Java Core Assignment

1. Introduction to Java

History of Java:

Java was developed by James Gosling at Sun Microsystems in 1995. Initially, it was named *Oak*, but later renamed Java. It was designed to be platform-independent and is widely used for developing applications ranging from web to enterprise solutions.

Features of Java:

1. **Platform Independent** – Java runs on the "Write Once, Run Anywhere" principle.
2. **Object-Oriented** – Java follows OOP principles like Encapsulation, Inheritance, and Polymorphism.
3. **Robust** – Java has strong memory management and exception handling.
4. **Secure** – It provides security features like bytecode verification and runtime checks.
5. **Multithreading** – Java supports multithreading for better performance.

---

2. Data Types, Variables, and Operators

Primitive Data Types in Java:

- int – Integer (4 bytes)
- float – Floating-point numbers (4 bytes)
- double – Double-precision floating-point numbers (8 bytes)
- char – Character type (2 bytes)
- boolean – Boolean values (true or false)

Program:

```
int age = 25;

float price = 99.99f;

char grade = 'A';

boolean isJavaFun = true;
```

Operators in Java:

- **Arithmetic Operators :** (+, -, *, /, %)
- **Relational Operators :** (>, <, >=, <=, ==, !=)
- **Logical Operators :** (&&, ||, !)
- **Bitwise Operators :** (&, |, ^, ~, <<, >>)

Type Casting Example:

java

```
double num = 10.5;
int castedNum = (int) num;  // Explicit Type Casting
System.out.println(castedNum);  // Output: 10
```

Control Flow Statements in Java :

1. If-Else Statements

The if-else statement is used to execute a block of code conditionally. If the condition is true, the if block executes; otherwise, the else block executes.

Program:

```
public class IfElseExample {
    public static void main(String[] args) {
        int number = 10;
 if (number % 2 == 0) {
            System.out.println(number + " is even.");
        } else {
            System.out.println(number + " is odd.");
        }
    }
}
```

2. Switch-Case Statements

The switch-case statement is used when you need to check a variable against multiple constant values.

Program:

```
import java.util.Scanner;

public class SwitchExample {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter a number (1-3): ");
        int choice = sc.nextInt();
 switch (choice) {
            case 1:
                System.out.println("You chose One.");
                break;
            case 2:
                System.out.println("You chose Two.");
                break;
            case 3:
                System.out.println("You chose Three.");
                break;
            default:
                System.out.println("Invalid choice.");
        }
    }
}
```

3. Loops in Java

Loops are used to execute a block of code multiple times based on a condition.

(a) For Loop

A for loop is used when the number of iterations is known beforehand.

Program:

```
public class ForLoopExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Number: " + i);
```

```
    }
    }
}
```

(b) While Loop

A while loop is used when the number of iterations is **not known beforehand**, and it runs as long as the condition is **true**.

Program:

```
public class WhileLoopExample {
    public static void main(String[] args) {
        int i = 1;
        while (i <= 5) {
            System.out.println("Number: " + i);
            i++;
        }
    }
}
```

(c) Do-While Loop

A do-while loop is similar to while, but it **executes at least once**, even if the condition is false.

Program:

```
public class DoWhileExample {
    public static void main(String[] args) {
        int i = 1;
        do {
            System.out.println("Number: " + i);
            i++;
        } while (i <= 5);
    }
}
```

4. Break and Continue Keywords

(a) Break Statement

- Used to **exit** the loop or switch statement immediately.
- Often used when a specific condition is met.

(b) Continue Statement

- Used to **skip** the current iteration and move to the next.
- Does **not** terminate the loop.

Class and Object in Java

1. What is a Class in Java?

A class in Java is a blueprint or template for creating objects. It defines attributes (variables) and behaviors (methods) that the objects of the class will have.

Program:

```
class Car {
   String brand;
   int speed;
   void display() {
      System.out.println("Brand: " + brand);
      System.out.println("Speed: " + speed + " km/h");
   }
}
```

2. What is an Object in Java?

An **object** is an instance of a class. It is created using the new keyword, and it has **its own unique values for attributes**.

Program:

```
public class CarDemo {
   public static void main(String[] args) {
      Car myCar = new Car();
      myCar.brand = "Toyota";
      myCar.speed = 120;
      myCar.display();
```

}

}

Constructors and Overloading in Java

1. What is a Constructor?

A constructor is a special method in Java used to initialize objects. It is called automatically when an object of a class is created. The constructor has the same name as the class and does not have a return type (not even void).

2. Types of Constructors

Java provides three types of constructors:

(a) Default Constructor

- A **default constructor** is automatically provided by Java **if no constructor is defined**.
- It initializes objects with default values.

(b) Parameterized Constructor

- A **parameterized constructor** accepts arguments to initialize an object with specific values.

3. Overloading in java :

Overloading means defining multiple constructors with different parameters in the same class. Java distinguishes them based on the number and type of arguments.

this Keyword in Java :

The this keyword in Java refers to the current instance of the class. It is used to differentiate between instance variables and local variables, call constructors and methods, and return the current object.

Program :

```
class Student {
   String name;
   int age;
Student(String name, int age) {
      this.name = name;
      this.age = age;
```

```
}
    void display() {
        System.out.println("Student Name: " + name);
        System.out.println("Student Age: " + age);
    }
 public static void main(String[] args) {
        Student student1 = new Student("John", 20);
        Student student2 = new Student("Alice", 22);
        student1.display();
        student2.display();
    }
}
```

Output:

Student Name: John

Student Age: 20

Student Name: Alice

Student Age: 22

Methods in Java :

1. Defining Methods in Java:

A method in Java is a block of code that performs a specific task. It helps in code reusability and modularity.

2. Method Parameters in Java

A method can accept **parameters (arguments)** to perform operations on given inputs.

3. Method Overloading in Java

Method overloading means defining **multiple methods with the same name but different parameters** (different number, type, or order of parameters). Java determines the correct method based on the arguments passed.

4. Static Methods and Variables in Java

- **Static Variables**: Belong to the class, not instances (objects). Shared among all objects.

- **Static Methods**: Can be called without creating an object. They can only access static variables.

Program :

Write a program to find the maximum of three numbers using a method.

```
import java.util.Scanner;
class MaxFinder {
   static int findMax(int a, int b, int c) {
      return Math.max(a, Math.max(b, c));
   }
public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.print("Enter first number: ");
      int num1 = sc.nextInt();
      System.out.print("Enter second number: ");
      int num2 = sc.nextInt();
      System.out.print("Enter third number: ");
      int num3 = sc.nextInt();
      int max = findMax(num1, num2, num3);
      System.out.println("The maximum number is: " + max);
sc.close();
   }
}
```

Output:

```
Enter first number: 25
Enter second number: 45
Enter third number: 30
The maximum number is: 45
```

Object-Oriented Programming (OOPs) Concepts :

Java follows the Object-Oriented Programming (OOP) concept, which allows us to design programs using real-world entities like objects and classes. The four fundamental OOP principles are Encapsulation, Inheritance, Polymorphism, and Abstraction.

(a) Encapsulation

Encapsulation is the process of **hiding the internal details** of a class and **restricting direct access** to data. It is achieved using **private variables** and **public getter and setter methods**.

(b) Inheritance

Inheritance is the process where a **child class (subclass)** acquires the properties and methods of a **parent class (superclass)**. It **promotes code reusability**.

(c) Polymorphism

Polymorphism means **"many forms"**. It allows **one method to perform different tasks**.
It is of two types:

1. **Compile-time Polymorphism (Method Overloading)**
2. **Runtime Polymorphism (Method Overriding)**

(d) Abstraction

Abstraction is the process of **hiding the implementation details** and only **showing the necessary features** to the user. It is implemented using:

- **Abstract Classes** (abstract keyword)
- **Interfaces**

Program :

Write a program demonstrating single inheritance.

```
class Animal {
   void eat() {
      System.out.println("This animal eats food.");
   }
}
class Dog extends Animal {
   // Additional method in child class
   void bark() {
      System.out.println("Dog barks.");
   }
```

```
}
public class SingleInheritanceExample {
   public static void main(String[] args) {
      Dog myDog = new Dog();
  myDog.eat();
      myDog.bark();
   }
}
```

Output:

This animal eats food.

Dog barks.

Arrays and Strings in Java :

1. One-Dimensional and Multidimensional Arrays

An array is a collection of elements of the same data type, stored in contiguous memory locations.

a) One-Dimensional Array

A **one-dimensional array** is a linear collection of elements.

(b) Multidimensional Arrays

A **multidimensional array** is an array of arrays, where data is stored in **rows and columns**.

2. String Handling in Java

A **string** is a sequence of characters. Java provides three ways to handle strings:

1. **String Class (Immutable)**
2. **StringBuffer Class (Mutable)**
3. **StringBuilder Class (Mutable, Faster)**

(a) String Class

A **String** in Java is immutable (cannot be changed once created).

(b) StringBuffer Class

StringBuffer allows **modification** of strings and is **thread-safe**.

(c) StringBuilder Class

StringBuilder is similar to StringBuffer but is **faster (not thread-safe)**.

3. Array of Objects:

An array of objects stores multiple objects in a single array.

Program :

Write a program to perform matrix addition and subtraction using 2D arrays .

```
import java.util.Scanner;
public class MatrixOperations {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the number of rows: ");
        int rows = sc.nextInt();
        System.out.print("Enter the number of columns: ");
        int cols = sc.nextInt();
        int[][] matrix1 = new int[rows][cols];
        int[][] matrix2 = new int[rows][cols];
        int[][] sumMatrix = new int[rows][cols];
        int[][] subMatrix = new int[rows][cols];
        System.out.println("Enter elements of first matrix:");
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                matrix1[i][j] = sc.nextInt();
            }
        }
        System.out.println("Enter elements of second matrix:");
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                matrix2[i][j] = sc.nextInt();
            }
        }
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
```

```
                sumMatrix[i][j] = matrix1[i][j] + matrix2[i][j]; // Addition
                subMatrix[i][j] = matrix1[i][j] - matrix2[i][j]; // Subtraction
            }
        }
        System.out.println("\nMatrix Addition Result:");
        displayMatrix(sumMatrix);
        System.out.println("\nMatrix Subtraction Result:");
        displayMatrix(subMatrix);
 sc.close();
    }
    static void displayMatrix(int[][] matrix) {
        for (int[] row : matrix) {
            for (int element : row) {
                System.out.print(element + " ");
            }
            System.out.println();
        }
    }
}
```

Output :

Enter the number of rows: 2

Enter the number of columns: 2

Enter elements of first matrix:

1 2

3 4

Enter elements of second matrix:

5 6

7 8

Matrix Addition Result:

6 8

10 12

Matrix Subtraction Result:

-4 -4

-4 -4

Program:

Create a program to reverse a string and check for palindromes.

```
import java.util.Scanner;
public class PalindromeCheck {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a string: ");
        String original = sc.nextLine();
        String reversed = reverseString(original);
        System.out.println("Reversed String: " + reversed);
        if (original.equalsIgnoreCase(reversed)) {
            System.out.println("The string is a palindrome.");
        } else {
            System.out.println("The string is NOT a palindrome.");
        }
  sc.close();
    }
    public static String reverseString(String str) {
        String reversed = "";
        for (int i = str.length() - 1; i >= 0; i--) {
            reversed += str.charAt(i);
        }
        return reversed;
    }
}
```

Output :

Enter a string: madam

Reversed String: madam

The string is a palindrome.

Interfaces and Abstract Classes in Java :

1. Abstract Classes and Methods

An abstract class in Java:

- Cannot be instantiated (cannot create objects of it).
- May contain abstract methods (without a body) and concrete methods (with implementation).
- Is used to provide partial abstraction.

2. Interfaces: Multiple Inheritance in Java

An **interface** in Java:

- Defines a contract that classes must follow.
- Contains only **abstract methods** (until Java 8).
- Supports **multiple inheritance** (as a class can implement multiple interfaces).
- **Cannot have instance variables** (only static final constants).

3. Implementing Multiple Interfaces in Java

Java **does not support multiple inheritance in classes** (to avoid ambiguity). However, **a class can implement multiple interfaces**, achieving multiple inheritance.

Exception Handling in Java :

1. What is Exception Handling?

An exception in Java is an unexpected event that occurs during the execution of a program, disrupting the normal flow. Exception handling allows the program to handle such errors gracefully and prevent crashes.

2. Types of Exceptions in Java

Exceptions in Java are classified into two main types:

(a) Checked Exceptions

- Checked exceptions are **checked at compile-time**.
- The program **will not compile** if a checked exception is not handled.
- Example: **FileNotFoundException, IOException, SQLException**

(b) Unchecked Exceptions

- Unchecked exceptions **occur at runtime**.
- These are caused by **logical errors** in the program.
- Example: **NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException**

3. Exception Handling Using try, catch, finally, throw, and throws

(a) try and catch

- The try block contains the code that may cause an exception.
- The catch block handles the exception.

(b) finally Block

- The finally block executes whether an exception occurs or not.
- It is used to release resources like closing files or database connections.

(c) throw Keyword

- The throw keyword is used to manually throw an exception.

(d) throws Keyword

- The **throws keyword** is used to declare an exception in a method signature.
- It **does not handle the exception** but **indicates that the method might throw an exception**.

Program :

```
import java.util.Scanner;
public class ExceptionHandlingExample {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);

      try {
```

```
            System.out.print("Enter numerator: ");
            int numerator = sc.nextInt();
 System.out.print("Enter denominator: ");
            int denominator = sc.nextInt();
            int result = numerator / denominator;
            System.out.println("Result: " + result);
  } catch (ArithmeticException e) {
            System.out.println("Error: Division by zero is not allowed.");
        } finally {
            System.out.println("Execution completed.");
            sc.close(); // Closing scanner resource
        }
    }
}
```

Output:

Enter numerator: 10

Enter denominator: 2

Result: 5

Execution completed.

Multithreading in Java :

Introduction to Threads

A thread in Java is a lightweight process that runs independently, allowing multiple operations to execute concurrently. Java provides built-in support for multithreading through the Thread class and Runnable interface.

Advantages of Threads:

- Efficient CPU utilization
- Faster execution by running tasks in parallel
- Supports asynchronous programming

- Helps in real-time applications like gaming, networking, and GUI-based programs

Thread Life Cycle :

A thread in Java goes through several states:

1. **New (Created)** – A thread is created but not yet started (Thread t = new Thread();).
2. **Runnable** – The thread is ready to run but waiting for CPU time (t.start();).
3. **Running** – The thread is executing (run() method is active).
4. **Blocked/Waiting** – The thread is waiting due to synchronization or inter-thread communication.
5. **Terminated (Dead)** – The thread execution is complete.

Synchronization in Java

When multiple threads access shared resources, **data inconsistency** can occur. Java provides synchronization to avoid such issues.

Inter-thread Communication

Inter-thread communication allows threads to **communicate** using wait(), notify(), and notifyAll().

Program :

Write a program to create and run multiple threads using the Thread class.

```
class MyThread extends Thread {
  private String threadName;
MyThread(String name) {
      this.threadName = name;
   }
  public void run() {
      for (int i = 1; i <= 5; i++) {
         System.out.println(threadName + " - Count: " + i);
         try {
            Thread.sleep(500); // Simulating some work
         } catch (InterruptedException e) {
            System.out.println(threadName + " interrupted.");
         }
```

```
        }
        System.out.println(threadName + " has finished execution.");
    }
public static void main(String[] args) {
        MyThread t1 = new MyThread("Thread 1");
        MyThread t2 = new MyThread("Thread 2");
        MyThread t3 = new MyThread("Thread 3");
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Output :

```
Thread 1 - Count: 1
Thread 2 - Count: 1
Thread 3 - Count: 1
Thread 1 - Count: 2
Thread 2 - Count: 2
Thread 3 - Count: 2
...
Thread 1 has finished execution.
Thread 2 has finished execution.
Thread 3 has finished execution.
```