // Create input and output streams to communicate with the client
            InputStream input = socket.getInputStream();
```

```
            BufferedReader reader = new BufferedReader(new
InputStreamReader(input));

            OutputStream output = socket.getOutputStream();
            PrintWriter writer = new PrintWriter(output, true);

           // Read the client's message
            String clientMessage = reader.readLine();
            System.out.println("Received from client: " + clientMessage);

            // Send a response back to the client
            writer.println("Hello, client! Message received.");

           // Close the connection
            socket.close();
            serverSocket.close();
        } catch (IOException ex) {
            System.out.println("Server error: " + ex.getMessage());
            ex.printStackTrace();
        }
    }
}
```

```
run:
Server is listening on port 8080
```

```
H:\One Drive KMC\OneDrive - Kailali Multiple Campus\
\Run from CMD>javac TCPServerExample.java

H:\One Drive KMC\OneDrive - Kailali Multiple Campus\
\Run from CMD>java TCPServerExample
Server is listening on port 8080
Client connected
Received from client: Hello, server!
```

TCP/IP Client socket

The TCP/IP client socket is used to communicate with the server. Following are the steps client needs to take in order to communicate with the server.

1. Create a socket with the server IP address
2. Connect to the server, this step also names the socket
3. Send data to the server
4. Read data returned (echoed) back from the server
5. Close the socket

TCP Client Program in Java:

The client program connects to the server via a Socket, sends a message, and then receives a response from the server.

Example:

```
package networkprogramming;

import java.io.*;

import java.net.*;

public class TCPClientExample {

    public static void main(String[] args) {

        try {

            // Connect to the server running on localhost and port 8080

             Socket socket = new Socket("localhost", 8080);

             System.out.println("Connected to the server");

            // Create output stream to send data to the server

             OutputStream output = socket.getOutputStream();

             PrintWriter writer = new PrintWriter(output, true);

            // Send a message to the server

             writer.println("Hello, server!");
```

```
            // Create input stream to receive the server's response
            InputStream input = socket.getInputStream();
            BufferedReader reader = new BufferedReader(new
InputStreamReader(input));

            // Read the server's response
            String serverMessage = reader.readLine();
            System.out.println("Received from server: " + serverMessage);

            // Close the connection
            socket.close();
        } catch (IOException ex) {
            System.out.println("Client error: " + ex.getMessage());
            ex.printStackTrace();
        }
    }
}
```

```
H:\One Drive KMC\OneDrive - Kailali Multiple Campus\Docu
\Run from CMD> javac TCPClientExample.java

H:\One Drive KMC\OneDrive - Kailali Multiple Campus\Docu
\Run from CMD>java TCPClientExample
Connected to the server
Received from server: Hello, client! Message received.
```


To Run the Program:

1. Compile and run the **TCPServerExaple** first.

   ```
   javac TCPServerExample.java
   java TCPServerExample
   ```

2. Compile and run the **TCPClient** in a separate terminal or environment.

   ```
   javac TCPClientExample.java
   java TCPClientExample
   ```

You will see the client and server exchanging messages.

User Datagram Protocol (UDP):

- **UDP** is a connectionless protocol that sends datagrams without ensuring their arrival or order. It's faster but less reliable than TCP.
- Suitable for scenarios where speed is prioritized over reliability (e.g., live video streaming).

In Java, the **java.net** package provides classes to handle UDP communication:

- **DatagramSocket**: Used for both sending and receiving datagram packets.
- **DatagramPacket**: Represents a datagram packet that can be sent or received.

UDP Server Program in Java:

The server listens on a specific port using **DatagramSocket** and waits for incoming datagram packets.

Example:

```
import java.net.*;
public class UDPServer {
    public static void main(String[] args) {
        try {
           // Create a datagram socket to listen on port 8080
            DatagramSocket socket = new DatagramSocket(8080);
            System.out.println("UDP Server is running on port 8080");

           // Buffer to store incoming data
            byte[] buffer = new byte[1024];

           // Prepare a datagram packet to receive data
            DatagramPacket packet = new DatagramPacket(buffer,
buffer.length);

           // Wait for a packet from the client
            socket.receive(packet);
            System.out.println("Packet received from client");
```

```
        // Extract data from the received packet
            String clientMessage = new String(packet.getData(), 0,
packet.getLength());
            System.out.println("Received from client: " + clientMessage);

           // Send a response to the client
            String responseMessage = "Hello, client! Message received.";
            byte[] responseData = responseMessage.getBytes();

           // Get client's address and port from the received packet
            InetAddress clientAddress = packet.getAddress();
            int clientPort = packet.getPort();

           // Send the response packet back to the client
            DatagramPacket responsePacket = new
DatagramPacket(responseData, responseData.length, clientAddress,
clientPort);
            socket.send(responsePacket);

           // Close the socket
            socket.close();
        } catch (Exception ex) {
            System.out.println("Server error: " + ex.getMessage());
            ex.printStackTrace();
        }
    }
}
```

```
H:\One Drive KMC\OneDrive - Kailali Multiple
\UDP CMD>javac UDPServerExample.java

H:\One Drive KMC\OneDrive - Kailali Multiple
\UDP CMD>java UDPServerExample
UDP Server is running on port 8080
Packet received from client
Received from client: Hello, server!
```

UDP Client Program in Java:

The client sends a datagram packet to the server using **DatagramSocket**, and then it waits for a response.

Example:

```
package networkprogramming;
import java.net.*;
public class UDPClientExample {

    public static void main(String[] args) {
        try {
            // Create a datagram socket to send/receive data
            DatagramSocket socket = new DatagramSocket();

            // Prepare the data to be sent (in bytes)
            String message = "Hello, server!";
            byte[] buffer = message.getBytes();

    // Create a datagram packet to send data to the server (localhost, port 8080)
            InetAddress serverAddress = InetAddress.getByName("localhost");
            DatagramPacket packet = new DatagramPacket(buffer, buffer.length,
serverAddress, 8080);

            // Send the packet to the server
            socket.send(packet);
            System.out.println("Message sent to server");

            // Buffer to store incoming data (response from the server)
            byte[] responseBuffer = new byte[1024];

            // Prepare a datagram packet to receive the response
            DatagramPacket responsePacket = new
DatagramPacket(responseBuffer, responseBuffer.length);

            // Receive the server's response
            socket.receive(responsePacket);
```

```
            // Extract and display the server's response
            String serverMessage = new String(responsePacket.getData(), 0,
responsePacket.getLength());
            System.out.println("Received from server: " + serverMessage);

            // Close the socket
            socket.close();
        } catch (Exception ex) {
            System.out.println("Client error: " + ex.getMessage());
            ex.printStackTrace();
        }
    }
}
```

```
H:\One Drive KMC\OneDrive - Kailali Multiple Campus\Documents
\UDP CMD>javac UDPClientExample.java

H:\One Drive KMC\OneDrive - Kailali Multiple Campus\Documents
\UDP CMD>java UDPClientExample
Message sent to server
Received from server: Hello, client! Message received.
```

How It Works:

1. The **server** listens on port **8080** using a **DatagramSocket**.
2. The **client** creates a **DatagramSocket** and sends a datagram packet containing a message ("Hello, server!") to the server.
3. The server receives the datagram packet, extracts the message, and prints it.
4. The server sends a response packet ("Hello, client! Message received.") back to the client.
5. The client receives the response and prints it.

Ports:

- **Ports** are logical endpoints that help identify specific services or applications running on a device within a network.
- Clients connect to servers via objects known as ports.
- A port serves as a channel through which several clients can exchange data with the same server or with different servers.
- Ports are usually specified by numbers. Some ports are dedicated to special servers or tasks.
- For example, many computers reserve port number 13 for the day/time server, which allows clients to obtain the date and time.
- Port number 80 is reserved for a Web server, and so forth.
- Most computers also have hundreds or even thousands of free ports available for use by network applications.
- Common ports include **80** (HTTP), **443** (HTTPS), **21** (FTP), and **25** (SMTP).

In Java, when you create a ServerSocket or DatagramSocket, you bind it to a specific port. This port will listen for incoming traffic.

Example:

```
ServerSocket serverSocket = new ServerSocket(8080); // Binds to port 8080
```

IP Address Network Classes in JDK:

IP Address:

- An **IP address** is a unique address used to identify devices on a network.
- Java provides the **InetAddress class** to work with IP addresses.

Example of retrieving the local IP address:

```
InetAddress localHost = InetAddress.getLocalHost();
System.out.println("Local IP address: " + localHost.getHostAddress());
```

Network Classes:

IP addresses are categorized into classes based on their range:

- **Class A**: Large networks (range: 1.0.0.0 to 126.255.255.255)
- **Class B**: Medium-sized networks (range: 128.0.0.0 to 191.255.255.255)
- **Class C**: Small networks (range: 192.0.0.0 to 223.255.255.255)
- **Class D**: Multicast (range: 224.0.0.0 to 239.255.255.255)
- **Class E**: Experimental (range: 240.0.0.0 to 255.255.255.255)

In Java, the **InetAddress** class handles both IPv4 and IPv6 addresses.

Example: Display the IP address of current host:

```
package networkprogramming;

import java.net.*;
public class HostInfo {

    public static void main(String args[]) {
        try {
            InetAddress ipaddress = InetAddress.getLocalHost();
            System.out.println("IP address:\n" + ipaddress);
        } catch (Exception e) {
            System.out.println("Unknown Host");
        }
    }
}
```

```
run:
IP address:
BHATTA_KAJI/192.168.1.67
BUILD SUCCESSFUL (total time: 0 seconds)
```

Example: Display the IP address of Remote host:

```
package networkprogramming;
import java.net.*;
public class GetRemoteIPAddress {

    public static void main(String[] args) {
        try {
           // Specify the domain name or the hostname of the remote server
            String host = "www.google.com";

            // Get the InetAddress object for the specified host
            InetAddress inetAddress = InetAddress.getByName(host);

           // Get the IP address as a string
            String ipAddress = inetAddress.getHostAddress();

           // Display the hostname and IP address
            System.out.println("Hostname: " + host);
            System.out.println("IP Address: " + ipAddress);
        } catch (UnknownHostException e) {
            System.out.println("Host not found: " + e.getMessage());
        }
    }
}
```



```
run:
Hostname: www.google.com
IP Address: 142.250.196.68
BUILD SUCCESSFUL (total time: 0 seconds)
```

5.2. Working with URLs: Connecting to URLS, Reading Directly from URLs, Inet Address Class.

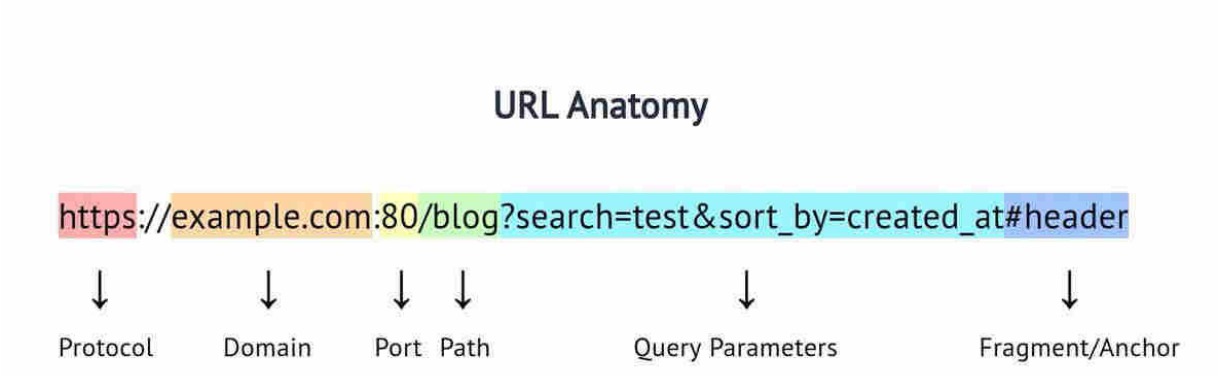
Working with URLS:

Working with URLs in Java involves classes such as **URL**, **URLConnection**, and **InetAddress** from the **java.net** package. These classes allow you to connect to web resources, read data from them, and work with IP addresses.

The URL allows uniquely identifying or addressing information on the Internet. Every browser uses them to identify information on the Web. Within Java's network class library, the URL class provides a simple, concise API to access information across the Internet using URLs. All URLs share the same basic format, although some variation is allowed.

A URL specification is based on four components.

1. The first is the protocol to use, separated from the rest of the locator by a colon (:). Common protocols are HTTP, FTP, gopher, etc.
2. The second component is the host name or IP address of the host to use; this is delimited on the left by double slashes (//) and on the right by a slash (/) or optionally a colon (:).
3. The third component, the port number, is an optional parameter, delimited on the left from the host name by a colon (:) and on the right by a slash (/). It defaults to port 80, the predefined HTTP port.
4. The fourth part is the actual file path.



Example showing the use of URL:

```
package networkprogramming;
import java.net.*;
public class URLExample {

    public static void main(String args[]) {
        URL url;
        try {
            url = new URL("http://localhost:8080/project");
            System.out.println("Protocol: " + url.getProtocol());
            System.out.println("Port: " + url.getPort());
            System.out.println("Host: " + url.getHost());
            System.out.println("File: " + url.getFile());

        } catch (MalformedURLException ex) {
            System.out.println("ERROR: Something went wrong! " + ex);
        }
    }
}
```

```
run:
Protocol: http
Port: 8080
Host: localhost
File: /project
BUILD SUCCESSFUL (total time: 0 seconds)
```

Connecting to URLs:

The URL class in Java represents a Uniform Resource Locator, which points to a resource on the web. You can use this class to connect to websites and interact with web resources.

URLConnection is a general-purpose class for accessing the attributes of a remote resource. Once we make a connection to a remote server, we can use **URLConnection** to inspect the properties of the remote object before actually transporting it locally. These attributes are exposed by the **HTTP** protocol specification and only make sense for **URL** objects that are using the **HTTP** protocol.

Example: Connecting to a URL

```
package networkprogramming;
import java.io.IOException;
import java.net.*;
public class URLConnectionExample {

    public static void main(String[] args) {
        try {
           // Create a URL object
            URL url = new URL("https://www.google.com");

           // Open a connection to the URL
            URLConnection urlConnection = url.openConnection();

           // Display some information about the connection
            System.out.println("URL: " + url);
     System.out.println("Content Type: " + urlConnection.getContentType());
 System.out.println("Content Length: " + urlConnection.getContentLength());
 System.out.println("Last Modified: " + urlConnection.getLastModified());
        } catch (IOException ex) {
            System.out.println("ERROR: Something went wrong! " + ex);
        }
    }
}
```

```
run:
URL: https://www.google.com
Content Type: text/html; charset=ISO-8859-1
Content Length: -1
Last Modified: 0
BUILD SUCCESSFUL (total time: 0 seconds)
```

Reading Directly from URLs:

Once connected to a URL, you can read its content (e.g., the HTML of a web page). You typically use input streams to read data.

Example: Reading Data from a URL:

```
package networkprogramming;
import java.io.*;
import java.net.*;
public class ReadingFromURLExample {

    public static void main(String[] args) {
        try {
            // Create a URL object for the target webpage
             URL url = new URL("https://www.example.com");
            // Open a BufferedReader to read text from the URL's input stream
             BufferedReader reader = new BufferedReader(new
InputStreamReader(url.openStream()));

            // Read and display the content line by line
             String line;
             while ((line = reader.readLine()) != null) {
                 System.out.println(line);
             }
            // Close the reader
             reader.close();
        } catch (IOException ex) {
             System.out.println("ERROR: Something went wrong! " + ex);
        }
    }
}
```

```
run:
<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <style type="text/css">
    body {
        background-color: #f0f0f2;
        margin: 0;
        padding: 0;
        font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI"

    }
```

Inet Address Class:

The **InetAddress** class provides methods to work with IP addresses. You can resolve domain names into IP addresses, check the local machine's address, and more.

Example: Working with InetAddress:

```
package networkprogramming;
import java.net.*;
public class InetAddressExample {

    public static void main(String[] args) {
        try {
            // Get the InetAddress object for a remote host (Google)
            InetAddress google = InetAddress.getByName("www.google.com");

            // Display information about the remote host
            System.out.println("Google Hostname: " + google.getHostName());
            System.out.println("Google IP Address: " + google.getHostAddress());

            // Get the InetAddress object for the local host
            InetAddress localHost = InetAddress.getLocalHost();

            // Display information about the local machine
            System.out.println("Local Hostname: " + localHost.getHostName());
            System.out.println("Local IP Address: " + localHost.getHostAddress());
        } catch (UnknownHostException ex) {
            System.out.println("ERROR: Something went wrong! " + ex);
        }
    }
}
```

```
run:
Google Hostname: www.google.com
Google IP Address: 142.250.196.68
Local Hostname: BHATTA_KAJI
Local IP Address: 192.168.1.67
BUILD SUCCESSFUL (total time: 4 seconds)
```

5.3. Sockets: TCP Sockets, UDP Sockets, Serving Multiple Clients, Half Close, Interruptible Sockets, Sending Email.

Sockets:

Socket is an object which establishes a communication link between two ports. A socket is an endpoint of a two-way communication link between two programs running on the network. Socket is bound to a port number so that the TCP layer can identify the application that Data is destined to be sent.

Sockets are the foundation for communication over networks. They allow programs to exchange data across a network, such as the internet. In Java, sockets can be used to create network-based applications such as web servers, email clients, etc. Let's cover various socket concepts like **TCP Sockets**, **UDP Sockets**, serving multiple clients, half-closing sockets, and more.

The working mechanism of the socket is as shown in the figure below:

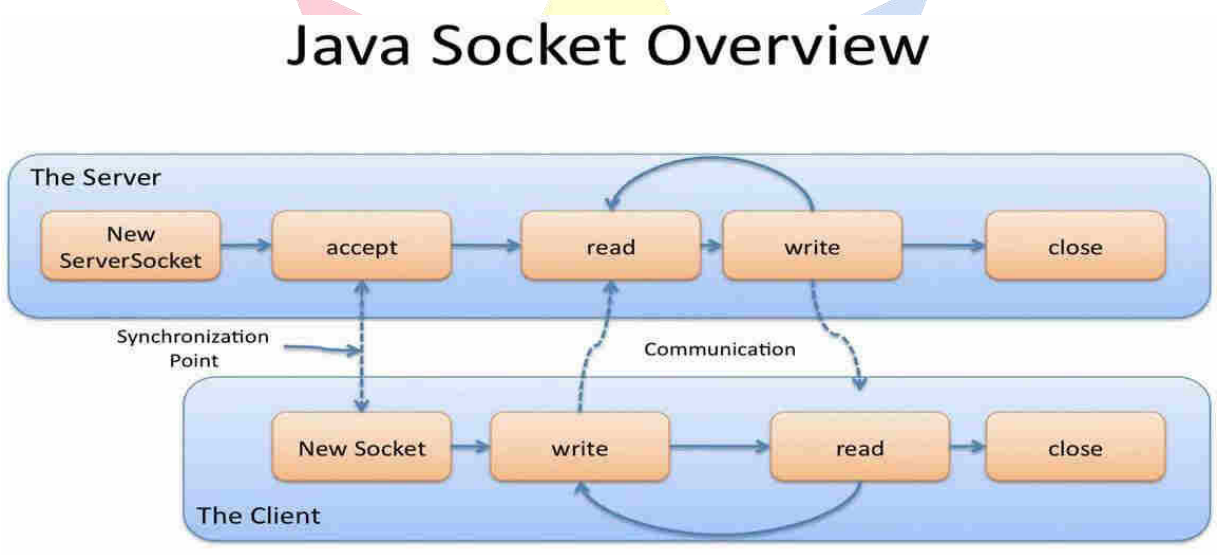


Fig: Overview of Java socket

TCP Sockets: Provide reliable, connection-oriented communication.

UDP Sockets: Provide fast, connectionless communication.

Please refer Topic 5.1 Page No. 2

Extra Dose:

Sometimes an error occurred regarding port (*address that is already in use*). Sometimes forcefully active port needs to be removed from network. For this you need to find the **PID ~ Process ID**.

Goto CMD ➔ **`netstat -a -o -n`**

```
C:\Users\dpkbh>netstat -a -o -n

Active Connections

  Proto  Local Address          Foreign Address        State           PID
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING       1696
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING       4
  TCP    0.0.0.0:1462           0.0.0.0:0              LISTENING       4
  TCP    0.0.0.0:3261           0.0.0.0:0              LISTENING       7800
  TCP    0.0.0.0:3306           0.0.0.0:0              LISTENING       9560
  TCP    0.0.0.0:5040           0.0.0.0:0              LISTENING       11916
  TCP    0.0.0.0:7070           0.0.0.0:0              LISTENING       6548
  TCP    0.0.0.0:7680           0.0.0.0:0              LISTENING       16700
  TCP    0.0.0.0:8080           0.0.0.0:0              LISTENING       26368
  TCP    0.0.0.0:8090           0.0.0.0:0              LISTENING       20760
  TCP    0.0.0.0:9095           0.0.0.0:0              LISTENING       4
```

Goto CMD ➔ **`tasklist`**

```
C:\Users\dpkbh>tasklist

Image Name                     PID Session Name        Session#    Mem Usage
========================= ======== ================ =========== ============
System Idle Process              0 Services                   0          8 K
System                           4 Services                   0      8,348 K
Secure System                  140 Services                   0     48,804 K
Registry                       176 Services                   0     48,400 K
smss.exe                       636 Services                   0      1,104 K
csrss.exe                     1092 Services                   0      5,308 K
wininit.exe                   1236 Services                   0      6,160 K
csrss.exe                     1244 Console                    1      6,128 K
services.exe                  1320 Services                   0     16,964 K
```

Goto CMD ➔ **`taskkill /f /pid 26368`**

```
C:\Users\dpkbh>taskkill /f /pid 26368
SUCCESS: The process with PID 26368 has been terminated.
```

Serving Multiple Clients:

To serve multiple clients simultaneously, you can create a multi-threaded server where each client connection is handled in its own thread.

Example: Multi-threaded TCP Server

```
import java.io.*;
import java.net.*;
public class MultiThreadedTCPServer {
    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(8080);
            System.out.println("Server is listening on port 8080");

            while (true) {
                Socket clientSocket = serverSocket.accept();
                System.out.println("New client connected");

               // Create a new thread for each client
                new ClientHandler(clientSocket).start();
            }
        } catch (IOException ex) {
            System.out.println("ERROR: Something went wrong! " + ex);
        }
    }
}

class ClientHandler extends Thread {

    private Socket clientSocket;

    public ClientHandler(Socket socket) {
        this.clientSocket = socket;
    }
```

```
    @Override
    public void run() {
        try {
            BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
            PrintWriter out = new
PrintWriter(clientSocket.getOutputStream(), true);

            // Handle client communication
            String message;
            while ((message = in.readLine()) != null) {
                System.out.println("Received: " + message);
                out.println("Echo: " + message);
            }
            clientSocket.close();
        } catch (IOException ex) {
            System.out.println("ERROR: Something went wrong! " + ex);
        }
    }
}
```

```
H:\One Drive KMC\OneDrive - Kailali Multiple Campus
\Multiple Clients>javac MultiThreadedTCPServer.java

H:\One Drive KMC\OneDrive - Kailali Multiple Campus
\Multiple Clients>java MultiThreadedTCPServer
Server is listening on port 8080
New client connected
Received: Client-1: Hello, server!
New client connected
Received: Client-2: Hello, server!
New client connected
Received: Client-3: Hello, server!
```

```
H:\One Drive KMC\OneDrive - Kailali Multiple Campus\Docu
\Multiple Clients>javac TCPClient1.java

H:\One Drive KMC\OneDrive - Kailali Multiple Campus\Docu
\Multiple Clients>java TCPClient1
Connected to the server
Received from server: Echo: Client-1: Hello, server!

H:\One Drive KMC\OneDrive - Kailali Multiple Campus\Docu
\Multiple Clients>javac TCPClient2.java

H:\One Drive KMC\OneDrive - Kailali Multiple Campus\Docu
\Multiple Clients>java TCPClient2
Connected to the server
Received from server: Echo: Client-2: Hello, server!

H:\One Drive KMC\OneDrive - Kailali Multiple Campus\Docu
\Multiple Clients>javac TCPClient3.java

H:\One Drive KMC\OneDrive - Kailali Multiple Campus\Docu
\Multiple Clients>java TCPClient3
Connected to the server
Received from server: Echo: Client-3: Hello, server!
```

Save below code in TCPClient1.java, TCPClient2.java, TCPClient3.java

```
import java.io.*;
import java.net.*;
public class TCPClient1 {
    public static void main(String[] args) {
        try {
            // Connect to the server running on localhost and port 8080
             Socket socket = new Socket("localhost", 8080);
             System.out.println("Connected to the server");

            // Create output stream to send data to the server
             OutputStream output = socket.getOutputStream();
             PrintWriter writer = new PrintWriter(output, true);

            // Send a message to the server
             writer.println("Client-1: Hello, server!");

            // Create input stream to receive the server's response
             InputStream input = socket.getInputStream();
             BufferedReader reader = new BufferedReader(new
InputStreamReader(input));

            // Read the server's response
             String serverMessage = reader.readLine();
             System.out.println("Received from server: " + serverMessage);

            // Close the connection
             socket.close();
        } catch (IOException ex) {
             System.out.println("Client - 1 error: " + ex.getMessage());
             ex.printStackTrace();
        }
    }
}
```

Half Close:

In TCP, the "**half-close**" mechanism allows one side to close its output stream while keeping its input stream open. This means one side can still receive data even after stopping sending data.

```
Socket socket = new Socket("localhost", 8080);
// Half-close: Close the output stream but keep receiving data
socket.shutdownOutput();
```

This will close the output side of the socket, meaning the client won't send data anymore, but it can still receive data from the server.

Interruptible Sockets:

Interruptible sockets allow you to handle a socket's blocking operations (e.g., waiting for data) in such a way that the thread can be interrupted. You can use **Thread.interrupt()** to cancel a blocking operation like **accept()** or **read()**.

Sending Email:

Sending emails can be done using Java's **JavaMail API**. Here is a basic example of sending an email via an **SMTP** server. Go to this website and signup https://mailtrap.io/

You need to download the **JavaMail API**, and Activation**-1.1.1** jar Files.

Example:

```
package networkprogramming;
import javax.mail.*;
import javax.mail.internet.*;
import java.util.Properties;

public class EmailSender {

    public static void main(String[] args) {
        // SMTP server information
        String host = "sandbox.smtp.mailtrap.io";
        String port = "2525";
        String ssl = "no";
        String tls = "yes";
        String username = "4d4043f4b4e9a2";
        String password = "b1d327121890c8";

        // Email details
        String toAddress = "dpkbhatta2051@gmail.com";
        String subject = "Test Email";
        String message = "This is a test email from Java!";

        Properties properties = new Properties();
        properties.put("mail.smtp.host", host);
        properties.put("mail.smtp.port", port);
        properties.put("mail.smtp.auth", "true");
        properties.put("mail.smtp.starttls.enable", "true");
        properties.put("mail.smtp.starttls.enable", tls);  // Enable TLS
        properties.put("mail.smtp.ssl.enable", ssl);       // SSL not enabled
```

```
        Session session = Session.getInstance(properties, new
Authenticator() {
            protected PasswordAuthentication getPasswordAuthentication() {
                return new PasswordAuthentication(username, password);
            }
        });

        try {
            Message mimeMessage = new MimeMessage(session);
            mimeMessage.setFrom(new InternetAddress(username));
            mimeMessage.setRecipients(Message.RecipientType.TO,
InternetAddress.parse(toAddress));
            mimeMessage.setSubject(subject);

           // Set the actual message
            mimeMessage.setText(message);

           // Send the email
            Transport.send(mimeMessage);

            System.out.println("Email sent successfully!");
        } catch (MessagingException ex) {
            System.out.println("ERROR: Something went wrong! " + ex);
        }
    }
}
```

THE END

Unit 6 5 hrs.

Java Beans

Specific Objectives

- Practice the creation, modification, and deletion of JAR files.
- Demonstrate the use of bean components.
- Write programs to create Java Beans.

6.1. Introduction: Creating, Updating and Reading From JAR Files, Java Beans, Advantages of Java Beans, Class vs Beans, BDK and Bean Box.

6.2. Java Bean: Creating a Java Bean, Creating a Bean Manifest File, Creating a Bean JAR File, Using a New Bean, Adding Controls to Beans, Giving a Bean Properties, Creating Bound Properties, Giving a Bean Methods, Giving a Bean an Icon.

6.1. Introduction: Creating, Updating and Reading From JAR Files, Java Beans, Advantages of Java Beans, Class vs Beans, BDK and Bean Box.

Introduction to JAR:

So, what exactly is JAR? Simply speaking, this is a format for archiving data. This is similar to you using **WinRAR** or **WinZIP**.

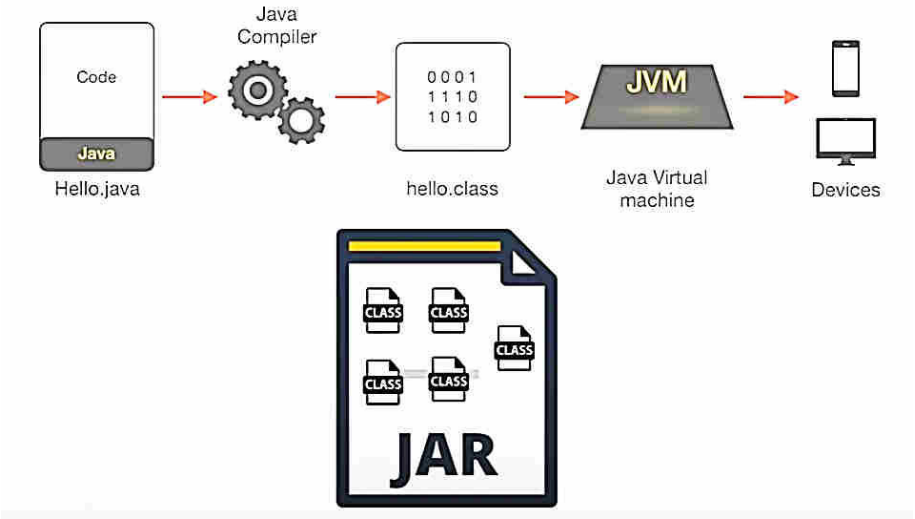
One major difference is that JAR files don't ask you to pay for a premium version.

Simply speaking, a single JAR file can contain compressed class files, interfaces, and even audio files.

A **JAR (Java ARchive)** file is a package file format used to aggregate multiple Java class files, along with associated resources (text, images, etc.), into a single file for distribution. It is essentially a ZIP file but includes a manifest file that contains metadata.

Let us look at some important points about JAR files in Java:

- This type of archiving is cross-platform in nature.
- This can handle audio and image files as well.
- There is backward compatibility. Without it being backward compatible, developers would have to rewrite existing applets.
- This is written in Java itself.
- This is a standard of bundling up Java applets.



Creating JAR Files:

You can create a JAR file using the jar command in the terminal. For example:

```
jar cf MyJarFile.jar .
```

- **cf:** Create a new JAR file.
- **MyJarFile.jar:** The name of the JAR file.

C ➔ Create

F ➔ Specify the file name

T ➔ Gives table content

Updating and Reading From JAR Files:

Updating a JAR File: To add or update files in an existing JAR file:

```
jar uf MyJarFile.jar .
```

- **uf:** Update the existing JAR file.

Reading from a JAR File: You can read the contents of a JAR file using:

```
jar tf MyJarFile.jar
```

- **tf:** List all the files in the JAR.

Extracting from a JAR File: You can read the contents of a JAR file using:

```
jar xf MyJarFile.jar
```

- **xf:** Extract all the files from the JAR.

1. **Compile your Java file:**

   ```
   javac SimpleFormTextFields.java
   ```

2. **Create a manifest file and your jar file:**

   ```
   ⇨ echo Main-Class: SimpleFormTextFields >manifest.txt
   ⇨ jar cfm SimpleForm.jar manifest.txt *.class
   ```

3. **Test your jar:**

   ```
   SimpleForm.jar
   ```

   or

   ```
   java -jar SimpleForm.jar
   ```

Java Beans:

Java Beans are reusable software components that follow certain conventions:

- Must have a public no-argument constructor.
- Properties must be accessible using getter and setter methods.
- Should implement the Serializable interface.

Advantages of Java Beans:

- **Reusability**: Java Beans are designed for reuse in a variety of applications.
- **Customization**: Beans can be easily customized by using property editors.
- **Persistence**: Bean properties can be saved and restored easily.
- **Component-Based Architecture**: Beans can be composed into larger applications.
- **Event Handling**: Java Beans use event-handling mechanisms, making them easy to work with in GUI applications.

Class vs Beans:

Class	Bean
Can have any kind of constructor.	Must have a no-argument constructor.
No specific naming conventions.	Follows getter/setter naming convention.
Not necessarily serializable.	Must implement Serializable.
Typically, doesn't follow a standard structure.	Follows a well-defined structure.

BDK and Bean Box:

BDK (Bean Development Kit): BDK is a framework provided by Sun Microsystems for developing and testing Java Beans. It includes tools and documentation to help developers create Java Beans.

BeanBox: It is a graphical environment provided with the BDK that allows developers to visually create and test Java Beans by placing them into the box and connecting them with other components.

6.2. Java Bean: Creating a Java Bean, Creating a Bean Manifest File, Creating a Bean JAR File, Using a New Bean, Adding Controls to Beans, Giving a Bean Properties, Creating Bound Properties, Giving a Bean Methods, Giving a Bean an Icon.

Java Bean:

A **Java Bean** is a reusable software component that follows specific design conventions, like having getter and setter methods for accessing properties, a no-argument constructor, and being serializable.

Below is a step-by-step guide for creating a Java Bean and related tasks:

Creating a Java Bean:

A Java Bean is a simple class that follows these conventions:

- Must have a **no-argument constructor**.
- Must implement **Serializable**.
- Must provide **getter** and **setter** methods for its properties.

Example:

```
package com.knightec.javabeans; //package myjavabeans;
import java.io.Serializable;
public class JavaBeans implements Serializable {

    private String userName;
    private int age;

    public JavaBeans() {
    }

    public String getUserName() {
        return userName;
    }
```

```
    public void setUserName(String userName) {
        this.userName = userName;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public static void main(String[] args) {

        JavaBeans jb = new JavaBeans();

        jb.setUserName("Deepak Bhatta");
        jb.setAge(30);
        System.out.println("User Name : " + jb.getUserName());
        System.out.println("Age : " + jb.getAge());
    }
}
```



```
--- exec:3.1.0:exec (default-cli) @ JavaBeans ---
User Name : Deepak Bhatta
Age : 30
```

Creating a Bean Manifest File:

A **Manifest file** is a special file that contains metadata about the JAR file's contents. For Java Beans, it declares the bean's properties.

Create a file named **manifest.mf** or **manifest.txt** with the following contents:

```
Main-Class: JavaBeans
Manifest-Version: 1.0
Created-By: Deepak Bhatta Kaji
Designation: Asst. Professor - KMC
Name: JavaBeans.class
Java-Bean: True
```

Creating a Bean JAR File:

After compiling your Java Bean, you can package it into a JAR file with the manifest.

Compile the bean:

```
javac JavaBeans.java
```

Package it into a JAR file with the manifest:

```
jar cfm JavaBeans.jar manifest.txt JavaBeans.class
```

This command creates a JAR file named **JavaBeans.jar**, which includes the compiled **JavaBeans.class** file and the **manifest.txt** manifest file.

Run the JAR file:

```
java -jar JavaBeans.jar
```

Using a New Bean:

You can now use the Java Bean in any application. Below is an example of how to use the **JavaBeans** class:

```
package myjavabeans;
public class BeanDemo {

    public static void main(String[] args) {
        JavaBeans jb = new JavaBeans();

        jb.setUserName("Billi");
        System.out.println("User Name : " +jb.getUserName());

        jb.setAge(22);
        System.out.println("Age : " +jb.getAge());
    }
}
```

```
run:
User Name : Billi
Age : 22
BUILD SUCCESSFUL (total time: 0 seconds)
```


Adding Controls to Beans:

You can add controls (like buttons, text fields, etc.) to your Java Beans if you're using them in GUI applications. You can do this by using **JButton**, **JTextField**, or any other Swing components within your Bean.

Giving a Bean Properties:

Bean properties are variables that have associated getter and setter methods. You can add as many properties as needed.

Example:

```
private String color;

public String getColor() {
    return color;
}

public void setColor(String color) {
    this.color = color;
}
```

Creating Bound Properties:

A **bound property** allows you to notify other objects when its value changes. You use **PropertyChangeSupport** to implement this behavior.

To implement a bound property in your application, follow these steps:

1. Import the **java.beans** package. This gives you access to the **PropertyChangeSupport** class.
2. Instantiate a **PropertyChangeSupport** object. This object maintains the property change listener list and fires property change events. You can also make your class a **PropertyChangeSupport** subclass.
3. Implement methods to maintain the property change listener list. Since a **PropertyChangeSupport** subclass implements these methods, you merely wrap calls to the property-change support object's methods.
4. Modify a property's set method to fire a property change event when the property is changed.

Example:

```
package myjavabeans;

import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.io.Serializable;

public class TemperatureBean implements Serializable {

    private int temp;
    private PropertyChangeSupport support;

    public TemperatureBean() {
        support = new PropertyChangeSupport(this);
    }

    public int getTemp() {
        return temp;
    }

    private int newTemp;

    public static final String PROP_NEWTEMP = "newTemp";

    public int getNewTemp() {
        return newTemp;
    }

    public void setNewTemp(int newTemp) {
        int oldNewTemp = this.newTemp;
        this.newTemp = newTemp;
        support.firePropertyChange(PROP_NEWTEMP, oldNewTemp, newTemp);
    }
```

```
    public void addPropertyChangeListener(PropertyChangeListener listener)
{
        support.addPropertyChangeListener(listener);
    }

    public void removePropertyChangeListener(PropertyChangeListener
listener) {
        support.removePropertyChangeListener(listener);
    }

    public static void main(String[] args) {
        TemperatureBean bean = new TemperatureBean();

        bean.addPropertyChangeListener(new PropertyChangeListener() {
            @Override
            public void propertyChange(PropertyChangeEvent evt) {

                System.out.println("Temperature Changed: Old-> " +
evt.getOldValue() + ", New-> " + evt.getNewValue());
            }

        });

        bean.setNewTemp(25);
        bean.setNewTemp(30);
    }
}
```

```
run:
Temperature Changed: Old-> 0, New-> 25
Temperature Changed: Old-> 25, New-> 30
BUILD SUCCESSFUL (total time: 0 seconds)
```

Giving a Bean Methods:

Java Beans can have methods, just like any other class. Methods are typically provided for actions the bean can perform.

Example:

```
public void reset() {
    this.message = "";
    this.number = 0;
}
```


Giving a Bean an Icon:

You can give a Bean an icon to represent it visually when used in a tool like the BeanBox or another IDE. Add the icon file to your JAR and use the following naming convention:

- **MyBeanNameIconColor16.png (16x16 icon)**
- **MyBeanNameIconColor32.png (32x32 icon)**

Example Steps to Add an Icon:

1. Create a 16x16 and/or 32x32 icon image.
2. Name it according to the convention.
3. Place the icon image in the same directory as your Bean class.
4. Add the image to the JAR file when packaging the bean.


```
H:\One Drive KMC\OneDrive - Kailali Multiple Campus\Documents\NetBeansProjects\JavaBeans\src\JB>jar tf JavaBeans.jar
META-INF/
META-INF/MANIFEST.MF
JavaBeans.class
DBK-ICON-32X32.png
```

THE END

Unit 7 — 5 hrs.

Servlets & JSP

Specific Objectives

- Understand Servlet basics and its life cycle.
- Configure web servers and create servlets by using different classes and interfaces.
- Demonstrate the use of session and cookies.
- Understand JSP architecture and compare it with servlets.
- Demonstrate the use of JSP tags by writing sample programs.
- Under exceptions and exception handling.

7.1. Servlets: Introduction to Servlets, Life cycle of servlets, Java Servlets Development Kit, Creating, Compiling and running servlet, The servlet API (javax.servlet package), Reading the servlet Parameters, Reading Initialization parameter, The javax.servlet.http.Package, Handling HTTP Request and Response (GET / POST Request), Using Cookies, Session Tracking.

7.2. Java Server Pages: Advantage of JSP technology (Comparison with ASP / Servlet), JSP Architecture, JSP Access Model, JSP Syntax Basic (Directives, Declarations, Expression, Scriptlets, Comments), JSP Implicit Object, Object Scope, Synchronization Issue, Exception Handling, Session Management, Creating and Processing Forms.

© Deepak Bhatta Kaji *(Assistant Professor)*
Department of B.Ed CSIT (KMC)

7.1. Servlets: Introduction to Servlets, Life cycle of servlets, Java Servlets Development Kit, Creating, Compiling and running servlet, The servlet API (javax.servlet package), Reading the servlet Parameters, Reading Initialization parameter, The javax.servlet.http.Package, Handling HTTP Request and Response (GET / POST Request), Using Cookies, Session Tracking.

Introduction to Servlets:

Servlets are server-side Java programs that extend the capabilities of a server. They can respond to any type of request but are commonly used to extend web servers by providing dynamic content, such as interacting with databases or generating HTML content. Servlets operate on the server side, and they handle client requests by following the HTTP protocol.

Servlet technology is used to create a web application (resides at server side and generates a dynamic web page).

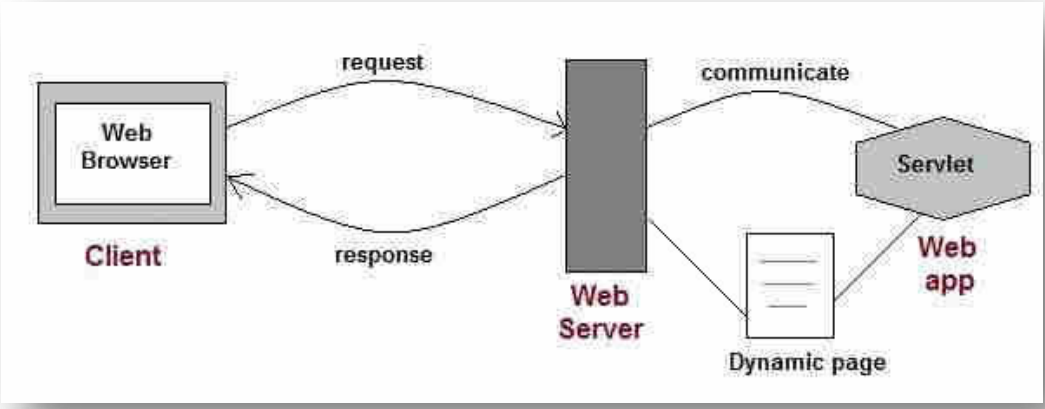
Servlet technology is robust and scalable because of java language. Before Servlet, CGI (Common Gateway Interface) scripting language was common as a server-side programming language. However, there were many disadvantages to this technology. We have discussed these disadvantages below.

There are many interfaces and classes in the Servlet API such as Servlet, GenericServlet, HttpServlet, ServletRequest, ServletResponse, etc.

What is a Servlet?

Servlet can be described in many ways, depending on the context:

- Servlet is a technology which is used to create a web application.
- Servlet is an API that provides many interfaces and classes including documentation.
- Servlet is an interface that must be implemented for creating any Servlet.
- Servlet is a class that extends the capabilities of the servers and responds to the incoming requests. It can respond to any requests.
- Servlet is a web component that is deployed on the server to create a dynamic web page.

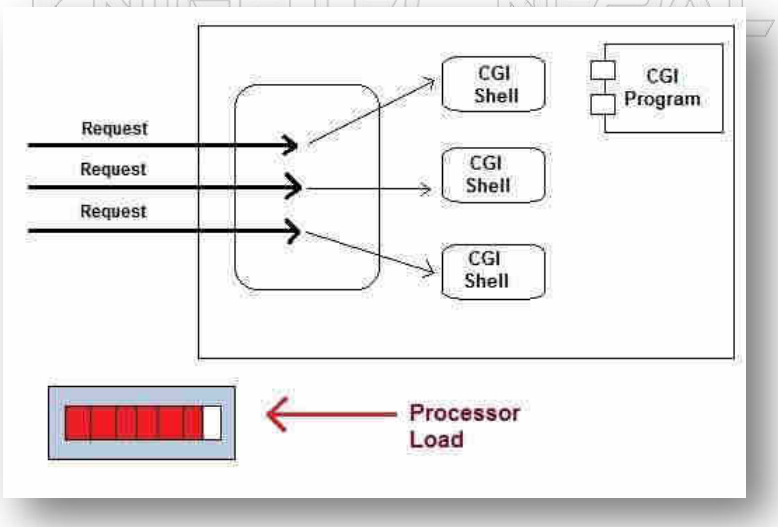


As Servlet Technology uses Java, web applications made using Servlet are Secured, Scalable and Robust.

CGI (Common Gateway Interface):

Before Servlets, CGI (Common Gateway Interface) programming was used to create web applications. Here's how a CGI program works:

- User clicks a link that has URL to a dynamic page instead of a static page.
- The URL decides which CGI program to execute.
- Web Servers run the CGI program in separate OS shell. The shell includes OS environment and the process to execute code of the CGI program.
- The CGI response is sent back to the Web Server, which wraps the response in an HTTP response and send it back to the web browser.



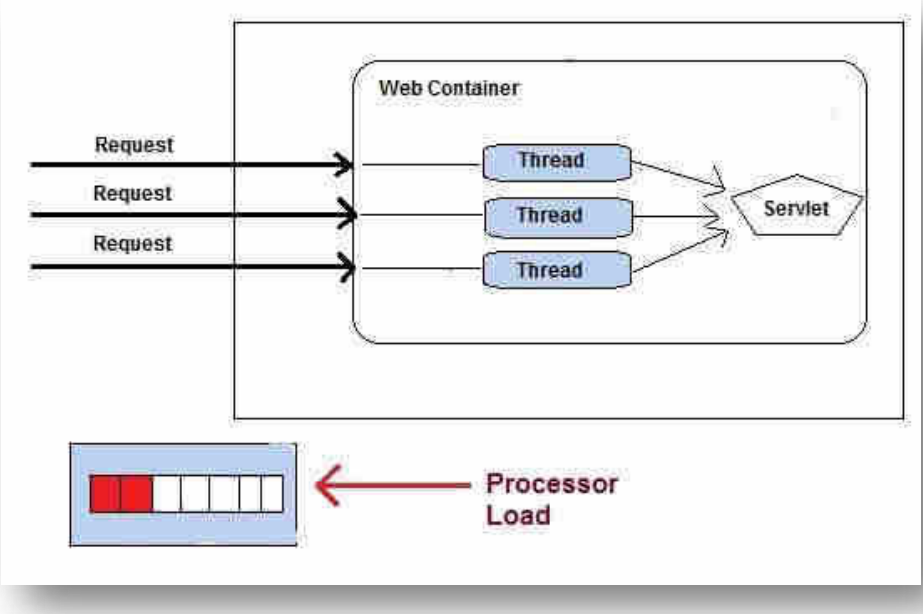
Drawbacks of CGI programs

- High response time because CGI programs execute in their own OS shell.
- CGI is not scalable.
- CGI programs are not always secure or object-oriented.
- It is Platform dependent.

Because of these disadvantages, developers started looking for better CGI solutions. And then Sun Microsystems developed **Servlet** as a solution over traditional CGI technology.

Advantages of using Servlets

- Less response time because each request runs in a separate thread.
- Servlets are scalable.
- Servlets are robust and object oriented.
- Servlets are platform independent.



Servlet Architecture:

Servlet architecture comes under a java programming language used to create dynamic web applications. Mainly servlets are used to develop server-side applications. Servlets are very robust and scalable. Before introducing servlets, CGI (common gateway interface) was used. Servlets are used to perform client request and response tasks dynamically. Servlets can be used to perform tasks like,

- ✓ Control the flow of the application.
- ✓ Generate dynamic web content.
- ✓ Server-side load balancing.
- ✓ implement business logic.

There are two types of Servlets:

1. Generic Servlets
2. HTTP Servlets

Servlets can be created in three possible ways:

1. Implementing Servlet Interface
2. Extending Generic Servlet
3. Extending HTTP Servlet

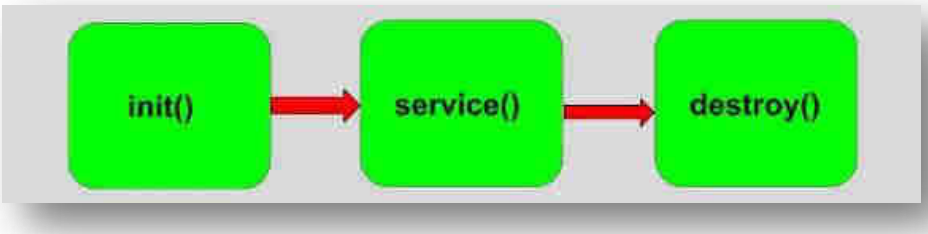
Life cycle of servlets:

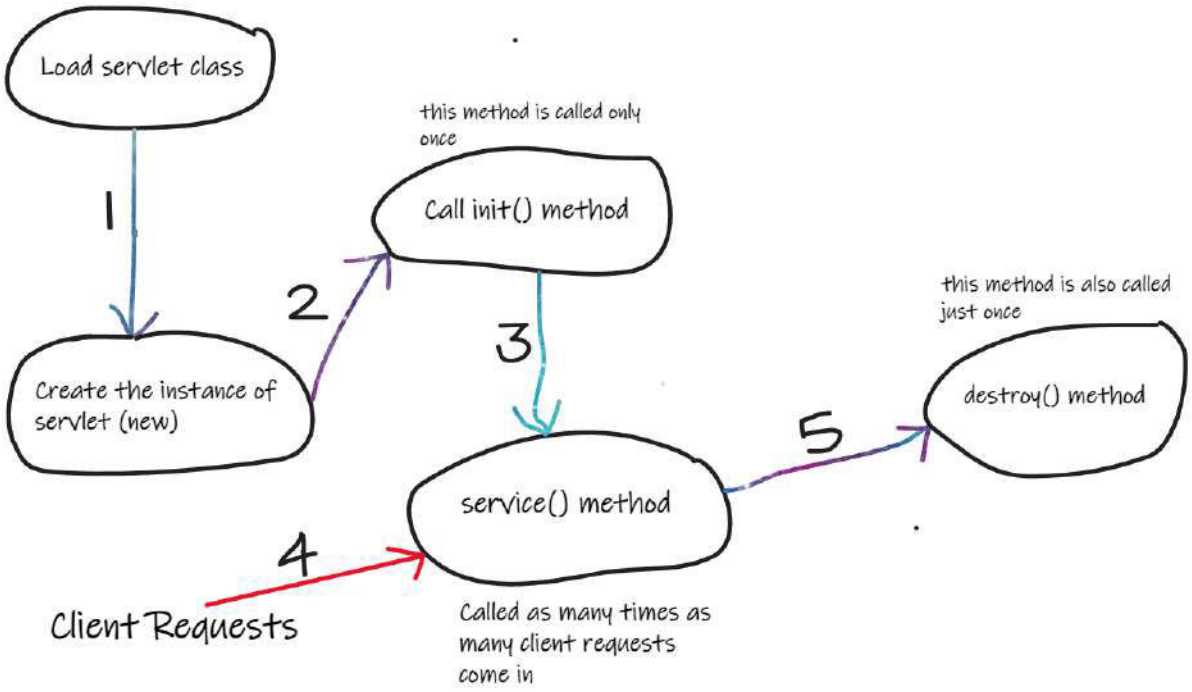
The life cycle of a servlet is defined by three main methods: init(), service(), and destroy(). These methods are called by the servlet container (like Apache Tomcat or Jetty).

Three life cycle methods available with servlets are:

1. init()
2. service() and
3. destroy()

Every servlet should override these methods.





Life Cycle of a Servlet

Let's look at each of these methods in details:

init() method: The Servlet.init() method is called by the Servlet container to indicate that this Servlet instance is instantiated successfully and is about to put into service.

```
//init() method
public class MyServlet implements Servlet{
   public void init(ServletConfig config) throws ServletException {
       //initialization code
   }
    //rest of code
}
```

service() method: The service() method of the Servlet is invoked to inform the Servlet about the client requests.

- This method uses **ServletRequest** object to collect the data requested by the client.
- This method uses **ServletResponse** object to generate the output content.

```
// service() method
public class MyServlet implements Servlet{
    public void service(ServletRequest res, ServletResponse res)
    throws ServletException, IOException {
            // request handling code
    }
    // rest of code
}
```

destroy() method: The destroy() method runs only once during the lifetime of a Servlet and signals the end of the Servlet instance.

```
//destroy() method
public void destroy()
```

As soon as the **destroy()** method is activated, the Servlet container releases the Servlet instance.

Java Servlets Development Kit:

The Java Servlet Development Kit (JSDK) provides the classes and interfaces necessary for developing servlets. It is included as part of the Java EE SDK, but you can also use standalone containers such as Tomcat or Jetty to run servlets.

Creating, Compiling and Running servlet:

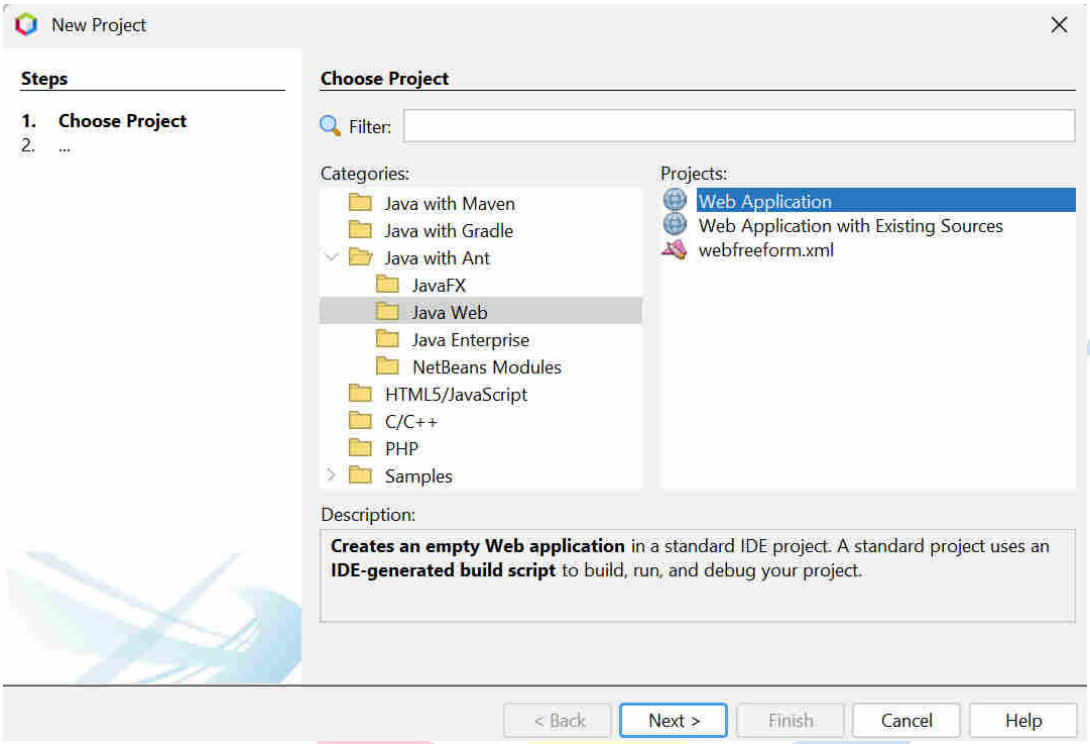
1. **Creating a Servlet**: A servlet is created by extending the **HttpServlet** class and overriding methods like **doGet()** and **doPost()** for handling HTTP requests.

2. **Compiling a Servlet:** The servlet is compiled like any other Java class using the javac command. Make sure that your classpath includes the servlet API (e.g., servlet-api.jar).

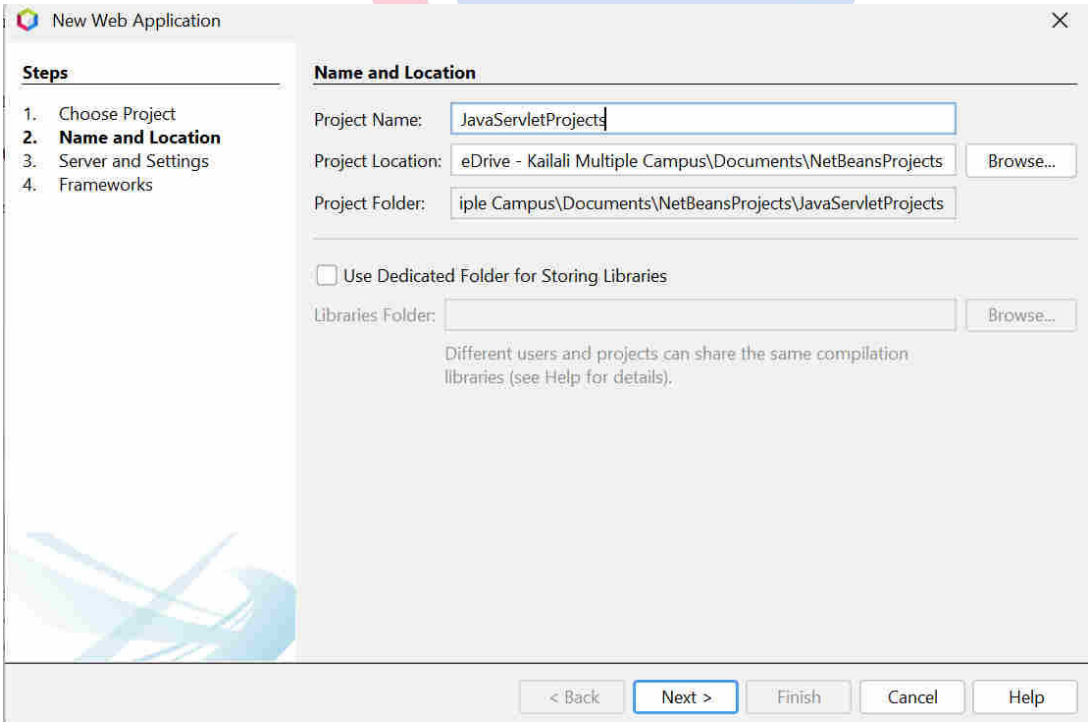
3. **Running a Servlet:** To run a servlet, it needs to be deployed to a servlet container (like **Tomcat**). The web server will handle client requests and delegate them to the appropriate servlet.

Here, I have shown the snapshot for creating the Servlet Project from Scratch:

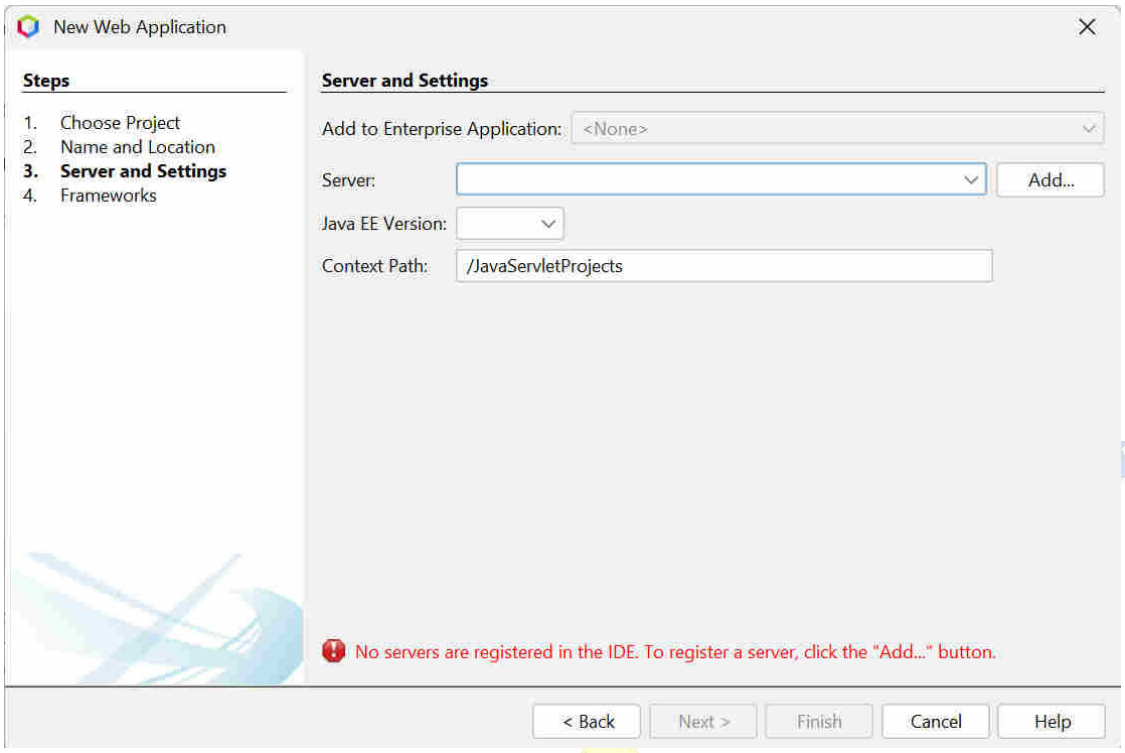
Step 1: Choose a Project, Here I have chosen the **Java with Ant ➔ Java Web ➔ Web Application**



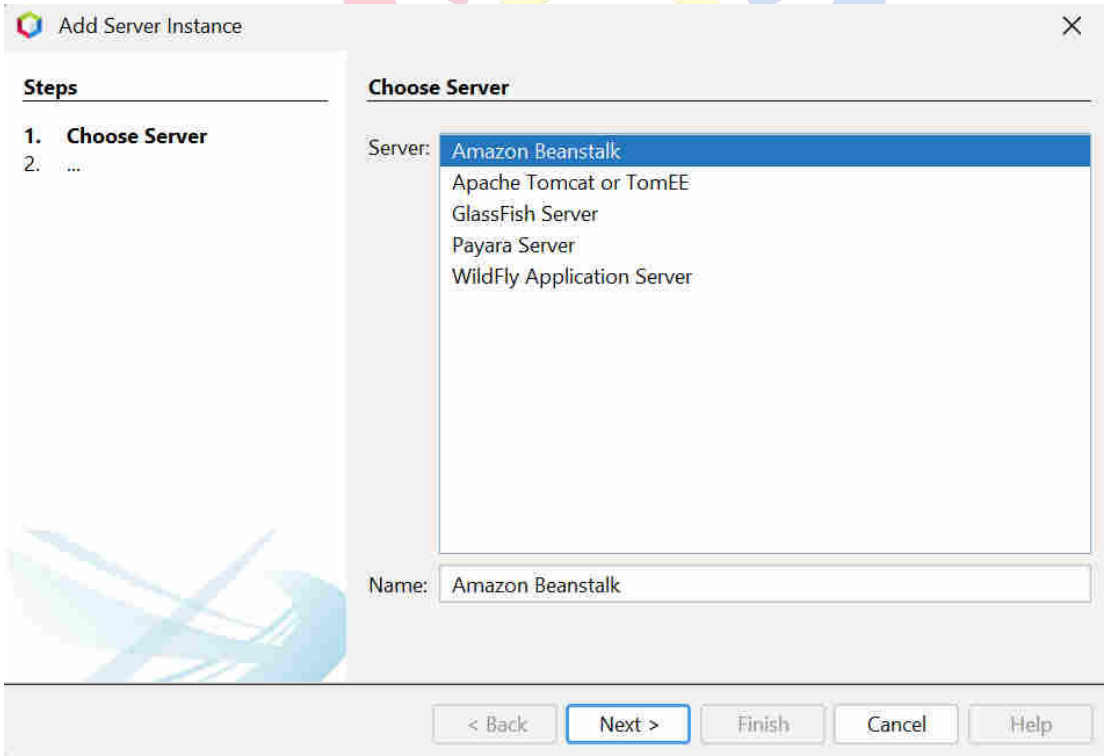
Step 2: Provide Name and Location for Servlet Project.



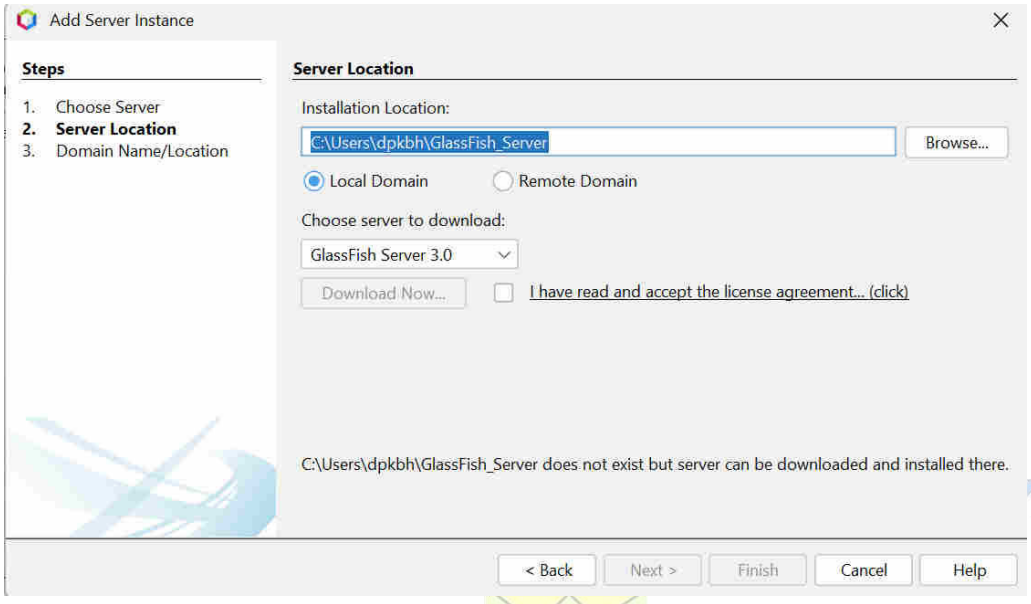
Step 3: Now time to Select the **Server** if you do not have any server in the list of servers, then click **Add…** button



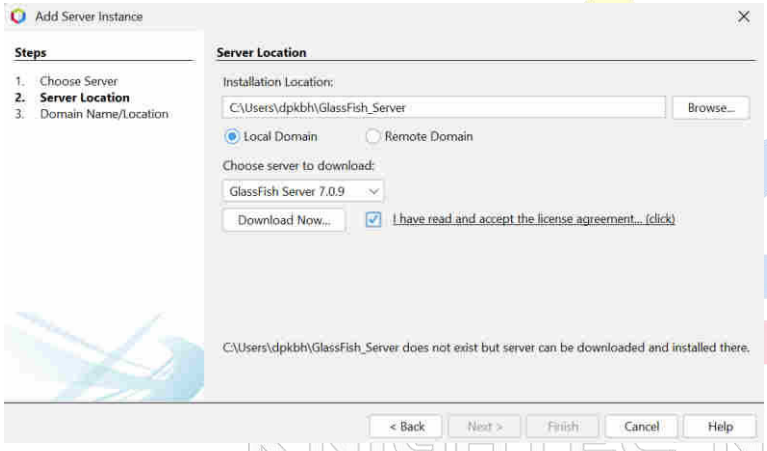
Step 4: You can see here list of Servers, You can select any of them here I am going to Select **GlassFish** Server,



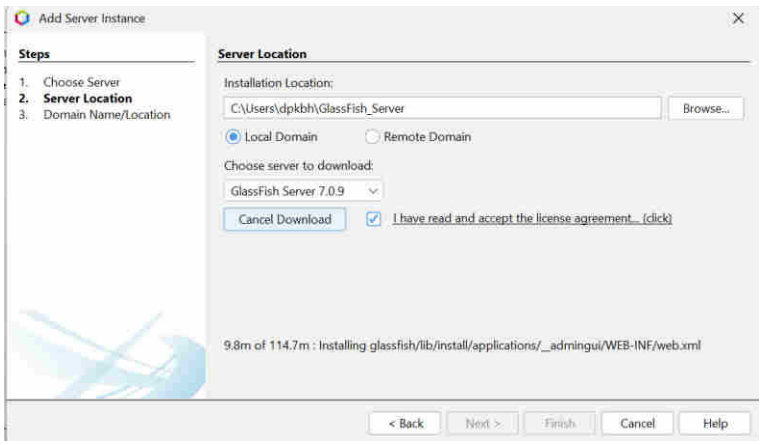
Step 5: Browse the Server Location, **or** Set it by default. Choose server version



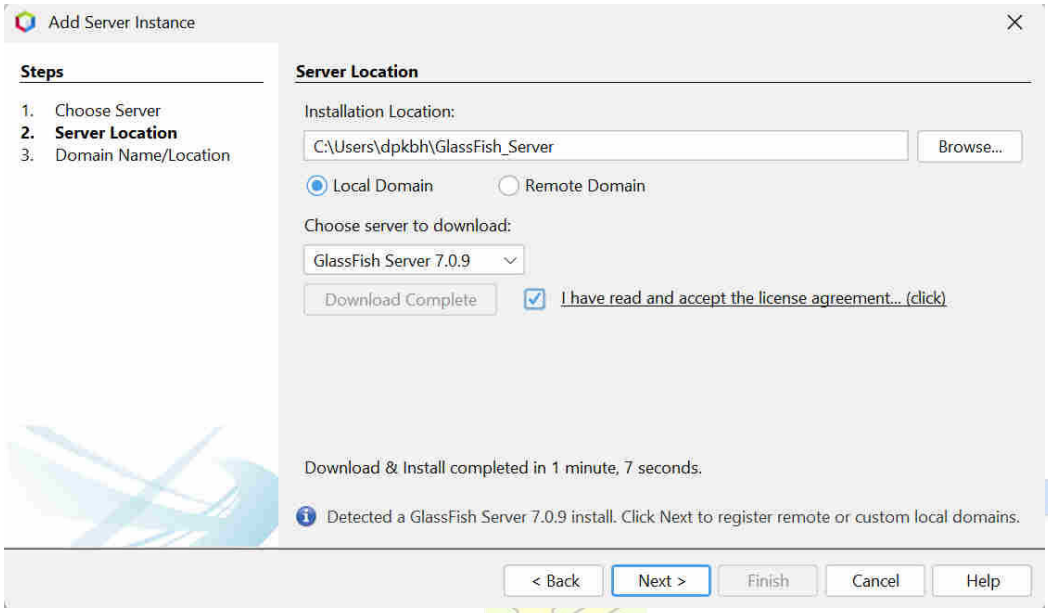
Step 6: Now, Choose GlassFish server version I have chosen **GlassFish Server 7.0.9.** And Check the I have read and accept the license agreement… (click)



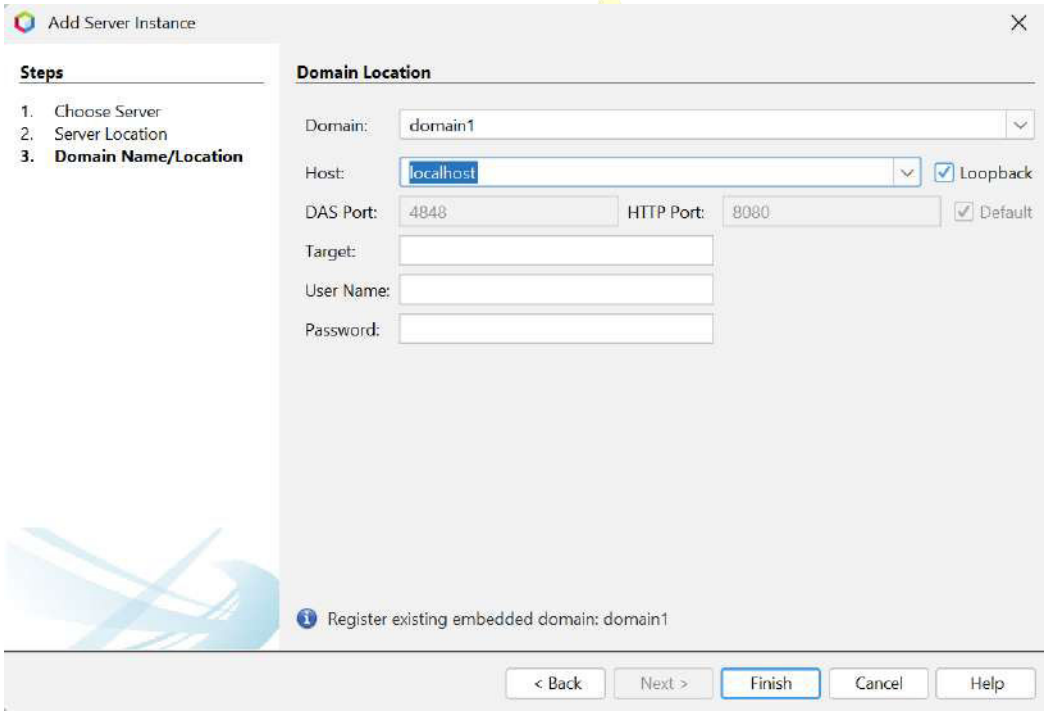
Step 7: Please wait till download complete.



Step 8: Once download is completed click **Next >**

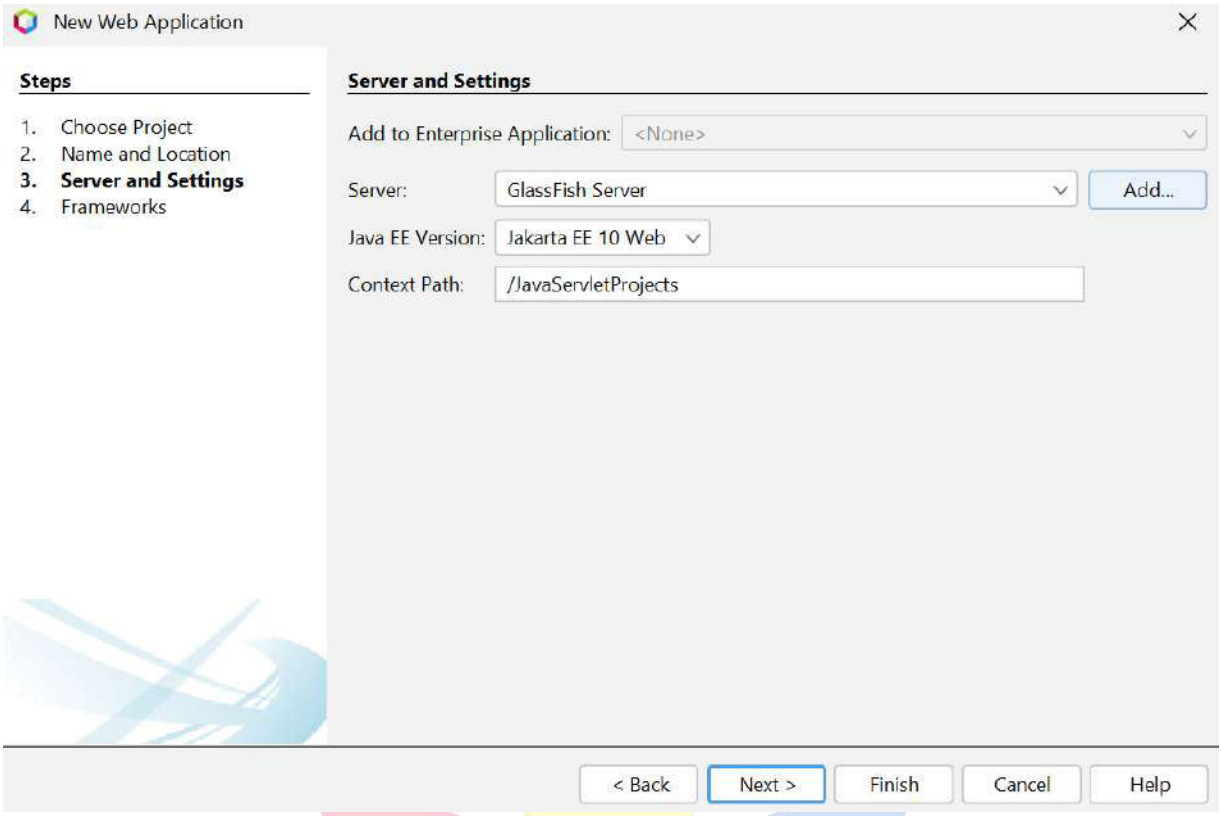


Step 9: For Now, left Domain Name / Location as it is. Click Fnish

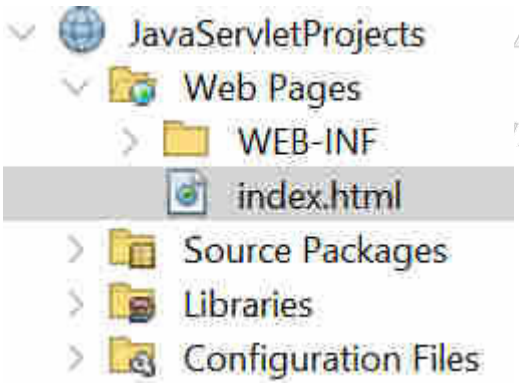


Step 10: You will automatically move to **Step 3** where you can see the Server name.

Click **Next** to Select the **Frame work for Servlet**. But we do not need for now. Click **Finish** button.

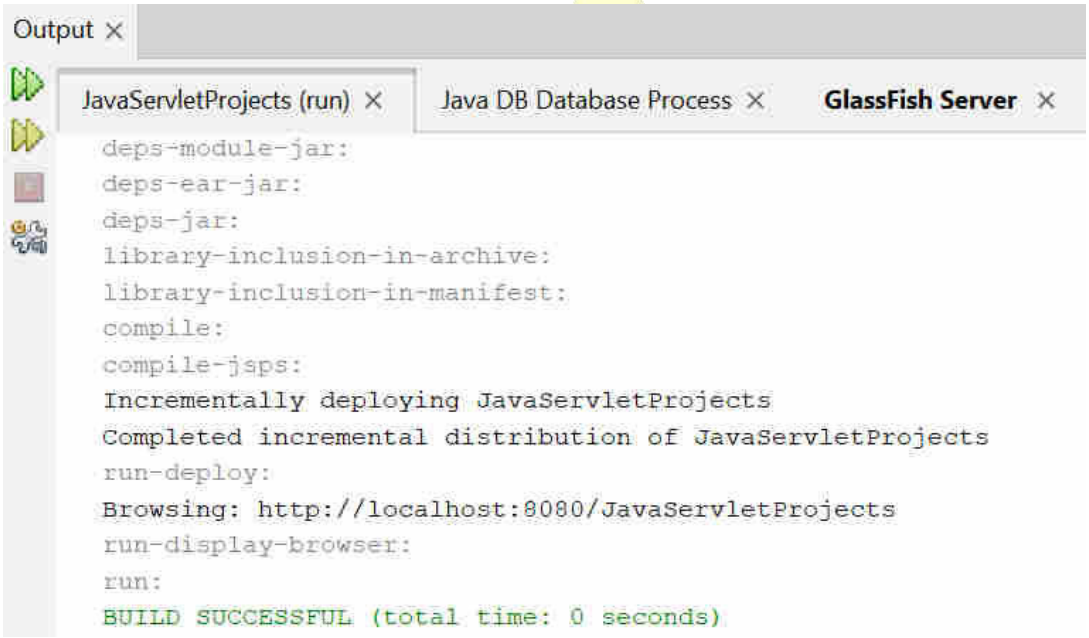


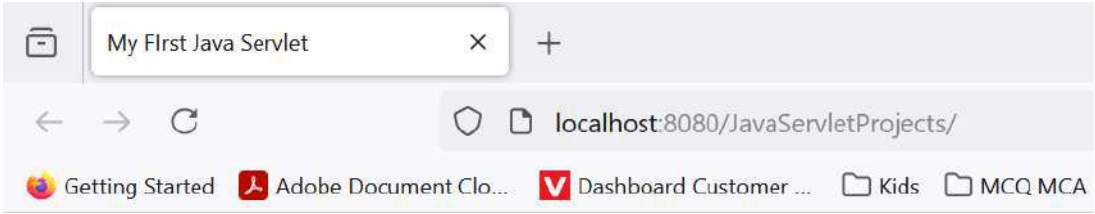
Step 11: Now you can see the Hierarchy for the Java Servlet Project. Here index.html is the default page for Servlet.



Example: Display Message in Web browser using Glassfish server.

```
<!DOCTYPE html>
<html>
    <head>
        <title>My FIrst Java Servlet</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    </head>
    <body>
        <div>Hello World! <br/> Greetings from CSIT - KMC</div>
    </body>
</html>
```





Hello World!
Greetings from CSIT - KMC

The servlet API (javax.servlet package):

The javax.servlet package defines classes and interfaces to support the development of servlets. Key classes include:

- **Servlet Interface**: Defines methods like *init()*, *service()*, and *destroy()*.
- **GenericServlet Class**: A generic, protocol-independent servlet base class.
- **HttpServlet Class**: A subclass of GenericServlet tailored for HTTP requests.

Reading the servlet Parameters:

Servlets can read parameters sent by the client using the **getParameter()** method of the **HttpServletRequest** object. These parameters are typically sent through an HTML form or as part of a query string.

```
String param = request.getParameter("paramName");
```

Reading Initialization parameter:

Initialization parameters are configured in the web.xml file or using annotations, and they provide the servlet with configuration data. The servlet reads them using the **getInitParameter()** method.

```
String configValue = getServletConfig().getInitParameter("configName");
```

The javax.servlet.http.Package:

The **javax.servlet.http** package contains classes specifically designed for HTTP-based servlets. Key classes include:

- **HttpServletRequest**: Encapsulates client request information such as headers and parameters.
- **HttpServletResponse**: Provides methods for responding to the client, like sending HTML or setting response headers.

Handling HTTP Request and Response (GET / POST Request):

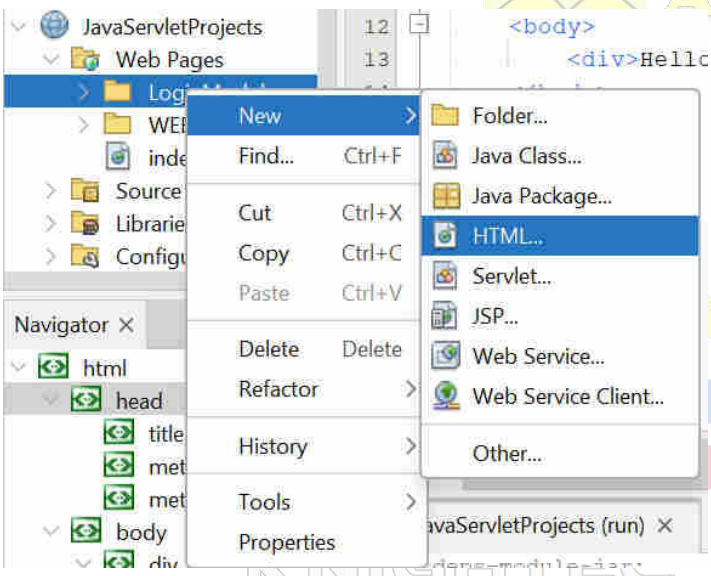
GET Request: This request method is used to request data from a specified resource. It appends parameters in the URL query string.

```
protected void doPost(HttpServletRequest request, HttpServletResponse
response)
```

POST Request: This method is used to send data to the server (e.g., from a form submission).

```
protected void doPost(HttpServletRequest request, HttpServletResponse
response)
```


How does the client or the Browser send these parameters using the methods GET or POST, is explained below:

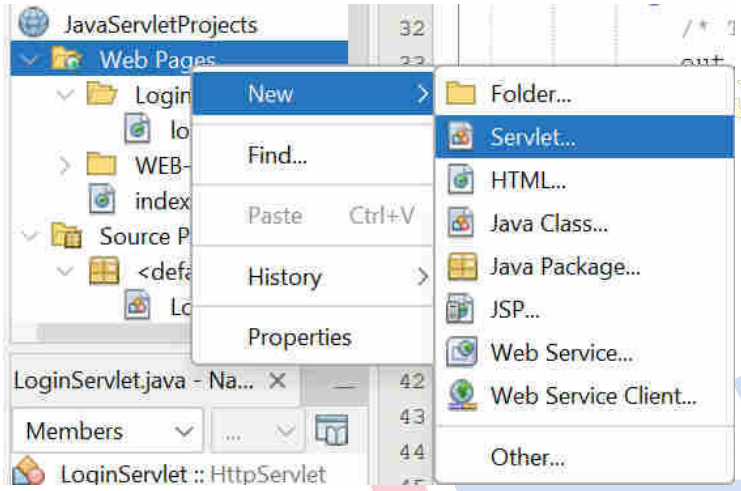


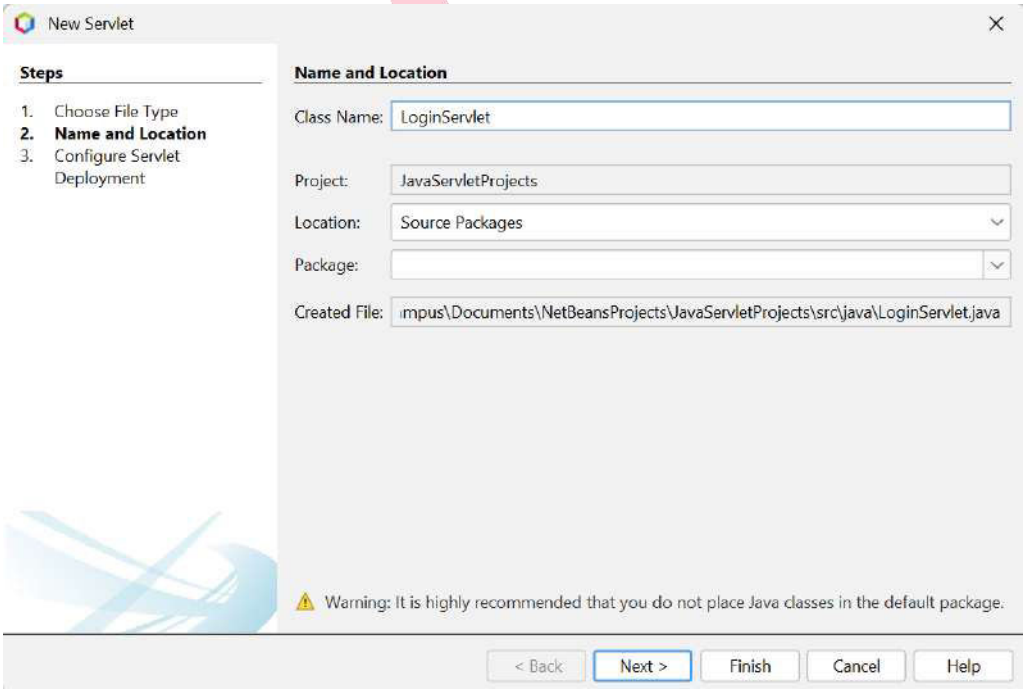
The first thing we are going to do is to create in our site a page "**login.html**" with the following content:

```
<html>
<body>
      <form action=" ../LoginServlet" method="get"> //or Change post method
            <table>
                  <tr>
                        <td>User:</td>
                        <td><input type="text" name="txtUser"/></td>
                  </tr>
                  <tr>
```

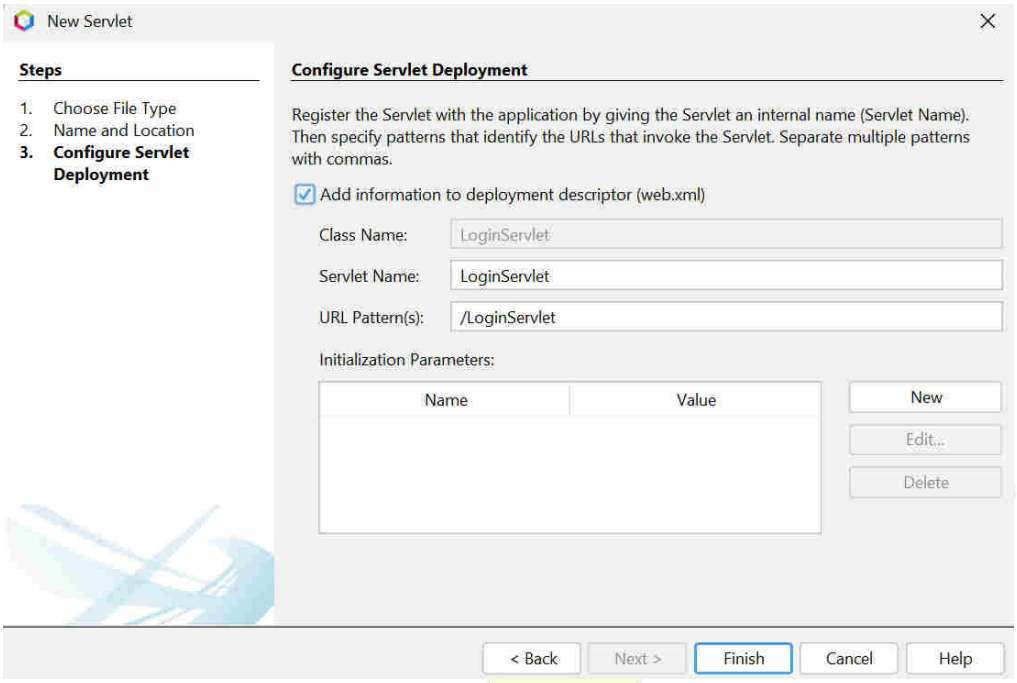
```
                    <td>password:</td>
                    <td>
                    <input type="password" name="txtPassword"/>
                    </td>

                </tr>
            </table>
            <input type="submit" name="btnSubmit" value=”Login”/>
        </form>
</body>
```

Then, we create a Servlet which receives the request in **../LoginServlet**, which is the indicated direction in the **action** attribute of the tag **<form>** of login.html





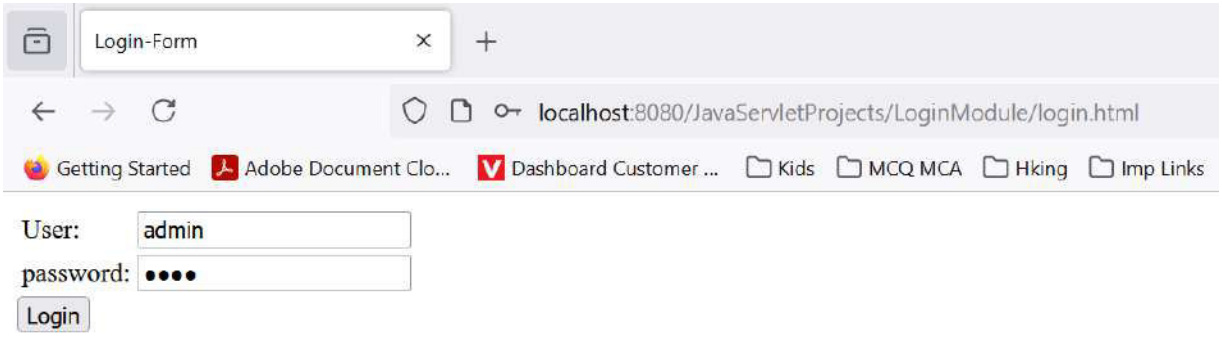
Check the Add information to deployment descriptor (web.xml)

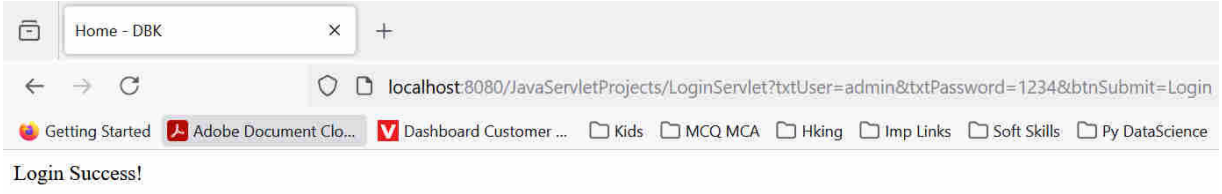


Servlet Code : LoginServlet – GET Method

```
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;

public class LoginServlet extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
    }
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
```

```
        String user = request.getParameter("txtUser");
        String pass = request.getParameter("txtPassword");
        if ("admin".equals(user) && "1234".equals(pass)) {
            response(response, "Login Success!");
        } else {
            response(response, "Invalid Login, Try Again!");
        }
    }
    private void response(HttpServletResponse response, String
login_Success)
            throws IOException {
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head><title>Home - DBK");
        out.println("</title></head>");
        out.println("<body>");
        out.println("<t1>" + login_Success + "</t1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```





NOTE:

The problem here is that the secret password is visible in the URL ***http://localhost:8080/JavaServletProjects/LoginServlet?txtUser=admin&txtPassword=1234&btnSubmit=Login***, it will be kept in the history of the Browser and anybody who access the Browser after us can obtain it easily.

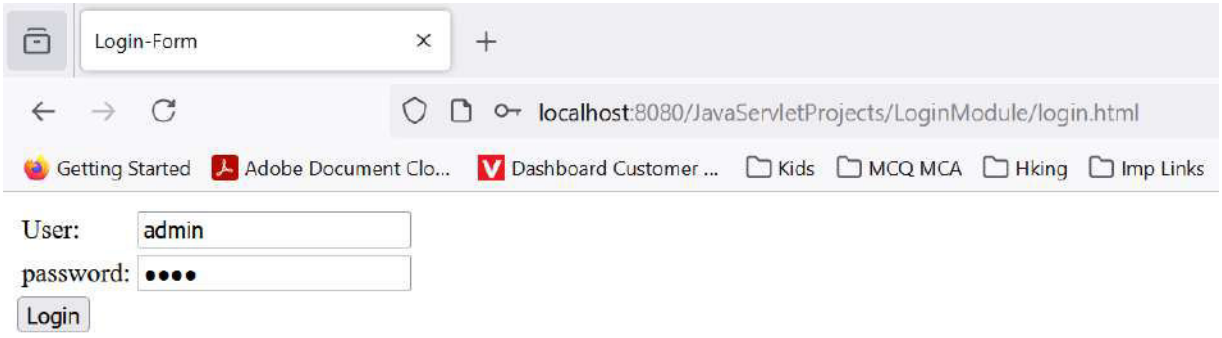
*This can be solved changing the way of sending the form and using the method **POST** in **login.html**.*

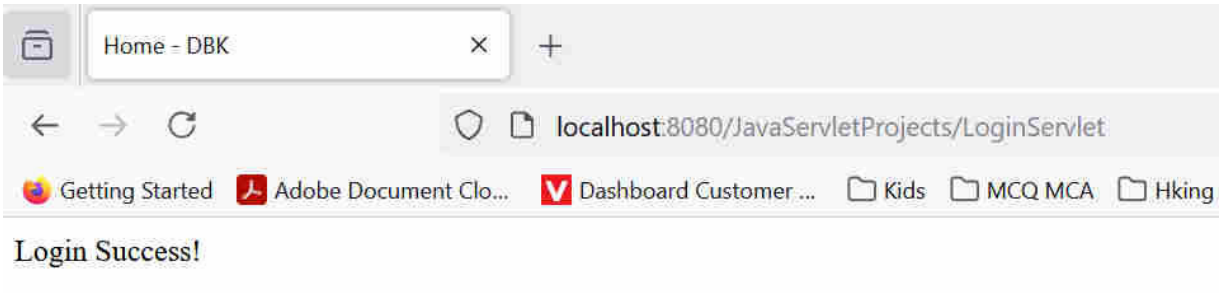
*The only change is the replacement of **doGet** for **doPost**. After the recompilation, the deployment of the Servlet, the restart of the server and the reuse of login.html we obtain.*

Servlet Code : LoginServlet – POST Method

```
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;

public class LoginServlet extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
    }
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
```

```
        String user = request.getParameter("txtUser");
        String pass = request.getParameter("txtPassword");
        if ("admin".equals(user) && "1234".equals(pass)) {
            response(response, "Login Success!");
        } else {
            response(response, "Invalid Login, Try Again!");
        }
    }
    private void response(HttpServletResponse response, String
login_Success)
            throws IOException {
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head><title>Home - DBK");
        out.println("</title></head>");
        out.println("<body>");
        out.println("<t1>" + login_Success + "</t1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```





Using Cookies:

Cookies are small pieces of data stored on the client side. Servlets use cookies to maintain user-specific information

- In order to use cookies in java, use a Cookie class that is present in **javax.servlet.http** package.
- To make a cookie, create an object of Cookie class and pass a name and its value.
- To add cookie in response, use **addCookie(Cookie)** method of **HttpServletResponse** interface.
- To fetch the cookie, **getCookies()** method of Request Interface is used.

Creating a Cookie:

```
Cookie cookie = new Cookie("name", "value");
response.addCookie(cookie);
```


Reading a Cookie:

```
Cookie[] cookies = request.getCookies();
for (Cookie cookie : cookies)
{
    String name = cookie.getName();
    String value = cookie.getValue();
}
```


NOTE:

Ensure that you're using the right Servlet package

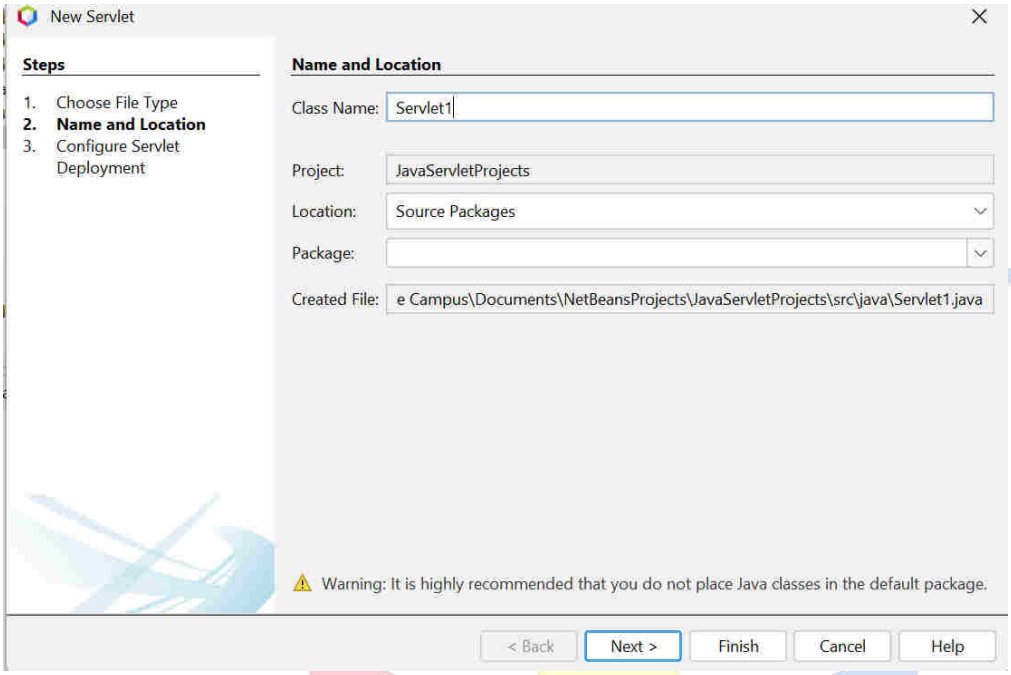
The **javax.*** package has been renamed to **jakarta.*** package since Servlet API version 5.0 which is part of Jakarta EE 9 (***Tomcat 10, TomEE 9, WildFly 22 Preview, GlassFish 6, Payara 6, Liberty 22, etc***). So, if you're targeting these server versions or newer, then you need to replace

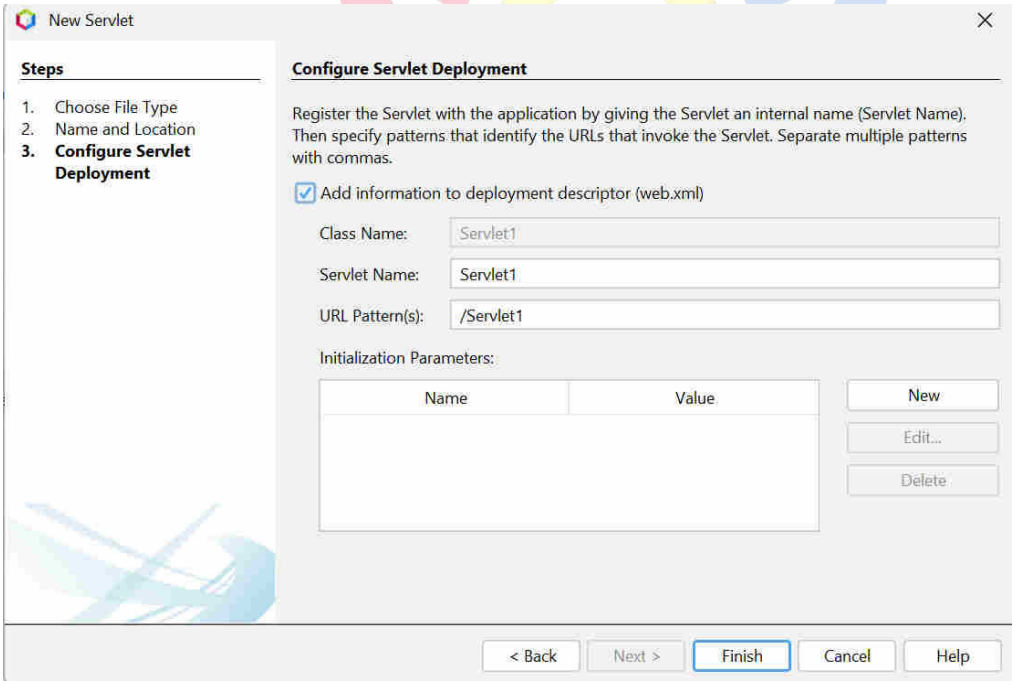
import javax.servlet.*;	by	import jakarta.servlet.*;
import javax.servlet.http.*;		import jakarta.servlet.http.*;

Example:

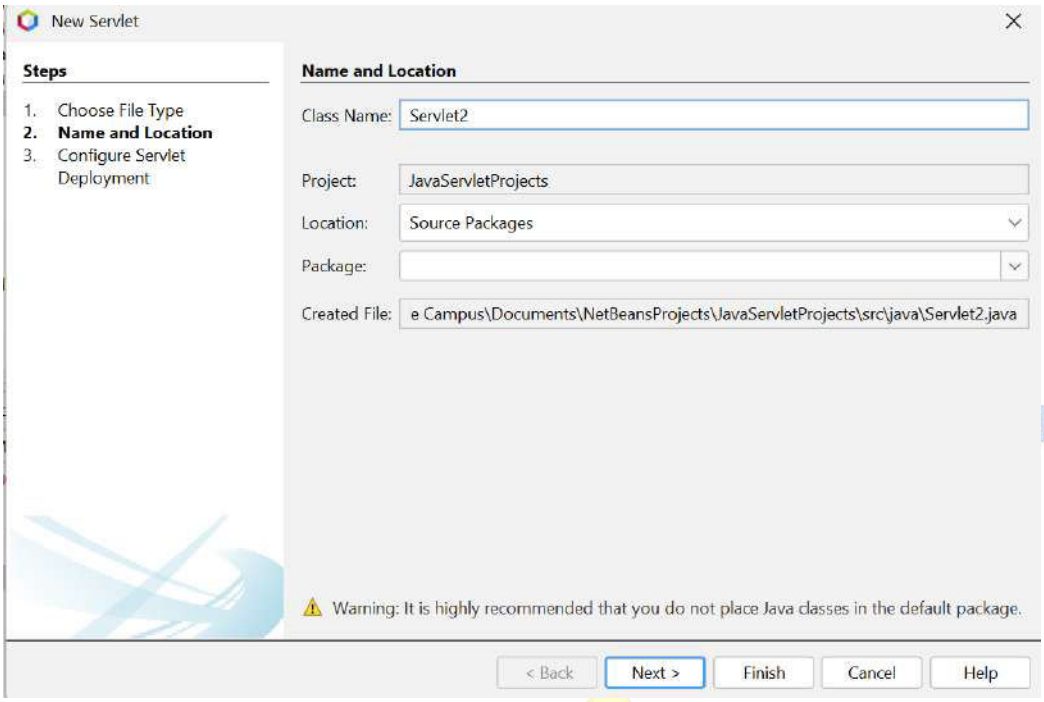
The name of the College is passed to Servlet 2 from Servlet 1 using Cookies.

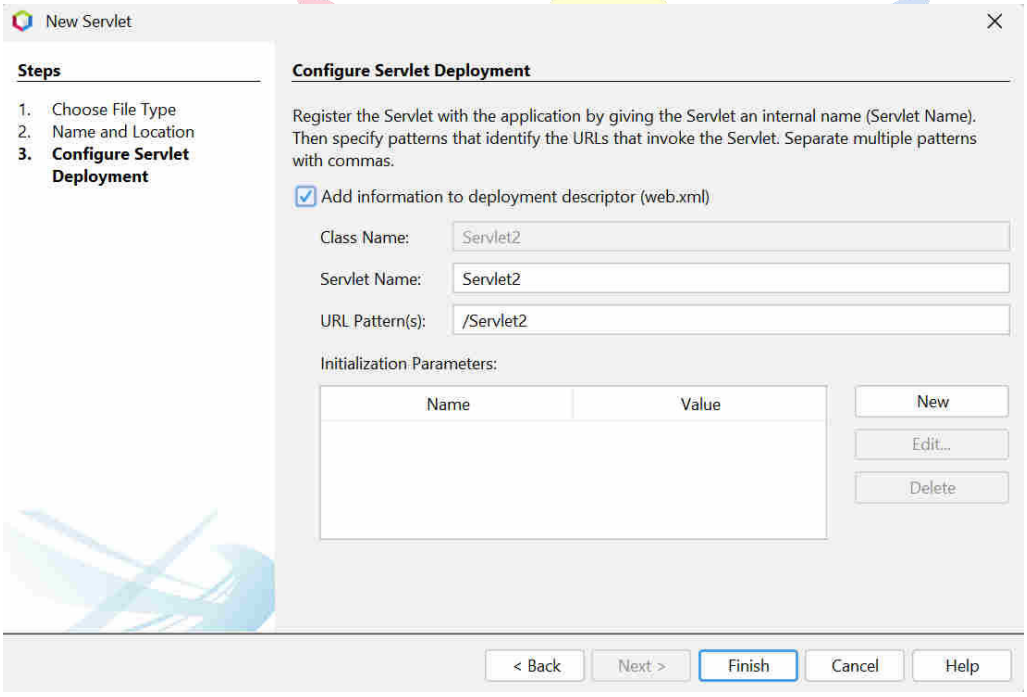
Servlet1.java



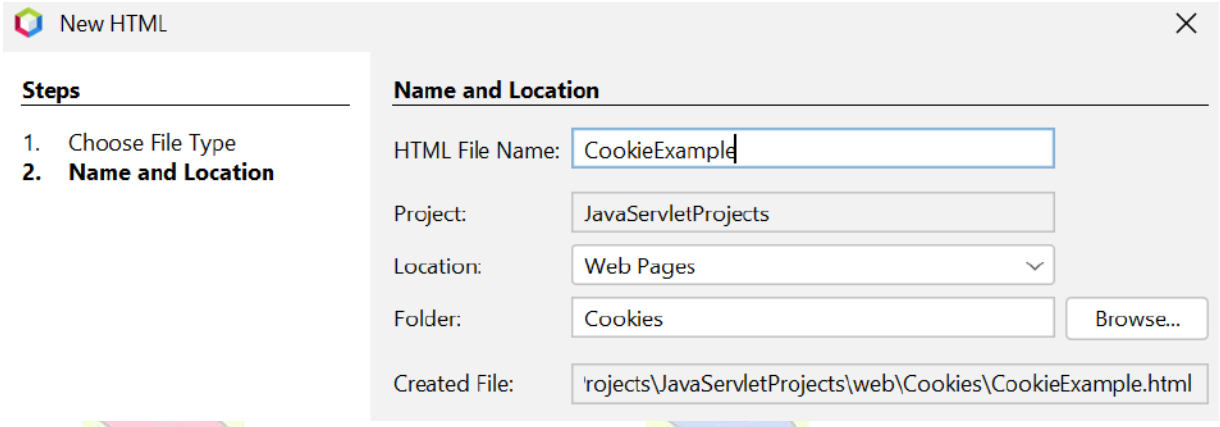


Servlet2.java





CookieExample.html




```
<!DOCTYPE html>
<html>
    <head>
        <title>Servlet Cookies</title>
    </head>
    <body>
        <form action="../Servlet1" method="POST">
            <h2>Enter your College name:</h2>
            <input type="text" name="txtCollegeName"  style="font-
size:30px;">
            <br>
            <br>
           <!-- button to redirect to Servlet1 -->
            <input type="submit" value="Submit" style="font-size:20px;">
        </form>
    </body>
</html>
```

Servlet1.java

```
import jakarta.servlet.ServletException;
import jakarta.servlet.http.Cookie;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import java.net.URLEncoder;
import java.nio.charset.StandardCharsets;

public class Servlet1 extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html");

        // Retrieving user input from the request parameter
        String name = request.getParameter("txtCollegeName");

        // Create and add a cookie before writing any response output
        if (name != null && !name.isEmpty()) {
      // URL-encode the cookie value to ensure special characters are handled
            String encodedName = URLEncoder.encode(name,
StandardCharsets.UTF_8.toString());
            Cookie c = new Cookie("user_name", encodedName);
            c.setPath("/"); // Make cookie available to the entire application
            c.setMaxAge(60 * 60 * 24); // Set cookie expiration to 1 day
            response.addCookie(c);
        }
```

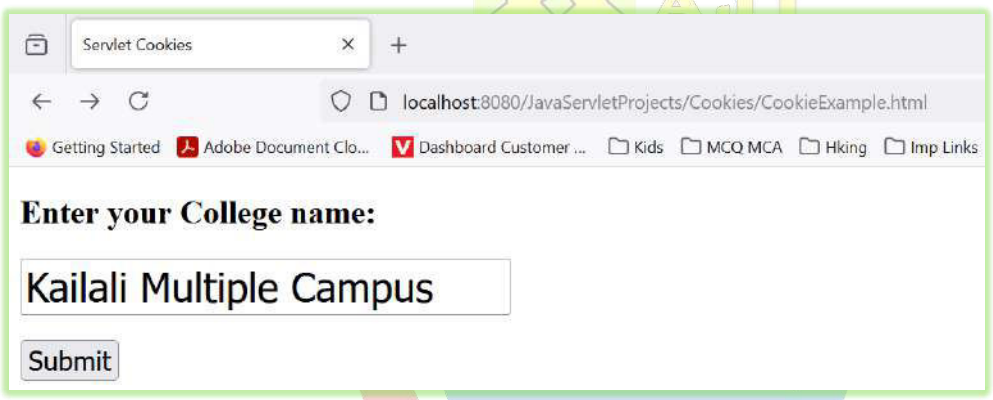
```
        try (PrintWriter out = response.getWriter()) {
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet Servlet1</title>");
            out.println("</head>");
            out.println("<body>");

            if (name != null && !name.isEmpty()) {
                out.println("<h1>Hello, welcome to " + name + "!</h1>");
                out.println(
                        "<h1><a href =\"Servlet2\">Go to
Servlet2</a></h1>");
            } else {
                out.println("<h1>College name not provided.</h1>");
                out.println("<h1><a href =\"Cookies/CookieExample.html\">
                  Go to Home</a></h1>");
            }

            out.println("</body>");
            out.println("</html>");
        }
    }
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        processRequest(request, response);
    }
}
```

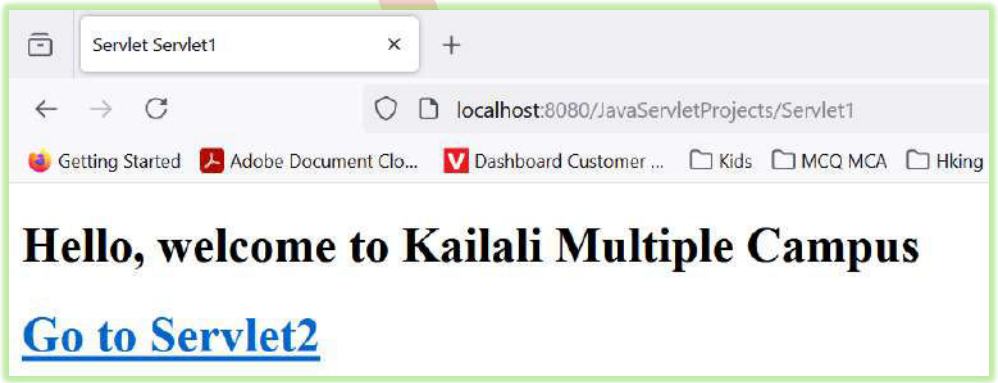
Servlet2.java

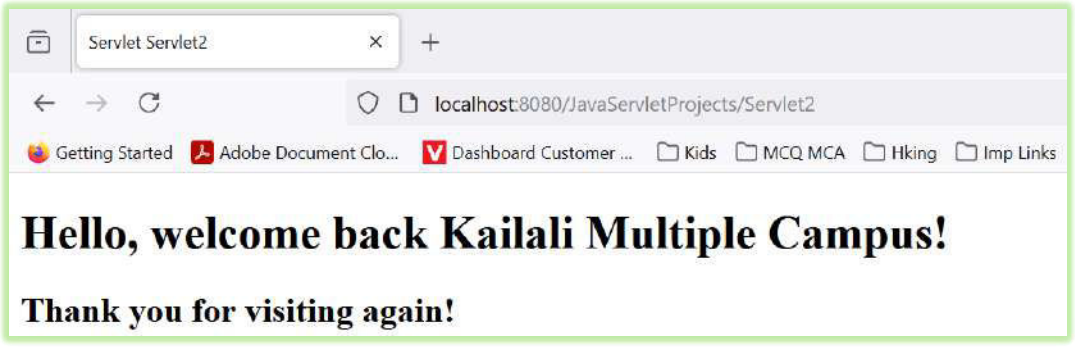
```
import jakarta.servlet.ServletException;
import jakarta.servlet.http.Cookie;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import java.net.URLDecoder;
import java.nio.charset.StandardCharsets;

public class Servlet2 extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet Servlet2</title>");
            out.println("</head>");
            out.println("<body>");
```

```
            String encodedName = getCookieValue(request, "user_name");
            String userName = null;
            if (encodedName != null) {
        userName = URLDecoder.decode(encodedName, StandardCharsets.UTF_8);
            }
            if (userName != null) {
             out.println("<h1>Hello, welcome back " + userName +
"!</h1>");
             out.println("<h2>Thank you for visiting again!</h2>");
            } else {
            out.println("<h1>You are a new user, please go to the homepage
and submit your college name.</h1>");
            out.println("<h1><a href =\"Cookies/CookieExample.html\">Go to
Home</a></h1>");
            }
            out.println("</body>");
            out.println("</html>");
        }
    }
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        processRequest(request, response);
    }
    private String getCookieValue(HttpServletRequest request, String
user_name) {
        Cookie[] cookies;
```

```
        cookies = request.getCookies();
        if (cookies != null) {
            for (Cookie cookie : cookies) {
                if (user_name.equals(cookie.getName())) {
                    return cookie.getValue();
                }
            }
        }
        return null; // Return null if cookie is not found
    }
}
```







Session Tracking:

Sessions allow the server to store user-specific information across multiple requests. This is essential for maintaining user state in a stateless HTTP protocol.

In web terminology, a session is simply the limited interval of time in which two systems communicate with each other. The two systems can share a client-server or a peer-to-peer relationship. However, in Http protocol, the state of the communication is not maintained. Hence, the web applications that work on http protocol use several different technologies that comprise **Session Tracking**, which means maintaining the state (data) of the user, in order to recognize him/her.

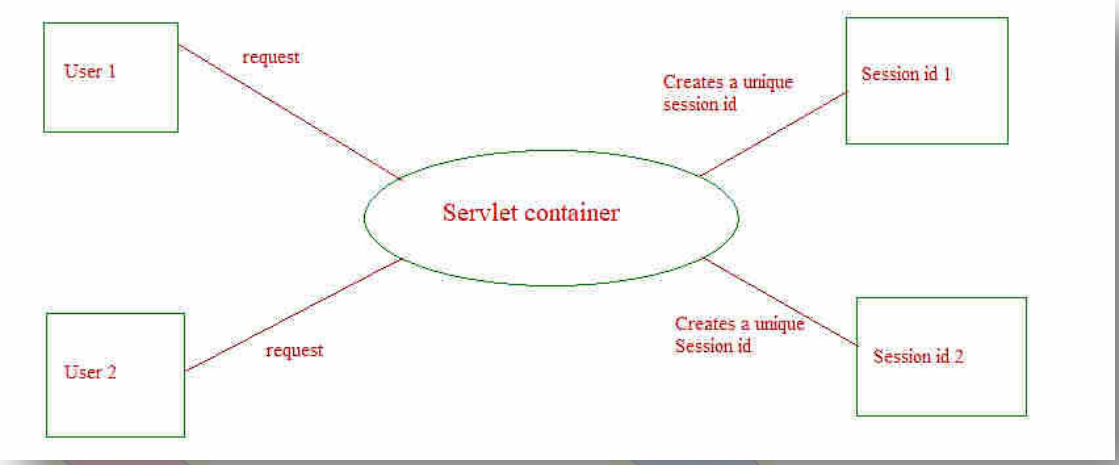
In order to achieve session tracking in servlets, cookies have been one of the most commonly used tech.

However, they have the following disadvantages:

- They can only keep textual information.
- They're browser dependent. Hence, if the client disables them, your web application can't make use of them
- Individual cookie can contain not more than 4kb of information

How to create sessions with a unique session id for each user in java servlet

For this, servlets provide an interface called **'HttpSession' Interface**. The following diagram explains how Http Sessions work in servlets:



Advantages of Http Sessions in Servlet

- Any kind of object can be stored into a session, be it a text, database, dataset etc.
- Usage of sessions is not dependent on the client's browser.
- Sessions are secure and transparent

Disadvantages of Http session

- Performance overhead due to session object being stored on server
- Overhead due to serialization and de-serialization of data

Example of Session tracking using HttpServlet Interface:

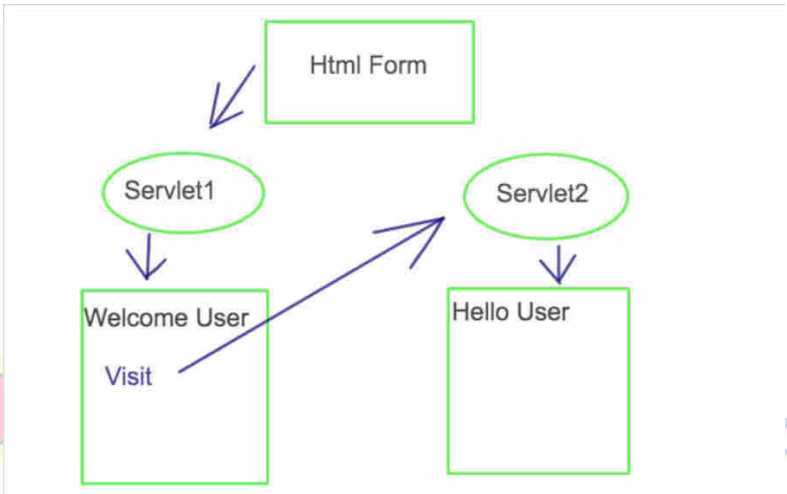
In the below example the **setAttribute()** and **getAttribute()** methods of the **HttpServlet** class is used to create an attribute in the session scope of one servlet and fetch that attribute from the session scope of another servlet.

Creating/Accessing a Session:

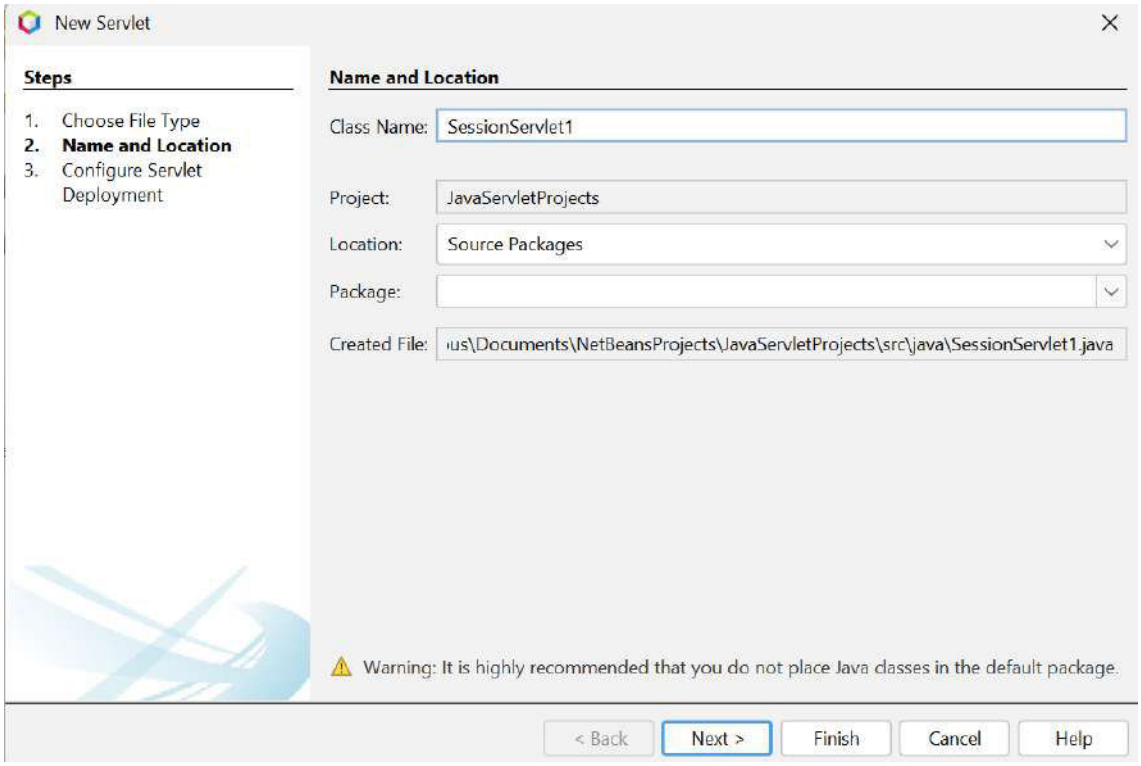
```
HttpSession session = request.getSession();
session.setAttribute("user", "John");
```

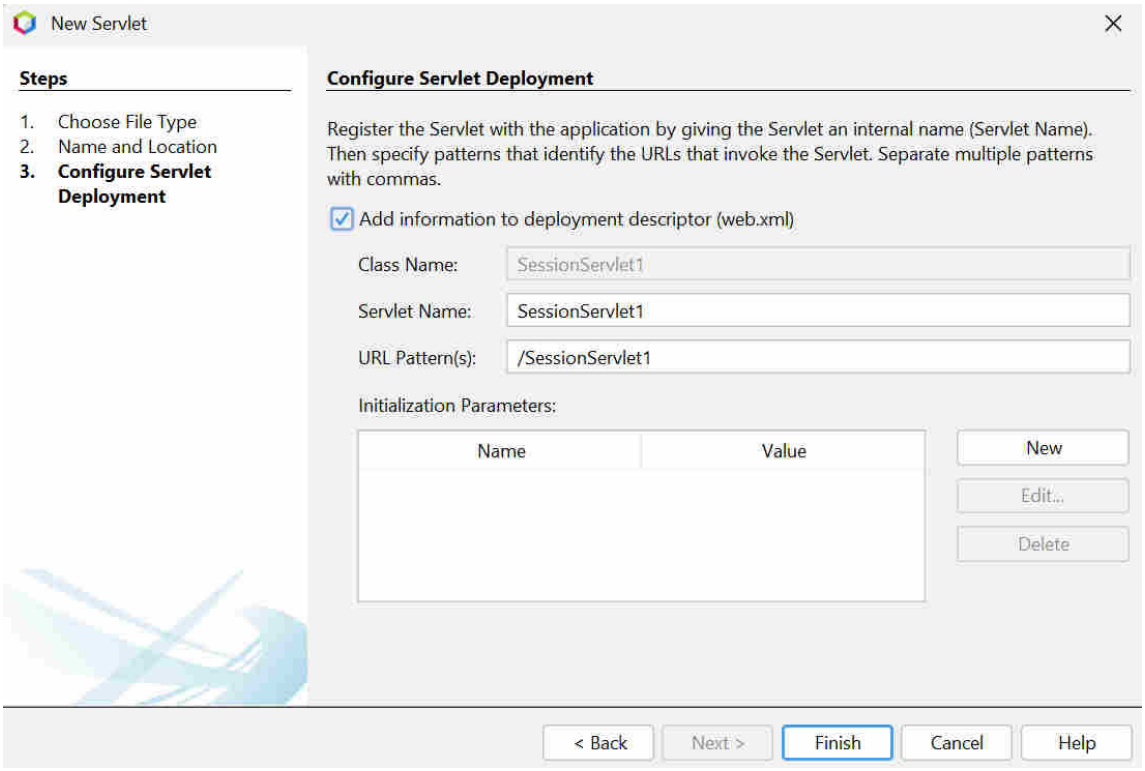
Retrieving Session Data:

```
String user = (String) session.getAttribute("user");
```

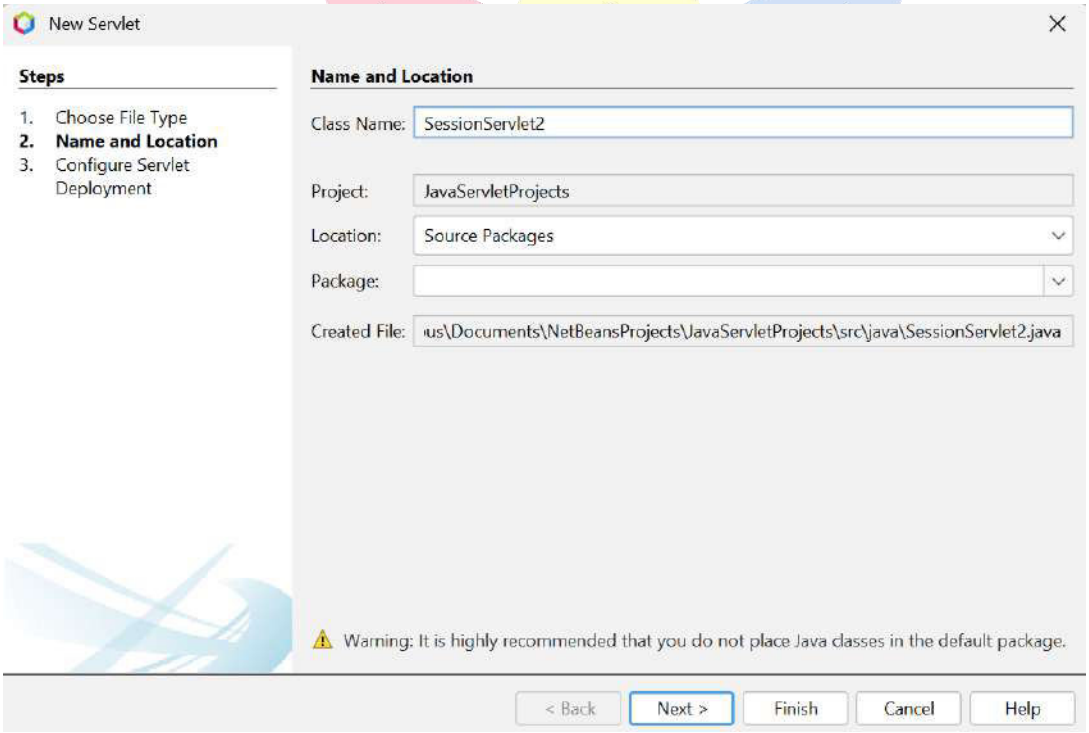


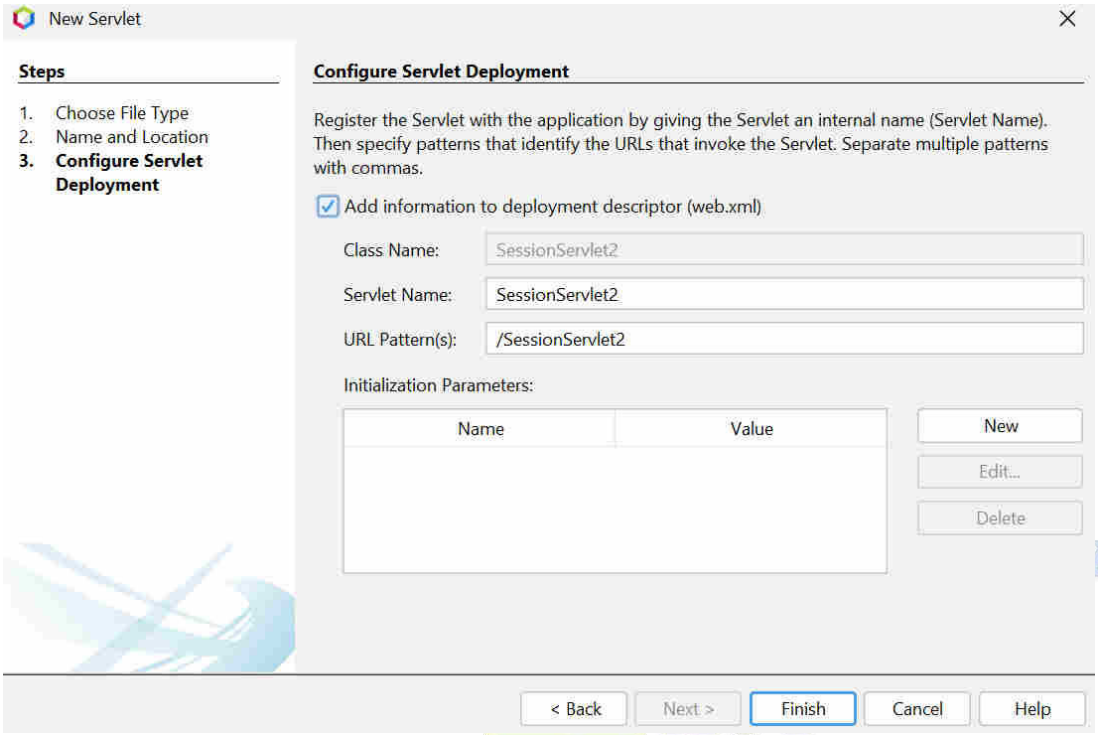
SessionServlet1.java



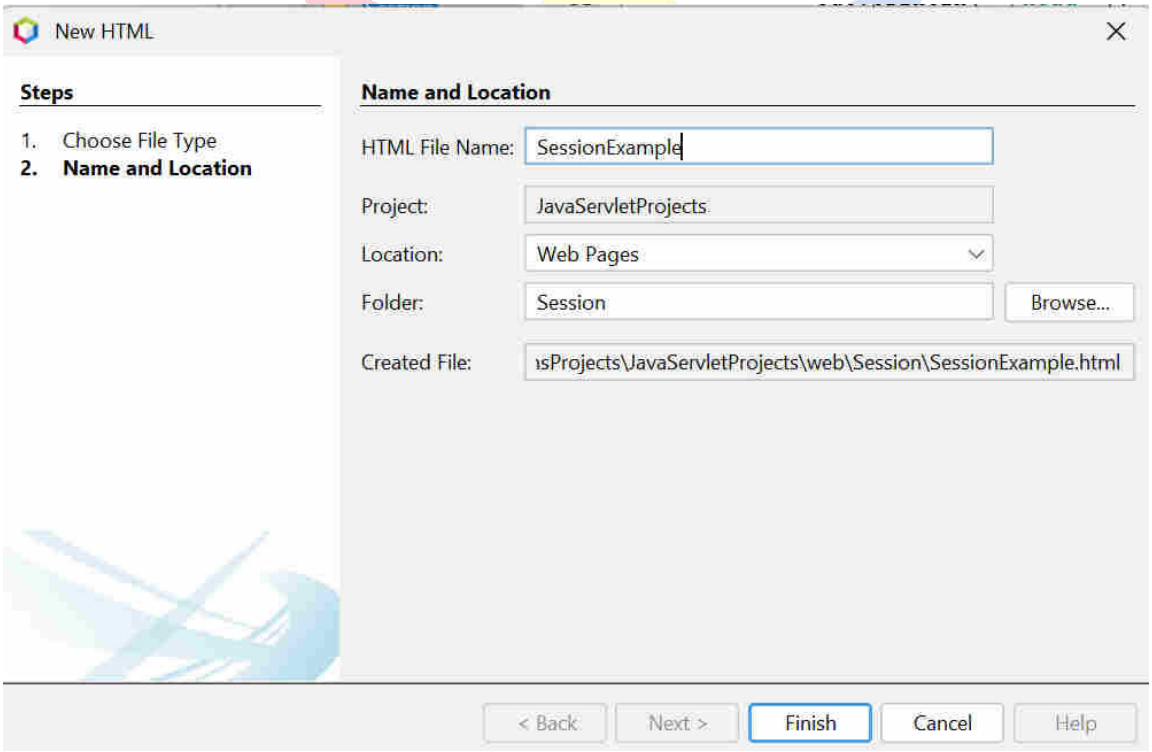


SessionServlet2.java





SessionExample.html




```
<!DOCTYPE html>
```

```
<html>
    <head>
        <title>Servlet Session</title>
        <meta charset="UTF-8">
        <meta    name="viewport"    content="width=device-width,    initial-
scale=1.0">
    </head>
    <body>
        <form action="../SessionServlet1" method="post">
            <h2>    Name:</h2><input    type="text"    name="txtUserName"
style="font-size:30px;"/> <br/><br/>
            <input type="submit" value="submit" style="font-size:20px;"/>
        </form>
    </body>
</html>
```

SessionServlet1.java

```
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.http.HttpSession;
import java.io.IOException;
import java.io.PrintWriter;

public class SessionServlet1 extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {

            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet SessionServlet1</title>");
            out.println("</head>");
            out.println("<body>");

          // Fetching the contents of the userName field from the form
            String name = request.getParameter("txtUserName");

            if (name != null && !name.isEmpty()) {
                out.print("<h1>Welcome " + name + "<h1><br/>");

              // Create a new session and set the "uname" attribute
                HttpSession session = request.getSession();
                session.setAttribute("uname", name);
```

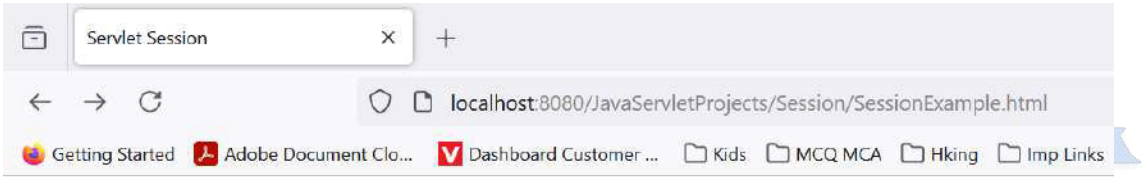
```
            // Link to the second servlet
              out.print("<h3><a href='SessionServlet2'> Visit SessionServlet2
</a></h3>");
                } else {
                    out.println("<h3>Please enter your username from the form
to create a session.</h3>");
                }
                out.println("</body>");
                out.println("</html>");
            }
        }

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

}
```

SessionServlet2.java

```
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.http.HttpSession;
import java.io.IOException;
import java.io.PrintWriter;

public class SessionServlet2 extends HttpServlet {
```

```
protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {

            HttpSession session = request.getSession(false); // Don't
create a new session if it doesn't exist
            String name = null; // Initialize name to null

            if (session != null) {
                name = (String) session.getAttribute("uname"); // Retrieve
the username from the session
            }

            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet SessionServlet2</title>");
            out.println("</head>");
            out.println("<body>");

            if (name != null) {
                out.println("<h1>Hello, " + name + "! Welcome back to
SessionServlet2.</h1>");
            } else {
                out.println("<h3>No user information found. Please start
from SessionServlet1.</h3>");
            }
            out.println("</body>");
            out.println("</html>");
        }
    }
```

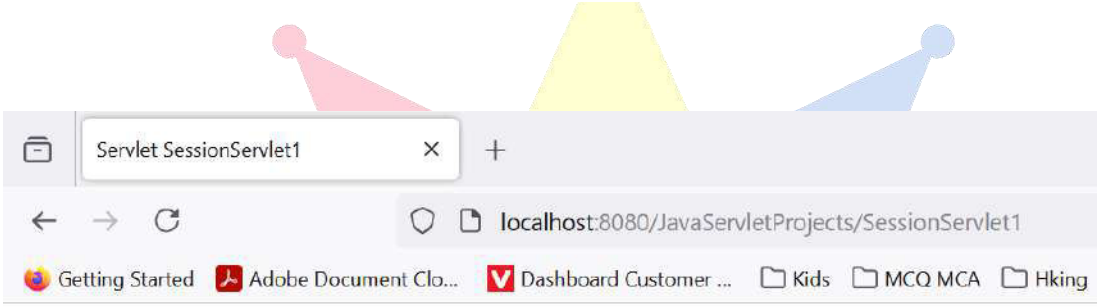
```
@Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        processRequest(request, response);
    }
}
```



Name:

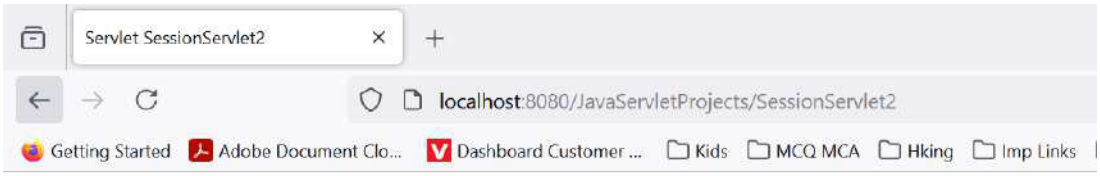
Deepak

submit



Welcome Deepak

Visit SessionServlet2



Hello, Deepak! Welcome back to SessionServlet2.

Key Differences between Session and Cookies

- Sessions are server-side files that store the user information, whereas Cookies are client-side files that contain user information on a local computer.
- Sessions are cookies dependent, whereas Cookies are not dependent on Session.
- The session ends when the user closes the browser or logout from the application, whereas Cookies expire at the set time.
- A session can store as much data as a user want, whereas Cookies have a limited size of 4KB.

Why Use Session?

- Sessions are used to store information such as UserID over the server more securely, where it cannot be tempered.
- It can also transfer the information in the form of value from one web page to another.
- It can be used as an alternative to cookies for browsers that don't support cookies to store variables in a more secure way.

Why use Cookies?

- HTTP is a stateless protocol; hence it does not store any user information. For this purpose, we can use Cookies.
- It allows us to store the information on the user's computer and track the state of applications.

7.2. Java Server Pages: Advantage of JSP technology (Comparison with ASP / Servlet), JSP Architecture, JSP Access Model, JSP Syntax Basic (Directives, Declarations, Expression, Scriptlets, Comments), JSP Implicit Object, Object Scope, Synchronization Issue, Exception Handling, Session Management, Creating and Processing Forms.

Introduction to JSP

Java Server Pages (JSP) is a server-side programming technology that enables the creation of dynamic, platform-independent method for building Web-based applications. JSP have access to the entire family of Java APIs, including the JDBC API to access enterprise databases.

JSP technology is used to create web application just like Servlet technology. It can be thought of as an extension to Servlet because it provides more functionality than servlet such as expression language, JSTL *(Jakarta Standard Tag Library)*, etc.

A JSP page consists of HTML tags and JSP tags. The JSP pages are easier to maintain than Servlet because we can separate designing and development. It provides some additional features such as Expression Language, Custom Tags, etc.

JSTL (JSP Standard Tag Library)

The JSP Standard Tag Library (JSTL) represents a set of tags to simplify the JSP development.

The Jakarta Standard Tag Library ***(JSTL; formerly JavaServer Pages Standard Tag Library)*** is a component of the Java EE Web application development platform. It extends the JSP specification by adding a tag library of JSP tags for common tasks, such as XML data processing, conditional execution, database access, loops and internationalization.

Advantage of JSTL

- Fast Development JSTL provides many tags that simplify the JSP.
- Code Reusability We can use the JSTL tags on various pages.
- No need to use scriptlet tag It avoids the use of scriptlet tag.

Advantage of JSP technology:

- **Simplified Syntax:** JSP allows you to embed Java code directly within HTML, making it easier to learn and implement for developers familiar with HTML.
- **Quick Development:** JSP's tag-based approach and pre-defined functions accelerate the development process compared to pure servlet programming.
- **Reduced Boilerplate Code:** JSP provides implicit objects and standard actions that minimize the amount of code required to perform common tasks.
- **Efficient Compilation:** JSP pages are compiled into servlets, which are then executed efficiently by the server.
- **Scalability:** JSP is well-suited for high-traffic websites, as it can handle multiple concurrent requests efficiently.
- **Caching Mechanisms:** JSP supports various caching techniques to improve performance and reduce server load.
- **Robust Security:** JSP inherits the security features of the Java platform, providing strong protection against common web vulnerabilities.
- **Reliable Platform:** JSP is built on a stable and mature platform, ensuring reliable and consistent performance.
- **Seamless Integration:** JSP can be easily integrated with other Java technologies like servlets, EJBs, and JDBC to create complex web applications.
- **Custom Tag Libraries:** You can create custom tag libraries to encapsulate reusable functionality, promoting code modularity and reusability.
- **Write Once, Run Anywhere:** JSP applications can be deployed on any platform that supports the Java Virtual Machine (JVM), ensuring portability and flexibility.

Comparison with ASP / Servlet:

Feature	JSP (JavaServer Pages)	ASP (Active Server Pages)	Servlets
Language	Java	VBScript, JScript (primarily on Windows platforms)	Java
Platform Independence	Platform-independent; runs on any server supporting Java	Primarily Windows-based, requires IIS	Platform-independent; runs on any server supporting Java
Integration with Web Server	Runs on Java-enabled servers (Tomcat, WebLogic, etc.)	Primarily designed for IIS on Windows	Runs on Java-enabled servers
Compilation	Compiled into Servlets, improving performance	Interpreted each time the page is requested, slower performance	Pre-compiled, higher performance
Separation of Code and HTML	Good separation; uses tags to embed Java code in HTML	Typically mixes HTML and VBScript, leading to less separation	Less separation; entire logic is embedded in Java code
Reuse of Components	Uses JavaBeans and tag libraries for code reuse	Limited support for component reuse	Reusable through encapsulated Java methods and classes
Object-Oriented Support	Fully object-oriented, leveraging Java's OO capabilities	Limited OO support, lacks features like inheritance	Fully object-oriented, leveraging Java's OO capabilities
Error Handling	Exception handling via try-catch and error pages	Error handling with On Error Resume Next, limited control	Exception handling via try-catch
Session Management	Built-in session management via session object	Supports session management, but limited to IIS functionality	Built-in session management via session object
Tag Libraries	Supports JSP tag libraries (e.g., JSTL)	Lacks standardized tag libraries	Does not support tag libraries
Extensibility	Highly extensible; can integrate with Java EE components	Less extensible outside of the Microsoft ecosystem	Highly extensible; part of Java EE stack
Thread Safety	Multithreaded by	Thread-safe on IIS	Multithreaded, thread-

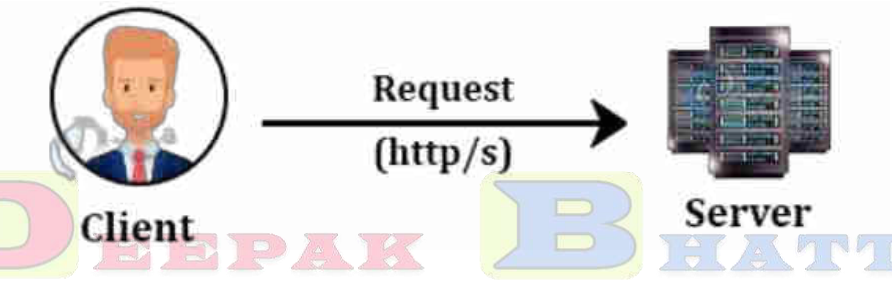
	default; needs synchronization handling		safety needs handling
Database Access	JDBC (Java Database Connectivity)	ODBC, ADO (ActiveX Data Objects)	JDBC
Cost	Free, open-source servers available (e.g., Apache Tomcat)	Typically requires licensing for Windows and IIS	Free, with open-source servers available
Lifecycle Management	Managed by the Servlet container	Managed by IIS	Managed by the Servlet container
Use Cases	Ideal for dynamic web pages with complex backend logic	Simple dynamic pages on Windows environments	Suitable for request handling and backend processing

JSP Architecture:

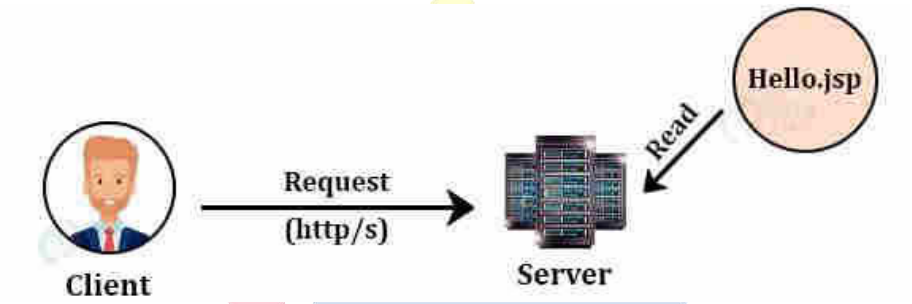
1. **JSP Page**: Written in HTML with JSP tags.
2. **JSP Engine**: Translates the JSP page into a Servlet.
3. **Servlet Engine**: Manages the lifecycle of the Servlet.
4. **JavaBeans / Enterprise JavaBeans (EJB)**: Can be used to encapsulate business logic.
5. **Database**: Interactions with the database can be managed via JDBC.

Following is the architecture of Java server pages:

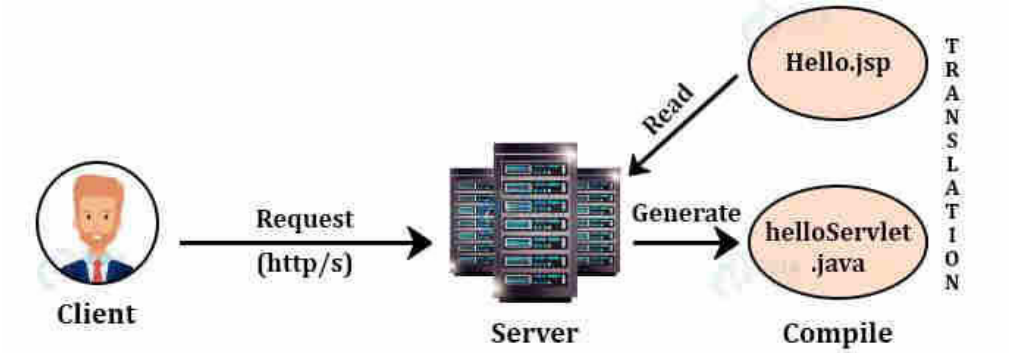
1. Firstly, Client sends an http request to the server. As with a normal page, your browser sends an http request to the web browser. For e.g. authentication windows where you enter user id and password in the system and click the submit button, the client thus sends this information to the server to check whether it is valid or not.



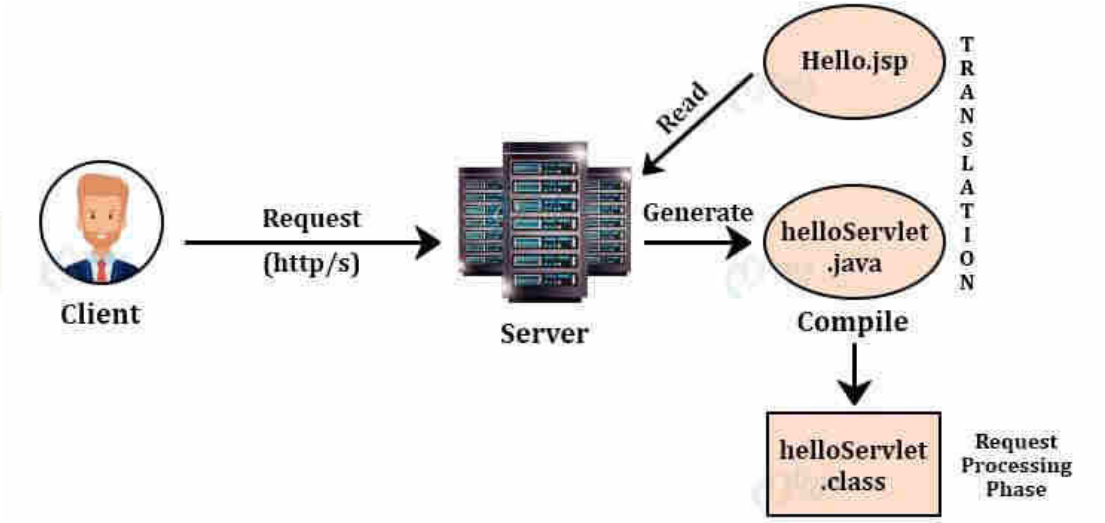
2. Web server recognizes the request made by the client that is it for a JSP page or any other HTML extension page. If the request made by the client is for a JSP page, then the Web server forwards it to the JSP engine. JSP engine processes the page. This is by using URL or JSP page which ends with **.jsp** extension.



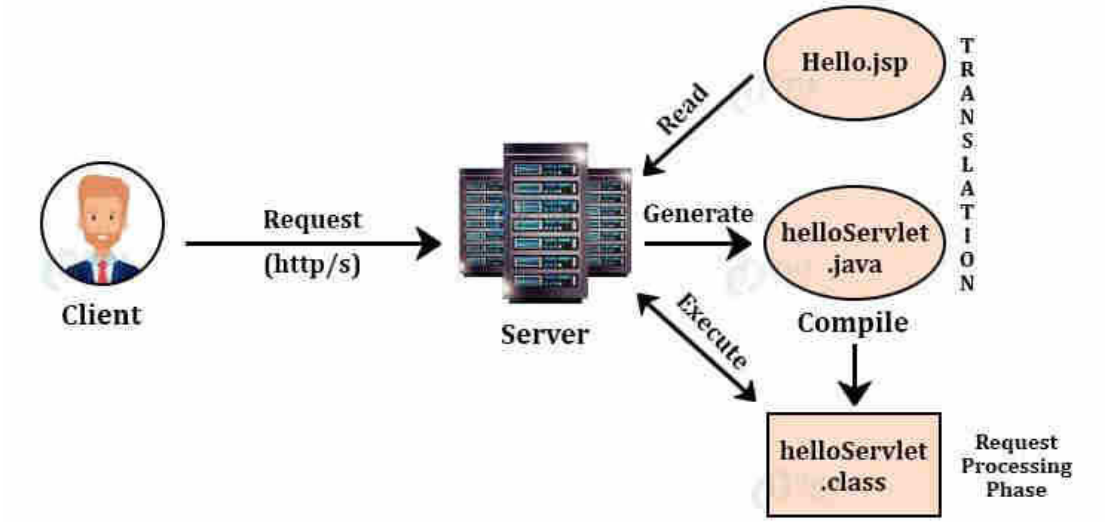
3. Then comes the translation phase. This phase is of utmost importance as the JSP engine loads the JSP page, and converts it into servlet content i.e. all the template text converts to println() statements and all JSP elements convert to Java code. As in the diagram **hello.jsp** gets converted to **helloServlet.java**. It is important to note that jsp content gets converted to java code in the translation phase.



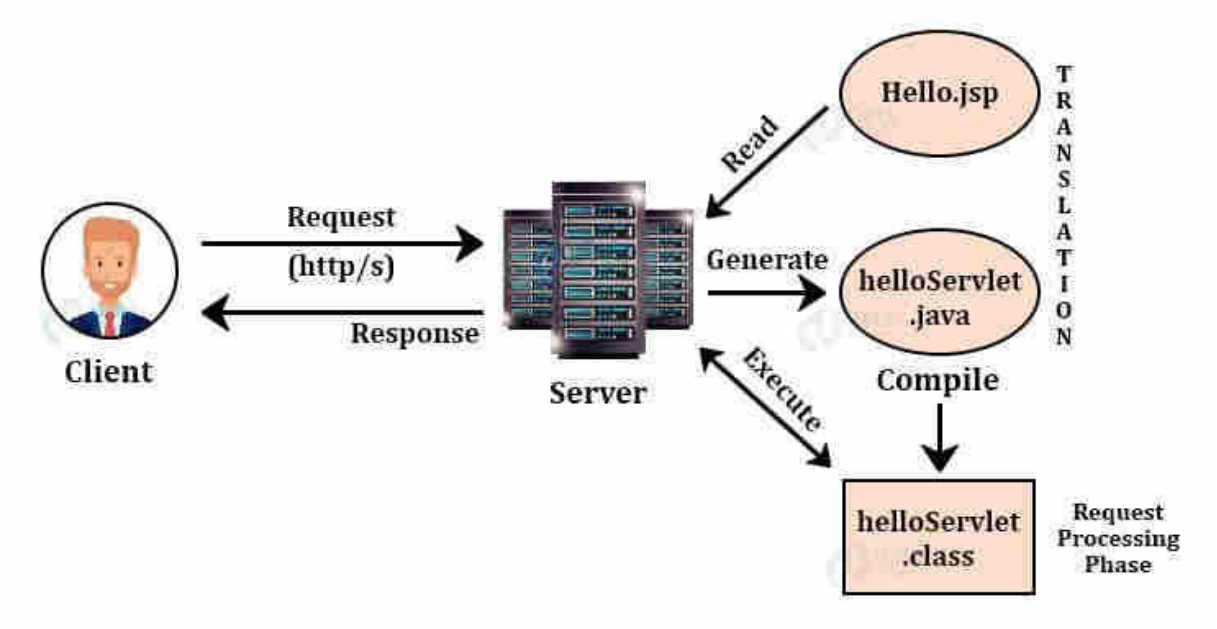
4. Translation phase is followed by the compilation phase. JSP engine compiles servlet into .class files that can execute. It forwards the initial request to the servlet engine. As shown in the diagram **hello.jsp** gets translated to helloServlet.java and then compiled to a class file called **helloSevlet.class**. This phase is also called the request processing phase as the request is getting processed here.



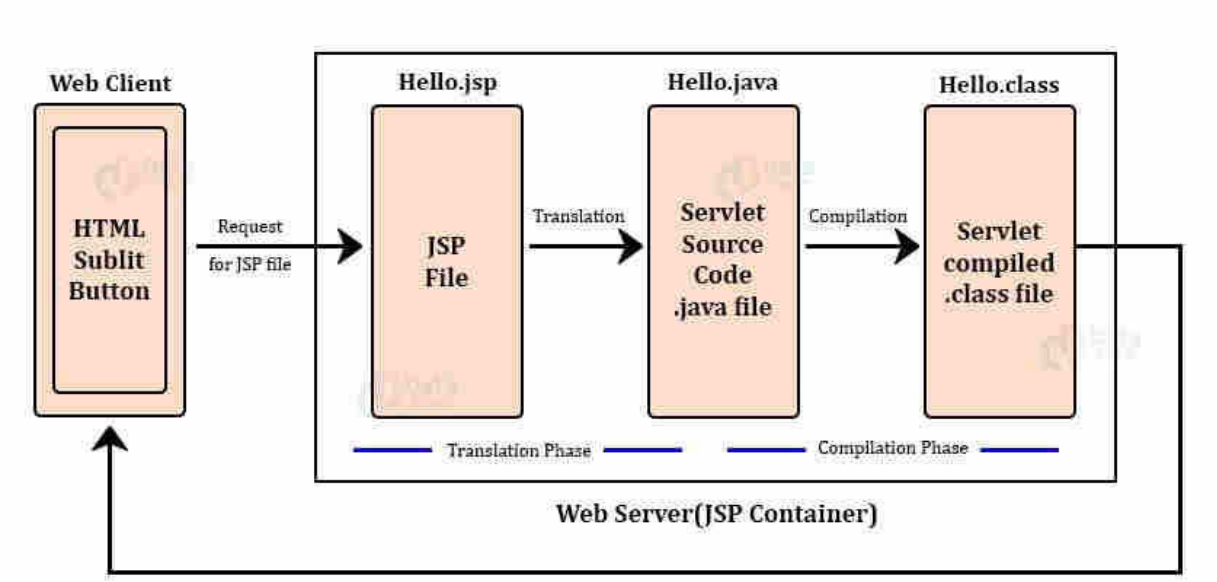
5. Thereafter the servlet class is loaded by a part of the web container, namely servlet engine, and This class file is then executed. The output is finally produced in HTML format after execution. The output is sent in an HTTP response to the web server by the servlet engine. The diagrams show the complete process.



6. Then The web server forwards the response to the client's browser. This HTTP response is in the form of static HTML content as it is more user friendly and easily understandable.



7. Finally, the dynamically generated web content is handled statically by the web browser. Then the HTML page inside the HTTP response is handled as if it is a static page.



JSP Access Model:

JSP primarily uses two architectural models to handle requests and generate responses:

1. JSP Model 1 Architecture (JSP-Centric):

- **Direct Request:** The client's request is directly sent to a JSP page.
- **Processing:** The JSP page handles both presentation logic and business logic.
- **Data Access:** It may interact with JavaBeans or EJBs to access and manipulate data.
- **Response Generation:** The JSP generates the final HTML response and sends it back to the client.

Advantages:

- Simple to implement.
- Suitable for small-scale applications.

Disadvantages:

- Tight coupling between presentation and business logic.
- Difficult to maintain and test as the application grows.

2. JSP Model 2 Architecture (MVC-Based):

- **Request Handling:** The client's request is initially handled by a servlet or controller.
- **Business Logic:** The controller processes the request, invokes business logic (often implemented in JavaBeans or EJBs), and retrieves data.
- **View Selection:** The controller selects the appropriate JSP page to render the view.
- **Response Generation:** The JSP page uses the data provided by the controller to generate the final HTML response.

Advantages:

- Clear separation of concerns (Model, View, Controller).
- Improved code organization and maintainability.
- Better testability.
- Scalability and flexibility.

Disadvantages:

- Increased complexity compared to Model 1.
- Requires more configuration and setup.

Choosing the Right Model:

The choice between Model 1 and Model 2 depends on the specific requirements of your application:

- **Model 1:** Suitable for small, simple applications where the business logic is straightforward and the presentation layer is not complex.
- **Model 2:** Ideal for large, complex applications that require a clear separation of concerns, improved maintainability, and scalability.

Key Differences:

Feature	Model 1	Model 2
Request Handling	JSP	Servlet
Data Access	JSP	Servlet
Business Logic	JSP	Servlet
View Rendering	JSP	JSP
Complexity	Simpler	More Complex
Scalability	Less Scalable	More Scalable
Maintainability	Less Maintainable	More Maintainable

Key Considerations:

- **Scalability:** Model 2 is generally more scalable due to its modular architecture.
- **Maintainability:** Model 2 promotes better code organization and easier maintenance.
- **Testability:** Model 2 allows for unit testing of individual components.
- **Performance:** While Model 2 can introduce additional overhead, it can be optimized for performance through caching and other techniques.

JSP Syntax Basic

Directives:

The **jsp directives** are messages that tells the web container how to translate a JSP page into the corresponding servlet.

There are three types of directives:

1. page directive
2. include directive
3. taglib directive

Syntax of JSP Directive

```
<%@ directive attribute="value" %>
```


1. JSP page directive:

The page directive defines attributes that apply to an entire JSP page.

Syntax of JSP page directive

```
<%@ page attribute="value" %>
```

Attributes of JSP page directive

- ✓ import
- ✓ contentType
- ✓ extends
- ✓ info
- ✓ buffer
- ✓ language
- ✓ isELIgnored
- ✓ isThreadSafe
- ✓ autoFlush
- ✓ session
- ✓ pageEncoding
- ✓ errorPage
- ✓ isErrorPage

2. JSP include directive:

The include directive is used to include the contents of any resource it may be **jsp** file, **html** file or **text** file. The include directive includes the original content of the included resource at page translation time (*the jsp page is translated only once so it will be better to include static resource*). The advantage of Include directive is code reusability.

Syntax of include directive

```
<%@ include file="resourceName" %>
```

Example of include directive

In this example, we are including the content of the **header.html** file. To run this example you must create an header.html file.

```
<html>
   <body>
      <%@ include file="header.html" %>
      Today is: <%= java.util.Calendar.getInstance().getTime() %>
   </body>
</html>
```

Note: The include directive includes the original content, so the actual page size grows at runtime.

3. JSP Taglib Directive:

The JSP taglib directive is used to define a tag library that defines many tags. We use the TLD (*Tag Library Descriptor*) file to define the tags. In the custom tag section, we will use this tag so it will be better to learn it in custom tag.

Syntax JSP Taglib directive

```
<%@ taglib uri="uriofthetaglibrary" prefix="prefixoftaglibrary" %>
```

Example of JSP Taglib directive

In this example, we are using our tag named currentDate. To use this tag, we must specify the taglib directive so the container may get information about the tag.

```
<html>
   <body>
      <%@ taglib uri="http://www.javatpoint.com/tags" prefix="mytag" %>
      <mytag:currentDate/>
   </body>
</html>
```

Declarations:

The **JSP declaration tag** is used *to declare fields and methods*.

The code written inside the jsp declaration tag is placed outside the **service()** method of auto generated servlet. So, it doesn't get memory at each request.

Syntax of JSP declaration tag

The syntax of the declaration tag is as follows:

```
<%!  field or method declaration %>
```

Example of JSP declaration tag that declares field

In this example of JSP declaration tag, we are declaring the field and printing the value of the declared field using the jsp expression tag.

File: index.jsp

```
<html>
   <body>
      <%! int data=50; %>
      <%= "Value of the variable is: "+data %>
   </body>
</html>
```

Example of JSP declaration tag that declares method

In this example of JSP declaration tag, we are defining the method which returns the cube of given number and calling this method from the jsp expression tag. But we can also use jsp scriptlet tag to call the declared method.

File: Cube.jsp

```
<html>
   <body>
      <%!
        int cube(int n){
        return n*n*n*;
        }
      %>
      <%= "Cube of 3 is: " +cube(3) %>
   </body>
</html>
```

Creating a simple JSP Page

To create the first JSP page, write some HTML code as given below, and save it by .jsp extension. We have saved this file as **addition.jsp**. To run the JSP page you need a server, here I have use Glassfish server, simply right click on the code view of JSP file and click on Run File or **(Shift +F6)**.

File: addition.jsp

Let's see the simple example of JSP where we are using the scriptlet tag to put Java code in the JSP page. We will learn scriptlet tag later.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Addition</title>
    </head>
    <body>
        <h1>Hello World!</h1>
        <%!
            int a = 20;
            int b = 30;
            int sum = a + b;
        %>
        <% out.println("The addition of " + a + " and " + b + " is : " + sum); %>
    </body>
</html>
```

How to run a simple JSP Page?

Follow the following steps to execute this JSP page:

- Start the server
- Put the JSP file in a folder and deploy on the server

Visit the browser by the URL `http://localhost:portno/contextRoot/jspfile`,

For example, `http://localhost:8080/Addition/addition.jsp`

Expression:

The code placed within **JSP expression tag** is *written to the output stream of the response*. So you need not write **out.print()** to write data. It is mainly used to print the values of variable or method.

Syntax of JSP expression tag

```
<%=  statement %>
```

Example of JSP expression tag

In this example of jsp expression tag, we are simply displaying a welcome message.

```
<html>
   <body>
      <%= "welcome to jsp" %>
   </body>
</html>
```

Note: Do not end your statement with semicolon in case of expression tag.

Example of JSP expression tag that prints current time

To display the current time, we have used the **getTime()** method of Calendar class. The **getTime()** is an instance method of Calendar class, so we have called it after getting the instance of Calendar class by the **getInstance()** method.

File: ShowTime.jsp

```
<html>
   <body>
      Current Time: <%= java.util.Calendar.getInstance().getTime() %>
   </body>
</html>
```

Example of JSP expression tag that prints the user name

In this example, we are printing the username using the expression tag. The index.html file gets the username and sends the request to the **welcome.jsp** file, which displays the username.

File: index.jsp

```
<html>
   <body>
      <form action="welcome.jsp">
         <input type="text" name="txtUserName"><br/>
         <input type="submit" value="Submit">
      </form>
   </body>
</html>
```

File: welcome.jsp

```
<html>
   <body>
      <%= "Welcome "+request.getParameter("txtUserName") %>
   </body>
</html>
```

Scriptlets:

Scriptlets in JSP allow you to embed Java code within HTML. They are enclosed within `<% ... %>` tags and are used to add dynamic content to a webpage.

```
<%
        // Declare a variable for the user's name
         String userName = "Deepak";
        // Display a welcome message
         out.println("<h2>Welcome, " + userName + "!</h2>");
<%
```

Comments:

JSP comments are not sent to the client.

```
      <%-- This is a JSP comment --%>
```

JSP Implicit Object:

JSP provides several implicit objects that can be accessed directly without being explicitly declared:

1. **request**: Represents the HTTP request.
2. **response**: Represents the HTTP response.
3. **out**: Used to send output to the client.
4. **session**: Represents the current session.
5. **application**: Represents the entire web application context.
6. **config**: Provides configuration information.
7. **page**: Refers to the current JSP page.
8. **pageContext**: Provides access to various context information.

Object Scope:

JSP defines four scopes for storing objects:

1. **Page Scope**: Accessible only within the current JSP page.
2. **Request Scope**: Accessible during the request-response cycle.
3. **Session Scope**: Accessible throughout a user session.
4. **Application Scope**: Accessible across the entire web application.

Synchronization Issue:

- **Thread Safety**: JSP pages are multithreaded by default, meaning multiple users can access the same page simultaneously.
- **Synchronization**: To prevent concurrency issues, you can use synchronized methods or blocks, or manage data through session scope.

Exception Handling:

JSP supports exception handling via:

1. Using errorPage and isErrorPage Directives

JSP allows you to specify error pages that handle exceptions automatically by defining an error page using the errorPage attribute in the JSP directive.

Step 1: Setting Up the Error Page (errorPage.jsp)

```
<%@ page isErrorPage="true" %>
<!DOCTYPE html>
<html>
<head>
    <title>Error Page</title>
</head>
<body>
    <h2>Oops! An error occurred.</h2>
    <p>Error Message: <%= exception.getMessage() %></p>
    <p>Please try again later or contact support.</p>
</body>
</html>
```

Step 2: Linking to the Error Page in the Main JSP (mainPage.jsp)

```
<%@ page errorPage="errorPage.jsp" %>
<!DOCTYPE html>
<html>
<head>
    <title>Main Page</title>
</head>
<body>
    <h2>Welcome to the Main Page</h2>
    <%
        // Example: Force an exception for demonstration
         int result = 10 / 0; // This will throw an ArithmeticException
    %>
</body>
</html>
```

2. Using try-catch Blocks

You can handle exceptions directly within JSP using **try-catch** blocks. This is useful for handling specific types of errors and providing alternative logic if an exception occurs.

```
<!DOCTYPE html>
<html>
<head>
    <title>Try-Catch Example</title>
</head>
<body>
    <h2>Calculating Division Result</h2>
    <%
        try {
            int numerator = 10;
            int denominator = 0; // Example to trigger an exception
            int result = numerator / denominator;
            out.println("Result: " + result);
        } catch (ArithmeticException e) {
            out.println("Error: Division by zero is not allowed.");
        } catch (Exception e) {
            out.println("An unexpected error occurred: " + e.getMessage());
        }
    %>
</body>
</html>
```


Session Management:

- **Session Creation**: Automatically created when a user accesses the JSP.
- **Accessing Session**: You can access the session using the session implicit object.
- **Managing Attributes**: You can set, get, and remove session attributes using:

```
session.setAttribute("username", "Deepak");
String username = (String) session.getAttribute("username");
session.removeAttribute("username");
```

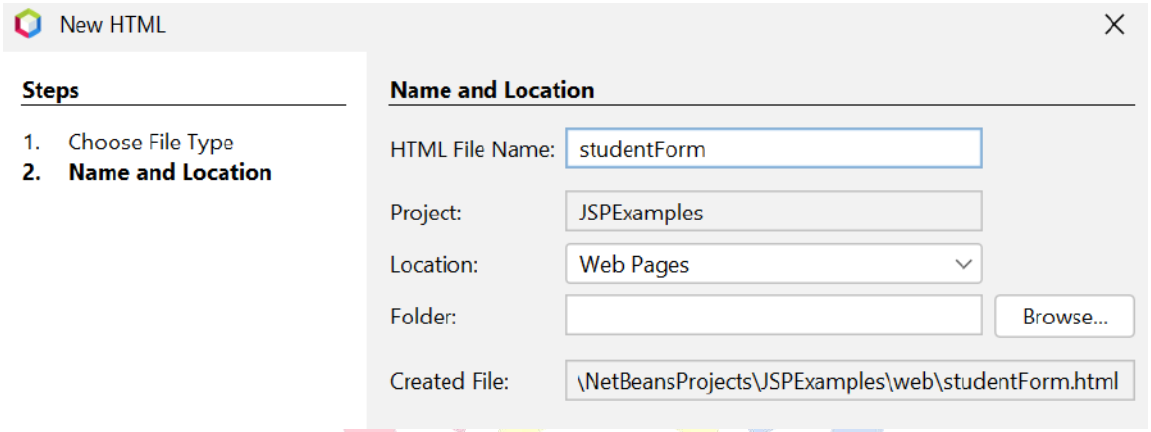
Creating and Processing Forms:

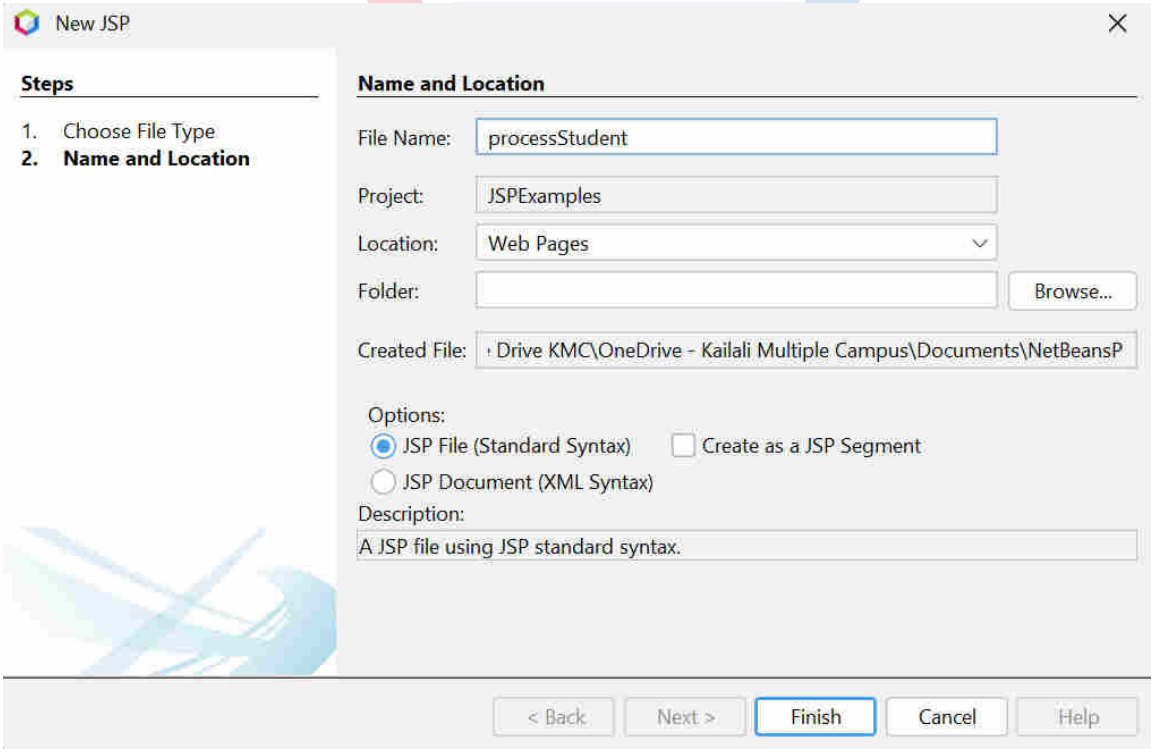
Creating Forms:

```
<form action="process.jsp" method="post">
    Name: <input type="text" name="name" />
    <input type="submit" value="Submit" />
</form>
```

Processing Forms: In **process.jsp**, you can retrieve form data:

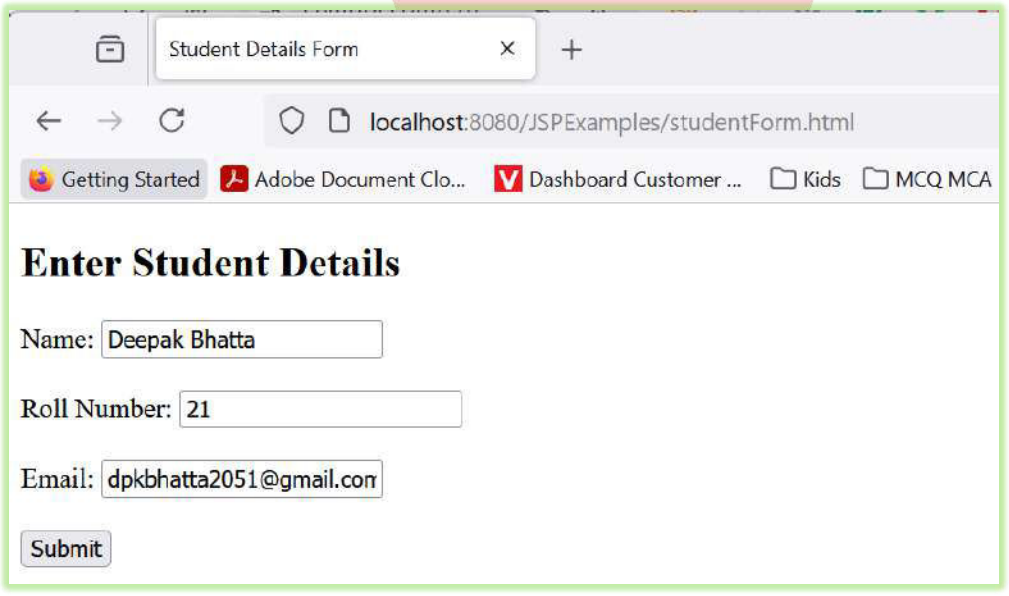
```
String name = request.getParameter("name");
out.println("Hello, " + name);
```



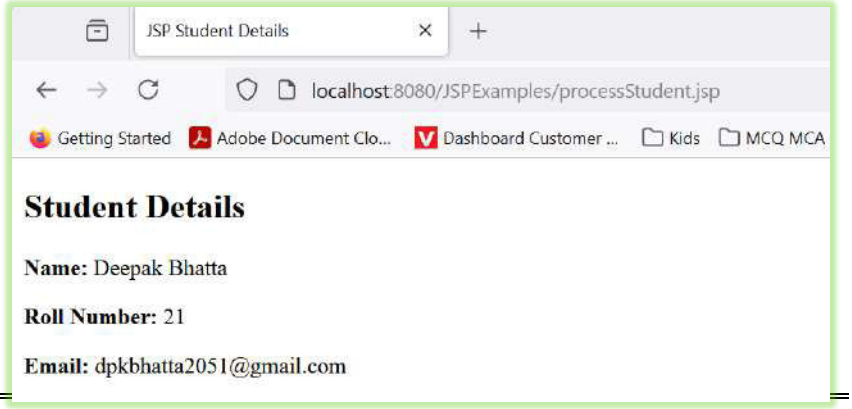
1. Creating the Form (studentForm.jsp)

```
<!DOCTYPE html>
<html>
    <head>
        <title>Student Details Form</title>
    </head>
    <body>
        <h2>Enter Student Details</h2>
        <form action="processStudent.jsp" method="post">
            <label>Name:</label>
            <input type="text" id="name" name="name" required /><br><br>
            <label>Roll Number:</label>
   <input type="text" id="rollNumber" name="rollNumber" required /><br><br>
            <label>Email:</label>
            <input type="email" id="email" name="email" required /><br><br>
            <input type="submit" value="Submit" />
        </form>
    </body>
</html>
```



2. Processing the Form (processStudent.jsp)

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Student Details</title>
    </head>
    <body>
        <h2>Student Details</h2>
        <%
           // Retrieving form data
            String name = request.getParameter("name");
            String rollNumber = request.getParameter("rollNumber");
            String email = request.getParameter("email");

           // Displaying the retrieved data
            if (name != null && rollNumber != null && email != null) {
        %>
        <p><strong>Name:</strong> <%= name %></p>
        <p><strong>Roll Number:</strong> <%= rollNumber %></p>
        <p><strong>Email:</strong> <%= email %></p>
        <%
            } else {
                out.println("<p>Error: Missing form data.</p>");
            }
        %>
    </body>
</html>
```



THE END

Unit 8 — 5 hrs.

RMI & CORBA

Specific Objectives

- Explain basics of RMI and CORBA.
- Write, Compile, and Execute sample RMI programs.
- Understand CORBA and its architecture.

8.1. Remote Method Invocation: Introduction of RMI, Architecture of RMI, Remote Objects, Creating and Executing RMI Applications.

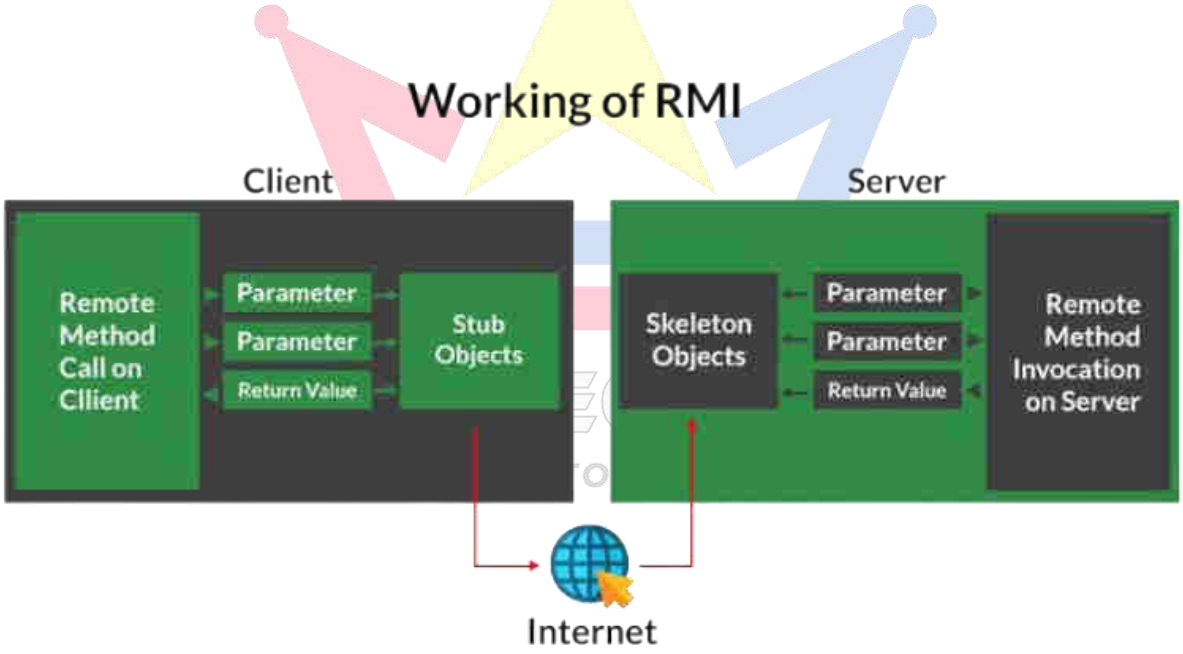
8.2. CORBA: Introduction to CORBA, Architecture of CORBA, Functioning of CORBA Applications, CORBA Service.

8.1. Remote Method Invocation: Introduction of RMI, Architecture of RMI, Remote Objects, Creating and Executing RMI Applications.

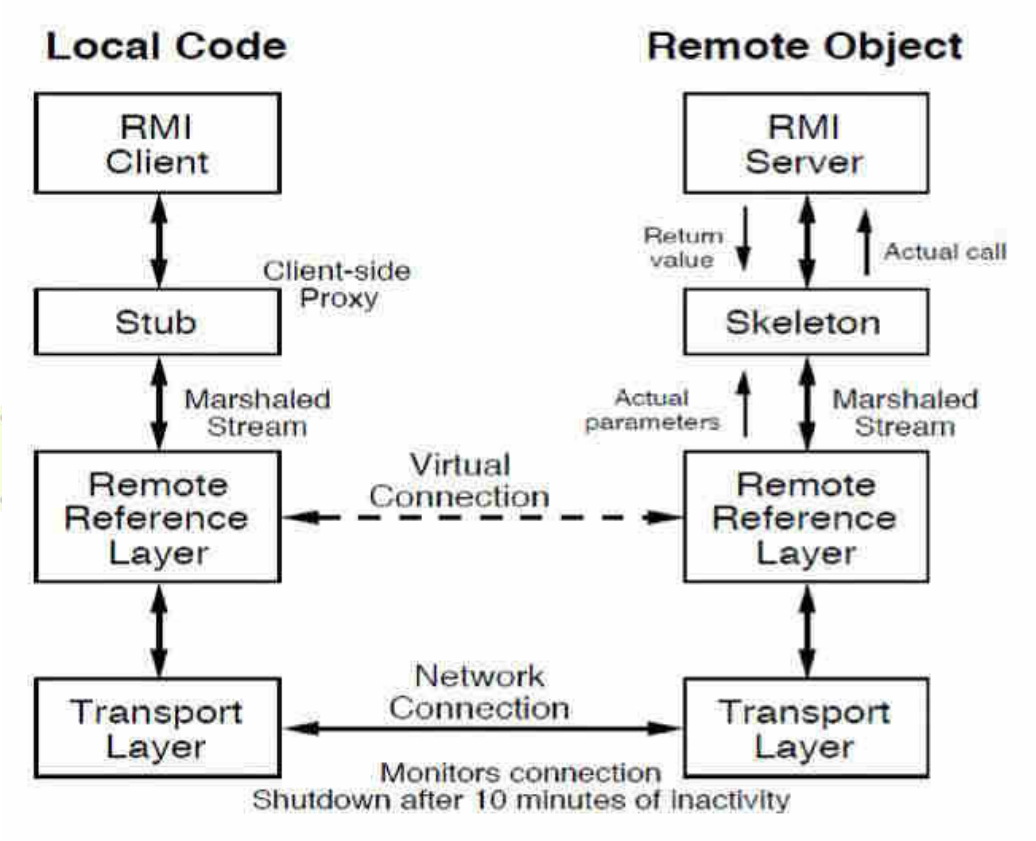
Introduction to RMI:

Remote Method Invocation (RMI) in Java is a framework that allows an object residing in one Java Virtual Machine (JVM) to invoke methods on an object located in another JVM. RMI is part of Java's standard library, and it simplifies the development of distributed applications by enabling remote communication between Java programs.

- RMI provides a way for Java programs to interact across networked computers, making it possible for a client program to invoke methods on a remote server object.
- RMI offers platform independence, as it is purely written in Java, and uses a registry service to manage remote objects.
- RMI handles the communication protocols and allows objects to be serialized, meaning objects are converted to a byte stream for transmission over the network.



Architecture of RMI:



The RMI architecture consists of three main layers:

- **Application Layer**
  - ✓ It's a responsible for the actual logic (implementation) of the client and server application.
  - ✓ Generally, at the server-side class contain implementation logic and also apply the reference to the appropriate object as per the requirement of the logic in application.
- **Stub and Skeleton Layer**:
  - ✓ ***Stub*** acts as a client-side proxy, forwarding the client's remote method calls to the server. It serializes parameters before sending them over the network.
  - ✓ ***Skeleton*** resides on the server side, receiving incoming method calls from the client, deserializing parameters, and invoking the appropriate method on the actual remote object.

- **Remote Reference Layer**:
  - ✓ It's a responsible for manage the references made by the client to the remote object on the server so it is available on both JVM (Client and Server).
  - ✓ Manages references to remote objects and is responsible for handling invocation and communication semantics. It hides the underlying complexity of the transport layer.
  - ✓ The Client side RRL receives the request for methods from the Stubs that is transferred into byte stream process called serialization (Marshaling) and then these data are send to the Server side RRL.
  - ✓ The Server side RRL doing reverse process and convert the binary data into object. This process called deserialization (Unmarshaling) and then sent to Skeleton class.
- **Transport Layer**:
  - ✓ It's also called the "Connection layer".
  - ✓ It's a responsible for the ,managing the existing connection and also setting up new connections.
  - ✓ Handles the low-level networking and manages the communication between the client and server, usually over TCP/IP. It is responsible for connection establishment, data transfer, and disconnection.

Remote Objects:

- A *remote object* in RMI is an object whose methods can be called from another JVM.
- These objects must implement a remote interface, which extends **java.rmi.Remote**.
- The interface defines the methods that can be invoked remotely, and these methods must throw a **RemoteException**.

Creating and Executing RMI Applications:

Steps to create and execute an RMI application involve:

Define a Remote Interface:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface MyRemote extends Remote {
    String sayHello() throws RemoteException;
}
```

Implement the Remote Interface:

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    public MyRemoteImpl() throws RemoteException {}

    public String sayHello() {
        return "Hello from the RMI server!";
    }
}
```

Create and Register the Remote Object:

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Server {
    public static void main(String[] args) {
        try {
            MyRemoteImpl obj = new MyRemoteImpl();
            Registry registry = LocateRegistry.createRegistry(1099);
```

```
            registry.rebind("HelloService", obj);
            System.out.println("Server is running...");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Create a Client to Access the Remote Object:

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {
    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.getRegistry("localhost",
1099);
            MyRemote stub = (MyRemote) registry.lookup("HelloService");
            System.out.println("Response from server: " + stub.sayHello());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```


Compile and Run the Application:

- Compile all Java files.
- Start the **rmiregistry**.
- Start the server program to register the remote object.
- Run the client program to invoke the remote methods.

8.2. CORBA: Introduction to CORBA, Architecture of CORBA, Functioning of CORBA Applications, CORBA Service.

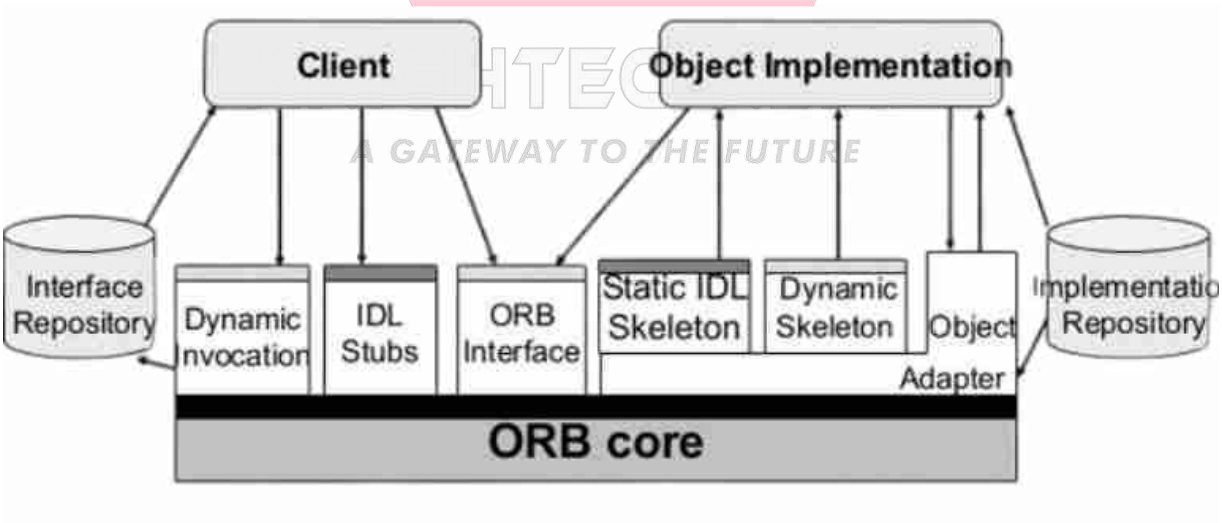
Introduction to CORBA:

Common Object Request Broker Architecture (CORBA) is a standard developed by the Object Management Group (OMG) for enabling communication between applications written in different programming languages and running on diverse platforms. CORBA provides a way to create distributed, language-independent applications where objects can communicate over a network.

- CORBA defines a mechanism that allows programs to send requests and receive responses from objects located on other computers in a language-agnostic way.
- This is achieved through the use of an Object Request Broker (ORB), which acts as the middleware layer handling communication, method invocation, and object location.
- CORBA uses the Interface Definition Language (IDL) to define the interfaces that objects present to the outside world, which can then be implemented in any supported programming language.

Architecture of CORBA:

The general architecture is shown in given figure:



1. Interface Repository:

- It provides representation of available object interfaces of all objects.
- It is used for dynamic invocation.

2. Implementation Repository:

- It stores implementation details for each object's interfaces. (Mapping from server object name to filename to implement service)
- The information stored may be OS specific.
- It is used by object adapter to solve incoming call and activate right object method.

3. Object Request Broker (ORB):

- It provides mechanisms by which objects can interact with each other transparently.

4. Static Invocation:

- It allows a client to invoke requests on an object whose compile time knowledge of server's interface specification is known.
- For client, object invocation is similar to local method of invocation, which automatically forwarded to object implementation through ORB, object adapter and skeleton.
- It has low overhead, and is efficient at run time.

5. Dynamic Invocation:

- It allows a client to invoke requests on object without having compile time knowledge of object's interface.
- The object and interface are detected at run time by inspecting the interface repository.
- The request is then constructed and invocation is performed as it static invocation.
- It has high overhead.

6. Object Adapter:

- It is the interface between server object implementation and ORB.

Functioning of CORBA Applications:

CORBA applications function through a well-defined process that involves several key components and interactions.

1. Interface Definition:

- Developers define the interfaces of objects using the Interface Definition Language (IDL). This language-neutral specification describes the operations and data types that objects can handle.

2. Compilation:

- IDL compilers generate language-specific stubs and skeletons from the IDL definitions.
    a. **Stubs:** Client-side code that provides a local interface to the remote object.
    b. **Skeletons:** Server-side code that handles incoming requests and forwards them to the actual object implementation.

3. Object Implementation:

- Developers implement the objects in their chosen programming language, adhering to the specified interfaces.

4. Deployment:

- Objects are deployed on servers, and the Object Request Broker (ORB) is configured to locate and manage them.

5. Client-Server Interaction:

- A client application obtains a reference to a remote object using the naming service.
- The client invokes operations on the object using the generated stubs.

CORBA Service:

CORBA provides a range of standardized services, known as CORBA services, to support distributed applications. Some key services include:

1. Naming service

- ✓ It allows clients to find and locate objects based on name.

2. Trading service

- ✓ It allows clients to find and locate objects based on their properties.

3. Notification service

- ✓ It allows objects to notify other objects that some event has occured.

4. Transaction Service

- ✓ It allows atomic transactions and rollback on failures.

5. Security Service

- ✓ It protects components from unauthorized access or users.

6. Concurrency control service

- ✓ It provides a lock manager that can obtain and free locks for transactions to manage concurrent transactions.

7. Life cycle service

- ✓ It defines conventions for creating, deleting, copying and moving CORBA objects.

8. Time service

- ✓ It provides interfaces for synchronizing time.

THE END