Advance Java BCA 6th Semester

EXAM BASED QUESTIONS WITH ANSWERS

1. Define inheritance. Discuss the benefits of using inheritance. Discuss multiple

inheritance with suitable example.(1+2+7)

Inheritance in Java

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a new class (subclass or child class) to inherit properties and behaviors (fields and methods) from an existing class (superclass or parent class). This concept promotes code reusability and establishes a natural hierarchy between classes.

Benefits of Using Inheritance

1. Code Reusability: Inheritance allows you to reuse existing code from a parent class, reducing redundancy and improving maintainability.

2. Method Overriding: It enables a subclass to provide a specific implementation of a method that is already defined in its parent class.

3. Polymorphism: Inheritance facilitates polymorphism, where a subclass can be treated as an instance of its parent class, allowing for more flexible and dynamic code.

4. Extensibility: You can extend existing classes to create new functionality without modifying the original class.

5. Modularity: Inheritance encourages a modular approach to program design, making code easier to manage and understand.

6. Data Abstraction: It helps in achieving data abstraction by allowing the use of general classes that can be specialized into more specific classes.

7. Maintenance: Enhancements and bug fixes in a parent class automatically propagate to all subclasses, simplifying maintenance.

Multiple Inheritance in Java

Java does not support multiple inheritance directly through classes to avoid the complexity and ambiguity that arises from the "Diamond Problem." However, Java allows multiple inheritance through interfaces.

Example of Multiple Inheritance Using Interfaces:

```
// Interface 1
interface Animal {
void eat();
}

// Interface 2
interface Bird {
void fly();
}

// A class that implements both interfaces
class Sparrow implements Animal, Bird {
@Override
public void eat() {
System.out.println("Sparrow is eating.");
}

@Override
public void fly() {
System.out.println("Sparrow is flying.");
}
}

// Main class
public class Main {
public static void main(String[] args) {
Sparrow sparrow = new Sparrow();
sparrow.eat();  // Output: Sparrow is eating.
sparrow.fly();  // Output: Sparrow is flying.
}
}
```

2. Why do we need event handling? Discuss the process of handling events with example.

Differentiate event listener interface with adapter class. (3+4+3)

Why Do We Need Event Handling?

Event handling is a crucial concept in software development, especially in graphical user interfaces (GUIs) and real-time applications. It allows programs to respond to user interactions or other types of events (such as mouse clicks, keyboard presses, or system-generated events). Without event handling, applications would be static, unable to interact dynamically with users or the environment.

Key reasons for event handling include:

1. Interactivity: Event handling enables the creation of interactive applications where the program can respond to user actions.

2. Asynchronous Processing: It allows applications to process events asynchronously, ensuring that the user interface remains responsive.

3. Custom Responses: Event handling allows developers to define custom behaviors for different events, making applications more versatile and user-friendly.

Process of Handling Events

Event handling in Java involves the following steps:

1. Event Source: The component (like a button) that generates an event when interacted with.

2. Event Object: When an event occurs, an event object is created containing details about the event (e.g., `ActionEvent`, `MouseEvent`).

3. Event Listener: An object that listens for events and responds accordingly. It must implement a specific event listener interface.

4. Event Handling Code: The code inside the event listener that defines what should happen when the event occurs.

Example: Handling a Button Click Event

```
import java.awt.*;

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



```
public class ButtonClickExample extends Frame implements ActionListener {
```

```
// Create a button
Button button;

ButtonClickExample() {
button = new Button("Click Me");
button.setBounds(100, 100, 80, 30);
button.addActionListener(this); // Register the button with the event listener

add(button); // Add button to the frame
setSize(300, 200);
setLayout(null);
setVisible(true);
}

// Event handler method
public void actionPerformed(ActionEvent e) {
System.out.println("Button clicked!");
}

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

Event Listener Interface vs. Adapter Class

Event Listener Interface:

1. Definition: An event listener interface is an interface that must be implemented by a class to handle specific events. Examples include `ActionListener`, `MouseListener`, `KeyListener`.

2. Multiple Methods: If an event listener interface has multiple methods (e.g., `MouseListener`), the implementing class must provide implementations for all of them, even if some are not needed.

3. Direct Implementation: The implementing class directly handles the events by overriding the required methods.

Example:

```java
class MyMouseListener implements MouseListener {
public void mouseClicked(MouseEvent e) { / handle click / }
public void mouseEntered(MouseEvent e) { / handle enter / }
public void mouseExited(MouseEvent e) { / handle exit / }
public void mousePressed(MouseEvent e) { / handle press / }
public void mouseReleased(MouseEvent e) { / handle release / }
}
```

Adapter Class:

1. Definition: An adapter class is an abstract class that provides default (empty) implementations for all methods of an event listener interface. A developer can extend the adapter class and override only the methods they need.

2. Convenience: Adapter classes are useful when you don't want to implement all methods of a listener interface, only the ones that are needed.

3. Simplified Implementation: The subclass only needs to override the methods relevant to the event handling.

Example:

```java
class MyMouseAdapter extends MouseAdapter {
@Override
public void mouseClicked(MouseEvent e) {
// handle click event only
}
}
```

Key Differences:

- Implementation Requirement: Event listener interfaces require implementing all methods, while adapter classes allow implementing only the needed methods.

- Ease of Use: Adapter classes simplify event handling when only a subset of listener methods is required.

- Inheritance: Using an adapter class involves inheritance, meaning the class that handles events must extend the adapter class. This limits multiple inheritances, which is not an issue with interfaces.

3. Write a program using swing components to find simple interest. Use text fields for

inputs and output. Your program should display output if the user clicks a button. (10)

Here's a concise Java program using Swing components to calculate simple interest. The program uses text fields for input and output, and the result is displayed when the user clicks a button.

```
import javax.swing.;

import java.awt.event.;


public class SimpleInterestCalculator {

public static void main(String[] args) {

// Create a frame

JFrame frame = new JFrame("Simple Interest Calculator");

frame.setSize(300, 200);

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

frame.setLayout(null);


// Create and position labels and text fields

JLabel principalLabel = new JLabel("Principal:");

principalLabel.setBounds(20, 20, 80, 25);

JTextField principalField = new JTextField();

principalField.setBounds(100, 20, 150, 25);


JLabel rateLabel = new JLabel("Rate (%):");

rateLabel.setBounds(20, 60, 80, 25);

JTextField rateField = new JTextField();

rateField.setBounds(100, 60, 150, 25);
```

```
JLabel timeLabel = new JLabel("Time (years):");
timeLabel.setBounds(20, 100, 80, 25);
JTextField timeField = new JTextField();
timeField.setBounds(100, 100, 150, 25);

// Create and position the button
JButton calculateButton = new JButton("Calculate");
calculateButton.setBounds(100, 140, 150, 25);

// Create and position the result field (non-editable)
JTextField resultField = new JTextField();
resultField.setBounds(20, 170, 230, 25);
resultField.setEditable(false);

// Add action listener to the button
calculateButton.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent e) {
// Parse input values
double principal = Double.parseDouble(principalField.getText());
double rate = Double.parseDouble(rateField.getText());
double time = Double.parseDouble(timeField.getText());

// Calculate simple interest
double interest = (principal  rate  time) / 100;

// Display the result
resultField.setText("Simple Interest: " + interest);
}
});

// Add components to the frame
frame.add(principalLabel);
```

```
frame.add(principalField);
frame.add(rateLabel);
frame.add(rateField);
frame.add(timeLabel);
frame.add(timeField);
frame.add(calculateButton);
frame.add(resultField);

// Set the frame visibility
frame.setVisible(true);
}
}
```

4. Write an object oriented program to find area and perimeter of rectangle. (5)

Here's a simple object-oriented program in Java to find the area and perimeter of a rectangle:

```
class Rectangle {
// Instance variables for the dimensions of the rectangle
private double length;
private double width;

// Constructor to initialize the length and width
public Rectangle(double length, double width) {
this.length = length;
this.width = width;
}

// Method to calculate the area
public double getArea() {
return length  width;
}
```

```
// Method to calculate the perimeter
public double getPerimeter() {
return 2  (length + width);
}

public static void main(String[] args) {
// Create a Rectangle object
Rectangle rect = new Rectangle(5.0, 3.0);

// Print the area and perimeter
System.out.println("Area: " + rect.getArea());
System.out.println("Perimeter: " + rect.getPerimeter());
}
}
```

5.Write a simple java program that reads data from one file and writes the data to another file (5)

Here's a simple Java program that reads data from one file and writes it to another file:

```
import java.io.;
public class FileCopy {
public static void main(String[] args) {
// Specify the source and destination file paths
String sourceFile = "input.txt";
String destinationFile = "output.txt";

try (
// Create file readers and writers
FileReader fr = new FileReader(sourceFile);
FileWriter fw = new FileWriter(destinationFile)
) {
int character;
```

```
// Read characters from the source file and write them to the destination file
while ((character = fr.read()) != -1) {
fw.write(character);
}
System.out.println("File copied successfully.");
} catch (IOException e) {
System.out.println("An error occurred: " + e.getMessage());
}
}
```

6. Discuss grid layout with example.

GridLayout is a layout manager in Java's AWT (Abstract Window Toolkit) and Swing that arranges components in a rectangular grid. It divides the container into equal-sized cells and places components into these cells. Each cell can hold only one component, and all cells are of equal size.

Key Features of GridLayout:

- Equal-Sized Cells: All cells in the grid are of equal size, which helps in creating a uniform layout.
- Row and Column Specification: You can specify the number of rows and columns when creating a `GridLayout` instance.
- Component Placement: Components are placed in the grid from left to right and top to bottom.

Example of GridLayout

Here's an example that demonstrates how to use `GridLayout` in a Swing application to arrange buttons in a grid:

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

public class GridLayoutExample {
public static void main(String[] args) {
// Create a frame
JFrame frame = new JFrame("GridLayout Example");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
// Create a GridLayout with 3 rows and 2 columns
frame.setLayout(new GridLayout(3, 2, 5, 5)); // (rows, columns, hgap, vgap)

// Add buttons to the frame
frame.add(new JButton("Button 1"));
frame.add(new JButton("Button 2"));
frame.add(new JButton("Button 3"));
frame.add(new JButton("Button 4"));
frame.add(new JButton("Button 5"));
frame.add(new JButton("Button 6"));

// Set frame size and make it visible
frame.setSize(300, 200);
frame.setVisible(true);
}
}
```

7. What is action event? Discuss (5)

An ActionEvent in Java is an event that is generated when a user interacts with a component that supports action events, such as buttons, menu items, or text fields. It is a fundamental part of event handling in Java's Abstract Window Toolkit (AWT) and Swing libraries.

Key Aspects of ActionEvent

1. Triggering Actions:

- ActionEvent is typically triggered by user actions such as clicking a button, selecting a menu item, or pressing Enter in a text field.

- It allows you to define specific behavior in response to these actions.

2. Event Object:

- An `ActionEvent` object contains information about the event, such as the source of the event (the component that triggered it) and the command string (if applicable).

- Common methods include `getSource()`, which returns the object that generated the event, and `getActionCommand()`, which returns the command string associated with the event.

3. ActionListener Interface:

- To handle `ActionEvent`, a class must implement the `ActionListener` interface, which requires defining the `actionPerformed(ActionEvent e)` method.

- The `actionPerformed` method is invoked when an action event occurs.

4. Registering Listeners:

- Components that generate action events must have an `ActionListener` registered with them. This is done using the `addActionListener(ActionListener listener)` method.

5. Use Cases:

- Commonly used in GUI applications to respond to user inputs, such as form submissions, button clicks, and menu selections.

Example of Handling ActionEvent

Here's a simple example demonstrating how to use `ActionEvent` in a Swing application to respond to a button click:

```
import javax.swing.;

import java.awt.event.;


public class ActionEventExample {

public static void main(String[] args) {

// Create a frame

JFrame frame = new JFrame("ActionEvent Example");

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

frame.setSize(300, 200);

frame.setLayout(null);


// Create a button

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

```
button.setBounds(100, 80, 100, 30);

// Add action listener to the button
button.addActionListener(new ActionListener() {
@Override
public void actionPerformed(ActionEvent e) {
// Handle the action event
JOptionPane.showMessageDialog(frame, "Button Clicked!");
}
});

// Add button to the frame
frame.add(button);

// Set frame visibility
frame.setVisible(true);
}
}
```

8. How do you execute SQL statement in JDBC? (5)

To execute SQL statements in JDBC (Java Database Connectivity), you follow a sequence of steps involving establishing a connection to a database, creating a `Statement` or `PreparedStatement` object, executing the SQL statement, and handling the results. Here's a step-by-step guide:

Steps to Execute SQL Statements in JDBC

1. Load the JDBC Driver:

- Ensure that the JDBC driver for your database is available in the classpath.

- Load the driver class (this step might be automatic for modern JDBC drivers).

2. Establish a Connection:

- Use `DriverManager` to establish a connection to the database using a connection URL, username, and password.

3. Create a Statement Object:

- Use the connection object to create a `Statement` or `PreparedStatement` object.

- `Statement` is used for simple queries, while `PreparedStatement` is used for parameterized queries and can offer better performance and security.

4. Execute the SQL Statement:

- Use the `executeQuery` method for SELECT statements to retrieve data.

- Use the `executeUpdate` method for INSERT, UPDATE, DELETE, or other DML statements to modify the database.

5. Process the Results:

- For queries, process the `ResultSet` object returned by `executeQuery`.

- For updates, check the number of affected rows returned by `executeUpdate`.

6. Close the Resources:

- Close the `ResultSet`, `Statement`, and `Connection` objects to free up database resources.

Example: Executing SQL Statements in JDBC

Here's a simple example demonstrating how to execute a SQL SELECT statement and an INSERT statement using JDBC:

```java
import java.sql.;

public class JdbcExample {
public static void main(String[] args) {
// Database URL, username, and password
String url = "jdbc:mysql://localhost:3306/mydatabase";
String username = "root";
```

```
String password = "password";

Connection connection = null;
Statement statement = null;
ResultSet resultSet = null;

try {
// Load the JDBC driver (optional for modern drivers)
Class.forName("com.mysql.cj.jdbc.Driver");

// Establish a connection
connection = DriverManager.getConnection(url, username, password);

// Create a statement object
statement = connection.createStatement();

// Execute a SELECT query
String selectQuery = "SELECT  FROM employees";
resultSet = statement.executeQuery(selectQuery);

// Process the result set
while (resultSet.next()) {
int id = resultSet.getInt("id");
String name = resultSet.getString("name");
System.out.println("ID: " + id + ", Name: " + name);
}

// Execute an INSERT query
String insertQuery = "INSERT INTO employees (name) VALUES ('John Doe')";
int rowsAffected = statement.executeUpdate(insertQuery);
System.out.println("Rows affected: " + rowsAffected);
```

```
} catch (SQLException | ClassNotFoundException e) {
e.printStackTrace();
} finally {
// Close resources
try {
if (resultSet != null) resultSet.close();
if (statement != null) statement.close();
if (connection != null) connection.close();
} catch (SQLException e) {
e.printStackTrace();
}
}
}
}
```

9. Write a Java program using servlet to display "Tribhuvan University". (5)

To create a Java servlet that displays "Tribhuvan University", follow these steps:

1. Servlet Code

Create a Java file named `DisplayServlet.java`:

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/display")
public class DisplayServlet extends HttpServlet {
private static final long serialVersionUID = 1L;
```

```
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
// Set the content type
response.setContentType("text/html");

// Write the response
response.getWriter().println("<html><body>");
response.getWriter().println("<h1>Tribhuvan University</h1>");
response.getWriter().println("</body></html>");
}
}
```

2. Web Deployment Descriptor (`web.xml`)

In your web application's `WEB-INF` directory, create or update `web.xml`:

```xml
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
version="3.1">

<servlet>
<servlet-name>DisplayServlet</servlet-name>
<servlet-class>DisplayServlet</servlet-class>
</servlet>

<servlet-mapping>
```

<servlet-name>DisplayServlet</servlet-name>

<url-pattern>/display</url-pattern>

</servlet-mapping>

</web-app>

```

**11.Discuss property design patterns for Java Beans. (5)**

Design Patterns for Properties

A property is a subset of a Bean's state. The values assigned to the properties determine the behavior and appearance of that component. A property is set through a setter method. A property is obtained by a getter method. There are two types of properties: simple and indexed.

Simple Properties

A simple property has a single value, where N is the name of the property and T is its type:

public T getN( )

public void setN(T arg)

A read/write property has both of these methods to access its values. A read- only property has only a get method. A write-only property has only a set method.

Boolean Properties:

A Boolean property has a value of true or false, where N is the name of the property:

public boolean isN( );

public boolean getN( );

public void setN(boolean value);

Either the first or second pattern can be used to retrieve the value of a Boolean property. However, if a class has both of these methods, the first pattern is used.

Indexed Properties

An indexed property consists of multiple values, where N is the name of the property and T is its type:

public T getN(int index);

public void setN(int index, T value); public T[ ] getN( );
```

public void setN(T values[ ]);

12.What is CORBA? How is it different from RMI? (2+3)

What is CORBA?

CORBA (Common Object Request Broker Architecture) is a framework developed by the Object Management Group (OMG) that allows software components written in different languages and running on different machines to communicate with each other. CORBA enables interoperability between heterogeneous systems by defining a standard for object communication and providing a way to invoke methods on objects in a distributed environment.

Key Features of CORBA:

1. Object Request Broker (ORB): The core component that handles communication between clients and servers.

2. Interface Definition Language (IDL): A language-neutral specification used to define the interfaces that objects expose.

3. Stubs and Skeletons: Generated code that helps in marshalling and unmarshalling data between clients and servers.

4. Interoperability: Allows components from different languages and platforms to work together.

How is CORBA Different from RMI?

RMI (Remote Method Invocation) and CORBA both provide mechanisms for remote communication in distributed systems, but they differ in several key aspects:

1. Language and Platform Support:

- CORBA: Language-agnostic and platform-independent. It uses IDL to define interfaces, which can be implemented in various programming languages (e.g., Java, C++, Python).

- RMI: Primarily Java-specific. It allows Java objects to invoke methods on other Java objects located on different JVMs.

2. Interface Definition:

- CORBA: Uses IDL (Interface Definition Language) to define interfaces, which are then mapped to specific programming languages.

- RMI: Uses Java interfaces directly; remote interfaces extend `java.rmi.Remote`, and method signatures must declare `throws RemoteException`.

3. Communication:

- CORBA: Utilizes IIOP (Internet Inter-ORB Protocol) for communication between ORBs. It can work over various transport protocols.

- RMI: Uses Java's RMI protocol for communication, which is built on top of Java's serialization mechanism and typically works over TCP/IP.

4. Flexibility and Complexity:

- CORBA: More complex due to its language-neutral approach and extensive features like CORBA Services (e.g., naming service, transaction service).

- RMI: More straightforward for Java applications but limited to Java environments.

13.Write short notes on: (2.5+2.5)

a) Multithreading

b) JSP

a) Multithreading

Multithreading is a programming concept that allows multiple threads to run concurrently within a single process. A thread is the smallest unit of execution in a program, and multithreading enables efficient utilization of CPU resources by performing multiple operations simultaneously.

Key Points:

- Concurrency: Multithreading enables concurrent execution, improving application performance and responsiveness.

- Threads: Threads share the same memory space but have their own stack, program counter, and local variables.

- Java Support: Java provides built-in support for multithreading through the `Thread` class and the `Runnable` interface.

- Synchronization: To avoid issues like race conditions, Java uses synchronization mechanisms such as synchronized methods and blocks to control access to shared resources.

Example:

```java
public class MyThread extends Thread {
public void run() {
System.out.println("Thread is running.");
}

public static void main(String[] args) {
MyThread t1 = new MyThread();
MyThread t2 = new MyThread();
t1.start(); // Starts thread t1
t2.start(); // Starts thread t2
}
}
```

b) JSP (JavaServer Pages)

JSP (JavaServer Pages) is a technology used to create dynamic web content in Java. It allows embedding Java code within HTML pages, enabling the generation of dynamic content based on user interactions or data from a server.

Key Points:

- Embedding Java: JSP files combine HTML with Java code to create dynamic web pages. Java code is embedded using special tags.

- Servlet Integration: JSPs are compiled into servlets by the server. They use standard JSP tags and expressions to handle logic and generate HTML.

- Directives and Tags: JSP includes directives (e.g., `<%@ page language="java" %>`) and custom tags (e.g., `<jsp:useBean>` for bean access) to manage content and functionality.

- Separation of Concerns: JSP promotes the separation of presentation from business logic. It is often used with JavaBeans or custom tags to manage business logic separately.

Example:


```jsp
```

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<title>JSP Example</title>
</head>
<body>
<h1>Hello, World!</h1>
<%
String name = "User";
out.println("Welcome, " + name + "!");
%>
</body>
</html>
```

Summary:

- Multithreading: Allows concurrent execution of threads, improving performance and responsiveness in applications.

- JSP: A Java technology for creating dynamic web pages by embedding Java code within HTML, compiled into servlets by the server.

14. Design a GUI form using swing with a text field, a text label for displaying the input message "Input any String", and three buttons with caption CheckPalindrome, Reverse, FindVowels. Write a complete program for above scenario and for checking palindrome in first button, reverse it after clicking second button and extract the vowels from it after clicking third button. (10)

Here is a complete Java Swing program that implements a GUI form with a text field, a label, and three buttons. Each button performs a specific action: checking if the input string is a palindrome, reversing the input string, and finding the vowels in the input string.

Complete Java Swing Program

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

public class StringManipulationGUI {
public static void main(String[] args) {
// Create the frame
JFrame frame = new JFrame("String Manipulation");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setSize(400, 200);
frame.setLayout(null);

// Create the label
JLabel label = new JLabel("Input any String:");
label.setBounds(10, 10, 150, 25);
frame.add(label);

// Create the text field
JTextField textField = new JTextField();
textField.setBounds(160, 10, 200, 25);
frame.add(textField);

// Create the "Check Palindrome" button
JButton checkPalindromeButton = new JButton("CheckPalindrome");
checkPalindromeButton.setBounds(10, 50, 150, 30);
```

```
frame.add(checkPalindromeButton);

// Create the "Reverse" button
JButton reverseButton = new JButton("Reverse");
reverseButton.setBounds(170, 50, 100, 30);
frame.add(reverseButton);

// Create the "Find Vowels" button
JButton findVowelsButton = new JButton("FindVowels");
findVowelsButton.setBounds(280, 50, 120, 30);
frame.add(findVowelsButton);

// Create a label to display the result
JLabel resultLabel = new JLabel();
resultLabel.setBounds(10, 90, 350, 25);
frame.add(resultLabel);

// Action listener for the "Check Palindrome" button
checkPalindromeButton.addActionListener(new ActionListener() {
@Override
public void actionPerformed(ActionEvent e) {
String text = textField.getText();
boolean isPalindrome = text.equals(new StringBuilder(text).reverse().toString());
resultLabel.setText("Palindrome: " + isPalindrome);
}
});

// Action listener for the "Reverse" button
reverseButton.addActionListener(new ActionListener() {
@Override
public void actionPerformed(ActionEvent e) {
String text = textField.getText();
```

```
String reversed = new StringBuilder(text).reverse().toString();
resultLabel.setText("Reversed: " + reversed);
}
});

// Action listener for the "Find Vowels" button
findVowelsButton.addActionListener(new ActionListener() {
@Override
public void actionPerformed(ActionEvent e) {
String text = textField.getText();
StringBuilder vowels = new StringBuilder();
for (char c : text.toCharArray()) {
if ("AEIOUaeiou".indexOf(c) != -1) {
vowels.append(c).append(" ");
}
}
resultLabel.setText("Vowels: " + vowels.toString().trim());
}
});

// Set frame visibility
frame.setVisible(true);
}
}
```

15. Why do we need to handle the exception? Distinguish error and exception, Write a program to demonstrate your own exception class. [1+2+7]

Why Do We Need to Handle Exceptions?

Exception handling is crucial in programming to manage errors gracefully, maintain application stability, and provide a better user experience. Here are the main reasons:

1. Error Management: Handle errors that occur during program execution without crashing the application.

2. Graceful Degradation: Allow the program to continue running or shut down gracefully, providing useful feedback to the user.

3. Maintainability: Helps in debugging and understanding issues by providing meaningful error messages and logs.

4. Control Flow: Allows developers to define how the application should respond to different error conditions.

Distinguishing Error and Exception

1. Error:

- Definition: Represents serious issues that are typically beyond the control of the application.

- Examples: `OutOfMemoryError`, `StackOverflowError`.

- Handling: Not usually handled by applications as they indicate severe problems that the application cannot recover from.

2. Exception:

- Definition: Represents conditions that a program can handle or recover from. Exceptions are used to indicate that something went wrong during execution.

- Examples: `IOException`, `SQLException`, `ArithmeticException`.

- Handling: Exceptions should be caught and handled using try-catch blocks to ensure the application can deal with errors and continue running.

Java Program to Demonstrate a Custom Exception

Here's a complete Java program that defines and uses a custom exception class:

Custom Exception Class

```java
// Define the custom exception class
```

```
class InvalidAgeException extends Exception {
public InvalidAgeException(String message) {
super(message);
}
}
```

Main Program

```java
public class CustomExceptionDemo {

// Method that throws the custom exception
public static void validateAge(int age) throws InvalidAgeException {
if (age < 0 || age > 150) {
throw new InvalidAgeException("Invalid age: " + age);
}
System.out.println("Valid age: " + age);
}

public static void main(String[] args) {
try {
validateAge(25);  // Valid age
validateAge(-5);  // This will cause an exception
} catch (InvalidAgeException e) {
System.out.println("Caught exception: " + e.getMessage());
}
}
}
```

16. Describe the process to deploy the servlet. Write a program to a JSP web form to take input of a student and submit it to second JSP file which may simply print the values of

form submission. [4+6]

Process to Deploy a Servlet

Deploying a servlet involves several steps to ensure it is properly set up and accessible within a web application. Here's a general outline of the process:

1. Write the Servlet Code:

- Create a Java class that extends `HttpServlet` and override the `doGet()` or `doPost()` methods to handle HTTP requests.

2. Compile the Servlet:

- Compile the servlet class using the Java compiler (`javac`). Ensure the servlet API is included in the classpath.

3. Package the Web Application:

- Place the compiled servlet class in the `WEB-INF/classes` directory of your web application.

- If using a build tool like Maven or Gradle, configure it to package your web application into a `.war` (Web Application Archive) file.

4. Configure the Web Application:

- Define the servlet in the `web.xml` deployment descriptor located in `WEB-INF` directory or use annotations in the servlet class for configuration.

5. Deploy the Application:

- Deploy the `.war` file or the web application directory to a servlet container like Apache Tomcat or Jetty.

- Copy the `.war` file to the `webapps` directory of Tomcat or use the deployment interface of the servlet container.

6. Start the Servlet Container:

- Start or restart the servlet container to deploy the application and make it accessible.

7. Access the Servlet:

- Open a web browser and navigate to the servlet URL (e.g., `http://localhost:8080/your-webapp/servlet-name`) to test the servlet.

JSP Web Form and Handling Example

Here’s a complete example with two JSP files:

1. First JSP File (input.jsp): A form to take student input.

2. Second JSP File (display.jsp): Displays the submitted values.

`input.jsp`

```jsp
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<title>Student Input Form</title>
</head>
<body>
<h1>Student Information Form</h1>
<form action="display.jsp" method="post">
<label for="name">Name:</label>
<input type="text" id="name" name="name" required>
<br><br>
<label for="age">Age:</label>
<input type="number" id="age" name="age" required>
<br><br>
<label for="email">Email:</label>
<input type="email" id="email" name="email" required>
<br><br>
<input type="submit" value="Submit">
</form>
```

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

`display.jsp`

```jsp
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<title>Student Information</title>
</head>
<body>
<h1>Submitted Student Information</h1>
<p><strong>Name:</strong> <%= request.getParameter("name") %></p>
<p><strong>Age:</strong> <%= request.getParameter("age") %></p>
<p><strong>Email:</strong> <%= request.getParameter("email") %></p>
</body>
</html>
```

17. An array is called balanced if it's even numbered elements (a[0], a[2], etc.) are even and its odd numbered elements (a[1], a[3],etc.) are Odd. Write a function named is Balanced that accepts an array of integers and returns 1 if the array is balanced otherwise it returns 0. [5]

To determine if an array is balanced, you need to check if all even-indexed elements are even and all odd-indexed elements are odd. Here's a Java function named `isBalanced` that performs this check:

Java Function

```java
public class ArrayBalancer {

// Function to check if the array is balanced
public static int isBalanced(int[] array) {
// Iterate through the array
for (int i = 0; i < array.length; i++) {
// Check even-indexed elements
if (i % 2 == 0) {
if (array[i] % 2 != 0) {
// Even index but element is not even
return 0;
}
} else {
// Check odd-indexed elements
if (array[i] % 2 == 0) {
// Odd index but element is not odd
return 0;
}
}
}
// Array is balanced
return 1;
}

public static void main(String[] args) {
// Test cases
int[] array1 = {2, 3, 4, 5};  // Balanced array
int[] array2 = {2, 3, 4, 6};  // Not balanced array

// Check if arrays are balanced
```

```
System.out.println("Array 1 is balanced: " + (isBalanced(array1) == 1 ? "Yes" : "No"));

System.out.println("Array 2 is balanced: " + (isBalanced(array2) == 1 ? "Yes" : "No"));

}

}
```

18. Explain the significance of cookies and sessions with suitable example? [5]

Cookies and sessions are fundamental concepts in web development used for maintaining state and storing user data between requests. Here's an explanation of their significance with suitable examples:

Cookies

Significance:

- State Persistence: Cookies are used to store small amounts of data on the client's browser. This data persists across multiple requests and sessions, making it useful for remembering user preferences or login information.

- Client-Side Storage: Data stored in cookies is maintained on the client's device, reducing the server's load.

Example:

A common use of cookies is to remember a user's login status or preferences.

```java
// Java Servlet example to set a cookie

import javax.servlet.*;

import javax.servlet.http.*;

import java.io.IOException;

public class CookieExample extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// Create a cookie

Cookie userCookie = new Cookie("username", "JohnDoe");

userCookie.setMaxAge(6060); // Cookie expires in 1 hour
```

```
// Add cookie to the response
response.addCookie(userCookie);

// Respond to the client
response.setContentType("text/html");
response.getWriter().println("Cookie has been set!");
}
}
```

Significance:

- Server-Side Storage: Sessions are used to store data on the server side. Each user is assigned a unique session ID, and data associated with that ID is stored on the server.

- State Management: Sessions are used to maintain state and user-specific information across multiple requests within the same user session, such as authentication status or shopping cart contents.

- Security: Since session data is stored on the server, it is more secure compared to cookies, which are stored on the client-side.

Example:

A typical use of sessions is to manage user login state.

```java
// Java Servlet example to use session
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.IOException;

public class SessionExample extends HttpServlet {
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
// Retrieve the session, create if not exists
```

```
HttpSession session = request.getSession();

// Store user information in the session
String username = request.getParameter("username");
session.setAttribute("username", username);

// Respond to the client
response.setContentType("text/html");
response.getWriter().println("Session data stored for user: " + username);
}
}
```

Summary

- Cookies:
- Stored on the client-side.
- Used for storing small amounts of data like user preferences or login states.
- Data is sent with every request to the server.

- Sessions:
- Stored on the server-side.
- Used for maintaining user-specific data and state across multiple requests.
- More secure as sensitive data is not exposed to the client.

Both cookies and sessions play a crucial role in managing user data and maintaining state in web applications. Cookies are useful for client-side persistence, while sessions provide secure server-side storage.

19. Define the chain of constructor. What is the purpose of private constructor? [5]

Chain of Constructors

Definition:

The chain of constructors refers to the process where one constructor in a class calls another constructor within the same class. This allows for constructor reuse and the initialization of default values or common setup code.

Purpose:

- Constructor Overloading: To provide multiple ways to initialize an object with different sets of parameters.

- Code Reuse: To avoid duplicating initialization code by centralizing it in one constructor and calling it from other constructors.

- Flexible Initialization: To offer different initialization options based on the provided parameters.

How It Works:

- Constructor Calling Another Constructor: In Java, you use `this()` to call another constructor in the same class. This must be the first statement in the calling constructor.

Example:

```
public class Rectangle {

private int width;

private int height;


// Constructor with no parameters

public Rectangle() {

this(0, 0); // Call the parameterized constructor

}


// Constructor with two parameters

public Rectangle(int width, int height) {

this.width = width;

this.height = height;

}
```

```
public int getArea() {
return width  height;
}

public static void main(String[] args) {
Rectangle rect1 = new Rectangle(); // Calls the no-arg constructor
Rectangle rect2 = new Rectangle(5, 10); // Calls the parameterized constructor

System.out.println("Area of rect1: " + rect1.getArea()); // Output: 0
System.out.println("Area of rect2: " + rect2.getArea()); // Output: 50
}
}
```

In this example:

- The no-argument constructor `Rectangle()` calls the parameterized constructor `Rectangle(int width, int height)` using `this(0, 0)`. This allows both constructors to share the initialization logic.

Purpose of Private Constructor

Definition:

A private constructor is a constructor that cannot be accessed from outside the class. It is declared with the `private` access modifier.

Purpose:

1. Singleton Pattern: To implement the Singleton design pattern, ensuring that only one instance of a class is created. By making the constructor private, the class itself controls the creation of its single instance.

2. Utility Classes: To prevent instantiation of utility classes that contain static methods only. These classes are not meant to be instantiated, so their constructors are made private.

3. Controlled Instantiation: To restrict object creation and control it through other static methods or factory methods within the class.

Example (Singleton Pattern):

```java
public class Singleton {
private static Singleton instance;

// Private constructor to prevent instantiation
private Singleton() {}

// Public method to provide access to the single instance
public static Singleton getInstance() {
if (instance == null) {
instance = new Singleton();
}
return instance;
}
}

public class Main {
public static void main(String[] args) {
// Accessing the single instance of Singleton
Singleton singleton = Singleton.getInstance();
System.out.println("Singleton instance: " + singleton);
}
}
```

In this example:

- The `Singleton` class has a private constructor to prevent external instantiation.

- The `getInstance()` method provides a controlled way to access the single instance of the class.

Summary

- Chain of Constructors:
- Refers to calling one constructor from another within the same class using `this()`.
- Provides constructor overloading, code reuse, and flexible initialization.

- Private Constructor:
- Prevents external instantiation of a class.
- Used in Singleton patterns, utility classes, and controlled instantiation scenarios.

20. Describe the process to run the RMI application. [5]

Running a Remote Method Invocation (RMI) application involves several steps, including setting up the RMI server, defining the remote interface, implementing the remote object, and running the RMI registry. Here's a detailed process to run an RMI application:

Summary

1. Define Remote Interface: Create an interface extending `Remote`.
2. Implement Remote Interface: Implement the interface and extend `UnicastRemoteObject`.
3. Create RMI Server: Set up and bind the remote object to the RMI registry.
4. Create RMI Client: Lookup the remote object and invoke methods.
5. Compile and Run: Compile the classes, start the RMI registry, and run the server and client.

Following these steps ensures that the RMI application is properly set up and can communicate between the server and client.

21. How prepared statements are different with statement? List the types of JDBC

driver.[2+3]

Differences Between `PreparedStatement` and `Statement`

1. SQL Injection Prevention:

- `Statement`: Vulnerable to SQL injection attacks if user input is included directly in the SQL query.

- `PreparedStatement`: Provides built-in protection against SQL injection by using placeholders (parameters) in the SQL query.

2. Performance:

- `Statement`: The SQL query is parsed, compiled, and executed each time it is run. This can be inefficient if the same query is executed multiple times with different parameters.

- `PreparedStatement`: The SQL query is precompiled and optimized once. Subsequent executions with different parameters are faster because the query plan is reused.

3. Parameter Handling:

- `Statement`: Does not support parameters. You need to manually concatenate query strings, which can be error-prone and unsafe.

- `PreparedStatement`: Supports parameterized queries, allowing you to set values for placeholders using setter methods (`setInt()`, `setString()`, etc.).

4. Readability and Maintainability:

- `Statement`: Requires manual concatenation of parameters into the SQL string, which can lead to messy code and harder maintenance.

- `PreparedStatement`: Uses placeholders and setter methods, making the code cleaner and easier to maintain.

5. Execution:

- `Statement`: Typically used for simple queries and updates where parameters are not required.

- `PreparedStatement`: Designed for executing parameterized queries, especially beneficial when the same query needs to be executed multiple times with different values.

Example of `Statement` vs `PreparedStatement`:

Using `Statement`:

```java
String query = "SELECT  FROM users WHERE username = '" + username + "'";
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(query);
```

Using `PreparedStatement`:

```java
String query = "SELECT  FROM users WHERE username = ?";
PreparedStatement pstmt = connection.prepareStatement(query);
pstmt.setString(1, username);
ResultSet rs = pstmt.executeQuery();
```

Types of JDBC Drivers

1. JDBC-ODBC Bridge Driver (Type 1):

- Description: Converts JDBC calls into ODBC calls and uses ODBC drivers to interact with the database.

- Advantages: Simple to use with existing ODBC drivers.

- Disadvantages: Performance overhead due to additional layer, and it is no longer supported in recent versions of Java.

2. Native-API Driver (Type 2):

- Description: Converts JDBC calls into database-specific native API calls. Requires native library installation on the client machine.

- Advantages: Better performance than Type 1, as it directly uses the database's native protocol.

- Disadvantages: Requires database-specific libraries, making it less portable.

3. Network Protocol Driver (Type 3):

- Description: Converts JDBC calls into a database-independent network protocol, which is then converted into database-specific calls by a server-side component.

- Advantages: Can be used with multiple databases by configuring the middle-tier server.

- Disadvantages: Requires a middle-tier server and additional configuration.

4. Pure Java Driver (Type 4):

- Description: Converts JDBC calls directly into the database-specific protocol using Java, without needing native libraries.

- Advantages: Best performance, platform-independent, and no need for native libraries.

- Disadvantages: Requires knowledge of the database's network protocol.

Summary

- `PreparedStatement` vs `Statement`:

- `PreparedStatement` is more secure, efficient, and easier to maintain due to precompilation and parameterization.

- JDBC Driver Types:

- Type 1: JDBC-ODBC Bridge (now deprecated).

- Type 2: Native-API Driver.

- Type 3: Network Protocol Driver.

- Type 4: Pure Java Driver.

Each JDBC driver type has its own advantages and is suitable for different scenarios based on the application's requirements and environment.

22. Write a Java program to find the sum of two numbers using swing components. Use text fields for input and output. Your program displays output if you press any key in keyboard. Use key adapter to handle events.

To create a Java Swing application that finds the sum of two numbers using text fields for input and output, and displays the result when any key is pressed, you can follow these steps. This example will use a `KeyAdapter` to handle key events.

Here's a complete Java program demonstrating this:

Java Program

```java
import javax.swing.*;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;

public class SumCalculator extends JFrame {
private JTextField inputField1;
private JTextField inputField2;
private JTextField resultField;

public SumCalculator() {
// Set up the frame
setTitle("Sum Calculator");
setSize(300, 150);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setLayout(null);

// Initialize text fields
inputField1 = new JTextField();
inputField1.setBounds(30, 30, 80, 25);
add(inputField1);

inputField2 = new JTextField();
inputField2.setBounds(120, 30, 80, 25);
add(inputField2);
```

```
resultField = new JTextField();
resultField.setBounds(30, 70, 170, 25);
resultField.setEditable(false);
add(resultField);

// Add key listener to the frame
KeyListener keyListener = new KeyAdapter() {
@Override
public void keyPressed(KeyEvent e) {
calculateSum();
}
};
addKeyListener(keyListener);
inputField1.addKeyListener(keyListener);
inputField2.addKeyListener(keyListener);

// Make frame visible
setVisible(true);
}

private void calculateSum() {
try {
// Get the values from text fields
String text1 = inputField1.getText();
String text2 = inputField2.getText();

// Parse the numbers
int num1 = Integer.parseInt(text1.isEmpty() ? "0" : text1);
int num2 = Integer.parseInt(text2.isEmpty() ? "0" : text2);

// Calculate the sum
int sum = num1 + num2;
```

```
// Display the result
resultField.setText(String.valueOf(sum));
} catch (NumberFormatException e) {
// Handle invalid input
resultField.setText("Invalid input");
}
}

public static void main(String[] args) {
// Create the GUI in the Event Dispatch Thread
SwingUtilities.invokeLater(() -> new SumCalculator());
}
}
```

Explanation

1. Setup Frame:
- Title: Sets the title of the JFrame.
- Size: Defines the size of the window.
- Layout: Uses `null` layout to manually set component bounds.
- Close Operation: Closes the application when the window is closed.

2. Initialize Components:
- `JTextField`: For input fields and result display.
- Bounds: Set the size and position of each component.

3. Add Key Listener:
- KeyAdapter: Custom implementation of `KeyAdapter` to listen for key events.
- `keyPressed` Method: Calls `calculateSum()` method when any key is pressed.

4. Calculate Sum:

- Get Values: Retrieve text from input fields.

- Parse Integers: Convert text to integers. Default to `0` if text is empty.

- Compute Sum: Add the two numbers.

- Display Result: Update the result field with the sum or an error message if input is invalid.

5. Main Method:

- SwingUtilities.invokeLater(): Ensures that the GUI creation runs on the Event Dispatch Thread for thread safety.

Summary

This program creates a simple Swing-based GUI where the user can input two numbers. When any key is pressed, it calculates and displays the sum of the two numbers in a non-editable text field. The use of `KeyAdapter` allows handling key events efficiently.

23. **Define servlet. Discuss life cycle of servlet. Differentiate servlet with JSP.**

A Servlet is a Java programming language class that is used to extend the capabilities of servers. It is commonly used to handle requests and responses in web applications. Servlets are part of the Java EE (Enterprise Edition) specification and are managed by a web container (or servlet container) such as Apache Tomcat or Jetty. They can process requests, generate dynamic web content, and interact with databases or other services.

Life Cycle of a Servlet

The life cycle of a servlet consists of several phases managed by the servlet container:

1. Loading and Instantiation:

- The servlet container loads the servlet class into memory and creates an instance of the servlet. This occurs when the servlet is first requested or when the container starts up (if the servlet is configured to load on startup).

2. Initialization:

- The servlet container calls the `init()` method of the servlet. This method is used for one-time initialization, such as setting up resources or configurations. The `init()` method is called once for the servlet instance.

- Signature: `public void init(ServletConfig config) throws ServletException`

3. Request Handling:

- After initialization, the servlet can handle multiple requests. For each request, the servlet container calls the `service()` method. This method processes the request and generates a response. It receives `HttpServletRequest` and `HttpServletResponse` objects.

- Signature: `public void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException`

4. Destruction:

- When the servlet is no longer needed or the server is shutting down, the servlet container calls the `destroy()` method. This method is used to release resources and perform cleanup operations. The `destroy()` method is called once before the servlet is removed from memory.

- Signature: `public void destroy()`

Differentiation Between Servlet and JSP

1. Definition:

- Servlet: A Java class used to handle requests and generate responses in a web application. It is typically used for complex logic and processing.

- JSP (JavaServer Pages): A technology used to create dynamic web pages with HTML, Java code, and JSP tags. It is primarily used for presentation and layout, allowing Java code to be embedded in HTML.

2. Usage:

- Servlet: More suitable for handling complex business logic, interacting with databases, and managing application state.

- JSP: Designed to simplify the creation of web content and presentation. It is often used for generating HTML dynamically and is generally more user-friendly for designing web pages.

3. Compilation:

- Servlet: Compiled into Java bytecode and executed on the server. Servlets are Java classes and require explicit code to generate HTML output.

- JSP: Compiled into a servlet by the JSP engine. The JSP file is transformed into a servlet and then compiled, allowing HTML and Java code to coexist.

4. Code Separation:

- Servlet: Java code and HTML are mixed within the servlet class. This can make the code harder to maintain.

- JSP: Separates HTML content and Java code, promoting a clearer separation between presentation and logic. JSP uses JSP tags and expressions for embedding Java code.

5. Development:

- Servlet: Typically involves more boilerplate code and manual handling of response generation.

- JSP: Offers a more concise syntax for mixing Java code with HTML, making it easier to develop and maintain web pages.

Simple Example of Servlet and JSP

Servlet Example:

```java
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class SimpleServlet extends HttpServlet {
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
response.setContentType("text/html");
response.getWriter().println("<html><body>");
response.getWriter().println("<h1>Hello from Servlet!</h1>");
response.getWriter().println("</body></html>");
```

```
}
}
```

JSP Example:

```jsp
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<title>Simple JSP</title>
</head>
<body>
<h1>Hello from JSP!</h1>
</body>
</html>
```

Summary

- Servlet: Java class for request handling and response generation, used for complex logic and server-side processing.
- JSP: Technology for creating dynamic web pages with a combination of HTML and Java code, mainly used for presentation.

The servlet life cycle involves loading, initializing, handling requests, and destruction phases, managed by the servlet container. While servlets are suited for handling logic, JSPs simplify web page creation and layout.

24. Discuss MVC design pattern with example.

The MVC (Model-View-Controller) design pattern is a software architectural pattern used for implementing user interfaces by separating an application into three interconnected components. This separation helps manage complexity and makes the application easier to maintain and scale.

Components of MVC

1. Model:

- Definition: Represents the data and the business logic of the application. It directly manages the data, logic, and rules of the application.

- Responsibilities:

- Retrieve data from the database or other sources.

- Update data.

- Notify the view of changes to the data.

- Example: A class representing a database entity, such as `User`, with methods to interact with the database.

2. View:

- Definition: Represents the presentation layer. It displays the data from the model to the user and sends user input to the controller.

- Responsibilities:

- Render the data provided by the model.

- Display the data in a user-friendly format.

- Provide user interface elements like forms and buttons.

- Example: A JSP page or an HTML template that displays user information.

3. Controller:

- Definition: Acts as an intermediary between the model and the view. It processes user inputs, interacts with the model, and updates the view.

- Responsibilities:

- Handle user input and update the model based on that input.

- Update the view to reflect changes in the model.

- Decide which view to display based on user actions and the state of the model.

- Example: A servlet or a controller class that processes form submissions and updates the model.

How MVC Works

1. User Interaction: The user interacts with the view (e.g., clicks a button or submits a form).

2. Controller Handling: The controller receives the input from the view and processes it (e.g., updates the model or performs some logic).

3. Model Update: The model is updated based on the controller's actions.

4. View Update: The view is updated to reflect changes in the model.

Example of MVC in Java with a Simple Application

Let's create a simple example of a web application that uses MVC to display and update user information.

1. Model: User.java

```java
public class User {
private String name;
private int age;

public User(String name, int age) {
this.name = name;
this.age = age;
}

public String getName() {
return name;
}

public void setName(String name) {
this.name = name;
}
```

```
public int getAge() {
return age;
}

public void setAge(int age) {
this.age = age;
}
}
```

2. View: user.jsp

```jsp
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<title>User Information</title>
</head>
<body>
<h2>User Information</h2>
<form action="updateUser" method="post">
Name: <input type="text" name="name" value="${user.name}"><br>
Age: <input type="text" name="age" value="${user.age}"><br>
<input type="submit" value="Update">
</form>
<p>Name: ${user.name}</p>
<p>Age: ${user.age}</p>
</body>
</html>
```

3. Controller: UserController.java

```java
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import java.io.IOException;

@WebServlet("/user")
public class UserController extends HttpServlet {

@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
HttpSession session = request.getSession();
User user = (User) session.getAttribute("user");

if (user == null) {
user = new User("John Doe", 25); // Default user
session.setAttribute("user", user);
}

request.setAttribute("user", user);
request.getRequestDispatcher("/user.jsp").forward(request, response);
}

@Override
```

```
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

String name = request.getParameter("name");

int age = Integer.parseInt(request.getParameter("age"));


HttpSession session = request.getSession();

User user = (User) session.getAttribute("user");


if (user == null) {

user = new User(name, age);

session.setAttribute("user", user);

} else {

user.setName(name);

user.setAge(age);

}


request.setAttribute("user", user);

request.getRequestDispatcher("/user.jsp").forward(request, response);

}

}
```


Summary

- Model: Manages data and business logic.

- View: Displays data to the user and provides UI elements.

- Controller: Handles user input, updates the model, and changes the view.

The MVC pattern promotes separation of concerns, making the application easier to maintain, test, and scale by clearly defining roles for each component.

25. What are the benefits of using JDBC? What is prepared statement?

Benefits of Using JDBC

JDBC (Java Database Connectivity) is a Java API that provides a standard interface for connecting to relational databases from Java applications. Here are the key benefits of using JDBC:

1. Database Independence:

- Description: JDBC provides a uniform interface for different databases. By using JDBC, you can switch between different databases with minimal changes to your code.

- Benefit: Helps in developing applications that are portable and can work with multiple database systems.

2. Standard API:

- Description: JDBC defines a standard set of interfaces and classes for database access, including handling SQL queries, updates, and results.

- Benefit: Provides a consistent way to interact with databases, making it easier for developers to work with different databases using the same API.

3. Support for Multiple Database Operations:

- Description: JDBC supports various database operations, including executing SQL queries, updates, and managing transactions.

- Benefit: Allows comprehensive interaction with databases, including CRUD (Create, Read, Update, Delete) operations.

4. Integration with Java Applications:

- Description: JDBC integrates seamlessly with Java applications, enabling Java programs to interact with databases directly.

- Benefit: Facilitates database access within Java applications, allowing for dynamic data manipulation and retrieval.

5. Transaction Management:

- Description: JDBC supports transaction management, including commit and rollback operations.

- Benefit: Ensures data integrity and consistency by managing transactions and handling errors gracefully.

6. Support for Batch Processing:

- Description: JDBC allows executing multiple SQL statements in a batch, reducing network round-trips to the database.

- Benefit: Improves performance and efficiency when dealing with large amounts of data.

Prepared Statement

A PreparedStatement is a special type of JDBC statement used to execute precompiled SQL queries with parameters. It extends the `Statement` interface and provides several advantages over regular `Statement` objects.

Key Features of PreparedStatement:

1. Parameterization:

- Description: Prepared statements use placeholders (`?`) in the SQL query, which are later replaced with actual values using setter methods.

- Benefit: Prevents SQL injection attacks by separating the SQL code from the data, and ensures that user input is treated safely.

2. Precompilation:

- Description: The SQL query is precompiled and optimized when the `PreparedStatement` object is created.

- Benefit: Improves performance by reusing the precompiled query plan for multiple executions, reducing the overhead of parsing and compiling the SQL statement.

3. Efficient Execution:

- Description: Prepared statements allow executing the same query with different parameters multiple times.

- Benefit: Reduces network traffic and processing time compared to executing individual statements.

4. Improved Readability:

- Description: The use of placeholders and setter methods makes the code cleaner and easier to read.

- Benefit: Enhances code maintainability and reduces errors related to string concatenation.

5. Automatic Handling of SQL Data Types:

- Description: Prepared statements automatically handle data types and format conversions.

- Benefit: Simplifies the code and reduces the chances of errors due to incorrect data types.

Example of PreparedStatement:

Using `Statement`:

```java
String query = "SELECT  FROM users WHERE username = '" + username + "'";
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(query);
```

Using `PreparedStatement`:

```java
String query = "SELECT  FROM users WHERE username = ?";
PreparedStatement pstmt = connection.prepareStatement(query);
pstmt.setString(1, username);
ResultSet rs = pstmt.executeQuery();
```

In the `PreparedStatement` example:

- The `?` placeholder is used in the query.

- The `setString()` method is used to safely insert the value of `username` into the query.

Summary

- Benefits of JDBC: Provides database independence, a standard API, supports multiple operations, integrates with Java applications, manages transactions, and supports batch processing.

- PreparedStatement: A type of statement in JDBC that supports parameterized queries, precompilation, efficient execution, improved readability, and automatic handling of SQL data types.

26. Define Java Bean. How is it different from other Java programs? What is design pattern?

Java Bean

A Java Bean is a reusable software component that follows certain conventions in its design. It is a special kind of Java class that adheres to specific rules which allow it to be manipulated in visual development environments and accessed through introspection.

Characteristics of Java Beans:

1. Serializable:

- A Java Bean must implement the `Serializable` interface, which allows its state to be saved and restored.

2. No-Arg Constructor:

- A Java Bean must have a public no-argument constructor, which allows for easy creation and initialization.

3. Properties:

- Java Beans use getter and setter methods to expose their properties. For example, for a property `name`, there should be `getName()` and `setName(String name)` methods.

4. Encapsulation:

- Properties are accessed through methods, maintaining encapsulation. Fields are usually private, and access is provided via public getter and setter methods.

5. Events (Optional):

- Java Beans can also handle events, allowing them to be used in event-driven environments.

Example of a Java Bean:

```java
import java.io.Serializable;
```

```
public class Person implements Serializable {
private String name;
private int age;

// No-argument constructor
public Person() {
}

// Getter for name
public String getName() {
return name;
}

// Setter for name
public void setName(String name) {
this.name = name;
}

// Getter for age
public int getAge() {
return age;
}

// Setter for age
public void setAge(int age) {
this.age = age;
}
}
```


Difference from Other Java Programs

1. Standard Conventions:

- Java Beans follow a specific set of conventions (e.g., naming conventions for getters and setters, having a no-arg constructor) which makes them more predictable and compatible with various tools and frameworks.

2. Reusable Components:

- Java Beans are designed to be reusable and can be easily manipulated in visual development environments or frameworks. Other Java programs may not adhere to these conventions and might not be as easily reusable or introspectable.

3. Serialization:

- Java Beans are required to be serializable, allowing their state to be persisted and restored. While other Java classes can also be serializable, it is a fundamental requirement for Java Beans.

4. Encapsulation through Methods:

- Java Beans use getter and setter methods for encapsulating properties. While other Java classes may also use methods for encapsulation, Java Beans strictly follow this pattern to ensure compatibility with various tools.

Design Pattern

A Design Pattern is a general, reusable solution to a common problem that occurs within a given context in software design. It provides a template for how to solve a problem in a way that is both effective and flexible.

Key Aspects of Design Patterns:

1. Problem-Solution Template:

- Design patterns offer a template for solving problems, which can be applied in various situations.

2. Best Practices:

- They represent best practices and solutions that have been proven effective over time.

3. Reusability:

- Patterns are designed to be reused across different projects, improving the consistency and quality of software design.

4. Types of Patterns:

- Creational Patterns: Deal with object creation mechanisms, e.g., Singleton, Factory Method, Abstract Factory.

- Structural Patterns: Focus on object composition and the organization of classes, e.g., Adapter, Composite, Decorator.

- Behavioral Patterns: Concerned with object interaction and responsibility, e.g., Observer, Strategy, Command.

Example of a Design Pattern: Singleton

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance.

Singleton Example in Java:

```java
public class Singleton {
private static Singleton instance;

// Private constructor to prevent instantiation
private Singleton() {
}

// Public method to provide access to the instance
public static synchronized Singleton getInstance() {
if (instance == null) {
instance = new Singleton();
}
return instance;
}
```

}

```

Summary

- Java Bean: A reusable Java component that follows conventions such as having a no-arg constructor, implementing `Serializable`, and using getter/setter methods.

- Difference from Other Java Programs: Java Beans adhere to specific conventions for easy manipulation, reuse, and compatibility with tools.

- Design Pattern: A general solution to common software design problems, offering best practices and reusable solutions. Examples include creational, structural, and behavioral patterns.

**27. How do you handle HTTP request (GET) using servlet?**

Handling HTTP GET requests using a servlet involves implementing a servlet class that overrides the `doGet()` method. Here's a step-by-step guide on how to handle GET requests in a servlet:

Steps to Handle HTTP GET Requests

1. Create a Servlet Class:

- Extend the `HttpServlet` class to create your servlet.

- Override the `doGet()` method to handle GET requests.

2. Configure the Servlet:

- Define the servlet in the `web.xml` configuration file or use annotations (if using Servlet 3.0 or later) to map the servlet to a URL pattern.

3. Implement the `doGet()` Method:

- In the `doGet()` method, process the request, interact with the model (if needed), and generate the response.

4. Deploy the Servlet:
```

- Deploy the servlet in a web container like Apache Tomcat and test it by sending GET requests.

Example Servlet Handling HTTP GET Request

Here's a simple example of a servlet that handles HTTP GET requests:

1. Create the Servlet Class

```java
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HelloWorldServlet extends HttpServlet {

@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
// Set the content type of the response
response.setContentType("text/html");

// Write the response to the client
response.getWriter().println("<html><body>");
response.getWriter().println("<h1>Hello, World!</h1>");
response.getWriter().println("</body></html>");
}
}
```


2. Configure the Servlet (web.xml)

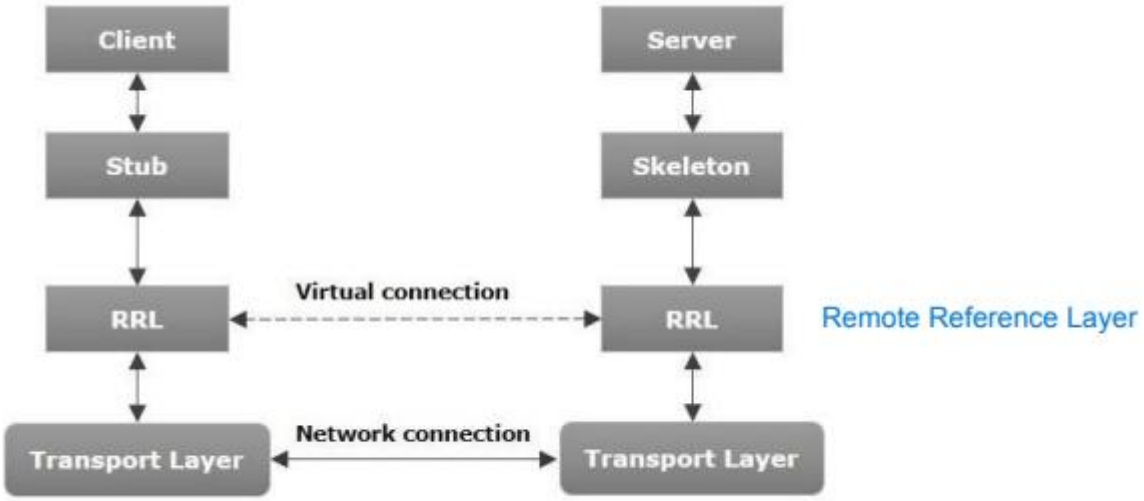
In `WEB-INF/web.xml`, define the servlet and map it to a URL pattern:

```xml
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
version="3.1">

<servlet>
<servlet-name>HelloWorldServlet</servlet-name>
<servlet-class>com.example.HelloWorldServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>HelloWorldServlet</servlet-name>
<url-pattern>/hello</url-pattern>
</servlet-mapping>

</web-app>
```

Alternatively, you can use annotations (Servlet 3.0 and later):

```java
import javax.servlet.annotation.WebServlet;

@WebServlet("/hello")
public class HelloWorldServlet extends HttpServlet {
// doGet method as shown earlier
}
```

```

3. Deploy and Test

- Deploy the application to a servlet container like Apache Tomcat.
- Access the servlet using a URL like `http://localhost:8080/your-app-context/hello`.

Summary

1. Create a Servlet Class: Extend `HttpServlet` and override `doGet()`.
2. Configure the Servlet: Map the servlet to a URL pattern in `web.xml` or using annotations.
3. Implement `doGet()`: Handle the request and generate a response.
4. Deploy and Test: Deploy the servlet to a web container and access it via a browser.

This approach allows you to process HTTP GET requests and generate dynamic content in response.

***28. What are different layers of RMI architecture? Explain.***

Architecture of an RMI Application

In an RMI application, we write two programs, a server program

(resides on the server) and a client program (resides on the client).

• Inside the server program, a remote object is created and

reference of that object is made available for the client (using

the registry).

• The client program requests the remote objects on the server

and tries to invoke its methods.

The following diagram shows the architecture of an RMI application.
```



• Transport Layer − This layer connects the client and the server. It manages the existing connection and also sets up new connections.

• Stub − A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.

• Skeleton − This is the object which resides on the server side. stub communicates with this skeleton to pass request to the remote object.

• RRL(Remote Reference Layer) − It is the layer which manages the references made by the client to the remote object.

29.. **Write short notes on:**

Servlet API

RMI vs CORBA

Servlet API

The Servlet API is a set of classes and interfaces in Java that allow developers to create web-based applications. A servlet is a Java class that extends the capabilities of a server and responds to requests from clients, typically over HTTP.

Key Components of Servlet API:

1. Servlet Interface:

- The core interface that all servlets must implement. The `javax.servlet.Servlet` interface defines the life cycle methods (`init()`, `service()`, and `destroy()`), which the servlet container calls to manage the servlet's life cycle.

2. HttpServlet:

- A subclass of `GenericServlet` that provides methods specifically for handling HTTP requests. The most commonly used methods are `doGet()` and `doPost()` to handle GET and POST requests respectively.

3. ServletRequest and ServletResponse:

- `ServletRequest`: Provides information about the client's request, including parameters, headers, and attributes.

- `ServletResponse`: Allows the servlet to send a response back to the client, such as HTML content.

4. ServletContext:

- An interface that provides a view of the web application's environment. It allows servlets to share information and resources, such as application-wide parameters, across different servlets within the same application.

5. ServletConfig:

- Provides configuration information to the servlet at initialization time, such as initialization parameters specified in the deployment descriptor (`web.xml`).

6. Session Management:

- The Servlet API provides session management capabilities through the `HttpSession` interface, allowing web applications to maintain state across multiple requests from the same client.

7. RequestDispatcher:

- Allows for request forwarding or including content from another resource (such as another servlet or JSP) within the same web application.

Use Cases:

- Creating dynamic web content, such as generating HTML, processing forms, handling file uploads, and interacting with databases.

- Managing sessions and cookies to maintain user state across multiple requests.

RMI vs CORBA

RMI (Remote Method Invocation) and CORBA (Common Object Request Broker Architecture) are both technologies that allow objects to communicate with each other across different processes, possibly on different machines. However, they differ in several aspects:

RMI (Remote Method Invocation):

1. Language-Specific:

- RMI is specific to Java. It allows Java objects to invoke methods on other Java objects located on different JVMs, making it tightly coupled with the Java programming language.

2. Ease of Use:

- RMI is simpler to use within Java applications since it leverages Java's native serialization and type system. Developers don't need to learn new IDLs (Interface Definition Languages) or protocols.

3. Transport Protocol:

- RMI uses Java's native networking capabilities, typically over TCP/IP, and uses Java Object Serialization for marshaling and unmarshaling objects.

4. Java-Centric:

- RMI is limited to environments where both the client and server are implemented in Java. It does not easily support communication with applications written in other languages.

CORBA (Common Object Request Broker Architecture):

1. Language-Independent:

- CORBA is designed to be language-agnostic. It allows objects written in different programming languages (e.g., Java, C++, Python) to communicate with each other. This is achieved through the use of an IDL (Interface Definition Language), which defines the interfaces in a language-neutral way.

2. Complexity:

- CORBA is more complex than RMI, both in terms of setup and usage. Developers must work with IDLs and understand the underlying ORB (Object Request Broker) architecture.

3. Interoperability:

- CORBA excels in environments where interoperability between different languages and platforms is required. It is designed to work across different operating systems and network protocols.

4. Transport Protocol:

- CORBA uses the IIOP (Internet Inter-ORB Protocol) for communication, which is a more sophisticated protocol that supports features like load balancing and fault tolerance.

Summary:

- RMI is best suited for Java-to-Java communication within a homogeneous environment, offering simplicity and tight integration with Java.

- CORBA is ideal for heterogeneous environments where different programming languages and platforms need to interact, offering greater flexibility but at the cost of increased complexity.

29. What is the significance of stub and skeleton In RMI? Create a RMI application such that

a client sends an Integer number to the server and the server return the factorial value

of that integer. Give a clear specification for every step. (10)

Significance of Stub and Skeleton in RMI

In Java RMI (Remote Method Invocation), Stub and Skeleton play critical roles in the communication between a client and a server:

1. Stub (Client-Side Proxy):

- The stub acts as a proxy on the client side. When a client calls a method on a remote object, it is the stub that actually receives the call.

- The stub is responsible for marshaling (converting) the method arguments into a format that can be sent over the network to the server.

- It then sends the request to the server, waits for the response, and unmarshals (converts back) the returned data into a format that the client can understand.

- From the client's perspective, the stub behaves like a local object, hiding the complexities of remote communication.

2. Skeleton (Server-Side Proxy):

- The skeleton acts as a proxy on the server side (Note: Skeletons were used in earlier versions of RMI, but starting with Java 2 (Java 1.2), skeletons are no longer needed, as they are replaced by dynamically generated code).

- The skeleton receives the request from the stub, unmarshals the arguments, and invokes the actual method on the remote object implementation.

- After the method execution, the skeleton marshals the result and sends it back to the stub on the client side.

RMI Application Example: Factorial Calculation

Let's create a simple RMI application where a client sends an integer to the server, and the server returns the factorial of that integer.

Step 1: Define the Remote Interface

The remote interface defines the methods that can be invoked remotely. It extends `java.rmi.Remote` and each method must throw a `RemoteException`.

```java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Factorial extends Remote {
// Method to calculate factorial
int getFactorial(int number) throws RemoteException;
}
```

Step 2: Implement the Remote Interface on the Server

The server-side implementation of the remote interface provides the actual logic for the remote methods.

```java
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class FactorialImpl extends UnicastRemoteObject implements Factorial {

// Constructor
protected FactorialImpl() throws RemoteException {
super();
}

// Implement the method to calculate factorial
@Override
public int getFactorial(int number) throws RemoteException {
int factorial = 1;
for (int i = 1; i <= number; i++) {
factorial = i;
}
return factorial;
}
}
```

Step 3: Create the Server Application

The server application will create an instance of the remote object and bind it to the RMI registry.


```java
```

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class FactorialServer {
public static void main(String[] args) {
try {
// Create an instance of the remote object
FactorialImpl factorial = new FactorialImpl();

// Bind the remote object to the RMI registry
Registry registry = LocateRegistry.createRegistry(1099);
registry.rebind("FactorialService", factorial);

System.out.println("Factorial Server is ready.");
} catch (Exception e) {
System.err.println("Server exception: " + e.toString());
e.printStackTrace();
}
}
}
```

Step 4: Create the Client Application

The client application looks up the remote object in the RMI registry and invokes the remote method.

```java
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

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

```
try {
// Get the registry
Registry registry = LocateRegistry.getRegistry("localhost");

// Lookup the remote object
Factorial stub = (Factorial) registry.lookup("FactorialService");

// Call the remote method
int number = 5;
int result = stub.getFactorial(number);
System.out.println("Factorial of " + number + " is: " + result);
} catch (Exception e) {
System.err.println("Client exception: " + e.toString());
e.printStackTrace();
}
}
}
```

Step 5: Compile and Run the Application

1. Compile the Code:
- Compile the Java files:

```sh
javac .java
```

2. Start the RMI Registry:
- Start the RMI registry on the default port 1099:

```sh
rmiregistry
```

3. Run the Server:

- Run the server application:

```sh
java FactorialServer
```

4. Run the Client:

- In a different terminal or command prompt, run the client application:

```sh
java FactorialClient
```

Summary

- Stub and Skeleton: The stub acts as the client-side proxy that handles the communication with the remote server, while the skeleton (in older RMI versions) acted as the server-side proxy.

- RMI Application: The application example demonstrates creating a simple RMI application to calculate the factorial of a number sent by the client to the server. The steps include defining the remote interface, implementing it, setting up the server, and writing the client to invoke the remote method.

30. You are hired by a reputed software company which is going to design an application for "Movie Rental System". Your responsibility is to design a schema named MRS and create a table named Movie(id, Tille, Genre, Language, Length). Write a program to design a GUI form to take input for this table and insert the data into table after clicking the OK button (10)

To design a simple Movie Rental System (MRS) application with a table named `Movie`, you will need to:

1. Create the database schema and table.
2. Design a GUI form using Swing to take input.
3. Insert the data into the database upon clicking the "OK" button.

Step 1: Create the Database Schema and Table

First, create the schema and the `Movie` table in your database. This can be done in any SQL-based database (e.g., MySQL, PostgreSQL, etc.).

```sql
CREATE SCHEMA MRS;

CREATE TABLE MRS.Movie (
id INT PRIMARY KEY AUTO_INCREMENT,
Title VARCHAR(100),
Genre VARCHAR(50),
Language VARCHAR(50),
Length INT
);
```

Step 2: Design the GUI Form Using Swing

Next, we'll design the Java Swing GUI form. The form will have fields to enter the `Title`, `Genre`, `Language`, and `Length` of the movie, and a button to insert this data into the `Movie` table.

Step 3: Java Program to Insert Data into the Database

Here's a Java program that creates the GUI and inserts the data into the database when the "OK" button is clicked:

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

```
public class MovieRentalSystem extends JFrame {

// Components of the GUI
private JTextField titleField, genreField, languageField, lengthField;
private JButton okButton;

// Database URL, Username, and Password
static final String DB_URL = "jdbc:mysql://localhost:3306/MRS"; // Change to your database URL
static final String USER = "root";  // Change to your database username
static final String PASS = "password";  // Change to your database password

public MovieRentalSystem() {
// Setting up the frame
setTitle("Movie Rental System");
setLayout(new GridLayout(5, 2, 10, 10));
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Adding GUI components
add(new JLabel("Title:"));
titleField = new JTextField(20);
add(titleField);

add(new JLabel("Genre:"));
genreField = new JTextField(20);
add(genreField);

add(new JLabel("Language:"));
languageField = new JTextField(20);
add(languageField);

add(new JLabel("Length (minutes):"));
```

```
lengthField = new JTextField(20);
add(lengthField);

okButton = new JButton("OK");
add(okButton);

// Button action listener
okButton.addActionListener(new ActionListener() {
@Override
public void actionPerformed(ActionEvent e) {
insertMovieData();
}
});

pack(); // Pack the frame to fit all components
setVisible(true); // Set the frame visible
}

// Method to insert movie data into the database
private void insertMovieData() {
String title = titleField.getText();
String genre = genreField.getText();
String language = languageField.getText();
int length = Integer.parseInt(lengthField.getText());

Connection conn = null;
PreparedStatement pstmt = null;

try {
// Register JDBC driver and open a connection
conn = DriverManager.getConnection(DB_URL, USER, PASS);
```

```
// Create SQL insert statement
String sql = "INSERT INTO Movie (Title, Genre, Language, Length) VALUES (?, ?, ?, ?)";
pstmt = conn.prepareStatement(sql);
pstmt.setString(1, title);
pstmt.setString(2, genre);
pstmt.setString(3, language);
pstmt.setInt(4, length);

// Execute the insert statement
pstmt.executeUpdate();
JOptionPane.showMessageDialog(this, "Movie data inserted successfully!");

} catch (SQLException se) {
se.printStackTrace();
JOptionPane.showMessageDialog(this, "Error inserting data.");
} finally {
try {
if (pstmt != null) pstmt.close();
if (conn != null) conn.close();
} catch (SQLException se) {
se.printStackTrace();
}
}
}

// Main method to run the application
public static void main(String[] args) {
SwingUtilities.invokeLater(new Runnable() {
public void run() {
new MovieRentalSystem();
}
});
```

}

}

```

**31. Describe the responsibility of Serializable interface.**

The Serializable interface in Java is a marker interface that indicates that a class can be serialized. Serialization is the process of converting an object's state into a byte stream, which can then be stored in a file, sent over a network, or saved to a database. The reverse process, deserialization, reconstructs the object from the byte stream.

Responsibilities of the `Serializable` Interface:

1. Enable Serialization of Objects:

- By implementing the `Serializable` interface, a class indicates that its instances can be serialized. This allows objects of that class to be converted into a byte stream, preserving their state.

2. Enable Deserialization:

- When a class implements `Serializable`, its objects can be deserialized, which means they can be reconstructed from the byte stream into a copy of the original object.

3. Compatibility Across Systems:

- Serialization allows objects to be transmitted across different JVMs (Java Virtual Machines) or saved and restored across different sessions. This is particularly useful for distributed systems, where objects need to be passed between different components of a system, potentially running on different machines.

4. Customization of Serialization:

- Although `Serializable` itself does not contain any methods (as it is a marker interface), it provides hooks for customization. A class can define the following methods to control the serialization process:

- `private void writeObject(ObjectOutputStream oos) throws IOException`: Customize the serialization process by writing to the `ObjectOutputStream`.

- `private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException`: Customize the deserialization process by reading from the `ObjectInputStream`.

5. Security and Data Integrity:
```

- By controlling which classes can be serialized, developers can manage security concerns related to object serialization, such as preventing sensitive information from being serialized inadvertently.

Example:

```java
import java.io.Serializable;

public class Employee implements Serializable {
private static final long serialVersionUID = 1L;  // Optional, but recommended for version control

private String name;
private int id;

public Employee(String name, int id) {
this.name = name;
this.id = id;
}

// Getters and setters
}
```

Summary:

The `Serializable` interface is crucial in Java for enabling the conversion of objects to a byte stream and vice versa, facilitating persistent storage, and transmission of objects across networks. Implementing this interface allows a class's instances to be easily saved, transferred, and reconstructed.

32. Compare AWT with Swing. Write a GUI program using components to find sum and difference of two numbers.Use two text fields for giving input and a label for output. The program should display sum if user presses mouse and difference if user release

mouse.(2+8)

Comparison Between AWT and Swing

AWT (Abstract Window Toolkit) and Swing are both used for building graphical user interfaces (GUIs) in Java, but they differ in several key aspects:

Aspect	AWT	Swing
Component Type	AWT components are heavyweight, meaning they rely on the native system's GUI controls.	Swing components are lightweight and written entirely in Java, providing more flexibility and control.
Look and Feel	AWT components have a look and feel that is dependent on the native operating system.	Swing allows for a pluggable look and feel that can be consistent across different platforms.
Performance	AWT is generally faster for simple interfaces but may be less flexible for complex UIs.	Swing is more versatile and capable of creating more complex, customizable interfaces, but might be slower.
Event Handling	AWT uses the older event model that is less powerful and flexible.	Swing uses a more advanced event-handling mechanism, including ActionListeners and other features.
Components	Limited number of components.	Richer set of components (JButton, JTextField, JLabel, JTable, etc.).
Extensibility	Less extensible due to reliance on native system components.	Highly extensible with a wide range of built-in and third-party components.

GUI Program Using Swing to Find Sum and Difference of Two Numbers

Below is a Java program that creates a Swing-based GUI. It uses two text fields for input and a label for displaying the result. The program computes the sum when the user presses the mouse and computes the difference when the user releases the mouse.

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

public class SumDifferenceCalculator extends JFrame {
```

```
// Components of the GUI
private JTextField textField1, textField2;
private JLabel resultLabel;

public SumDifferenceCalculator() {
// Setting up the frame
setTitle("Sum and Difference Calculator");
setLayout(new FlowLayout());
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Initialize components
textField1 = new JTextField(10);
textField2 = new JTextField(10);
resultLabel = new JLabel("Result: ");

// Add components to the frame
add(new JLabel("Number 1:"));
add(textField1);
add(new JLabel("Number 2:"));
add(textField2);
add(resultLabel);

// Mouse listener for handling mouse press and release events
MouseAdapter mouseAdapter = new MouseAdapter() {
@Override
public void mousePressed(MouseEvent e) {
calculateSum();
}

@Override
public void mouseReleased(MouseEvent e) {
calculateDifference();
```

```
}
};

// Attach mouse listener to the frame
addMouseListener(mouseAdapter);

pack(); // Pack the frame to fit all components
setVisible(true); // Set the frame visible
}

// Method to calculate the sum of the two numbers
private void calculateSum() {
try {
int num1 = Integer.parseInt(textField1.getText());
int num2 = Integer.parseInt(textField2.getText());
int sum = num1 + num2;
resultLabel.setText("Result: Sum = " + sum);
} catch (NumberFormatException ex) {
resultLabel.setText("Invalid input!");
}
}

// Method to calculate the difference between the two numbers
private void calculateDifference() {
try {
int num1 = Integer.parseInt(textField1.getText());
int num2 = Integer.parseInt(textField2.getText());
int difference = num1 - num2;
resultLabel.setText("Result: Difference = " + difference);
} catch (NumberFormatException ex) {
resultLabel.setText("Invalid input!");
}
```

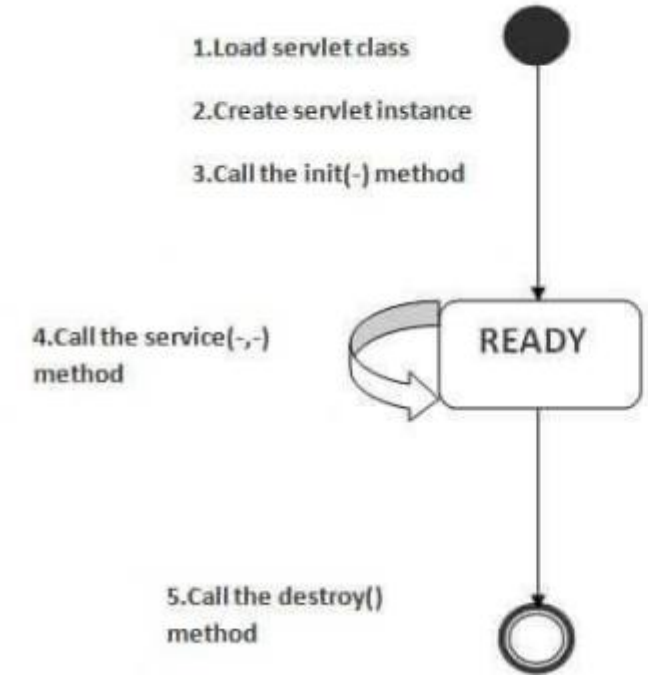
```
}

// Main method to run the application
public static void main(String[] args) {
SwingUtilities.invokeLater(new Runnable() {
public void run() {
new SumDifferenceCalculator();
}
});
}
}
```

33. Explain life-cycle of servlet in detail.Create a simple servlet that reads and displaysdata from HTML form. Assume form with two fields username and password.(5+5)

Life Cycle of a Servlet

A servlet life cycle can be defined as the entire process from its creation till the destruction.

1. Servlet class is loaded.
2. Servlet instance is created.
3. init method is invoked.
4. service method is invoked.
5. destroy method is invoked.



As displayed in the above diagram, there are three states of a servlet: new, ready and end. The servlet is in new state if servlet instance is created. After invoking the init() method, Servlet comes in the ready state. In the ready state, servlet performs all the tasks. When the web container invokes the destroy() method, it shifts to the end state.

1) Servlet class is loaded

The classloader is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the web container.

2) Servlet instance is created

The web container creates the instance of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.

3) init method is invoked

The web container calls the init method only once after creating the servlet instance. The init method is used to initialize the servlet. It is the life cycle method of the javax.servlet.Servlet interface. Syntax of the init method is given below:

public void init(ServletConfig config) throws ServletException

4) service method is invoked

The web container calls the service method each time when request for the servlet is received. If servlet is not initialized, it follows the first three steps as described above then calls the service method. If servlet is initialized, it calls the service method. Notice that

servlet is initialized only once. The syntax of the service method of the Servlet interface is

given below:

public void service(ServletRequest request, ServletResponse response)

throws ServletException, IOException

5) destroy method is invoked

The web container calls the destroy method before removing the servlet instance from the

service. It gives the servlet an opportunity to clean up any resource for example memory,

thread etc. The syntax of the destroy method of the Servlet interface is given below:

public void destroy()

Simple Servlet Example: Reading Data from an HTML Form

Here's how you can create a simple servlet that reads and displays data from an HTML form with two fields: username and password.

HTML Form (index.html):

html

Copy code

<!DOCTYPE html>

<html>

<head>

<title>Login Form</title>

</head>

<body>

<form action="LoginServlet" method="POST">

<label for="username">Username:</label>

<input type="text" id="username" name="username">

<label for="password">Password:</label>

<input type="password" id="password" name="password">

<input type="submit" value="Submit">

</form>

</body>

</html>

Servlet Code (LoginServlet.java):

java

Copy code

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

public class LoginServlet extends HttpServlet {
public void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

// Set response content type
response.setContentType("text/html");
PrintWriter out = response.getWriter();

// Read form data
String username = request.getParameter("username");
String password = request.getParameter("password");

// Display the data
out.println("<html><body>");
out.println("<h2>Login Details</h2>");
out.println("<p>Username: " + username + "</p>");
out.println("<p>Password: " + password + "</p>");
out.println("</body></html>");
}
}
```

34. Explain RMI architecture layers in detail. Write a Java programs using RMI to find product of two numbers.(4+6)

RMI stands for Remote Method Invocation. It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM.

RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package java.rmi.

Following are the goals of RMI −

• To minimize the complexity of the application.

• To preserve type safety.

• Distributed garbage collection.

• Minimize the difference between working with local and remote objects.

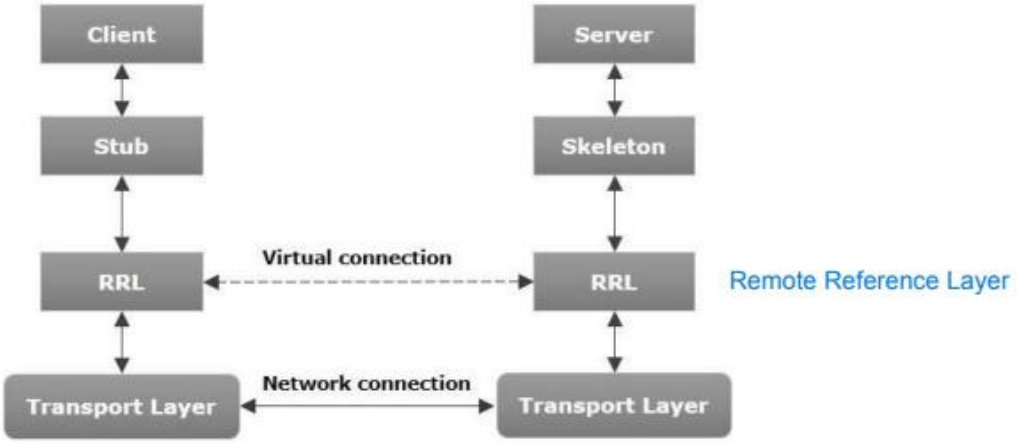
Architecture of an RMI Application

In an RMI application, we write two programs, a server program (resides on the server) and a client program (resides on the client).

• Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry).

• The client program requests the remote objects on the server and tries to invoke its methods.

The following diagram shows the architecture of an RMI application.



Transport Layer − This layer connects the client and the server.

It manages the existing connection and also sets up new

connections.

• Stub − A stub is a representation (proxy) of the remote object

at client. It resides in the client system; it acts as a gateway for

the client program.

• Skeleton − This is the object which resides on the server side.

stub communicates with this skeleton to pass request to the

remote object.

• RRL(Remote Reference Layer) − It is the layer which manages

the references made by the client to the remote object.

Java Program Using RMI to Find the Product of Two Numbers

Below is a step-by-step guide to create a simple RMI application that calculates the product of two numbers.

Step 1: Define the Remote Interface

Create an interface that declares the method to calculate the product. This interface must extend java.rmi.Remote.

java

Copy code

```
import java.rmi.Remote;

import java.rmi.RemoteException;


// Remote Interface

public interface Product extends Remote {

// Method to calculate product of two numbers

public int multiply(int a, int b) throws RemoteException;

}
```

Step 2: Implement the Remote Interface

Create a class that implements the remote interface. This class provides the actual implementation of the remote method.

java

Copy code

```
import java.rmi.RemoteException;

import java.rmi.server.UnicastRemoteObject;
```

```
// Implementation of the remote interface
public class ProductImpl extends UnicastRemoteObject implements Product {
// Constructor
protected ProductImpl() throws RemoteException {
super();
}

// Implementation of the multiply method
public int multiply(int a, int b) throws RemoteException {
return a  b;
}
}
```

Step 3: Create the Server

Write the server code to register the remote object with the RMI registry.

java

Copy code

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class ProductServer {
public static void main(String[] args) {
try {
// Create an instance of the ProductImpl class
ProductImpl obj = new ProductImpl();

// Bind the remote object's stub in the registry
Registry registry = LocateRegistry.createRegistry(1099); // Default port 1099
registry.bind("Product", obj);

System.out.println("Product Server is ready.");
} catch (Exception e) {
```

```
System.out.println("Product Server failed: " + e);
}
}
}
```

Step 4: Create the Client

Write the client code that looks up the remote object in the RMI registry and invokes the remote method.

java

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class ProductClient {
public static void main(String[] args) {
try {
// Locate the registry at localhost and default port 1099
Registry registry = LocateRegistry.getRegistry("localhost", 1099);

// Look up the remote object "Product" in the registry
Product stub = (Product) registry.lookup("Product");

// Call the multiply method
int result = stub.multiply(5, 7);
System.out.println("Product of 5 and 7 is: " + result);
} catch (Exception e) {
System.out.println("Product Client exception: " + e);
}
}
}
```


35. What is package? How can you create your own package in Java? Explain with

example.(1+4)

What is a Package in Java?

A package in Java is a namespace that organizes a set of related classes and interfaces. Packages help avoid name conflicts and make it easier to locate and use the classes, interfaces, and sub-packages within your program. They also provide access control and can contain hidden classes that are used only within the package and are not accessible to classes outside the package.

Creating Your Own Package in Java

To create your own package in Java, follow these steps:

1. Define a Package:

- Use the `package` keyword at the beginning of your Java source file.

- All the classes and interfaces defined in this file will belong to the specified package.

2. Compile the Package:

- Compile the Java file using the `javac` command.

- The compiled `.class` files will be placed in a directory structure matching the package name.

3. Use the Package:

- To use the classes and interfaces from your package, you can import them into other classes using the `import` statement.

Example: Creating and Using a Custom Package

Step 1: Create a Package

Let's say you want to create a package named `mypackage` that contains a class `Rectangle`.

Rectangle.java:

```java
// Step 1: Define the package
```

```
package mypackage;

public class Rectangle {
private int length;
private int width;

// Constructor
public Rectangle(int length, int width) {
this.length = length;
this.width = width;
}

// Method to calculate area
public int getArea() {
return length  width;
}
}
```

Step 2: Compile the Package

- Compile the `Rectangle.java` file. This will create the `mypackage` directory with the `Rectangle.class` file inside it.

```bash
javac Rectangle.java
```

Step 3: Use the Package in Another Program

Now, create a new Java file in the same directory or a different directory that uses the `Rectangle` class from the `mypackage`.

TestRectangle.java:

```java
// Step 1: Import the package
import mypackage.Rectangle;

public class TestRectangle {
public static void main(String[] args) {
// Step 2: Create an instance of Rectangle
Rectangle rect = new Rectangle(10, 5);

// Step 3: Use the method from Rectangle class
int area = rect.getArea();
System.out.println("The area of the rectangle is: " + area);
}
}
```

Step 4: Compile and Run the Program

- Compile `TestRectangle.java`:

```bash
javac TestRectangle.java
```

- Run the program:

```bash
java TestRectangle
```

- Output:

```bash
The area of the rectangle is: 50
```

Summary:

- Package: A namespace that organizes related classes and interfaces.

- Creating a Package: Use the `package` keyword, then compile and organize your classes within that package.

- Using a Package: Import the package using the `import` statement and then use its classes or interfaces in your application.

36. Why do we need swing components? Explain the uses of check boxes and radio buttons in GUI programming. (2+3)

Why Do We Need Swing Components?

Swing components are essential for building modern, flexible, and rich graphical user interfaces (GUIs) in Java. They offer more advanced features, better look and feel, and greater flexibility compared to AWT, allowing developers to create more sophisticated and visually appealing applications.

Uses of Check Boxes and Radio Buttons in GUI Programming

- Check Boxes:

- Allow users to select multiple options independently.

- Useful for scenarios where more than one option can be chosen, such as selecting multiple hobbies or preferences.

- Radio Buttons:

- Allow users to select only one option from a group.

- Ideal for scenarios where a single choice is required, such as choosing a gender or selecting a payment method.

37. How can we use listener interface to handle events? Compare listener interface with

adapter class. (3+2)

Using Listener Interface to Handle Events

In Java GUI programming, listener interfaces are used to handle events such as button clicks or mouse movements. You implement these interfaces to define how your application should respond to various user actions. Here's a brief overview:

1. Implement the Listener Interface:

- Create a class that implements one or more listener interfaces (e.g., `ActionListener`, `MouseListener`).

2. Override the Event-Handling Methods:

- Provide concrete implementations for the methods defined in the listener interfaces (e.g., `actionPerformed` for `ActionListener`).

3. Register the Listener:

- Add an instance of your class to the component you want to listen to using methods like `addActionListener`, `addMouseListener`, etc.

Example:

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

public class ButtonExample implements ActionListener {
private JButton button;
```

```
public ButtonExample() {
JFrame frame = new JFrame("Button Example");
button = new JButton("Click Me");
button.addActionListener(this); // Register listener
frame.add(button);
frame.setSize(200, 200);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
}

@Override
public void actionPerformed(ActionEvent e) {
button.setText("Clicked!");
}

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

Comparing Listener Interface and Adapter Class

Listener Interface:

- Definition: A listener interface defines one or more methods for handling specific events.

- Usage: You must implement all methods of the interface, which can lead to verbose code if you only need to handle a few methods.

- Flexibility: More flexible when handling multiple types of events, as you can implement several interfaces in a single class.

Adapter Class:

- Definition: An adapter class provides default (empty) implementations of all methods in the corresponding listener interface.

- Usage: You can extend the adapter class and override only the methods you need, reducing boilerplate code.

- Flexibility: More concise and easier to use if you need to handle only a few events from the interface.

Example:

Using Listener Interface:

```java
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MyActionListener implements ActionListener {
@Override
public void actionPerformed(ActionEvent e) {
// Handle action
}
}
```

Using Adapter Class:

```java
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

public class MyMouseAdapter extends MouseAdapter {
@Override
public void mouseClicked(MouseEvent e) {
// Handle mouse click
}
}
```

```

Summary:

- Listener Interface: Directly implements methods to handle events but may require implementing multiple methods.

- Adapter Class: Provides default implementations to simplify code by overriding only the methods you need.

**38. What is row set? Explain cached row set in detail.(1+4)**

What is a RowSet?

A RowSet is a type of JDBC object that extends the capabilities of a `ResultSet` by providing additional features such as connectivity, scrollability, and updatability. Unlike a `ResultSet`, a `RowSet` can be disconnected from the database, which allows it to work offline.

Cached RowSet

CachedRowSet is a type of `RowSet` that provides a disconnected operation, meaning it can be used without an active connection to the database. It stores its data in memory and can be used to work with the data offline.

Features of CachedRowSet:

1. Disconnected Operation: It can be created, manipulated, and then reconnected to the database when needed, allowing offline data handling.

2. Serialization: CachedRowSet can be serialized, making it possible to save the data to disk and reload it later.

3. Concurrency Control: It allows modifications to be made to the data while disconnected, and these changes can be synchronized back to the database.

4. Scrolling and Updating: Provides support for scrolling through rows and updating data, even when disconnected.
```

Example:

```java
import javax.sql.rowset.CachedRowSet;
import javax.sql.rowset.RowSetProvider;
import java.sql.;

public class CachedRowSetExample {
public static void main(String[] args) {
try {
// Create a CachedRowSet object
CachedRowSet crs = RowSetProvider.newFactory().createCachedRowSet();

// Connect to the database
Connection conn = DriverManager.getConnection("jdbc:yourdriver:yourdb", "username", "password");

// Set the SQL query
crs.setCommand("SELECT  FROM your_table");
crs.execute(conn);

// Work with data
while (crs.next()) {
System.out.println(crs.getString("column_name"));
}

// Modify data
crs.moveToInsertRow();
crs.updateString("column_name", "new_value");
crs.insertRow();

// Update the database
crs.moveToCurrentRow();
```

```
crs.acceptChanges(conn);

// Clean up
conn.close();
} catch (SQLException e) {
e.printStackTrace();
}
}
}
```

Summary:

- RowSet: Extends `ResultSet` with additional features.
- CachedRowSet: Allows disconnected operations, supports serialization, and enables offline data manipulation.

39. What is servlet? Write a simple JSP file to display "Tribhuwan University" five times.

(2+3)

What is a Servlet?

A Servlet is a Java programming class that handles HTTP requests and responses in a web application. It operates on the server side and is used to extend the capabilities of servers. Servlets can process requests, generate dynamic content, and manage session data. They are a core component of Java EE (Enterprise Edition) for web applications.

Simple JSP File to Display "Tribhuvan University" Five Times

Here is a simple JSP (JavaServer Pages) file that displays "Tribhuvan University" five times:

`display.jsp`:

```jsp
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Display Message</title>
</head>
<body>
<%
// Loop to display the message five times
for (int i = 0; i < 5; i++) {
out.println("Tribhuvan University<br>");
}
%>
</body>
</html>
```

Explanation:

- `<%@ page ... %>`: Directives that define page-level settings like content type and character encoding.
- `<% ... %>`: Scriptlet that allows embedding Java code within the JSP page.
- `out.println(...)`: Outputs text to the client.

Usage:

1. Save this code in a file named `display.jsp`.
2. Deploy the JSP file in a servlet container (like Apache Tomcat).
3. Access the JSP page via a web browser to see the output.

40. Why CORBA is important? Compare CORBA with RMI. (3+2)

Importance of CORBA

CORBA (Common Object Request Broker Architecture) is important because it provides a framework for enabling communication between objects in different programming languages and across different platforms. This interoperability makes it easier for heterogeneous systems to work together in distributed computing environments. Key benefits include:

1. Language and Platform Independence: CORBA allows objects written in different languages to communicate, promoting integration across diverse systems.

2. Scalability: It supports distributed systems and helps in scaling applications across multiple servers.

3. Standardization: CORBA provides a standardized approach to object communication, which helps in building complex systems with consistent interfaces.

Comparison of CORBA and RMI

| Feature | CORBA | RMI |

|---|---|---|

| Language Support | Supports multiple programming languages (e.g., Java, C++, Python). | Primarily designed for Java-to-Java communication. |

| Platform Independence | Designed to work across different platforms and operating systems. | Generally used in Java environments, with limited cross-platform support. |

| Complexity | More complex due to its broad range of features and specifications. | Simpler and easier to use within Java environments. |

| Standard | Based on OMG (Object Management Group) standards. | Based on Java's own specifications and standards. |

| Interoperability | Provides interoperability between different languages and platforms. | Limited to Java-to-Java communication. |

Summary:

- CORBA: Provides broad interoperability across different languages and platforms with a more complex setup.

- RMI: Simplifies distributed communication within Java environments but lacks cross-language and cross-platform support.

41. Write short notes on: (22.5 = 5)

a. JDBC drivers

b. Java server pages

a. JDBC Drivers

JDBC Drivers are components that enable Java applications to interact with databases via the Java Database Connectivity (JDBC) API. They translate Java calls into database-specific calls. There are four types of JDBC drivers:

1. Type 1: JDBC-ODBC Bridge Driver:

- Uses ODBC (Open Database Connectivity) drivers to connect to databases.

- Disadvantage: Requires ODBC driver and is generally slower and less efficient.

2. Type 2: Native-API Driver:

- Converts JDBC calls into database-specific calls using native API libraries.

- Advantage: Faster than Type 1 but requires native database libraries.

3. Type 3: Network Protocol Driver:

- Translates JDBC calls into a database-independent network protocol, which is then translated into database-specific calls by a server-side component.

- Advantage: Can work with multiple databases and is more flexible.

4. Type 4: Thin Driver:

- Directly converts JDBC calls into database-specific protocol calls, without needing additional software.

- Advantage: Efficient and platform-independent.

b. JavaServer Pages (JSP)

JavaServer Pages (JSP) is a technology used for creating dynamic web content. JSP allows embedding Java code directly into HTML pages, which is compiled into servlets by the server. Here are some key aspects:

1. Simplified Syntax:

- JSP syntax is similar to HTML, with embedded Java code within special tags (`<% %>`). This makes it easier to create dynamic web content compared to writing pure servlets.

2. Separation of Concerns:

- JSP separates the presentation layer from business logic. It allows developers to design the HTML structure while keeping Java code for dynamic content generation separate.

3. Standard Tags and Libraries:

- JSP supports standard tags for common tasks (JSP Standard Tag Library - JSTL) and custom tags for extending functionality.

4. Compilation:

- JSP pages are compiled into servlets by the web server (e.g., Apache Tomcat) before being executed, providing the benefits of servlet-based processing with an easier-to-write syntax.

In summary, JDBC Drivers are crucial for database connectivity in Java applications, and JSP provides a streamlined way to generate dynamic web content by embedding Java code in HTML.