ਜਗਤ ਗੁਰੂ ਨਾਨਕ ਦੇਵ
ਪੰਜਾਬ ਸਟੇਟ ਓਪਨ ਯੂਨੀਵਰਸਿਟੀ
ਪਟਿਆਲਾ

JAGAT GURU NANAK DEV

PUNJAB STATE OPEN UNIVERSITY, PATIALA

(Established by Act No. 19 of 2019 of the Legislature of State of Punjab)

The Motto of the University

(SEWA)

SKILL ENHANCEMENT EMPLOYABILITY WISDOM ACCESSIBILITY



M.Sc. (Computer Science)

Course Name: Web Programming Lab

Course Code: MSCS-3-01P

ADDRESS: C/28, THE LOWER MALL, PATIALA-147001
WEBSITE: www.psou.ac.in

SELF-INSTRUCTIONAL STUDY MATERIAL FOR JGND PSOU, ALL COPYRIGHTS WITH JGND PSOU, PATIALA

JAGAT GURU NANAK DEV
PUNJAB STATE OPEN UNIVERSITY PATIALA
(Established by Act No.19 of 2019 of Legislature of the State of Punjab)

Faculty of School of Science and Emerging Technologies:

Dr. Baljit Singh Khera (Head)
Professor, School of Sciences and Emerging Technologies
Jagat Guru Nanak Dev Punjab State Open University, Patiala

Dr. Kanwalvir Singh Dhindsa
Professor, School of Sciences and Emerging Technologies
Jagat Guru Nanak Dev Punjab State Open University, Patiala

Dr. Amitoj Singh
Associate Professor, School of Sciences and Emerging Technologies
Jagat Guru Nanak Dev Punjab State Open University, Patiala

Dr. Karan Sukhija
Assistant Professor, School of Sciences and Emerging Technologies
Jagat Guru Nanak Dev Punjab State Open University, Patiala

Dr. Monika Pathak
Assistant Professor, School of Sciences and Emerging Technologies
Jagat Guru Nanak Dev Punjab State Open University, Patiala

Dr. Gaurav Dhiman
Assistant Professor, School of Sciences and Emerging Technologies
Jagat Guru Nanak Dev Punjab State Open University, Patiala

JAGAT GURU NANAK DEV
PUNJAB STATE OPEN UNIVERSITY PATIALA
(Established by Act No.19 of 2019 of Legislature of the State of Punjab)

PROGRAMME COORDINATOR
Dr. Karan Sukhija (Assistant Professor)
School of Sciences and Emerging Technologies
JGND PSOU, Patiala

COURSE COORDINATOR AND EDITOR:
Dr. Gaurav Dhiman (Assistant Professor)
School of Sciences and Emerging Technologies
JGND PSOU, Patiala

JAGAT GURU NANAK DEV
PUNJAB STATE OPEN UNIVERSITY PATIALA
(Established by Act No.19 of 2019 of Legislature of the State of Punjab)

PREFACE

Jagat Guru Nanak Dev Punjab State Open University, Patiala was established in Decembas 2019 by Act 19 of the Legislature of State of Punjab. It is the first and only Open Universit of the State, entrusted with the responsibility of making higher education accessible to all especially to those sections of society who do not have the means, time or opportunity to pursue regular education.

In keeping with the nature of an Open University, this University provides a flexible education system to suit every need. The time given to complete a programme is double the duration of a regular mode programme. Well-designed study material has been prepared in consultation with experts in their respective fields.

The University offers programmes which have been designed to provide relevant, skill-based and employability-enhancing education. The study material provided in this booklet is self instructional, with self-assessment exercises, and recommendations for further readings. The syllabus has been divided in sections, and provided as units for simplification.

The Learner Support Centres/Study Centres are located in the Government and Government aided colleges of Punjab, to enable students to make use of reading facilities, and for curriculum-based counselling and practicals. We, at the University, welcome you to be a part of this institution of knowledge.

Prof. G. S. Batra,
Dean Academic Affairs

MSCS-3-01P: Web Programming Lab

Total Marks: 50
External Marks: 35
Internal Marks: 15
Credits: 2
Pass Percentage: 40%

Course: Web Programming Lab	
Course Code: MSCS-3-01P	
Course Outcomes (COs) After the completion of this course, the students will be able to:	
CO1	Understand and articulate the characteristics of the Java programming language.
CO2	Classify and explain different types of inheritance in Java.
CO3	Understand the concept of streams and effectively use predefined streams for reading console input and writing console output.
CO4	Achieve multiple inheritance using interfaces and comprehend the relationship between interfaces and abstract classes.
CO5	Demonstrate proficiency in multi-threaded programming, including creating and managing threads, implementing.

Detailed List of Programs:

Program No.	**Name of Program**
P1	Install Java Development Kit (JDK) on your system and set up the Java runtime environment. Verify the installation using command-line tools
Solution:	Installing the Java Development Kit (JDK) and setting up the Java Runtime Environment (JRE) involves several steps. Here's an explanation of what it means and how to do it: **What is JDK and JRE?** **JDK (Java Development Kit):** 1. A software development environment for building Java applications. 2. Includes tools such as the Java compiler (javac), debugger, and libraries. 3. Necessary for writing and compiling Java code. **JRE (Java Runtime Environment):** 1. A software layer that runs Java programs. 2. Includes the Java Virtual Machine (JVM), core libraries,

and other files required to execute Java applications.

The JDK contains the JRE, so installing the JDK is sufficient to both develop and run Java applications.

Steps to Install JDK and Set Up the Environment

Download JDK:

1. Visit the Oracle Java Downloads page or use an open-source version like OpenJDK.
2. Choose the appropriate version for your operating system (Windows, macOS, or Linux).

Install JDK:

1. Run the downloaded installer and follow the on-screen instructions.
2. By default, the JDK is installed in directories such as:
    1. Windows: C:\Program Files\Java\jdk-XX
    2. macOS: /Library/Java/JavaVirtualMachines/jdk-XX
    3. Linux: /usr/lib/jvm/jdk-XX

Set Up Environment Variables:

1. Add the JDK installation directory to your system's PATH environment variable.
2. Define the JAVA_HOME variable to point to the JDK installation directory.

On Windows:

1. Open "System Properties" > "Environment Variables."
2. Add a new variable named JAVA_HOME with the path to the JDK directory (e.g., C:\Program Files\Java\jdk-XX).
3. Edit the Path variable and add %JAVA_HOME%\bin.

On macOS/Linux:

1. Edit the shell configuration file (~/.bashrc, ~/.zshrc, or ~/.bash_profile):

```
export JAVA_HOME=/path/to/jdk
export PATH=$JAVA_HOME/bin:$PATH
```

	2. Save the file and run source ~/.bashrc or source ~/.zshrc. **Verify the Installation:** 1. Open a terminal or command prompt and run the following commands: java -version 2. Expected output: Displays the installed Java version. javac -version 3. Expected output: Displays the Java compiler version. If both commands return version information, your installation and setup are successful.
P2	Write a simple Java program and compile it using the Java compiler. Run the compiled program and observe the output.

Solution:	public class HelloWorld {
	public static void main(String[] args) {
	System.out.println("Hello, World!");
	}
	}

P3	Compare and contrast Java and C++ programming languages, highlighting key differences.
Solution:	**1. Platform Independence** • **Java:** o Platform-independent due to the **Java Virtual Machine (JVM)**. o Java code is compiled into **bytecode**, which can run on any device with a JVM. • **C++:** o Platform-dependent. o C++ programs are compiled into machine code, which is

specific to the target operating system and processor.

2. Programming Paradigm

- **Java:**
    - Purely **object-oriented** (with minor exceptions like primitive types).
    - Every piece of code must be part of a class.
- **C++:**
    - Supports both **object-oriented** and **procedural programming**.
    - You can write code without using classes.

3. Memory Management

- **Java:**
    - Automatic memory management using **garbage collection**.
    - Developers don't need to explicitly free allocated memory.
- **C++:**
    - Manual memory management.
    - Developers must explicitly allocate and deallocate memory using new and delete.

4. Syntax Differences

- **Java:**
    - Simpler and more standardized syntax.
        - Example:

      System.out.println("Hello, World!");

    -
- **C++:**

- More complex syntax due to multiple programming styles and features.
    - Example:

```
std::cout << "Hello, World!" << std::endl;
```

    -

5. Multiple Inheritance

- **Java:**
    - Does not support multiple inheritance directly (to avoid ambiguity issues like the **Diamond Problem**).
    - Achieves multiple inheritance using **interfaces**.
- **C++:**
    - Supports multiple inheritance directly but may lead to ambiguity in case of conflicting methods.

6. Pointers

- **Java:**
    - Does not support explicit pointer manipulation for security and simplicity.
    - References are managed by the JVM.
- **C++:**
    - Provides full support for pointers.
    - Developers can directly manipulate memory addresses.

7. Speed and Performance

- **Java:**
    - Generally slower due to the JVM and garbage collection overhead.
    - Just-In-Time (JIT) compilation improves performance.

- **C++:**
    - Faster because it compiles directly to machine code.
    - Provides fine-grained control over system resources.

8. Standard Libraries

- **Java:**
    - Rich set of built-in libraries (e.g., java.util, java.io) for tasks like data structures, networking, and multithreading.
- **C++:**
    - Standard Template Library (STL) provides data structures (e.g., vectors, maps) and algorithms.
    - Fewer built-in libraries compared to Java.

9. Exception Handling

- **Java:**
    - Robust exception handling with try-catch-finally.
    - Checked exceptions force the developer to handle potential errors at compile time.
- **C++:**
    - Supports exception handling, but it is less enforced and optional.
    - No concept of checked exceptions.

10. Application Areas

- **Java:**
    - Commonly used for web applications, enterprise software, mobile development (Android), and cross-platform apps.
- **C++:**
    - Used for system-level programming, game development,

real-time applications, and performance-critical software.

Key Features Java Lacks (Present in C++)

- Operator overloading.
- Direct hardware access (via pointers).
- Templates for generic programming.
- Inline assembly code.

Key Features C++ Lacks (Present in Java)

- Built-in garbage collection.
- Platform independence through JVM.
- Simpler multithreading support.
- Built-in support for networking and database connectivity.

Summary Table

Feature	Java	C++
Platform Independence	Yes (via JVM)	No
Memory Management	Automatic (Garbage Collection)	Manual (new/delete)
Multiple Inheritance	Not supported (use interfaces)	Fully supported
Speed	Slower due to JVM	Faster (compiled to machine code)
Pointers	Not exposed to developers	Fully supported
Programming Paradigm	Object-Oriented only	Procedural + Object-Oriented
Standard Libraries	Extensive built-in libraries	STL for data structures and algorithms
Use Cases	Web, mobile, enterprise software	System-level programming, games

P4	Create a Java program that demonstrates the concept of byte code and

	explain how it is executed by the Java Virtual Machine (JVM).

Solution:	public class BytecodeDemo {
	public static void main(String[] args) {
	System.out.println("This program demonstrates Java bytecode execution.");
	int a = 10;
	int b = 20;
	int sum = a + b;
	System.out.println("The sum of " + a + " and " + b + " is: " + sum);
	}
	}

P5	Implement a Java program that explores the usage of constants, variables, data types, and operators.

Solution	public class JavaBasicsDemo {
	public static void main(String[] args) {
	// Constants (declared as final)
	final double PI = 3.14159; // A constant value for π
	System.out.println("Value of PI: " + PI);
	
	// Variables
	int age = 25; // Integer variable
	double salary = 55000.50; // Double variable
	char grade = 'A'; // Character variable
	boolean isEmployed = true; // Boolean variable
	String name = "John Doe"; // String variable
	
	// Printing variables
	System.out.println("Name: " + name);
	System.out.println("Age: " + age);
	System.out.println("Salary: " + salary);
	System.out.println("Grade: " + grade);

```
        System.out.println("Employed: " + isEmployed);

        // Data types and type casting
        float height = 5.9f;        // Float variable
        long population = 7_900_000_000L; // Long variable with underscores for readability
        short year = 2025;          // Short variable
        byte month = 12;            // Byte variable

        System.out.println("Height: " + height + " feet");
        System.out.println("World Population: " + population);
        System.out.println("Year: " + year);
        System.out.println("Month: " + month);

        // Operators
        int num1 = 10, num2 = 20;

        // Arithmetic operators
        System.out.println("Addition: " + (num1 + num2));
        System.out.println("Subtraction: " + (num2 - num1));
        System.out.println("Multiplication: " + (num1 * num2));
        System.out.println("Division: " + (num2 / num1));
        System.out.println("Modulus: " + (num2 % num1));

        // Relational operators
        System.out.println("Is num1 equal to num2? " + (num1 == num2));
        System.out.println("Is num1 not equal to num2? " + (num1 != num2));
        System.out.println("Is num1 greater than num2? " + (num1 > num2));
        System.out.println("Is num1 less than or equal to num2? " + (num1 <= num2));
```

```
        // Logical operators
        boolean condition1 = (num1 > 5);
        boolean condition2 = (num2 < 50);
        System.out.println("Condition1 AND Condition2: " + (condition1
&& condition2));
        System.out.println("Condition1 OR Condition2: " + (condition1 ||
condition2));
        System.out.println("NOT Condition1: " + (!condition1));

        // Increment and Decrement operators
        System.out.println("Original num1: " + num1);
        System.out.println("Post-increment of num1: " + (num1++));
        System.out.println("After post-increment: " + num1);
        System.out.println("Pre-decrement of num1: " + (--num1));

        // Assignment operators
        int x = 15;
        x += 10;  // Equivalent to x = x + 10
        System.out.println("Value of x after += 10: " + x);
    }
}
```

Output:-

```
Name: John Doe
Age: 25
Salary: 55000.5
Grade: A
Employed: true
Height: 5.9 feet
World Population: 7900000000
Year: 2025
Month: 12
Addition: 30
Subtraction: 10
```

	Multiplication: 200 Division: 2 Modulus: 0 Is num1 equal to num2? false Is num1 not equal to num2? true Is num1 greater than num2? false Is num1 less than or equal to num2? true Condition1 AND Condition2: true Condition1 OR Condition2: true NOT Condition1: false Original num1: 10 Post-increment of num1: 10 After post-increment: 11 Pre-decrement of num1: 10 Value of x after += 10: 25
P6	Create a Java program to illustrate different types of inheritance, including single, multiple, and multilevel inheritance.

Solution	// Single Inheritance: Parent class and Child class
	class Animal {
	  void eat() {
	  System.out.println("This animal can eat.");
	  }
	}
	
	// Single Inheritance: Dog inherits Animal
	class Dog extends Animal {
	  void bark() {
	  System.out.println("The dog barks.");
	  }
	}

```
// Multilevel Inheritance: Puppy inherits Dog
class Puppy extends Dog {
    void weep() {
        System.out.println("The puppy weeps.");
    }
}

// Multiple Inheritance using Interfaces
interface Flyable {
    void fly();
}

interface Swimable {
    void swim();
}

// A class implementing multiple interfaces
class Duck implements Flyable, Swimable {
    @Override
    public void fly() {
        System.out.println("The duck can fly.");
    }

    @Override
    public void swim() {
        System.out.println("The duck can swim.");
    }
}

// Main class to demonstrate inheritance types
public class InheritanceDemo {
    public static void main(String[] args) {
        // Single Inheritance demonstration
```

```
        Dog dog = new Dog();
        System.out.println("Single Inheritance:");
        dog.eat(); // Inherited from Animal
        dog.bark();

        // Multilevel Inheritance demonstration
        Puppy puppy = new Puppy();
        System.out.println("\nMultilevel Inheritance:");
        puppy.eat();  // Inherited from Animal
        puppy.bark(); // Inherited from Dog
        puppy.weep();

        // Multiple Inheritance using interfaces demonstration
        Duck duck = new Duck();
        System.out.println("\nMultiple Inheritance using Interfaces:");
        duck.fly();
        duck.swim();
    }
}
```

Output:-

Single Inheritance:

This animal can eat.

The dog barks.

Multilevel Inheritance:

This animal can eat.

The dog barks.

The puppy weeps.

Multiple Inheritance using Interfaces:

The duck can fly.

The duck can swim.

P7	Implement a program that showcases the use of the 'super' keyword to call superclass constructors.

Solution	// Superclass
	class Animal {
	  String name;
	
	  // Constructor of Animal class
	  Animal(String name) {
	  this.name = name;
	  System.out.println("Animal constructor called: " + name);
	  }
	
	  // Method in the Animal class
	  void displayInfo() {
	  System.out.println("Animal Name: " + name);
	  }
	}
	
	// Subclass
	class Dog extends Animal {
	  String breed;
	
	  // Constructor of Dog class
	  Dog(String name, String breed) {
	  super(name); // Calls the superclass (Animal) constructor
	  this.breed = breed;
	  System.out.println("Dog constructor called: Breed - " + breed);
	  }
	
	  // Method in the Dog class
	  void displayInfo() {
	  super.displayInfo(); // Calls the superclass (Animal) method

	System.out.println("Breed: " + breed);
	}
	}
	
	// Main Class
	public class SuperKeywordDemo {
	public static void main(String[] args) {
	// Creating a Dog object
	Dog dog = new Dog("Buddy", "Golden Retriever");
	
	System.out.println("\nCalling the overridden displayInfo method:");
	dog.displayInfo(); // Calls the Dog's overridden method, which uses super
	}
	}
	Output:-
	Animal constructor called: Buddy
	Dog constructor called: Breed - Golden Retriever
	Calling the overridden displayInfo method:
	Animal Name: Buddy
	Breed: Golden Retriever

P8	Develop a Java application that demonstrates method overriding and dynamic method dispatch.

Solution	// Superclass
	class Animal {
	// Overridden method
	void sound() {
	System.out.println("Animals make different sounds.");
	}
	}

```
// Subclass 1
class Dog extends Animal {
    // Overriding the sound method
    @Override
    void sound() {
        System.out.println("Dog barks: Woof Woof!");
    }
}

// Subclass 2
class Cat extends Animal {
    // Overriding the sound method
    @Override
    void sound() {
        System.out.println("Cat meows: Meow Meow!");
    }
}

// Subclass 3
class Cow extends Animal {
    // Overriding the sound method
    @Override
    void sound() {
        System.out.println("Cow moos: Moo Moo!");
    }
}

// Main Class
public class MethodOverridingDemo {
    public static void main(String[] args) {
        // Reference of Animal class pointing to Dog object
        Animal animal;
```

	System.out.println("Dynamic Method Dispatch Demonstration:");
	
	// Dynamic dispatch: assigning Dog object to Animal reference
	animal = new Dog();
	animal.sound(); // Calls Dog's overridden method
	
	// Dynamic dispatch: assigning Cat object to Animal reference
	animal = new Cat();
	animal.sound(); // Calls Cat's overridden method
	
	// Dynamic dispatch: assigning Cow object to Animal reference
	animal = new Cow();
	animal.sound(); // Calls Cow's overridden method
	}
	}
	Output:-
	Dynamic Method Dispatch Demonstration:
	Dog barks: Woof Woof!
	Cat meows: Meow Meow!
	Cow moos: Moo Moo!

P9	Write a program to read input from the console and display it using predefined input/output streams.

Solution	import java.io.*;
	
	public class ConsoleInputOutput {
	public static void main(String[] args) {
	// Creating a BufferedReader to read input from the console
	BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
	
	try {

	// Prompt the user for input
	System.out.print("Enter your name: ");
	String name = reader.readLine(); // Read input
	
	System.out.print("Enter your age: ");
	String ageString = reader.readLine(); // Read input
	int age = Integer.parseInt(ageString); // Convert age to integer
	
	// Displaying the collected information
	System.out.println("\n--- Collected Information ---");
	System.out.println("Name: " + name);
	System.out.println("Age: " + age);
	
	} catch (IOException e) {
	System.err.println("Error reading input: " + e.getMessage());
	} catch (NumberFormatException e) {
	System.err.println("Invalid input for age. Please enter a valid integer.");
	}
	}
	}
	Input:-
	Enter your name: John
	Enter your age: 30
	Output:-
	Name: John
	Age: 30

P10	Create a Java program that utilizes one-dimensional and two-dimensional arrays. Perform operations such as sorting or searching on these arrays.
Solution	import java.util.Arrays;

```
public class ArrayOperations {

    // Method to perform sorting and searching on a one-dimensional array
    public static void oneDimensionalArrayOperations() {
        int[] numbers = {12, 45, 23, 56, 89, 4, 32};

        System.out.println("Original One-Dimensional Array:");
        System.out.println(Arrays.toString(numbers));

        // Sorting the array
        Arrays.sort(numbers);
        System.out.println("\nSorted One-Dimensional Array:");
        System.out.println(Arrays.toString(numbers));

        // Searching for an element in the sorted array
        int target = 45;
        int index = Arrays.binarySearch(numbers, target);

        if (index >= 0) {
            System.out.println("\nElement " + target + " found at index " +
index);
        } else {
            System.out.println("\nElement " + target + " not found.");
        }
    }

    // Method to perform sorting and searching on a two-dimensional array
    public static void twoDimensionalArrayOperations() {
        int[][] matrix = {
            {12, 45, 23},
            {56, 89, 4},
            {32, 19, 71}
        };
```

```
        System.out.println("\nOriginal Two-Dimensional Array (Matrix):");
        printMatrix(matrix);

        // Sorting each row of the matrix
        System.out.println("\nSorted Two-Dimensional Array (Rows sorted):");
        for (int i = 0; i < matrix.length; i++) {
            Arrays.sort(matrix[i]);
        }
        printMatrix(matrix);

        // Searching for an element in the matrix
        int target = 19;
        boolean found = false;
        for (int i = 0; i < matrix.length; i++) {
            int index = Arrays.binarySearch(matrix[i], target);
            if (index >= 0) {
                System.out.println("\nElement " + target + " found at matrix[" + i + "][" + index + "]");
                found = true;
                break;
            }
        }
        if (!found) {
            System.out.println("\nElement " + target + " not found in the matrix.");
        }
    }

    // Helper method to print a 2D matrix
    public static void printMatrix(int[][] matrix) {
        for (int i = 0; i < matrix.length; i++) {
```

	System.out.println(Arrays.toString(matrix[i]));
	}
	}
	
	public static void main(String[] args) {
	// Performing operations on one-dimensional array
	oneDimensionalArrayOperations();
	
	// Performing operations on two-dimensional array
	twoDimensionalArrayOperations();
	}
	}
	Output:-
	Original One-Dimensional Array:
	[12, 45, 23, 56, 89, 4, 32]
	Sorted One-Dimensional Array:
	[4, 12, 23, 32, 45, 56, 89]
	Element 45 found at index 4
	Original Two-Dimensional Array (Matrix):
	[12, 45, 23]
	[56, 89, 4]
	[32, 19, 71]
	Sorted Two-Dimensional Array (Rows sorted):
	[12, 23, 45]
	[4, 56, 89]
	[19, 32, 71]
	Element 19 found at matrix[2][0]

P11	Implement a program that explores string handling using both the String and StringBuffer classes.
Solution	public class StringAndStringBuffer { public static void main(String[] args) {

```
        // Example using String (immutable)
        String str = "Hello, World!";
        System.out.println("Original String: " + str);

        // Modifying String (creates a new String object)
        str = str.concat(" Welcome to Java.");
        System.out.println("Modified String: " + str);

        // Example using StringBuffer (mutable)
        StringBuffer stringBuffer = new StringBuffer("Hello, World!");
        System.out.println("\nOriginal StringBuffer: " + stringBuffer);

        // Modifying StringBuffer (modifies the original object)
        stringBuffer.append(" Welcome to Java.");
        System.out.println("Modified StringBuffer: " + stringBuffer);

        // Demonstrating other StringBuffer operations
        stringBuffer.insert(13, " - Java Programming");
        System.out.println("After Insertion: " + stringBuffer);

        stringBuffer.reverse();
        System.out.println("After Reversal: " + stringBuffer);

        stringBuffer.replace(0, 5, "Hi");
        System.out.println("After Replacement: " + stringBuffer);
    }
}
```

Output:-

```
Original String: Hello, World!
Modified String: Hello, World! Welcome to Java.
Original StringBuffer: Hello, World!
Modified StringBuffer: Hello, World! Welcome to Java.
After Insertion: Hello, World! - Java Programming Welcome to Java.
```

	After Reversal: .avaJ ot emocleW gnimmargorP avaJ - dlroW ,olleH After Replacement: Hi, World! - Java Programming Welcome to Java.
P12	Define a package and demonstrate its usage in different Java classes. Import and utilize classes from other packages.

Solution	package vehicles;
	
	public class Car {
	private String model;
	private double fuelEfficiency; // in miles per gallon
	
	// Constructor to initialize the car's model and fuel efficiency
	public Car(String model, double fuelEfficiency) {
	this.model = model;
	this.fuelEfficiency = fuelEfficiency;
	}
	
	// Method to get the fuel efficiency
	public double getFuelEfficiency() {
	return fuelEfficiency;
	}
	
	// Method to display car details
	public void displayDetails() {
	System.out.println("Car Model: " + model);
	System.out.println("Fuel Efficiency: " + fuelEfficiency + " mpg");
	}
	}
	package vehicles;
	
	public class Truck {
	private String model;

```
    private double loadCapacity; // in tons

    // Constructor to initialize the truck's model and load capacity
    public Truck(String model, double loadCapacity) {
        this.model = model;
        this.loadCapacity = loadCapacity;
    }

    // Method to get the load capacity
    public double getLoadCapacity() {
        return loadCapacity;
    }

    // Method to display truck details
    public void displayDetails() {
        System.out.println("Truck Model: " + model);
        System.out.println("Load Capacity: " + loadCapacity + " tons");
    }
}
```

Step 2:-

```
package app;
import vehicles.Car;
import vehicles.Truck;

public class MainApp {
    public static void main(String[] args) {
        // Create objects of Car and Truck from the vehicles package
        Car car = new Car("Toyota Prius", 50);  // Model and fuel efficiency (mpg)
        Truck truck = new Truck("Ford F-150", 3);  // Model and load capacity (tons)

        // Display details of the car and truck
```

	car.displayDetails();
	truck.displayDetails();
	}
	}
	Output:-
	Car Model: Toyota Prius
	Fuel Efficiency: 50.0 mpg
	Truck Model: Ford F-150
	Load Capacity: 3.0 tons

P13	Create interfaces with variables and implement them in Java classes to achieve multiple inheritance.

Solution	**Step 1.**
	// Swimmable.java
	public interface Swimmable {
	// A constant variable (implicitly public, static, final)
	int MAX_SWIM_SPEED = 10; // in meters per second
	
	// Abstract method to be implemented by classes
	void swim();
	}
	Step 2.
	// Flyable.java
	public interface Flyable {
	// A constant variable (implicitly public, static, final)
	int MAX_FLY_ALTITUDE = 5000; // in meters
	
	// Abstract method to be implemented by classes
	void fly();
	}
	Step 3.
	// Duck.java

```
public class Duck implements Swimmable, Flyable {
    private String name;

    // Constructor to initialize the duck's name
    public Duck(String name) {
        this.name = name;
    }

    // Implementing the swim() method from Swimmable interface
    @Override
    public void swim() {
        System.out.println(name + " is swimming at a speed of " +
MAX_SWIM_SPEED + " m/s.");
    }

    // Implementing the fly() method from Flyable interface
    @Override
    public void fly() {
        System.out.println(name + " is flying at an altitude of " +
MAX_FLY_ALTITUDE + " meters.");
    }

    // Method to display duck details
    public void displayDetails() {
        System.out.println("Duck Name: " + name);
    }
}
```

Step 4.

```
// MainApp.java
public class MainApp {
    public static void main(String[] args) {
        // Create a Duck object
        Duck myDuck = new Duck("Donald");
```

	// Display details of the duck
	myDuck.displayDetails();
	// Call methods from both interfaces
	myDuck.swim(); // From Swimmable interface
	myDuck.fly(); // From Flyable interface
	}
	}
	
	Output :-
	Duck Name: Donald Donald is swimming at a speed of 10 m/s. Donald is flying at an altitude of 5000 meters.

P14	Develop a program that uses both interfaces and abstract classes, showcasing their differences.

Solution	**Step 1**
	// Swimmable.java
	public interface Swimmable {
	// Abstract method for swimming
	void swim();
	}
	Step 2
	// Animal.java
	public abstract class Animal {
	// Instance variable
	protected String name;
	// Constructor to initialize the animal's name
	public Animal(String name) {
	this.name = name;
	}
	// Abstract method (to be implemented by subclasses)
	public abstract void makeSound();

```
    // Concrete method
    public void eat() {
        System.out.println(name + " is eating.");
    }
}
```

Step 3

```
// Dolphin.java
public class Dolphin extends Animal implements Swimmable {

    // Constructor to initialize the dolphin's name
    public Dolphin(String name) {
        super(name);
    }

    // Implementing the makeSound() method from Animal
    @Override
    public void makeSound() {
        System.out.println(name + " says: Eee Eee!");
    }

    // Implementing the swim() method from Swimmable
    @Override
    public void swim() {
        System.out.println(name + " is swimming.");
    }
}
```

Step 4

```
// Dolphin.java
public class Dolphin extends Animal implements Swimmable {
    // Constructor to initialize the dolphin's name
    public Dolphin(String name) {
        super(name);
    }
```

	// Implementing the makeSound() method from Animal
	@Override
	public void makeSound() {
	System.out.println(name + " says: Eee Eee!");
	}
	// Implementing the swim() method from Swimmable
	@Override
	public void swim() {
	System.out.println(name + " is swimming.");
	}
	}

P15	Write a Java program that demonstrates the different types of exceptions. Implement try-catch blocks to handle exceptions effectively.

Solution	// ExceptionDemo.java
	import java.io.File;
	import java.io.FileNotFoundException;
	import java.io.IOException;
	
	public class ExceptionDemo {
	
	public static void main(String[] args) {
	// Demonstrating a Checked Exception: FileNotFoundException
	try {
	readFile("nonexistent_file.txt");
	} catch (FileNotFoundException e) {
	System.out.println("Checked Exception: File not found: " + e.getMessage());
	}
	
	// Demonstrating an Unchecked Exception: ArithmeticException
	try {

```
            int result = 10 / 0;  // This will cause division by zero
        } catch (ArithmeticException e) {
            System.out.println("Unchecked Exception: Arithmetic error: " + e.getMessage());
        }

        // Demonstrating an Unchecked Exception: NullPointerException
        try {
            String str = null;
            System.out.println(str.length());  // This will cause a NullPointerException
        } catch (NullPointerException e) {
            System.out.println("Unchecked Exception: Null pointer error: " + e.getMessage());
        }

        // Demonstrating an Unchecked Exception: ArrayIndexOutOfBoundsException
        try {
            int[] arr = new int[5];
            System.out.println(arr[10]);  // This will cause an ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Unchecked Exception: Array index out of bounds: " + e.getMessage());
        }

        // Demonstrating Generic Exception Handling (catching any exception)
        try {
            throw new IOException("A generic IO error occurred");
        } catch (Exception e) {
            System.out.println("Generic Exception: " + e.getMessage());
```

	}
	
	System.out.println("Program execution continues...");
	}
	
	// A method that throws FileNotFoundException (checked exception)
	public static void readFile(String filename) throws FileNotFoundException {
	File file = new File(filename);
	if (!file.exists()) {
	throw new FileNotFoundException("File " + filename + " does not exist.");
	}
	}
	}
	Output:-
	Checked Exception: File not found: nonexistent_file.txt
	Unchecked Exception: Arithmetic error: / by zero
	Unchecked Exception: Null pointer error: Cannot invoke "String.length()" because "str" is null
	Unchecked Exception: Array index out of bounds: Index 10 out of bounds for length 5
	Generic Exception: A generic IO error occurred
	Program execution continues...

P16	Create a program with multiple try and catch clauses and observe how the program behaves in different scenarios.

Solution	**Step 1**
	// MultipleCatchDemo.java
	public class MultipleCatchDemo {
	public static void main(String[] args) {
	// Scenario 1: Division by zero (ArithmeticException)

```
        try {
            int num = 10;
            int result = num / 0; // This will cause ArithmeticException
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Caught ArithmeticException: Division by zero.");
        }

        // Scenario 2: Array index out of bounds (ArrayIndexOutOfBoundsException)
        try {
            int[] arr = new int[5];
            System.out.println(arr[10]); // This will cause ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Caught ArrayIndexOutOfBoundsException: Invalid array index.");
        }

        // Scenario 3: Invalid number format (NumberFormatException)
        try {
            String str = "abc";
            int number = Integer.parseInt(str); // This will cause NumberFormatException
        } catch (NumberFormatException e) {
            System.out.println("Caught NumberFormatException: Invalid number format.");
        }

        // Scenario 4: Null pointer exception (NullPointerException)
        try {
            String str = null;
```

```
            System.out.println(str.length()); // This will cause NullPointerException
        } catch (NullPointerException e) {
            System.out.println("Caught NullPointerException: Attempt to access a method on a null object.");
        }

        // Scenario 5: Multiple exceptions in one try block
        try {
            int[] arr = new int[2];
            System.out.println(arr[2]); // ArrayIndexOutOfBoundsException
            String str = null;
            System.out.println(str.length()); // NullPointerException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Caught ArrayIndexOutOfBoundsException: Invalid array index.");
        } catch (NullPointerException e) {
            System.out.println("Caught NullPointerException: Attempt to access a method on a null object.");
        }

        // Scenario 6: Catching generic exception
        try {
            String text = "Hello";
            char ch = text.charAt(10); // This will cause StringIndexOutOfBoundsException
        } catch (Exception e) {
            System.out.println("Caught generic Exception: " + e.getClass().getSimpleName());
        }

        System.out.println("Program continues execution...");
    }
```

	} **Output:-** Caught ArithmeticException: Division by zero. Caught ArrayIndexOutOfBoundsException: Invalid array index. Caught NumberFormatException: Invalid number format. Caught NullPointerException: Attempt to access a method on a null object. Caught ArrayIndexOutOfBoundsException: Invalid array index. Caught NullPointerException: Attempt to access a method on a null object. Caught generic Exception: StringIndexOutOfBoundsException Program continues execution...
P17	Implement a custom exception class and use it in your program to handle specific error conditions.

Solution	// AgeNotValidException.java (Custom Exception)
	public class AgeNotValidException extends Exception {
	  // Constructor that accepts a message
	  public AgeNotValidException(String message) {
	  super(message);
	  }
	}
	
	// Main class to test the custom exception
	import java.util.Scanner;
	
	public class CustomExceptionDemo {
	
	  // Method to check age validity
	  public static void checkAge(int age) throws AgeNotValidException {
	  if (age < 0 || age > 150) {
	  throw new AgeNotValidException("Age must be between 0 and

	150.");
	} else {
	System.out.println("Age is valid: " + age);
	}
	}
	
	public static void main(String[] args) {
	Scanner scanner = new Scanner(System.in);
	
	System.out.print("Enter your age: ");
	int age = scanner.nextInt();
	
	try {
	checkAge(age); // Calling the method that might throw the custom exception
	} catch (AgeNotValidException e) {
	System.out.println("Caught custom exception: " + e.getMessage());
	}
	
	scanner.close();
	}
	}
	Output:-
	Enter your age: 25
	Age is valid: 25

P18	Develop a Java program with multiple threads, each performing a different task. Use thread priorities and observe the execution order.

Solution	// Task1.java (Thread 1)
	class Task1 extends Thread {
	public void run() {

```
        for (int i = 0; i < 5; i++) {
            System.out.println("Task 1 - Low priority thread: " + i);
            try {
                Thread.sleep(500);  // Simulating some task with sleep
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}

// Task2.java (Thread 2)
class Task2 extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Task 2
```

Output:-

```
Task 3 - High priority thread: 0
Task 3 - High priority thread: 1
Task 3 - High priority thread: 2
Task 3 - High priority thread: 3
Task 3 - High priority thread: 4
Task 2 - Medium priority thread: 0
Task 1 - Low priority thread: 0
Task 2 - Medium priority thread: 1
Task 2 - Medium priority thread: 2
Task 1 - Low priority thread: 1
Task 2 - Medium priority thread: 3
Task 1 - Low priority thread: 2
Task 2 - Medium priority thread: 4
Task 1 - Low priority thread: 3
Task 1 - Low priority thread: 4
```

P19	Implement synchronization mechanisms in a multi-threaded program to avoid data race conditions.

Solution	class Counter {
	  private int count = 0;
	
	  // Synchronized method to ensure that only one thread modifies count at a time
	  public synchronized void increment() {
	  count++;
	  }
	
	  public int getCount() {
	  return count;
	  }
	}
	
	class Task extends Thread {
	  private Counter counter;
	
	  public Task(Counter counter) {
	  this.counter = counter;
	  }
	
	  @Override
	  public void run() {
	  for (int i = 0; i < 1000; i++) {
	  counter.increment(); // Increment the shared counter
	  }
	  }
	}
	
	public class SynchronizationDemo {

	public static void main(String[] args) {
	// Creating the shared counter object
	Counter counter = new Counter();
	
	// Creating multiple threads that will operate on the shared counter
	Thread t1 = new Task(counter);
	Thread t2 = new Task(counter);
	Thread t3 = new Task(counter);
	
	// Starting the threads
	t1.start();
	t2.start();
	t3.start();
	
	try {
	// Waiting for all threads to finish
	t1.join();
	t2.join();
	t3.join();
	} catch (InterruptedException e) {
	Output:-
	Final counter value: 3000

P20	Create a program that demonstrates inter-thread communication and includes features like deadlock, thread suspension, resumption, and stopping.

Solution	// Thread A that waits for Thread B to complete its task
	class ThreadA extends Thread {
	private final Object lock;
	
	public ThreadA(Object lock) {
	this.lock = lock;

```
    }

    @Override
    public void run() {
        synchronized (lock) {
            try {
                System.out.println("ThreadA: Waiting for ThreadB to complete...");
                lock.wait(); // ThreadA waits for ThreadB to notify
                System.out.println("ThreadA: Resumed and completed.");
            } catch (InterruptedException e) {
                System.out.println("ThreadA: Interrupted.");
            }
        }
    }
}

// Thread B that notifies ThreadA to proceed
class ThreadB extends Thread {
    private final Object lock;

    public ThreadB(Object lock) {
        this.lock = lock;
    }

    @Override
    public void run() {
        synchronized (lock) {
            try {
                System.out.println("ThreadB: Performing some task...");
                Thread.sleep(2000);  // Simulate some work
                lock.notify(); // Notify ThreadA to proceed
                System.out.println("ThreadB: Task completed and notified
```

```
ThreadA.");
            } catch (InterruptedException e) {
                System.out.println("ThreadB: Interrupted.");
            }
        }
    }
}

// Deadlock scenario: Two threads trying to acquire locks on each other
class ThreadC extends Thread {
    private final Object lock1;
    private final Object lock2;

    public ThreadC(Object lock1, Object lock2) {
        this.lock1 = lock1;
        this.lock2 = lock2;
    }

    @Override
    public void run() {
        synchronized (lock1) {
            System.out.println("ThreadC: Holding lock1...");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                System.out.println("ThreadC: Interrupted.");
            }

            synchronized (lock2) {
                System.out.println("ThreadC: Acquired lock2!");
            }
        }
    }
```

```
}

class ThreadD extends Thread {
    private final Object lock1;
    private final Object lock2;

    public ThreadD(Object lock1, Object lock2) {
        this.lock1 = lock1;
        this.lock2 = lock2;
    }

    @Override
    public void run() {
        synchronized (lock2) {
            System.out.println("ThreadD: Holding lock2...");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                System.out.
```

Output:-

```
ThreadB: Performing some task...
ThreadA: Waiting for ThreadB to complete...
ThreadB: Task completed and notified ThreadA.
ThreadA: Resumed and completed.
ThreadC: Holding lock1...
ThreadD: Holding lock2...
```