USN | | | | | | | | | | |

Internal Assessment Test 2 – May 2025

Sub	Advanced Java					Sub Code:	BCS613D	Branch:	CSE
Date:	24/05/2025	Duration:	90 mins	Max Marks:	50	Sem / Sec:	VI Professional Elective		OBE

	Answer any FIVE FULL Questions	MARKS	CO	RBT
1 (a)	Explain a simple Swing application with example program	[05]	CO3	L2, L3
(b)	Create a Swing application that has two Buttons named "CSE" and "ISE", when either of the buttons is clicked it should display "CSE Pressed" and "ISE Pressed" respectively.	[05]	CO3	L3
2 (a)	Explain the components and containers used in Swings with key features.	[05]	CO3	L2
(b)	Explain how components and container work together with a program	[05]	CO3	L2
3	What is Servlet? Explain the life cycle of servlets.	[10]	CO4	L3
4	Explain core classes and interfaces in javax.servlet package.	[10]	CO4	L2
5 (a)	Explain scrollableResultSet and UpdatableResultSet Object in JDBC with example program	[07]	CO5	L2
(b)	Explain the types of exceptions that occur in JDBC	[03]	CO5	L2
6	List and explain various statement objects in JDBC with example program	[10]	CO5	L2

Solution

1.Explain a simple Swing application with example program

Ans: **A Simple Swing Application**

A Swing application is a Java program that uses the Swing library to create a graphical user interface (GUI). Unlike console-based programs, a Swing app displays windows, buttons, and other visual elements users can interact with. This section explains a simple Swing application step-by-step, showing how to create a basic window with a label. It's the starting point for learning Swing and demonstrates core concepts like containers, components, and setup.

It uses two Swing components:
JFrame - top level container that is commonly used for Swing applications.
JLabel - JLabel - is the Swing component that creates a label, which is a component that displays information. The label is Swing's simplest component because it is passive. It does not respond to user input. It just displays output.

This program uses a JFrame container to hold an instance of a JLabel. The label displays a short text message.

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

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

        // Step 1: Create a new JFrame container
        JFrame frame = new JFrame("My First Swing App");

        // Step 2: Give the frame an initial size
        frame.setSize(300, 200);

        // Step 3: Terminate the program when the user closes the application
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Step 4: Create a text-based label.
        JLabel label = new JLabel("Hello, Swing!");

        // Step 5: Add the label to the content pane
        frame.add(label);

        // Step 6: Display the frame.
        frame.setVisible(true);
    }
}
```

Detailed Explanation

Line: import javax.swing.*;
- This imports the core Swing package, giving access to classes like JFrame and JLabel.
- The javax.swing package contains Swing's components and containers, essential for GUI building.
- The * means all classes in this package are available, though here we only use JFrame and JLabel.

Line: public class SimpleSwingApp {
- Defines a public class named SimpleSwingApp. Every Java program needs a class, and this is ours.
- The name can be anything, but it must match the file name (SimpleSwingApp.java).

Line: public static void main(String[] args) {
- The main method is the entry point for the program, where execution starts.
- It's static so Java can run it without creating an object, and args is for command-line arguments (unused here).

Line: JFrame frame = new JFrame("My First Swing App");
- Creates a JFrame object, the top-level container (window) for the app.
- JFrame comes from javax.swing and is a lightweight wrapper around AWT's Frame class.
- The string "My First Swing App" sets the window's title, shown in the title bar.
- frame is the variable name we'll use to refer to this window.

Line: frame.setSize(300, 200);
- Sets the window's size to 300 pixels wide and 200 pixels tall.
- The setSize method comes from AWT's Component class, which JFrame inherits.
- Without this, the window would be tiny and empty.

Line: frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
- Tells the app what to do when the user clicks the window's close button (the X).
- JFrame.EXIT_ON_CLOSE means the program exits completely when closed.
- Other options include DO_NOTHING_ON_CLOSE (ignore the click) or HIDE_ON_CLOSE (hide the window), but EXIT_ON_CLOSE is standard for simple apps.
- Without this, closing the window might leave the program running invisibly.

Line: JLabel label = new JLabel("Hello, Swing!");
- Creates a JLabel, a lightweight Swing component that displays text or images.
- The string "Hello, Swing!" is the text to show.
- label is the variable name for this component.
- JLabel is simple but versatile—here it's just text, but it could also display an icon.

Line: frame.add(label);
- Adds the JLabel to the JFrame so it appears in the window.
- The add method comes from AWT's Container class, which JFrame inherits.

- By default, Swing places the component in the center of the frame's content pane (a special area inside the JFrame) if no layout is specified.
- A JFrame can hold multiple components, but this simple example uses just one.

Line: frame.setVisible(true);
- Makes the window visible on screen.
- Until this line, the JFrame exists in memory but isn't shown.
- The true argument tells Swing to display it; false would hide it.
- This is always the last setup step to ensure everything (size, components) is ready before showing.

How It Works
When you run this code:
1. Java starts at main and creates a JFrame (the window).
2. The window is sized and configured to exit when closed.
3. A JLabel is created and added to the frame's content pane.
4. setVisible(true) tells Swing to draw the window and its contents.

Swing handles all rendering internally (lightweight), so the window and label look the same on any platform, unlike AWT's native dependency.

Speculative Output
- A window appears, 300 pixels wide and 200 pixels tall.
- The title bar says "My First Swing App".
- In the center, the text "Hello, Swing!" is displayed.
- Clicking the close button (X) shuts down the program.

Why This Matters
- It's the simplest complete Swing app, showing the essentials: a container (JFrame), a component (JLabel), and basic setup.
- It introduces Swing's lightweight nature and key methods like setSize and setVisible.
- It's a foundation—you can build on it by adding more components (e.g., buttons) or event handling (e.g., clicks).

Extending the Example
To see more, add a button:

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

public class ExtendedSwingApp {
   public static void main(String[] args) {
      JFrame frame = new JFrame("Extended Swing App");
      frame.setSize(300, 200);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
JLabel label = new JLabel("Hello, Swing!");
JButton button = new JButton("Click Me");
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        label.setText("Button Clicked!");
    }
});

frame.setLayout(new java.awt.FlowLayout());
frame.add(label);
```

1 b. Create a Swing application that has two Buttons named “CSE” and “ISE”, when either of the buttons is clicked it should display “CSE Pressed” and “ISE Pressed” respectively.

Ans:-

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

public class DepartmentButtons extends JFrame implements ActionListener {
    JButton cseButton, iseButton;

    public DepartmentButtons() {
        // Set title and layout
        setTitle("Department Buttons");
        setLayout(new FlowLayout());

        // Initialize buttons
        cseButton = new JButton("CSE");
        iseButton = new JButton("ISE");

        // Add action listeners
        cseButton.addActionListener(this);
        iseButton.addActionListener(this);

        // Add buttons to frame
        add(cseButton);
        add(iseButton);

        // Frame settings
        setSize(300, 100);
```

```
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    // Handle button clicks
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == cseButton) {
            JOptionPane.showMessageDialog(this, "CSE Pressed");
        } else if (e.getSource() == iseButton) {
            JOptionPane.showMessageDialog(this, "ISE Pressed");
        }
    }

    // Main method
    public static void main(String[] args) {
        new DepartmentButtons();
    }
}
```

2(a). Explain the components and containers used in Swings with key features
(b) Explain how components and container work together with a program

Ans:- **Components and Containers**

A Swing GUI consists of two key items:

1. Components
2. Components

All containers are also components. The difference between the two is found in their intended purpose.

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.

In order for a component to be displayed , it must be held within a container. Hence, all Swing GUI's will have at leaset one container.

In Swing (and Java GUIs in general), components and containers are the building blocks for creating user interfaces. They work together like pieces of a puzzle: components are the individual elements (e.g., buttons, labels), and containers are the structures that hold and organize them (e.g., windows, panels).

1. Components

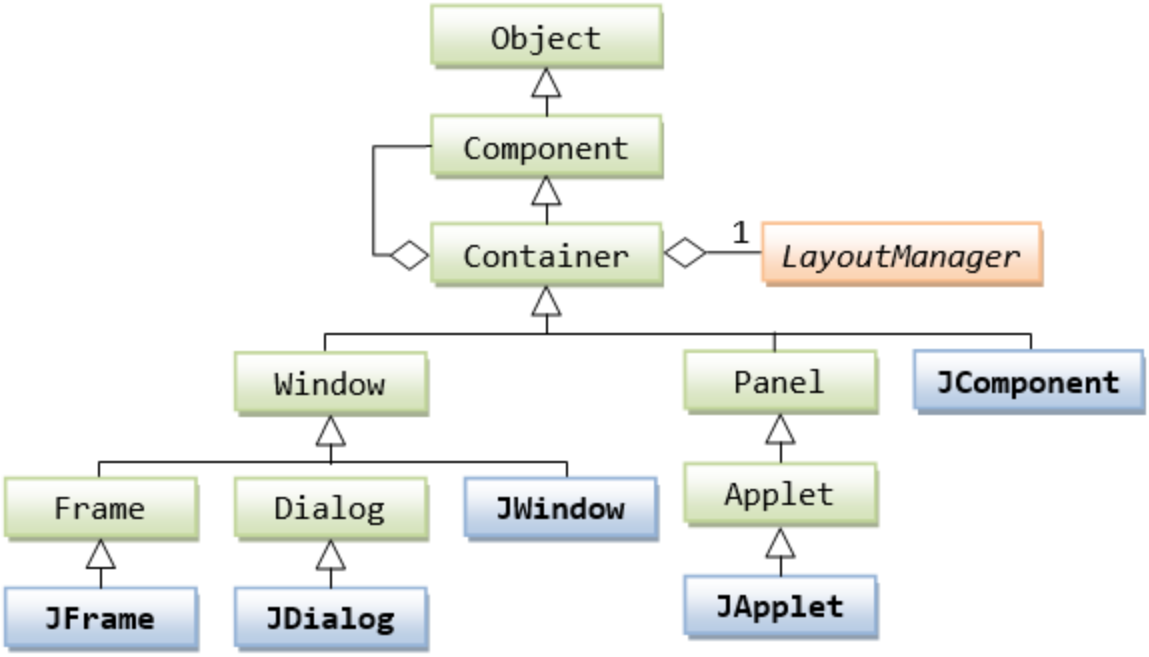
What Are They?

- **Components are the interactive or visual elements users see and use in a GUI, such as buttons, text fields, labels, and checkboxes.**
- In Swing, these are lightweight objects (drawn by Java, not the OS) with a “J” prefix, like JButton, JLabel, or JTextField.
- Swing components are derived from the JComponent class.
- JComponent provides the functionality that is common to all components, it 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**

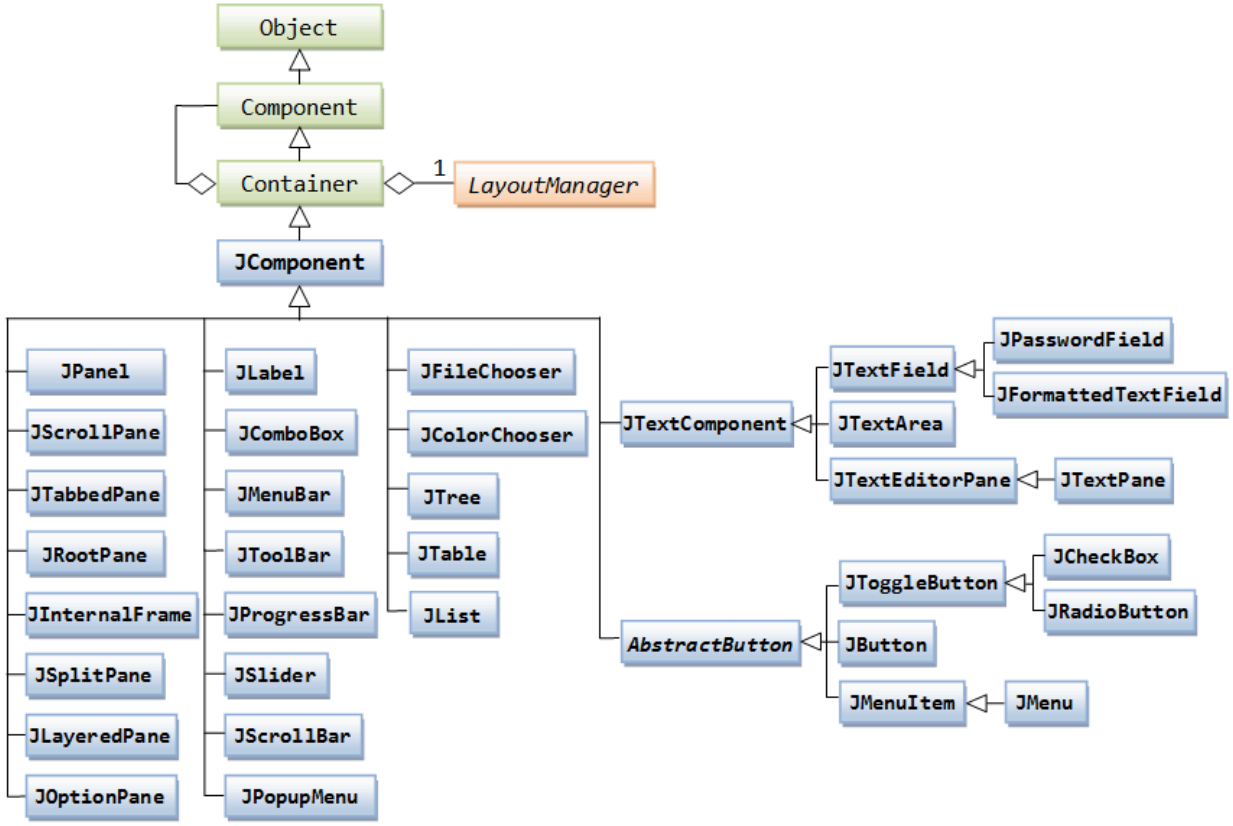
The following table shows the class names for Swing components (including those used as containers).

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		

The class hierarchy of Swing's JComponent is shown below:



JComponent and its descendants - Swing components classes

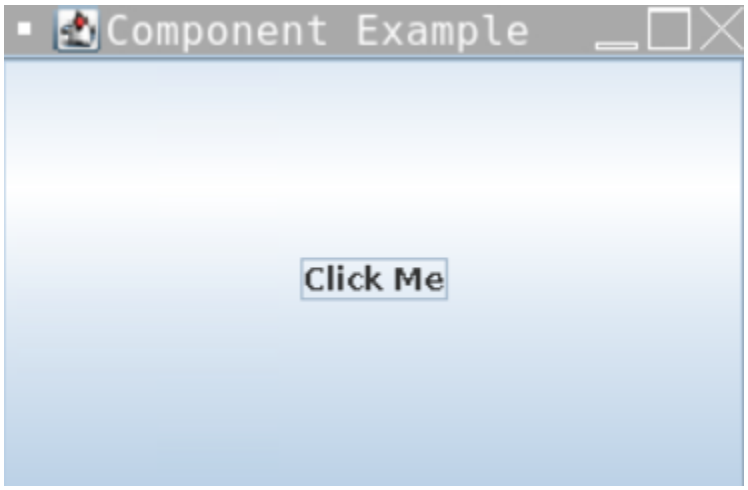


Examples of Swing Components
- JButton: A clickable button.
- JLabel: A text or image display.
- JTextField: A single-line text input.
- JCheckBox: A checkbox for on/off choices.
- JRadioButton: A radio button for single-selection groups.

Key Features
- Lightweight: Rendered by Java, ensuring consistency across platforms.
- Customizable: Appearance can be changed via Pluggable Look and Feel (PLAF) or properties (e.g., font, color).
- Event Handling: Respond to user actions (e.g., clicks) via listeners.

Example: A Simple Component

```
import javax.swing.*;
public class ComponentDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Component Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton button = new JButton("Click Me"); // A Swing component
        frame.add(button); // Added to a container

        frame.setVisible(true);
    }
}
```



- Explanation: The JButton is a component—users can see and click it. Alone, it can't display; it needs a container (here, JFrame) to appear on screen.

2. Containers

What Are They?

- **Containers are special objects that hold and organize components. They act as "parents" to components, providing a space where components can be displayed and arranged.**
- In Swing, containers are also components (they inherit from **java.awt.Container via JComponent**), but they have the added ability to contain other components or even other containers.
- **Swing defines two types of containers:**
  - Top-Level Containers
  - Lightweight Containers

- Top-Level Containers:

- **JFrame**: A standalone window with a title bar, borders, and close button.
- **JDialog**: A pop-up dialog box (e.g., for alerts).
- **JWindow**: A borderless window (less common).
- **JApplet**: A container for Java applets (older technology).

- **These containers do not inherit JComponent.**
- They inherit the **AWT classes Component and Container.**
- They are heavyweitght as they rely on AWT's heavyweight equivalents (e.g., Frame, Dialog) for OS integration.
- A top-level container must be at the top of a containment hierarchy.
- **A top-level container is not contained within any other container.**
- The one most commonly used container for applications is **JFrame**

- Lightweight Containers:

- **Lightweight containers inherit JComponent.**
- 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.

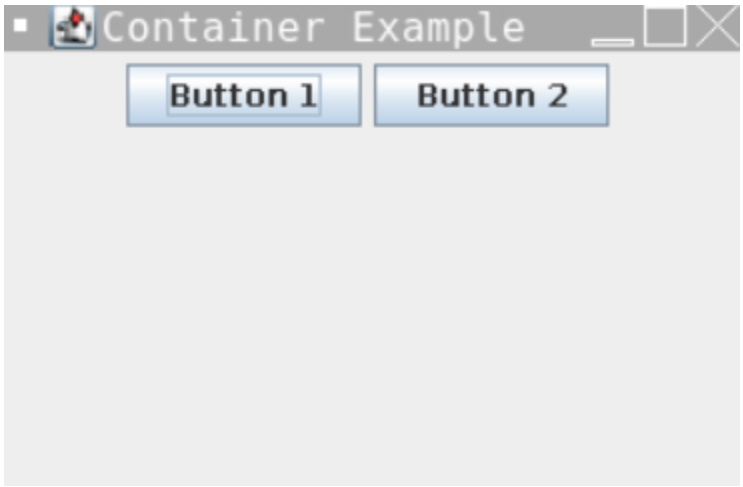
Key Features

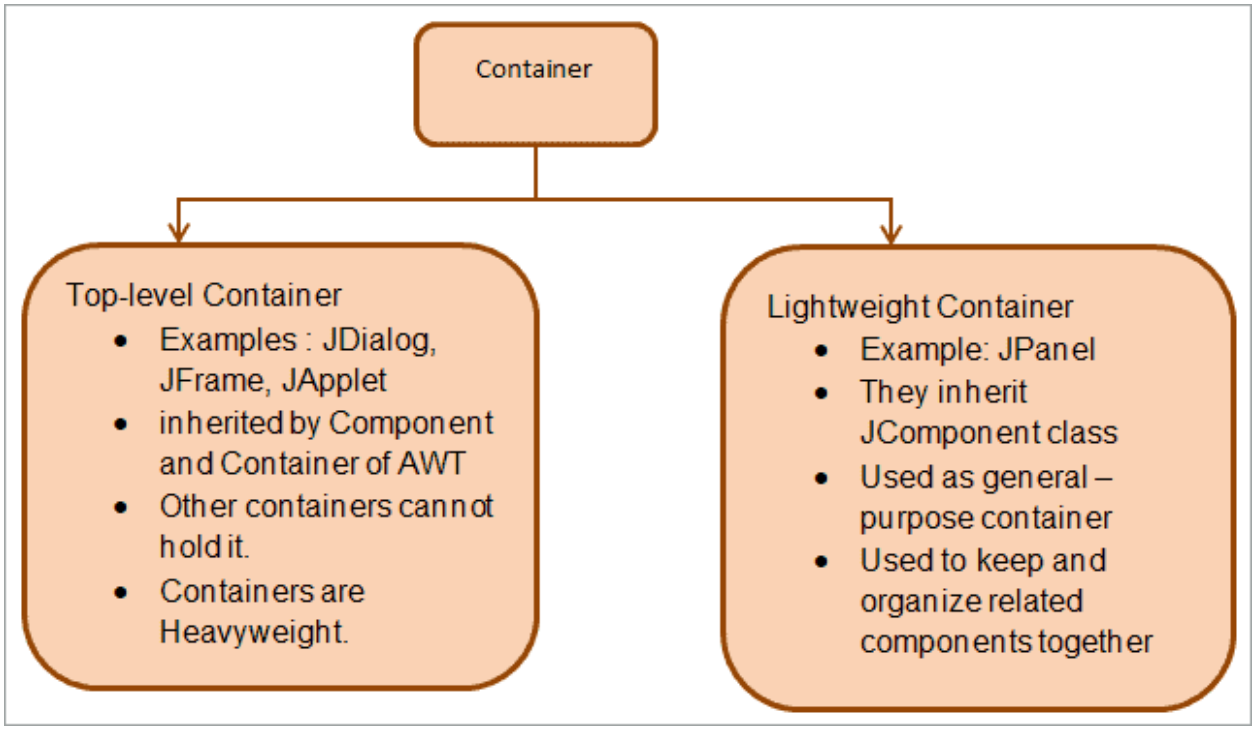
- Hierarchy: Containers can nest inside each other (e.g., a JPanel inside a JFrame).
- Layout Management: Containers use layout managers (e.g., FlowLayout, BorderLayout) to arrange their components.
- Lightweight (Mostly): While top-level containers have a heavyweight base (from AWT), their content areas are lightweight, managed by Swing.

Example: Using a Container

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

public class ContainerDemo {
   public static void main(String[] args) {
      JFrame frame = new JFrame("Container Example"); // Top-level container
      frame.setSize(300, 200);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      JPanel panel = new JPanel(); // General-purpose container
      panel.setLayout(new FlowLayout()); // Layout manager

      JButton button1 = new JButton("Button 1");
      JButton button2 = new JButton("Button 2");
      panel.add(button1); // Add components to panel
      panel.add(button2);

      frame.add(panel); // Add panel to frame
      frame.setVisible(true);
   }
}
```

- Explanation:
  - JFrame: The top-level container (the window).
  - JPanel: A lightweight container inside the JFrame, holding two buttons.
  - FlowLayout: Arrange the buttons side by side.
- Output: A window with two buttons next to each other.





2 (b) How Components and Containers Work Together

- Parent-Child Relationship: Components must be placed inside a container to be visible. A lone JButton won't show up without a JFrame or JPanel to host it.
- Nesting: Containers can hold other containers, creating a hierarchy. For example:
- **A JFrame might contain multiple JPanel`s.**
- Each JPanel could hold buttons, labels, or even more JPanel`s.
- Rendering: **The top-level container (e.g., JFrame) connects to the OS's windowing system, while Swing's lightweight components and nested containers are drawn inside it.**

Visualizing the Hierarchy
Imagine a `JFrame` as a house:
- JFrame: The house itself (top-level container).
- JPanel: A room inside the house (general-purpose container).
- JButton, JLabel: Furniture in the room (components).

Example: Nested Containers

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

public class NestedDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Nested Example");
        frame.setSize(300, 200);
```

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

        JPanel outerPanel = new JPanel(new BorderLayout());
        JPanel innerPanel = new JPanel(new FlowLayout());

        JButton button = new JButton("Click");
        JLabel label = new JLabel("Hello");

        innerPanel.add(button); // Components in inner panel
        innerPanel.add(label);
        outerPanel.add(innerPanel, BorderLayout.CENTER); // Inner panel in outer panel

        frame.add(outerPanel); // Outer panel in frame
        frame.setVisible(true);
    }
}
```

- Explanation:
  - JFrame: The window.
  - outerPanel: A container using BorderLayout to position the inner panel.
  - innerPanel: A container using FlowLayout to hold the button and label.
- Output: A window with a button and label side by side, centered.

Why This matters
- Organization: Containers let you group and arrange components logically (e.g., a form with labels and fields).
- Flexibility: Nesting and layout managers make complex GUIs possible.
- Swing's Design: Lightweight components in lightweight containers (like JPanel) ensure portability, while top-level containers tie into the OS via AWT.

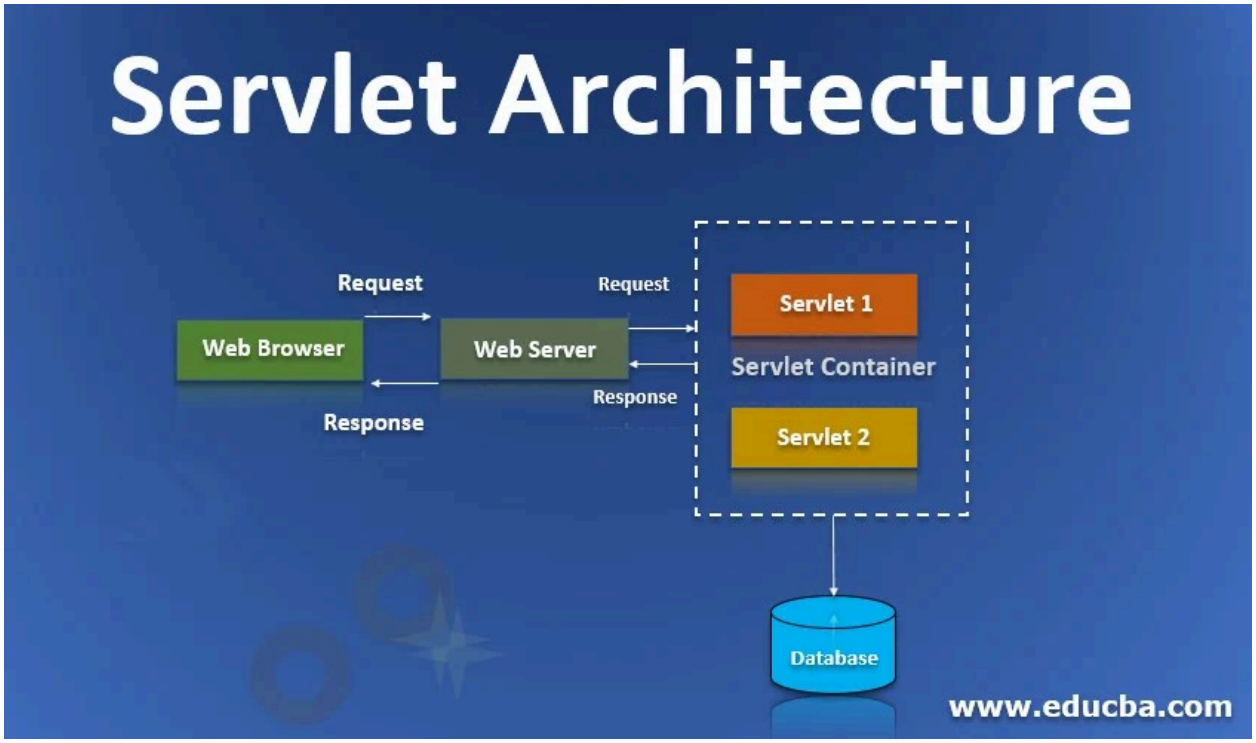
3 What is Servlet? Explain the life cycle of servlets.

Ans:-

A servlet is a Java class that extends the capabilities of a server, typically a web server, to process client requests and generate responses. Most commonly, they handle HTTP requests (e.g., a user clicking a link or submitting a form) and produce HTML or other content (e.g., JSON, images) as responses. Unlike standalone Java applications, **servlets don't run on their own—they need a servlet container to manage their life cycle and interactions.**

Servlets are:
- **Server-Side**: They run on the server, not the client's browser.
- **Request-Driven**: They respond to specific client actions (e.g., GET or POST requests).
- **Dynamic**: They generate content on-the-fly, unlike static HTML files.

How Servlets Work
Servlets operate within a web container, which:
1. Receives an HTTP request from a client (e.g., a browser).
2. Passes the request to the appropriate servlet based on URL mapping.
3. Lets the servlet process the request and create a response.
4. Sends the response back to the client.



For example:
- User visits http://localhost:8080/hello.
- The container maps "/hello" to a servlet (e.g., HelloServlet).

- The servlet generates "Hello, World!" in HTML and sends it back.
- The browser displays the text.

Example: A Simple Servlet
Here's a basic servlet that displays a greeting:

```
import jakarta.servlet.*;
import jakarta.servlet.http.*;
import java.io.*;

public class HelloServlet extends HttpServlet {
        public void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html><body>");
    out.println("<h1>Hello, World!</h1>");
    out.println("</body></html>");
  }
}
```

Explanation
- import jakarta.servlet.*: **Core servlet classes (ServletException).**
- import jakarta.servlet.http.*: HTTP-specific classes (HttpServlet, HttpServletRequest).
- import java.io.*: For PrintWriter and IOException.
- public class HelloServlet extends HttpServlet: Defines the servlet, inheriting from HttpServlet.
- public void doGet(...): Handles HTTP GET requests (e.g., typing a URL).
  - HttpServletRequest request: Carries client data (e.g., parameters).
  - HttpServletResponse response: Sends data back to the client.
- response.setContentType("text/html"): Tells the browser the response is HTML.
- PrintWriter out = response.getWriter(): Gets a writer to send text.
- out.println(...): Writes HTML to the response.
- throws ServletException, IOException: Handles potential errors (e.g., network issues).

Setup
1. Compile this into HelloServlet.class.
2. Place it in Tomcat's webapps/ROOT/WEB-INF/classes folder.
3. Add a web.xml file (or use annotations in modern Java) to map the servlet:

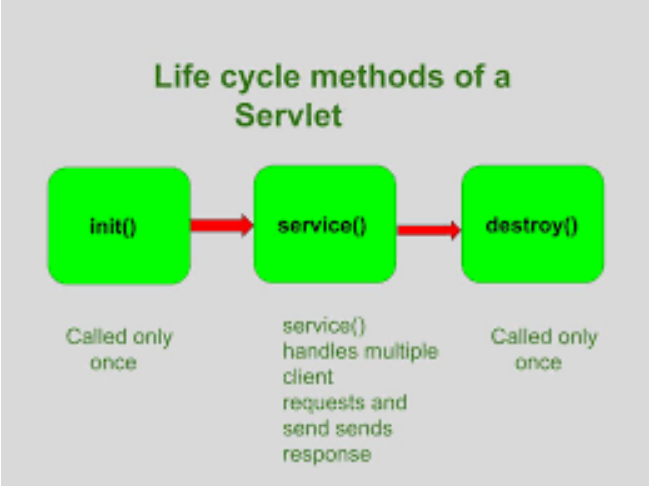
```
<web-app>
  <servlet>
    <servlet-name>HelloServlet</servlet-name>
    <servlet-class>HelloServlet</servlet-class>
  </servlet>
```

```
    <servlet-mapping>
        <servlet-name>HelloServlet</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>
</web-app>
```
4. Start Tomcat and visit http://localhost:8080/hello.

Speculative Output
- Browser displays: A page with "Hello, World!" as a heading.

The Life Cycle of a Servlet

The life cycle of a servlet describes the stages it goes through from creation to destruction, managed by a **web container (like Apache Tomcat).** Unlike standalone Java programs, servlets don't run independently—they're controlled by the container, which handles their instantiation, request processing, and cleanup. Understanding this life cycle is key to writing effective servlets, as it dictates when and how your code executes. **The life cycle has three main phases: initialization, service, and destruction, each tied to specific methods in the Servlet interface.**



Overview of the Life Cycle
1. **Initialization**: The servlet is created and set up (happens once).
2. **Service**: The servlet handles client requests (happens multiple times).
3. **Destruction**: The servlet is shut down and cleaned up (happens once).

These phases are triggered by the container based on server events, like starting up, receiving requests, or shutting down.

The Servlet Interface

The life cycle is defined by the **jakarta.servlet.Servlet interface**, which all servlets implement (directly or via classes like HttpServlet). It includes five methods, but three are central to the life cycle:
- **init(ServletConfig config):** Called once when the servlet is initialized.
- **service(ServletRequest req, ServletResponse res):** Called for each client request.
- **destroy():** Called once when the servlet is removed.

The other two methods—**getServletConfig() and getServletInfo()—provide metadata but aren't part of the core life cycle.**

Detailed Phases

1. Initialization (init Method)
- When: The container loads the servlet, either at server startup or the first request (configurable).
- Purpose: Sets up the servlet before it handles requests—e.g., initializing variables, database connections, or resources.
- How: The container:
  1. Instantiates the servlet (calls new MyServlet()).
  2. Calls init(ServletConfig config) with a ServletConfig object containing configuration data (e.g., from web.xml).
- Key Points:
  - Runs only once per servlet instance.
  - The servlet isn't ready for requests until init completes.
  - You override init() to add setup code.

Example

```
import jakarta.servlet.*;
import jakarta.servlet.http.*;
import java.io.*;

public class LifeCycleServlet extends HttpServlet {
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        System.out.println("Servlet initialized");
    }
}
```

- super.init(config): Calls the parent's init to store the ServletConfig.
- System.out.println("Servlet initialized"): Logs a message (in practice, you'd see this in server logs).

2. Service (service Method)
- When: Every time a client request arrives (e.g., a browser sends GET or POST).

- Purpose: Processes the request and generates a response—e.g., sending HTML to the browser.
- How: The container:
  1. Receives a request (e.g., http://localhost:8080/myServlet).
  2. Calls service(ServletRequest req, ServletResponse res) with request and response objects.
- Key Points:
  - Runs multiple times—once per request.
  - For HttpServlet, service() delegates to methods like doGet() or doPost() based on the HTTP method.
  - Thread-safe: One servlet instance handles multiple requests via threads, so instance variables need caution.

Example (Continued)

```
public class LifeCycleServlet extends HttpServlet {
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        System.out.println("Servlet initialized");
    }

        public void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><body>Servlet is serving</body></html>");
    }
}
```

- service(...): Overrides the generic service method (though HttpServlet usually uses doGet/doPost).
- Output: Browser shows "Servlet is serving" for each request.

HttpServlet Specifics

For HTTP servlets (extending HttpServlet):
- service() checks the request type (GET, POST, etc.) and calls:
  - doGet(): For GET requests (e.g., loading a page).
  - doPost(): For POST requests (e.g., form submissions).
- Example with doGet:

```
public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html><body>Hello via GET</body></html>");
}
```

3. **Destruction (destroy Method)**
- When: The container removes the servlet, usually at server shutdown or if the servlet is reloaded.
- Purpose: Cleans up resources—e.g., closing database connections or files.
- How: The container calls destroy() after all requests are handled and before the servlet is garbage-collected.
- Key Points:
  - Runs only once.
  - No requests are processed after this.
  - You override destroy() for cleanup.

Full Example

```
import jakarta.servlet.*;
import jakarta.servlet.http.*;
import java.io.*;

public class LifeCycleServlet extends HttpServlet {
   public void init(ServletConfig config) throws ServletException {
      super.init(config);
      System.out.println("Servlet initialized");
   }

      public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<html><body>Servlet is serving</body></html>");
   }

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

Setup
- Deploy in Tomcat (webapps/ROOT/WEB-INF/classes).
- Map in web.xml:

```
<web-app>
   <servlet>
      <servlet-name>LifeCycleServlet</servlet-name>
      <servlet-class>LifeCycleServlet</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>LifeCycleServlet</servlet-name>
```

```
        <url-pattern>/lifecycle</url-pattern>
    </servlet-mapping>
</web-app>
```
- Access: http://localhost:8080/lifecycle.

Output
- Server starts: Console logs "Servlet initialized".
- Browser visits URL: Shows "Servlet is serving" (multiple visits repeat this).
- Server stops: Console logs "Servlet destroyed".

4. Explain core classes and interfaces in javax.servlet package.

Ans:-

The Servlet API

- Two packages contain the classes and interfaces that are required to build the servlets.
- These are **javax.servlet** and **javax.servlet.http**.
- They constitute the core of the Servlet API.
- These packages are not part of the Java core packages. Therefore, they are not included with Java SE. Instead, they are provided by Tomcat. They are also provided by Java EE.

javax.servlet package

- The javax.servlet package contains a number of interfaces and classes that establish the framework in which servlets operate.
- The following table summarizes several key interfaces that are provided in this package.
- The most significant of these is **Servlet**.
- All servlets must implement this interface or extend a class that implements the interface.
- The **ServletRequest** and **ServletResponse** interfaces are also very important.

Core Interfaces	
Servlet	Defines lifecycle methods for a servlet
ServletConfig	Allows servlet to get initialization parameters
ServletContext	Defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file.
ServletRequest	Used to read data from a client request
ServletResponse	Used to write data to a client response

The following table summarizes the core classes that are provided in the javax.servlet package:

Core Classes	
GenericServlet	Defines a generic servlet that implements Servlet and ServletConfig interfaces
ServletInputStream	Provides an input stream for reading binary data from a client request.
ServletOutputStream	Provides an output stream for sending binary data to the client.

Exception Summary	
ServletException	Defines a general exception a servlet can throw when it encounters difficulty.
UnavailableException	Defines an exception that a servlet or filter throws to indicate that it is permanently or temporarily unavailable.

Servlet Interface

all servlets must implement the servlet interface

- Defines methods that all servlets must implement.
- A servlet is a small Java program that runs within a Web server.
- Servlets receive and respond to requests from Web clients, usually across HTTP, the HyperText Transfer Protocol.
- To implement this interface, you can write a generic servlet that extends javax.servlet GenericServlet or an HTTP servlet that extends javax.servlet.http.HttpServlet.
- This interface defines methods to initialize a servlet, to service requests, and to remove a servlet from the server.
- These are known as life-cycle methods and are called in the following sequence:
- The servlet is constructed, then initialized with the **init** method.
- Any calls from clients to the **service** method are handled.
- The servlet is taken out of service, then destroyed with the **destroy** method, then garbage collected and finalized.

Method Summary of Servlet Interface	
`void destroy()`	Called when servlet is unloaded

	This method gives the servlet an opportunity to clean up any resources that are being held (for example, memory, file handles, threads) make sure that any persistent state is synchronized with the servlet's current state in memory.
ServletConfig **getServletConfig**()	Returns a ServletConfig object, which contains initialization and startup parameters for this servlet.
String **getServletInfo**()	Returns information about the servlet, such as author, version, and copyright.
void **init**(ServletConfig config) throws ServletException	Called when servlet is initialized. An UnavailableException is thrown if servlet cannot be initialized. The servlet container calls the init method exactly once after instantiating the servlet. The init method must complete successfully before the servlet can receive any requests.
Void **service**(ServletRequest req, ServletResponse res) throws ServletException, IOException	Called by the servlet container to allow the servlet to respond to a request. Response can be written to res. IOException is returned if a problem occurs.

Servlets typically run inside multithreaded servlet containers that can handle multiple requests concurrently.

Developers must be aware to synchronize access to any shared resources such as files, network connections, and as well as the servlet's class and instance variables.

ServletConfig Interface

allows a servlet to obtain configuration data when it is loaded

Method Summary	
String **getInitParameter**(Strin g param)	Returns vlue of initialization parameter named param
Enumeration **getInitParameterNames**()	Returns names of the servlet's initialization parameters as an Enumeration of String objects, or an empty Enumeration if the servlet has no initialization parameters.

ServletContext **getServletContext**()	Returns a reference to the ServletContext.
String **getServletName**()	Returns the name of this servlet instance.

ServletContext Interface

enables servlet to obtain information about their environment

Method Summary	
Object **getAttribute**(String name)	Returns value of servlet container with the given name
String **getMimeType**(String file)	Returns the MIME (Multipurpose Internet Mail Extensions) type of the specified file
String **getRealPath**(String vpath)	Returns a String containing the real path for a given virtual path.
String **getServerInfo**()	Returns the name and version of the servlet container on which the servlet is running.
void **log**(String msg)	Writes the specified message to a servlet log file.
void **log**(String message, Throwable throwable)	Writes an explanatory message and a stack trace for a given Throwable exception to the servlet log file.
void **setAttribute**(String name, Object object)	Binds an object to a given attribute name in this servlet context.

ServletRequest Interface

enables a servlet to obtain information about a client request

Method Summary	
Object **getAttribute**(String name)	Returns the value attribute as an Object.
String **getCharacterEncoding**()	Returns the name of the character encoding used in the body of this request.
int **getContentLength**()	Returns the length, in bytes, of the request body and made available by the input stream, or -1 if the length is not known.
String **getContentType**()	Returns the MIME type of the body of the request
ServletInputStream **getInputStream**() throws IOException	Retrieves the body of the request as binary data using a ServletInputStream. IllegalStateException is throws if getReader() has already been invoked.
String **getParameter**(String name)	Returns the value of a request parameter as a String

Enumeration **getParameterNames**()	Returns an `Enumeration` of `String` objects containing the names of the parameters contained in this request.
String[]**getParameterValues**(String name)	Returns an array of `String` objects containing all of the values the given request parameter has
BufferedReader **getReader**()	Returns a `BufferedReader`. that can be used to read text form the request
String **getRemoteAddr**()	Returns the Internet Protocol (IP) address of the client
String**getRemoteHost**()	Returns client host name
String **getScheme**()	Returns the name of the scheme used to make this request, for example, `http`, `https`, or `ftp`.
String **getServerName**()	Returns the host name of the server
int **getServerPort**()	Returns the port number to which the request was sent.

ServletResponse Interface

enables a servlet to formulate a response for a client.

Method Summary	
String **getCharacterEncoding**()	Returns the name of the character encoding (MIME charset) used for the body sent in this response.
ServletOutputStream **getOutputStream**() throws IOException	Returns a `ServletOutputStream` suitable for writing binary data in the response.
PrintWriter **getWriter**()	Returns a `PrintWriter` object that can send character text to the client.
void **setContentLength**(int len)	Sets the length of the content body in the response to len
void **setContentType**(String type)	Sets the content type of the response being sent to the client

The GenericServlet Class

- Defines a generic, protocol-independent servlet.
- provides implementations of the Servlet and ServletConfig interaces.
- In addition, two log methods
  - **void log(String s)**
  - **void log(String s, Throwable e)** are used.

ServletInputStream class

- extends **InputStream**
- implemented by servlet container.
- provides an input stream that a servlet developer can use to read the data from a client request
- provides default contructor

int readLine(byte[] *buffer*, int *offset*, int *size* throws *IOException*.

- *buffer* is the array into which *size* bytes are placed starting at *offset*.
- returns actual number of bytes read or -1 if end-of-stream is reached

ServletOutputStream Class

- extends **OutputStream** class
- implemented by servlet container
- provides an output stream that can be used to write data to a client response
- print()
- println().

ServletException Class

- defines 2 exceptions
- ServletException – indicates a servlet problem has occurred
- UnavailableException – extends ServletException.

A program using GenericServlet class to handle parameters.

PostParam.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Post parameters Demo</title>
</head>
<body>
<h1>  Pass parameters using post </h1>
<div style= "width:350px; height:200px; padding: 10px;
background-color:#ccccff; color: #333388">
   <form name = "form1" method="post"
action="servlets/postParamServlet">
   <label>Parameter 1 : </label>
   <input type = text name = "e" size ="25" value = "" />
   <br><br>
   <label>Parameter 2 : </label>
   <input type = text name = "p" size = "25" value = "" />
   <br><br>
   <input type = submit value = "Submit" />
</form>
```

```
</div>
</body>
</html>
```

web.xml entry under tag <web-app>

```
<servlet>
    <description></description>
    <display-name>postParamServlet</display-name>
    <servlet-name>postParamServlet</servlet-name>
    <servlet-class>postParamServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>postParamServlet</servlet-name>
    <url-pattern>/servlets/postParamServlet</url-pattern>
  </servlet-mapping>
```

PostParamServlet.java

```
import java.io.*;
import javax.servlet.*;
import java.util.*;
public class postParamServlet extends GenericServlet {
   public void service(ServletRequest req, ServletResponse
resp)
        throws IOException, ServletException  {

PrintWriter pw = resp.getWriter();

//Get enumeration of parameter names.
   Enumeration e = req.getParameterNames();

   pw.println("<h1> Parameters passed using get method are :
</h1>");

//Display param name and values
   while(e.hasMoreElements() ) {
        String pname = (String) e.nextElement();
        pw.print("<h3>" +pname+" = ");
        String pvalue = req.getParameter(pname);
        pw.print(pvalue+"</h3>");
   }

   pw.close();
   }
}
```

-- compile the servlet.
place the .class file in the classes directory.
update the web.xml file

Start Tomcat
Display the web page in browser
Enter two parameters
submit form

- javax.servlet.http Package

the javax.servlet.http package contains a number of classes and interfaces that describe and define the contracts between a servlet class running under the HTTP protocol and the runtime environment.

Core Interfaces	
HttpServletRequest	Enables servlets to read form an HTTP request
HttpServletResponse	Enables servlets to write data to an HTTP response
HttpSession	Allows session data to be read and written
HttpSessionBindingListener	Causes an object to be notified when it is bound to or unbound from a session.
Core Classes	
Cookie	Creates a cookie, a small amount of information sent by a servlet to a Web browser, saved by the browser, and later sent back to the server.
HttpServlet	Provides methods to handle HTTP request and responses

HttpSessionBindingEvent	Indicates when a listener is bound or unbound from a session.
HttpSessionEvent	This is the class representing event notifications for changes to sessions within a web application.

HttpServletRequest Interface.

Method Summary	
String **getAuthType**()	Returns the name of the authentication scheme.
Cookie[]**gtCookies**()	Returns an array containing all of the `Cookie` objects the client sent with this request.
long **getDateHeader**(String name)	Returns the value of the specified request header as a `long` value that represents a `Date` object.
String **getHeader**(String name)	Returns the value of the specified request header as a `String`.
Enumeration **getHeaderNames**()	Returns an enumeration of all the header names this request contains.
int **getIntHeader**(String name)	Returns the value of the specified request header as an `int`.
String **getMethod**()	Returns the name of the HTTP method with which this request was made, for example, GET, POST
String **getPathInfo**()	Returns any extra path information associated with the URL the client sent before the query string.
String **getPathTranslated**()	Returns any extra path information after the servlet name but before the query string, and translates it to a real path.
String **getQueryString**()	Returns the query string that is contained in the request URL after the path.
String **getRemoteUser**()	Returns the login of the user making this request.
String **getRequestedSessionId**()	Returns the session ID specified by the client.
String **getRequestURI**()	Returns the part of this request's URL from the protocol name up to the query string in the first line of the HTTP request.
StringBuffer **getRequestURL**()	Reconstructs the URL the client used to make the request.
String **getServletPath**()	Returns the part of this request's URL that calls the servlet.

HttpSession **getSession**()	Returns the current session associated with this request, or if the request does not have a session, ***creates one***.
HttpSession **getSession**(boolean create)	Returns the current `HttpSession` associated with this request or, if there is no current session and `create` is true, returns a new session.
Boolean **isRequestedSessionIdFromCookie**()	Checks whether the requested session ID came in as a cookie.
Boolean **isRequestedSessionIdFromURL**()	Checks whether the requested session ID came in as part of the request URL.
Boolean **isRequestedSessionIdValid**()	Checks whether the requested session ID is still valid.

HttpServletResponse Interface

- enables a servlet to formulate an HTTP response to a client.
- several constants are defined.
- they correspond to different status codes assigned to an HTTP response, eg.,
  - SC_OK, SC_NOT_FOUND

Method Summary	
void **addCookie**(Cookie cookie)	Adds the specified cookie to the response.
boolean **containsHeader**(String *name*)	Returns true if HTTP response header contains a field named *name*
String **encodeRedirectURL**(String url)	Encodes the specified URL for use in the `sendRedirect` method or, if encoding is not needed, returns the URL unchanged.
String **encodeURL**(String url)	Encodes the specified URL by including the session ID in it, or, if encoding is not needed, returns the URL unchanged.
void **sendError**(int sc) throws IOException	Sends an error response to the client using the specified status code and clearing the buffer.
void **sendError**(int sc, String msg) throws IOException	Sends an error response to the client using the specified status.
void **sendRedirect**(String *location*) throws IOException	redirects to redirect *location* URL.
void **setDateHeader**(String name, long date)	Sets a response header with the given name and date-value (in milliseconds since Jan 1, 1970, GMT)

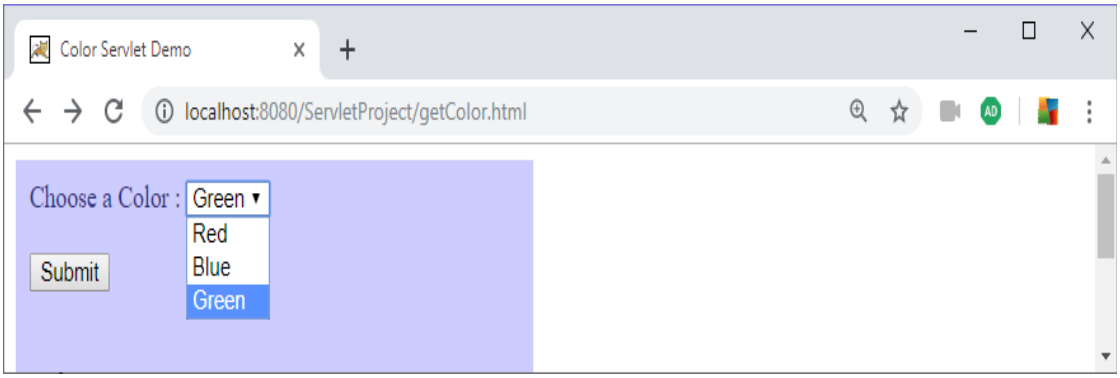
void **setHeader**(String name, String value)	Sets a response header with the given name and value.
void **setIntHeader**(String name, int value)	Sets a response header with the given name and integer value.
void **setStatus**(int sc)	Sets the status code for this response.

The HttpServlet Class

extends GenericServlet

Method Summary	
void **doDelete**(HttpServletRequest req, HttpServletResponse resp)	Called by the server to handle a DELETE request.
void **doGet**(HttpServletRequest req, HttpServletResponse resp)	Called by the server to handle a GET request.
void **doHead**(HttpServletRequest req, HttpServletResponse resp)	handles HTTP HEAD request
void **doOptions**(HttpServletRequest req, HttpServletResponse resp)	Called by the server to handle a HTTP OPTIONS request.
void **doPost**(HttpServletRequest req, HttpServletResponse resp)	Called by the server to handle a POST request.
void **doPut**(HttpServletRequest req, HttpServletResponse resp)	Called by the server to handle a PUT request.
void **doTrace**(HttpServletRequest req, HttpServletResponse resp)	Called by the server to handle a TRACE request.
long **getLastModified**(HttpServletRequest req)	Returns the time the object was last modified, in milliseconds since midnight January 1, 1970 GMT.
void **service**(HttpServletRequest req, HttpServletResponse resp)	called by server when an HTTP request arrives for this servlet. The arguments provide access to the HTTP request and response.

- **Program to handle HTTP requests and Responses.**



getColor.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Color Servlet Demo</title>
</head>
<body>
<div style= "width:350px; height:200px; padding: 10px;
background-color:#ccccff; color: #333388">
<form name = "form1" method="get"
action="servlets/ColorGetServlet">
     <label>Choose a Color : </label>

         <select name ="color">
              <option value = "red">Red</option>
              <option value = "blue">Blue</option>
              <option value = "green">Green</option>
         </select>
     <br><br>
     <input type = submit value = "Submit" />
</form>
</div>
</body>
</html>
```

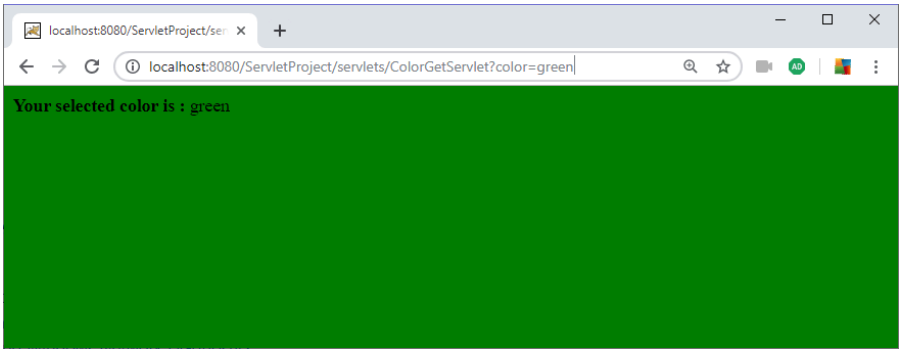
ColorGetServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ColorGetServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;
```

```
   protected void doGet(HttpServletRequest req,
HttpServletResponse res) throws ServletException, IOException
{

        String color = req.getParameter("color");
        res.setContentType("text/html");
        PrintWriter pw = res.getWriter();
        pw.println("<body bgcolor = "+color+ ">");
        pw.println("<strong> Your selected color is :
</strong>");
        pw.println(color); pw.println("</body>");
        pw.close();
   }
}
```

web.xml entry

```
<servlet>
    <description></description>
    <display-name>ColorGetServlet</display-name>
    <servlet-name>ColorGetServlet</servlet-name>
    <servlet-class>sp.ColorGetServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>ColorGetServlet</servlet-name>
    <url-pattern>/servlets/ColorGetServlet</url-pattern>
  </servlet-mapping>
```



When user submits the ColorGet.htm page the *query string* will be attached to the URL
http://localhost:8080/servlets/ColorGetServlet?color=green

Handling Http POST requests.

- use the same ColorGet.html, except change the form method to post
- Save it in file ColorPost.html.
- In the ColorGetServlet.java, use the doPost() method instead of doGet() method, rest of the code is same.

5 (a) Explain scrollableResultSet and UpdatableResultSet Object in JDBC with example program

Ans:-

Scrollable ResultSet

- until the release of JDBC 2.1 API, virtual cursor could only be moved down the ResultSet object.
- 6 methods used in JDBC 2.1 API to position the cursor
  - first()
  - last()
  - previous()
  - absolute() – positions cursor at row number specified
  - relative() – moves the virtual cursor by a specified number of rows, positive or negative
  - getRow() - returns an integer that contains the number of the current row in the ResultSet.
- 3 constants can be passed to the createStatement() method of the Connection object to handle ResultSet Scrollability,
  - TYPE_FORWARD_ONLY – restricts the virtual cursor to downward movement
  - TYPE_SCROLL_INSENSITIVE
    - permits cursor to move up and down
    - makes the ResultSet insensitive to changes made by another J2EE component to data in the table whose rows are reflected in the ResultSet.
  - TYPE_SCROLL_SENSITIVE
    - permits cursor to move up and down
    - makes the ResultSet sensitive to those changes.

Q. What is the output given the table above?

Ans:
rs.first() at 1 Jane
rs.last() at 8 Meera
rs.previous() at 7 Sunio
rs.absolute(5) at 5 Mary
rs.relative(-2) at 3 Likitha
rs.relative(3) at 6 Saritha

Code to use Scrollable ResultSet

```
Connection db;
Statement stmt;
ResultSet rs;
String fname, lname;
….
/* Scrollable ResultSet Demo  */
  try {
        System.out.println("\n** Scrollable ResultSet Demo "
  String query = "Select fname, lname from custTbl";
  Statement st = db.createStatement
```

```
            (ResultSet.TYPE_SCROLL_INSENSITIVE,
            ResultSet.CONCUR_READ_ONLY);
  ResultSet scrs= st.executeQuery(query);
  boolean rec = scrs.next();
  scrs.first();
  String name= scrs.getString(1)+" "
  +scrs.getString(2);
  System.out.println(name);

  scrs.last();
  scrs.previous();
  scrs.absolute(5);
  scrs.relative(-2);
  scrs.relative(3);
  name= scrs.getString(1)+" "+scrs.getString(2);
  System.out.println(name);
st.close();
}catch(SQLException err) {
        err.printStackTrace();
  }
```

Check whether the database supports scrollability using DataBaseMetaData Object

- not all JDBC drivers are scrollable
- To test whether or not JDBC driver supports a Scrollable Result Set,

```
boolean forward, insensitive, sensitive;
DatabaseMetaData meta = db.getMetaData();
forward = meta.supportsResultSetType(ResultSet.TYPE_FORWARD_ONLY);
insensitive = meta.supportsResultSetType(ResultSet.TYPE_SCROLL_INSENSITIVE);
sensitive = meta.supportsResultSetType(ResultSet.TYPE_SCROLL_SENSITIVE);
System.out.println("forward: "+forward+" Insensitive "+ insensitive + " sensitive:
"+sensitive);
```

Fetch Size

- when J2EE component requests row from the ResultSet, some rows are fetched into the driver and returned at one time.
- sometimes not all rows are retrieved at the same time
- in this case, driver returns to the DBMS and requests another set of rows that are defined by the fetch size and discards the current set of rows.
- fetch size is set using the setFetchSize() method of the Statement object.

Code to set fetch size

```
Connection db;
Statement stmt;
ResultSet rs;
….
try{
```

```
      String query = "SELECT * FROM customers";
      stmt = db.createStatement
             (ResultSet.TYPE_SCROLL_INSENSITIVE,
             ResultSet.CONCUR_READ_ONLY);
      stmt.setFetchSize(500);
      rs = stmt.executeQuery(query);
      stmt.close();
}catch(SQLException e) {
      System.err.println("SQL Error"  + err);
}
```

Updatable ResultSet

- rows in a ResultSet are updatable.
- two constants are passed to the createStatement() method to manage updation
  - CONCUR_UPDATABLE – enables updatability of the ResultSet
  - CONCUR_READ_ONLY – prevents ResultSet from being updated
- 3 ways in which a ResultSet can be changed
  - Update values in a row
  - insert a new row
  - delete a row

1.Update ResultSet

- updatexxx() method() of the statement object is used to change the value of a column in the current row of the ResultSet.
  - xxx is replaced with the data type of the column being updated
  - two parameters,
    - column name or column number of the column to be updated,
    - the value that will replace the existing value
- updateNull() : replaces value in column with a null
  - takes 1 parameter: the name/number of the column
- updateRow() : is called after all the updatexxx() methods. It changes values in columns of the current row of the ResultSet.

Code to update the ResultSet

```
Connection db;
Statement stmt;
ResultSet rs;
….
try{
      String query = "SELECT firstName, lastName FROM customers WHERE
firstName='Mary' AND lastName = 'Brown'";
      stmt = db.createStatement();
      rs = stmt.executeQuery(query);
      Boolean rec = rs.next();
      if(! rec){
        System.out.println("No data returned");
        System.exit();
      }
      rs.updateString("lastName","Smith");
```

```
      rs.updateRow();

      stmt.close();
}catch(SQLException e) {
      System.err.println("SQL Error"  + err);
}
```

2. Delete row from ResultSet

- deletes rows from the ResultSet and the underlying database.
- an integer that contains the number of the row to be deleted is passes as the parameter.
  rs.deleteRow()
- deletes the current row

```
if(rs_ur.getInt("ID") == 4) {
      rs_ur.deleteRow();
}
```

3. Insert row in the ResultSet

- use the updatexxx(0 method
- then use the insertRow() method - to add a new row to the ResultSet.

```
Connection db;
Statement stmt;
ResultSet rs;
….
try{
      String query = "SELECT firstName, lastName FROM customers WHERE
firstName='Mary' AND lastName = 'Brown'";
      stmt = db.createStatement();
      rs = stmt.executeQuery(query);
      Boolean rec = rs.next();
      if(! rec){
        System.out.println("No data returned");
        System.exit();
      }
      rs_ur.moveToInsertRow();
      rs_ur.updateString("custNumber", "CS205");
      rs_ur.updateString("fname", "Saritha");
      rs_ur.updateString("lname", "K");
      rs_ur.updateString("Street", "#43, KT Layout");
      rs_ur.updateInt("balance", 9000);
      rs_ur.insertRow();
      rs_ur.moveToCurrentRow();
      stmt.close();
}catch(SQLException e) {
      System.err.println("SQL Error"  + err);
}
```

5.(b). Explain the types of exceptions that occur in JDBC

Ans:

- **Exceptions**

3 kinds of Exception thrown by JDBC methods

1. **SQLException** : commonly reflect a SQL syntax error in the query.
    - has getNextException()
    - getErrorCode() – to receive vendor specific error code.

2. **SQLWarning**: throws warnings received by the Connection from the DBMS.
    - getWarning()
    - getNextWarnings()
3. **DataTruncationException** : whenever data is lost due to the truncation of the data value.

6. List and explain various statement objects in JDBC with example program

Ans:

- **Statement Objects**
- J2EE component creates and sends a query to access data in database.
- 3 types of statement objects are used to execute the query
    - The **Statement** Object
    - **PreparedStatement** Object
    - **CallableStatement** Object

1. The Statement Object

- used whenever a J2EE component needs to immediately execute a query without having the query compiled.
    - executeQuery()
        - returns ResultSet object containing rows, columns and metadata
    - execute()
        - when multiple results are returned.
    - executeUpdate()
        - used to execute queries that contain UPDATE and DELETE SQL statements.
        - returns an integer indicating a number of rows updated.
        - also used for the INSERT statement

Code using executeQuery()

```
….
Connection db;
Statement stmt;
ResultSet rs;
try{
      String query = "SELECT * FROM customers";
      stmt = db.createStatement();
      rs = stmt.excecuteQuery(query);
```

```
      //…….
      stmt.close();
}catch(SQLException e) {
      System.err.println("SQL Error"  + err);
}
```

Code using executeUpdate() to update

```
….
Connection db;
Statement stmt;
int numRowsUpdated;
try{
      String query = "UPDATE customers SET PAID= 'Y' WHERE BALANCE =
0";
      stmt = db.createStatement();
      numRowsUpdated= stmt.excecuteUpdate(query);
      //…….
      stmt.close();
}catch(SQLException e) {
      System.err.println("SQL Error"  + err);
}
```

2. PreparedStatement Object

- An SQL query must be compiled before the DBMS processes the query
- Compiling occurs after the Statement objects' execution method.
- Compiling a query is an acceptable overhead if the query is called once.
- compiling several times becomes an expensive overhead
- Solution: PreparedStatement Object
    - **precompiles** and executes an SQL statement, known as **late binding**
    - a? placeholder is used in the query to later insert the value for the query.

Steps to use PreparedStatement Object

1. prepareStatement() method of the Connection object is called to return the PreparedStatement
    - prepareStatement() method is passed the query.
2. setxxx() method is used to replace the ? placeholder.
    - xxx – refers to a data type, eg., setString()
        - 2 parameters are passed -
            - The first parameter: identifies the position of the ? placeholder
            - Second parameter: the late binding variable.
3. the executeQuery() method of the PreparedStatement object is called.
    - it doesn't require a parameter 'cos query is already associated with the PreparedStatement object.

Code using PreparedStatement

```
Connection db;
ResultSet rs;
….
```

```
try {
        String qPrep ="Select * from customers where custNumber = ?";
        PreparedStatement pstmt =db.prepareStatement(qPrep);
        pstmt.setString(1, "C0100");
        ResultSet prs = pstmt.executeQuery();
        prs.next();
        System.out.println("\n**Customer with Number : C0100 is ");
        System.out.println(prs.getString(3)+" "+prs.getString(4));
  }catch(SQLException err) {
        err.printStackTrace();
  }
```

3. CallableStatement Object

- used to call a stored procedure from within a J2EE object.
    - stored procedure: a block of code identified by a unique name, executed by invoking the name of the stored procedure.
- uses 3 types of parameters
    - IN:
        - contains data that needs to be passed to a stored procedure.
        - value is assigned using setxxx() method
    - OUT
        - contains value returned by the stored procedure
        - must be registered using the registerOutParameter() method.
        - later received using getxxx() method
    - INOUT
        - used to pass and retrieve information from a stored procedure.

Steps to use CallableStatement Object

1. **prepareCall()** of the Connection object is called and passed the query
    - returns a CallableStatement object.
2. **registerOutParameter()** has two arguments
    - 1st parameter: represents the number of the parameter in the stored procedure
    - 2nd parameter: data type of the value returned by the stored procedure, eg. VARCHAR.
3. **execute()** method of CallableStatement object is used to execute the query
    - need not be passed as it is already identified in the prepareCall() method.

```
delimiter; $$

create procedure LastOrderNumber(IN var1 int, OUT firstName CHAR(20))
  -> BEGIN
```

```
   -> select fname INTO firstName from customers where ID = var1;
   -> END$$
```

call LastOrderNumber(1, @L);
Query OK, 1 row affected (0.05 sec)

mysql> select @L;

Code to use CallableStatement Object.

```
Connection db;
ResultSet rs;
….
  /*  CallableStatement Demo */
  try {
        String cQry = "CALL LastOrderNumber(1,?)";
       CallableStatement cstmt = db.prepareCall(cQry);
       cstmt.registerOutParameter(1, Types.VARCHAR);
       cstmt.execute();
       String fname = cstmt.getString(1);
       System.out.println("\n**Call Statement Demo");
       System.out.println(fname);

  }catch(SQLException err) {
        err.printStackTrace();
  }
```