Java Final Question Paper Solution

Q1. What is the role of bitwise operators? and write a program in java to swap two numbers without using third variable using bitwise operator. And Using Multiplication and Division operation.

Role of Bitwise Operators

Bitwise operators are used to perform operations directly on the binary representations of numbers. They are useful for tasks that involve low-level data manipulation, optimization, or when working with flags and masks.

Common Bitwise Operators

- **AND (&)**: Compares each bit and returns 1 if both bits are 1, otherwise 0.
- **OR (|)**: Compares each bit and returns 1 if either bit is 1.
- **XOR (^)**: Compares each bit and returns 1 if the bits are different, otherwise 0.
- **NOT (~)**: Inverts the bits.
- **Left Shift (<<)**: Shifts bits to the left, adding 0 on the right.
- **Right Shift (>>)**: Shifts bits to the right, discarding the bits shifted out.

```
public class BitwiseSwap {
    public static void main(String[] args) {
        int a = 5; // First number
        int b = 7; // Second number
        System.out.println("Before Swap:");
        System.out.println("a = " + a + ", b = " + b);
        a = a ^ b; // Step 1: a becomes a ^ b
        b = a ^ b; // Step 2: b becomes (a ^ b) ^ b = a
        a = a ^ b; // Step 3: a becomes (a ^ b) ^ a = b
        // Display swapped values
        System.out.println("After Swap:");
        System.out.println("a = " + a + ", b = " + b);
    }
}
```

```
public class SwapNumbers {
    public static void main(String[] args) {
        // Initialize two variables
        int a = 10;
        int b = 5;
        System.out.println("Before swapping:");
        System.out.println("a = " + a + ", b = " + b);
        // Swapping using multiplication and division
        a = a * b;  // Step 1: a = a * b
        b = a / b;  // Step 2: b = a / b (b becomes the original a)
        a = a / b;  // Step 3: a = a / b (a becomes the original b)
        System.out.println("After swapping:");
        System.out.println("a = " + a + ", b = " + b);
    }
}
```

Q2. Design a code to handle ArrayIndexOutOfBounds, ArithmeticException, NullPointerException exceptions. There should also be an exception handler available for any user defined exception. OR

Design a code to handle multiple exceptions. The code must use concept of built-

in and custom exceptions. Assume suitable data and run-time cases.

Ans=>// Custom Exception class

```
class MyCustomException extends Exception {
    public MyCustomException(String message) {
        super(message);
    }
}
public class ExceptionHandlingExample {
    public static void main(String[] args) {
        // Handling ArrayIndexOutOfBoundsException
        try {
            int[] arr = new int[5];
            System.out.println(arr[10]);  // Accessing invalid index
```

```
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("ArrayIndexOutOfBoundsException caught: " + e.getMessage());
        }
        // Handling ArithmeticException
        try {
            int a = 10;
            int b = 0;
            System.out.println(a / b);  // Division by zero
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException caught: " + e.getMessage());
        }
        // Handling NullPointerException
        try {
            String str = null;
            System.out.println(str.length());  // Null reference access
        } catch (NullPointerException e) {
            System.out.println("NullPointerException caught: " + e.getMessage());
        }


// Handling User-defined Exception
        try {
            throw new MyCustomException("This is a user-defined exception!");
        } catch (MyCustomException e) {
            System.out.println("MyCustomException caught: " + e.getMessage());
        }
        catch (Exception e) {
            System.out.println("Exception caught: "+ e.getMessage());
        }
    }
}
```

Q3. Write a menu driven code using methods to handle following: 1. Start counting from the input number incrementing the value by 1 för each iteration. The counting stops once the current value is divisible by 10. 2. Find the count of factors for a given number. For example, the factors of 10 are 1,2,5,10 and count in this case is 4.

Ans=>

```
import java.util.Scanner;

public class MenuDrivenProgram {
   // Method to start counting and stop when the number is divisible by 10
   public static void countUntilDivisibleByTen(int num) {
      while (num % 10 != 0) {
         System.out.println(num);
         num++;
      }
      System.out.println("Final value divisible by 10: " + num);
   }
   // Method to count the number of factors of a given number
   public static int countFactors(int num) {
      int count = 0;
      for (int i = 1; i <= num; i++) {
         if (num % i == 0) {
            count++;
         }
      }
      return count;
   }
   // Main method to display the menu and handle user input
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int choice;
      do {
         // Displaying the menu
         System.out.println("Menu:");
```

```
System.out.println("1. Start counting from a number until it is divisible by 10");
System.out.println("2. Find the count of factors of a number");
System.out.println("3. Exit");
System.out.print("Enter your choice: ");
choice = sc.nextInt();

switch (choice) {
   case 1:
      // Handling option 1
      System.out.print("Enter the starting number: ");
      int startNum = sc.nextInt();
      countUntilDivisibleByTen(startNum);
      break;



   case 2:
     // Handling option 2
      System.out.print("Enter the number to find factors: ");
      int number = sc.nextInt();
      int factorCount = countFactors(number);
      System.out.println("The number of factors of " + number + " is: " + factorCount);
      break;
   case 3:
      // Exit option
      System.out.println("Exiting the program...");
      break;
   default:
      // Invalid option
      System.out.println("Invalid choice, please try again.");
}
```

```
        } while (choice != 3);
        sc.close(); // Close the scanner object
    }
}
```

Q4. Design a package to contain the class Student that contains data members such as name, roll number and another package contains the interface Sports which contains some sports information. Import these two packages in a package called Report which process both Student and Sport and give the report.

Ans=> **Create Packages for Student and Sports, and a Report Class**

Let's create a package structure as per the problem statement:

1. **Package studentinfo**: This package will contain the Student class with data members like name, roll number, and marks.
2. **Package sportsinfo**: This package will contain the Sports interface which holds information related to sports.
3. **Package report**: This package will import both studentinfo and sportsinfo packages and generate a report combining student details and sports information.

Step-by-step Implementation:

1. Student class (in package studentinfo)

// File: Student.java in package studentinfo

```
package studentinfo;
public class Student {
    private String name;
    private int rollNumber;
    public Student(String name, int rollNumber) {
        this.name = name;
        this.rollNumber = rollNumber;
    }
    public String getName() {
        return name;
    }
```

```
    public int getRollNumber() {
        return rollNumber;
    }
}
```

2. Sports interface (in package sportsinfo)

```
// File: Sports.java in package sportsinfo
package sportsinfo;
public interface Sports {
    String getSportDetails();
}
```

3. Cricket class implementing Sports (in package sportsinfo)

```
// File: Cricket.java in package sportsinfo
package sportsinfo;
public class Cricket implements Sports {
    private String sportName = "Cricket";
    private String country = "India";
    @Override
    public String getSportDetails() {
        return "Sport: " + sportName + ", Country: " + country;
    }
}
```

4. Report class (in package report)

```
// File: Report.java in package report
package report;
import studentinfo.Student;
import sportsinfo.Sports;
import sportsinfo.Cricket;

public class Report {
    public void generateReport(Student student, Sports sports) {
        System.out.println("Student Report:");
```

```
        System.out.println("Name: " + student.getName());
        System.out.println("Roll Number: " + student.getRollNumber());
        System.out.println(sports.getSportDetails());
    }
}
```

5. Main Class to Run the Report

```
// File: Main.java
import studentinfo.Student;
import sportsinfo.Sports;
import sportsinfo.Cricket;
import report.Report;

public class Main {
    public static void main(String[] args) {
        // Create instances of Student and Sports
        Student student = new Student("John Doe", 101);
        Sports sports = new Cricket();

        // Generate the report
        Report report = new Report();
        report.generateReport(student, sports);
    }
}
```

Explanation:

- **Student class**: Contains the student's name and roll number, along with getters.
- **Sports interface**: Contains a method getSportDetails() which will be implemented by classes that represent specific sports (e.g., Cricket).
- **Cricket class**: Implements the Sports interface and provides details about the sport.
- **Report class**: Generates a report combining the student information and sport details.

Output:

```
Student Report:
```

Name: John Doe

Roll Number: 101

Sport: Cricket, Country: India

New Paper

Q1. How do bitwise operators differ from relational operators in Java?

Key Differences:

Aspect	Bitwise Operators	Relational Operators
Purpose	Operate on individual bits of numbers.	Compare two values to determine their relationship.
Return Type	Returns a numeric result (integer).	Returns a boolean result (`true` or `false`).
Operands	Operands must be integers or integral types (`int`, `long`, etc.).	Operands can be any data types that can be compared (numbers, strings, etc.).
Common Use Cases	Bit manipulation, low-level programming, performance optimization.	Conditional statements, loops, decision making.
Examples	`&`, `	`, `^`, `<<`, `>>`, `~` `
Behavior	Operates on bits (0s and 1s) of numbers.	Compares values (e.g., greater, less, equal).

```
public class Main {
    public static void main(String[] args) {
        int a = 5, b = 3;

        // Bitwise operation:
        int resultBitwise = a & b;  // Binary AND (0101 & 0011 = 0001), result = 1
        System.out.println("Bitwise AND result: " + resultBitwise);

        // Relational operation:
        boolean resultRelational = a > b;  // Checks if a is greater than b, result = true
        System.out.println("Is a greater than b? " + resultRelational);
    }
}
```

Output:

Bitwise AND result: 1

Is a greater than b? true

Q2. Explain the purpose of access specifiers in Java, citing proper programs. **OR**

Design a program to elaborate the visibility of class and their members for different

access specifier.

Ans=>**Purpose of Access Specifiers in Java**

Access specifiers in Java are used to control the visibility and accessibility of classes, methods, variables, and constructors. They define how the members of a class can be accessed from other parts of the program. The primary purpose of access specifiers is to implement **encapsulation**, which is one of the core principles of Object-Oriented Programming (OOP). By using access specifiers, you can restrict access to certain parts of your code, ensuring better security and code maintenance.

Types of Access Specifiers in Java

1. **Public (public)**:
    - Members (variables, methods, classes) declared as public are accessible from any other class, inside or outside the package.
2. **Private (private)**:
    - Members declared as private are accessible only within the same class. They cannot be accessed from outside the class, even if they are in the same package or subclass.
3. **Protected (protected)**:
    - Members declared as protected are accessible within the same package and by subclasses (including subclasses in different packages).
4. **Default (no specifier)**:
    - If no access specifier is mentioned, it is known as "package-private" or default access. Members are accessible only within the same package, but not from classes in other packages.

Examples of Access Specifiers

1. Public Access Specifier

```
class PublicExample {
   public int publicVariable = 10;  // Accessible from anywhere


   public void publicMethod() {
      System.out.println("This is a public method.");
   }
}
public class TestPublic {
   public static void main(String[] args) {
```

```
        PublicExample obj = new PublicExample();
        System.out.println(obj.publicVariable);  // Accessing public variable
        obj.publicMethod();  // Calling public method
    }
}
```

Explanation:

- The publicVariable and publicMethod() can be accessed from any other class, including the TestPublic class in the same or different package.

2. Private Access Specifier

```
class PrivateExample {
    private int privateVariable = 20;  // Accessible only within this class
    private void privateMethod() {
        System.out.println("This is a private method.");
    }
    public void accessPrivateMethod() {
        System.out.println(privateVariable);  // Accessing private variable within the same class
        privateMethod();  // Calling private method within the same class
    }
}


public class TestPrivate {
    public static void main(String[] args) {
        PrivateExample obj = new PrivateExample();
        // The following lines will cause compile-time error:
        // System.out.println(obj.privateVariable);
        // obj.privateMethod();
        obj.accessPrivateMethod();  // Access private members through a public method
    }
}
```

Explanation:

- privateVariable and privateMethod() are accessible only within the PrivateExample class. They cannot be accessed directly from outside the class.

- However, a public method (accessPrivateMethod()) allows controlled access to the private members.

3. Protected Access Specifier

```
class ProtectedExample {
    protected int protectedVariable = 30;  // Accessible within same package and subclasses

    protected void protectedMethod() {
        System.out.println("This is a protected method.");
    }
}
public class SubClassExample extends ProtectedExample {
    public void accessProtectedMembers() {
        System.out.println(protectedVariable);  // Accessing protected variable in subclass
        protectedMethod();  // Calling protected method in subclass
    }
}

public class TestProtected {
    public static void main(String[] args) {
        SubClassExample obj = new SubClassExample();
        obj.accessProtectedMembers();  // Accessing protected members through subclass
    }
}
```

Explanation:

- The protectedVariable and protectedMethod() can be accessed within the same package and in subclasses (even if the subclass is in a different package).

4. Default (Package-Private) Access Specifier

```
class DefaultExample {
    int defaultVariable = 40;  // Accessible only within the same package

    void defaultMethod() {
        System.out.println("This is a default method.");
```

```
    }
}


public class TestDefault {
    public static void main(String[] args) {
        DefaultExample obj = new DefaultExample();
        System.out.println(obj.defaultVariable);  // Accessible because it's in the same package
        obj.defaultMethod();  // Accessible because it's in the same package
    }
}
```

Q3. Combine the principles of abstraction and inheritance to illustrate how Java supports the creation of complex software systems with proper program.

Ans=>**Combining Abstraction and Inheritance in Java**

In Java, **abstraction** and **inheritance** are two key principles of Object-Oriented Programming (OOP) that help create complex software systems. By combining these principles, you can design flexible, scalable, and maintainable systems.

- **Abstraction** allows you to hide complex implementation details and expose only the essential features of an object. This is done using abstract classes or interfaces.
- **Inheritance** allows one class to inherit the properties and behaviors of another, enabling code reuse and building on top of existing functionality.

```
// Abstract class defining the blueprint for all vehicles
abstract class Vehicle {
    // Common properties
    String make;
    String model;

    // Constructor to initialize the vehicle
    public Vehicle(String make, String model) {
        this.make = make;
        this.model = model;
    }
    // Abstract method (does not have a body)
    public abstract void startEngine();  // Different vehicles will implement this differently
```

```
    // Concrete method (has a body)
    public void displayInfo() {
        System.out.println("Vehicle Make: " + make);
        System.out.println("Vehicle Model: " + model);
    }
}
// Car class inherits from Vehicle and provides specific implementation
class Car extends Vehicle {
    public Car(String make, String model) {
        super(make, model);  // Call the constructor of the superclass (Vehicle)
    }
    // Implement the abstract method to start the engine for a car
    public void startEngine() {
        System.out.println("The car's engine starts with a roar.");
    }
}



// Bike class inherits from Vehicle and provides specific implementation
class Bike extends Vehicle {
    public Bike(String make, String model) {
        super(make, model);  // Call the constructor of the superclass (Vehicle)
    }
    // Implement the abstract method to start the engine for a bike
    public void startEngine() {
        System.out.println("The bike's engine starts with a smooth sound.");
    }
}
// Main class to test the implementation
public class VehicleTest {
```

```
    public static void main(String[] args) {
        // Create objects of Car and Bike
        Vehicle car = new Car("Toyota", "Camry");
        Vehicle bike = new Bike("Yamaha", "YZF-R1");
        // Display vehicle information
        car.displayInfo();
        car.startEngine();  // Calling the specific startEngine method of the Car class
        System.out.println();
        bike.displayInfo();
        bike.startEngine();  // Calling the specific startEngine method of the Bike class
    }
}
```

note for another purpose: Celsius=5/9 x (Fahrenheit - 32)

Q4. How does Java's String class facilitate efficient string handling through its various constructors, length attribute, special operations, character extraction methods, comparison mechanisms, search functionalities, and modification capabilities? **This question is covering all string chapter.**

Ans=>Java's String class provides various features to handle strings efficiently. It offers constructors, attributes, methods for searching, comparing, modifying, and converting strings, as well as for extracting characters and changing cases. Here's a comprehensive breakdown with code snippets demonstrating these features:

1. Constructors

The String class provides several constructors to create string objects:

- **Default constructor**: Creates an empty string.
- **Character array constructor**: Creates a string from a character array.
- **String constructor with specific characters**: Initializes a string with a substring of another string.

```
// Default constructor
String str1 = new String();
// String from a character array
char[] chars = {'J', 'a', 'v', 'a'};
String str2 = new String(chars);
// String from a substring of another string
```

String str3 = new String("Hello, World!".substring(0, 5)); // "Hello"

2. Length Attribute

The length() method returns the number of characters in the string.

String str = "Java";

int length = str.length();

System.out.println("Length of the string: " + length); // Output: 4

3. Special Operations

- **String concatenation**: Use + operator or concat() method to join strings.
- **String equality**: Use equals() to compare contents, and == for reference comparison.

// Concatenation

String str1 = "Hello";

String str2 = "World";

String result = str1 + " " + str2;

Or // String result = str1.concat(str2);

System.out.println(result); // Output: Hello World

// Equality check

String s1 = "Java";

String s2 = "java";

System.out.println(s1.equals(s2)); // Output: false (case-sensitive)

System.out.println(s1.equalsIgnoreCase(s2)); // Output: true (case-insensitive)

4. Character Extraction Methods

- **charAt(int index)**: Returns the character at the specified index.
- **getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)**: Copies characters from the string into a character array.

String str = "Java";

char c = str.charAt(2);

System.out.println(c); // Output: v

// Extracting characters into an array

char[] arr = new char[3];

str.getChars(0, 3, arr, 0);

System.out.println(arr); // Output: Jav

5. Comparison Mechanisms

- **compareTo()**: Compares two strings lexicographically.
- **compareToIgnoreCase()**: Compares strings lexicographically ignoring case.

```
String s1 = "Java";
String s2 = "JavaScript";
int result = s1.compareTo(s2);  // Result is negative as "Java" is lexicographically less than "JavaScript"
System.out.println(result);  // Output: -3
String s3 = "java";
System.out.println(s1.compareToIgnoreCase(s3));  // Output: 0 (case-insensitive comparison)
```

6. Search Functionalities

- **indexOf()**: Finds the index of a character or substring.
- **lastIndexOf()**: Finds the last occurrence of a character or substring.
- **contains()**: Checks if a string contains a specific sequence of characters.

```
String str = "Hello, World!";
System.out.println(str.indexOf("World"));  // Output: 7 (index of "World")
System.out.println(str.lastIndexOf('o'));  // Output: 8 (last 'o' in "Hello, World!")
System.out.println(str.contains("World"));  // Output: true
```

7. Modification Capabilities

- **replace()**: Replaces characters or substrings.
- **substring()**: Extracts a substring.
- **trim()**: Removes leading and trailing whitespace.
- **toLowerCase() / toUpperCase()**: Changes the case of characters.
- **concat()**: Concatenates two strings.

```
String str = "  Java Programming  ";
System.out.println(str.trim());  // Output: Java Programming

// Replace characters
String replaced = str.replace("Java", "Python");
System.out.println(replaced);  // Output: Python Programming

// Substring extraction
```

```
String sub = str.substring(2, 6);
System.out.println(sub);  // Output: va

// Change case
String lower = str.toLowerCase();
System.out.println(lower);  // Output: "  java programming  "
String upper = str.toUpperCase();
System.out.println(upper);  // Output: "  JAVA PROGRAMMING  "
```

8. Data Conversion

- **valueOf()**: Converts various types (e.g., int, boolean, etc.) to strings.
- **toString()**: Converts an object to its string representation.

```
int num = 123;
String numStr = String.valueOf(num);
System.out.println(numStr);  // Output: "123"
Object obj = new Object();
System.out.println(obj.toString());  // Output: String representation of the object
```

9. Changing the Case of Characters

- **toLowerCase()**: Converts all characters to lowercase.
- **toUpperCase()**: Converts all characters to uppercase.

```
String text = "Java Programming";
String lowerText = text.toLowerCase();
System.out.println(lowerText);  // Output: java programming
String upperText = text.toUpperCase();
System.out.println(upperText);  // Output: JAVA PROGRAMMING
```

New paper

Q1.How arguments passed from the console can be received in the Java progranl and it can be used as an input? Give example.

Ans=>In Java, arguments passed from the console (command line) can be received in the program using the main method's parameter String[] args. This array contains the command-line arguments passed to the program when it's executed.

```
public class CommandLineArgsExample {
```

```
    public static void main(String[] args) {
        // Check if arguments are passed
        if (args.length == 0) {
            System.out.println("No arguments passed.");
        } else {
            System.out.println("Arguments passed:");
            // Loop through the arguments and print them
            for (int i = 0; i < args.length; i++) {
                System.out.println("Argument " + (i + 1) + ": " + args[i]);
            }
        }
    }
}
```

steps to Run:

1. Save the code in a file named CommandLineArgsExample.java.
2. Compile the Java program:

   javac CommandLineArgsExample.java
3. Run the compiled program and pass arguments:

   java CommandLineArgsExample Hello World 123

Output:

Arguments passed:

Argument 1: Hello

Argument 2: World

Argument 3: 123

Q2. a) Compare objects with variables in terms of passing them as parameters. (4)

b) Explain constructors in derived class with the help of a program. (8)

ans=>

a) Comparison of Objects with Variables in Terms of Passing Them as Parameters (4 marks)

In Java, when you pass **objects** and **variables** as parameters to methods, there are key differences in how they are passed:

1. **Passing Variables (Primitive types):**

- When you pass a variable of a **primitive data type** (e.g., int, float, char), Java passes the **value** of the variable to the method.
- Changes made to the variable within the method do not affect the original variable outside the method.
- This is known as **pass-by-value**.

Example:

```
public class Test {
   public static void modifyValue(int x) {
      x = x + 10;
   }

   public static void main(String[] args) {
      int num = 5;
      modifyValue(num);
      System.out.println(num); // Outputs: 5 (original value remains unchanged)
   }
}
```

2. **Passing Objects:**
   - When you pass an **object** as a parameter, Java passes the **reference** to the object, not the actual object itself.
   - If the object is modified inside the method, the changes affect the original object outside the method.
   - However, reassigning the object inside the method (changing the reference) will not affect the original reference outside the method.
   - This is also known as **pass-by-reference** (but technically, it's still pass-by-value, as the value passed is the reference, not the actual object).

Example:

```
public class Test {
   static class Person {
      String name;
   }

   public static void modifyObject(Person p) {
```

```
        p.name = "John";
    }

    public static void main(String[] args) {
        Person person = new Person();
        person.name = "Alice";
        modifyObject(person);
        System.out.println(person.name); // Outputs: John (original object is modified)
    }
}
```

b) Explanation of Constructors in Derived Class with the Help of a Program (8 marks)

In Java, constructors in a derived (child) class are used to initialize the object of the child class. A constructor in a derived class can call the constructor of its base (parent) class using the super() keyword, which allows initialization of inherited fields before the child class adds its specific fields.

Key Points:

- If a constructor is not explicitly defined in a derived class, the default constructor of the parent class (if available) is called.
- A constructor in the child class can explicitly call a constructor of the parent class using super() to initialize the parent class's fields.

Example Program:

```
class Animal {
    String name;
    // Constructor of the parent class
    public Animal(String name) {
        this.name = name;
        System.out.println("Animal constructor called");
    }

    public void makeSound() {
        System.out.println("Animal makes a sound");
    }
}
```

```
class Dog extends Animal {
    int age;
    // Constructor of the child class, calling the parent class constructor using super()
    public Dog(String name, int age) {
        super(name);  // Call the parent class constructor
        this.age = age;
        System.out.println("Dog constructor called");
    }
    public void makeSound() {
        System.out.println("Bark! Bark!");
    }
    public void displayInfo() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
public class Main {
    public static void main(String[] args) {
        // Create an object of the Dog class
        Dog dog = new Dog("Buddy", 3);
        // Display information about the dog
        dog.displayInfo();
        // Call the method of the derived class
        dog.makeSound();
    }
}
```

Output:

```
Animal constructor called
Dog constructor called
Name: Buddy, Age: 3
Bark! Bark!
```

New paper

Q1. Write a program to display current data and time.

```
import java.time.*;

public class CurrentDateTime {

   public static void main(String[] args) {

      // Display the current date and time

      System.out.println("Current Date and Time: " + LocalDateTime.now());

   }

}
```

Q2. Explain the role of the Just-In Time (JIT) compiler.

Ans=>**Role of the Just-In-Time (JIT) Compiler:**

The **Just-In-Time (JIT) compiler** is a key component of the JVM that improves the performance of Java applications during runtime.

- **What JIT Does**:
    - The JIT compiler translates bytecode into **native machine code** for the host machine, which is directly executed by the CPU.
    - This translation happens **at runtime** when a method or class is invoked, instead of compiling the entire program upfront.
- **How JIT Works**:

1. The JVM initially interprets the bytecode (line-by-line) to run the program.
2. During execution, the JVM identifies **hot spots** (frequently executed methods or loops).
3. The JIT compiler then compiles these hot spots into native machine code, which is stored for future use, speeding up execution.
4. The next time the same bytecode is encountered, the JVM uses the compiled machine code rather than interpreting the bytecode.

Q3. Explain the concept of variable scope in Java and how it affects program behavior. Provide examples of local, instance, and class variables.

Ans=>**Variable Scope in Java**

In Java, **variable scope** refers to the region of the program where a variable is accessible or valid. The scope of a variable determines where it can be referenced or modified within the program. The scope of a variable depends on where it is declared, and Java defines several types of scopes:

1. **Local Variables**
2. **Instance Variables**
3. **Class (Static) Variables**

1. Local Variables

Local variables are declared inside methods, constructors, or blocks and can only be accessed within the method, constructor, or block where they are defined. They are created when the method is called and destroyed when the method execution is completed.

- **Scope**: The scope of a local variable is limited to the method, constructor, or block in which it is defined.

2. Instance Variables

Instance variables are declared inside a class but outside any method, constructor, or block. They belong to an instance of the class (i.e., each object of the class has its own copy of the instance variables).

- **Scope**: The scope of an instance variable is the entire class (except within static methods if they are not referenced via an object).

3. Class (Static) Variables

Class variables (also known as **static variables**) are declared with the static keyword inside a class but outside any method or constructor. These variables are shared by all instances of the class.

- **Scope**: The scope of a class variable is the entire class. It can be accessed by both static and non-static methods.

```
public class VariableScopeExample {
    private int instanceVar = 5;  // Instance variable
    private static int staticVar = 10;  // Class (static) variable

    public void method() {
        int localVar = 20;  // Local variable
        System.out.println("Instance variable: " + instanceVar);
        System.out.println("Class variable: " + staticVar);
        System.out.println("Local variable: " + localVar);
    }

    public static void main(String[] args) {
        VariableScopeExample example = new VariableScopeExample();
        example.method();

        // System.out.println(localVar); // Error: localVar is not in scope
    }
}
```

Output:

Instance variable: 5

Class variable: 10

Local variable: 20

Q4. Describe the garbage collection mechanism in Java. Explain how it works, including the different types of garbage collectors available in the JVM. Discuss the role of the finalize() method and why relying on it for resource cleanup is discouraged. Provide examples to illustrate proper resource management techniques in Java.

Ans=>**Garbage Collection in Java**

Garbage collection (GC) in Java is an automatic process of reclaiming memory that is no longer in use by objects, preventing memory leaks and ensuring efficient memory management. The Java Virtual Machine (JVM) has a garbage collector that helps manage memory automatically, freeing developers from manual memory management.

How Garbage Collection Works:**Mark-and-Sweep and Reclaim**

- **Mark phase**: The garbage collector traverses the object graph starting from root references (like local variables, static fields, and active threads) and marks all reachable objects.
- **Sweep phase**: After marking, the collector sweeps through the heap and collects all objects that were not marked, i.e., objects that are no longer reachable.
- **Reclaim**: After collecting unreachable objects, the garbage collector reclaims the memory, making it available for new objects.

Types of Garbage Collectors in the JVM

1. **Serial Garbage Collector**:

- It is the simplest collector and is designed for single-threaded applications.
- It performs all garbage collection tasks using a single thread.
- Suitable for small applications with low memory consumption.

2. **Parallel Garbage Collector (Throughput Collector)**:

- It uses multiple threads for garbage collection, which improves the performance in multi-threaded environments.
- Ideal for applications that require a balance between throughput and pause times.

3. **CMS (Concurrent Mark-Sweep) Garbage Collector**:

- Designed to minimize pause times by performing most of its work concurrently with the application threads.
- Good for applications that require low pause times.

finalize() Method

- The finalize() method was introduced in Java to allow developers to define cleanup actions (like releasing resources or closing file handles) before an object is garbage collected.
- The JVM calls finalize() before destroying an object, but the timing of this call is unpredictable.
- **Disadvantages of `finalize()`**:
  - **Unpredictable Execution**: The method might not be called immediately after the object becomes unreachable, leading to delayed resource cleanup.
  - **Performance Impact**: Objects with `finalize()` are subject to additional processing, which can slow down the GC process.
  - **Lack of Guarantees**: The garbage collector may not call `finalize()` if the program exits before GC occurs, leaving resources unreleased.

For these reasons, using `finalize()` for resource management is generally discouraged.

Proper Resource Management Techniques in Java

1. Using try-with-resources

Automatically closes resources that implement the AutoCloseable interface after the try block finishes, reducing boilerplate code.

```
class Resource implements AutoCloseable {
   public void use() { System.out.println("Using resource!"); }
   public void close() { System.out.println("Resource closed!"); }
}
public class TryWithResourcesExample {
   public static void main(String[] args) {
      try (Resource resource = new Resource()) {
         resource.use();
      }
   }
}
```

Output:

```
Using resource!
Resource closed!
```

---

2. Explicit Resource Management

Manually closes resources in a finally block, ensuring proper cleanup regardless of exceptions.

```
class Resource {
    public void use() { System.out.println("Using resource!"); }
    public void close() { System.out.println("Resource closed!"); }
}
public class ExplicitResourceExample {
    public static void main(String[] args) {
        Resource resource = null;
        try {
            resource = new Resource();
            resource.use();
        } finally {
            if (resource != null) {
                resource.close();
            }
        }
    }
}
```

Output:

```
Using resource!
Resource closed!
```

Q5.Write a Java program to demonstrate inheritance and polymorphism. Create a superclass Shape with a method draw(). Create three subclasses Circle, Rectangle, and Triangle that override the draw() method. In your main method, create an array of Shape objects and call the draw() method on each object. Show how dynamic method dispatch works in this scenario.

Ans=>

```
// Superclass
class Shape {
    public void draw() {
        System.out.println("Drawing a shape");
    }
}
// Subclass 1
class Circle extends Shape {
```

```
    @Override
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}
// Subclass 2
class Rectangle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Rectangle");
    }
}
// Subclass 3
class Triangle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Triangle");
    }
}



// Main Class
public class InheritancePolymorphismExample {
    public static void main(String[] args) {
        // Create an array of Shape references
        Shape[] shapes = { new Circle(), new Rectangle(), new Triangle() };
        // Call the draw() method on each Shape object
        for (Shape shape : shapes) {
            shape.draw(); // Dynamic method dispatch
```

```
    }
  }
}
```

New paper

Q1. Explain the following methods of String class: (i) indexOf( ) (ii) substring();

Ans=> **1. indexOf() Method**

The indexOf() method returns the index (position) of the first occurrence of a specified character or substring within a string. If the character or substring is not found, it returns -1.

```
public class IndexOfExample {
  public static void main(String[] args) {
    String text = "Hello, World!";
    System.out.println(text.indexOf('W'));      // Output: 7
    System.out.println(text.indexOf("World")); // Output: 7
    System.out.println(text.indexOf('o', 5));  // Output: 8
    System.out.println(text.indexOf("Java"));  // Output: -1
  }
}
```

2. substring() Method

The substring() method extracts a portion of the string, starting from a specified index. It can also end at a specified index (exclusive).

```
public class SubstringExample {
  public static void main(String[] args) {
    String text = "Hello, World!";
    System.out.println(text.substring(7));       // Output: World!
    System.out.println(text.substring(0, 5));    // Output: Hello
    System.out.println(text.substring(7, 12));   // Output: World
  }
}
```

Q2. Write the difference between input and output stream class with example **OR** Develop a code for reading from and writing to a file.

Ans=> **Difference Between InputStream and OutputStream Classes in Java**

In Java, the **InputStream** and **OutputStream** classes are part of the java.io package, and they are used for reading from and writing to byte streams respectively. The primary difference between the two lies in their purpose: one handles input (reading), and the other handles output (writing).

---

1. InputStream Class

The InputStream class is used to read data from a source (like a file, network socket, or memory). It reads data as bytes.

- **Primary purpose**: To read byte data.
- **Common methods**:
    - int read(): Reads a byte of data.
    - int read(byte[] b): Reads bytes into a byte array.
    - void close(): Closes the stream.

Example of InputStream:

```
import java.io.*;

public class InputStreamExample {
   public static void main(String[] args) {
      try (InputStream inputStream = new FileInputStream("example.txt")) {
         int data;
         while ((data = inputStream.read()) != -1) {
            System.out.print((char) data);  // Print each byte as a character
         }
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

- **Explanation**: This program reads bytes from a file using InputStream and prints each byte as a character.

---

2. OutputStream Class

The OutputStream class is used to write data to a destination (like a file, network socket, or memory). It writes data as bytes.

- **Primary purpose**: To write byte data.

- **Common methods**:
    - void write(int b): Writes a byte of data.
    - void write(byte[] b): Writes a byte array to the stream.
    - void close(): Closes the stream.

Example of OutputStream:

```
import java.io.*;

public class OutputStreamExample {
   public static void main(String[] args) {
      try (OutputStream outputStream = new FileOutputStream("output.txt")) {
         String data = "Hello, OutputStream!";
         outputStream.write(data.getBytes());  // Write data to file as bytes
         System.out.println("Data written to file!");
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

- **Explanation**: This program writes a string to a file using OutputStream by converting the string into bytes.

Q3. Write a program r.o convert a String to a List of Characters in Java progvamming.. Also implement the try, catch and throw method to handle the exception in the program. Input: String "JavaProgramming" Output:[j,a,v,a,P,r....]

Ans=>

```
import java.util.ArrayList;

import java.util.List;

public class StringToListExample {
   public static void main(String[] args) {
      try {
         String inputString = "JavaProgramming";  // Input string
         // Call the method to convert the string to a list of characters
         List<Character> charList = convertStringToList(inputString);
```

```
            // Print the output List of characters
            System.out.println("List of characters: " + charList);
        } catch (NullPointerException e) {
            System.out.println("Error: String cannot be null.");
        } catch (Exception e) {
            System.out.println("An unexpected error occurred: " + e.getMessage());
        }
    }
    // Method to convert String to List of characters
    public static List<Character> convertStringToList(String input) throws Exception {
        // Check if input is null, throw an exception if it is
        if (input == null) {
            throw new NullPointerException("Input string is null.");
        }
        // Create a List to store characters
        List<Character> charList = new ArrayList<>();
        // Convert each character of the string to the List
        for (int i = 0; i < input.length(); i++) {
            charList.add(input.charAt(i));  // Add each character to the list
        }
        return charList;  // Return the List of characters
    }
}
```

Q4. What are different iypes of inheritance supported by java? Explain with examples. Why multiple Inheritance is not, supported by Java? jusiif'y in detail.

Ans=> **Types of Inheritance Supported by Java**

Java supports several types of inheritance, allowing classes to inherit properties and behaviors from other classes. The types of inheritance are:

1. **Single Inheritance**: In single inheritance, a class can inherit from only one superclass.

Example:

```
class Animal {
    void sound() {
```

```
    System.out.println("Animal makes a sound");
  }
}

class Dog extends Animal {
  void display() {
    System.out.println("Dog barks");
  }
}

public class Test {
  public static void main(String[] args) {
    Dog d = new Dog();
    d.sound();   // Inherited method
    d.display(); // Method of Dog class
  }
}
```

Explanation: In this example, the Dog class extends the Animal class, inheriting the sound() method from it.

2. **Multilevel Inheritance**: In multilevel inheritance, a class is derived from another class, which is itself derived from another class.

Example:

```
class Animal {
  void sound() {
    System.out.println("Animal makes a sound");
  }
}

class Dog extends Animal {
  void bark() {
    System.out.println("Dog barks");
  }
```

```
}

class Puppy extends Dog {
   void play() {
      System.out.println("Puppy plays");
   }
}

public class Test {
   public static void main(String[] args) {
      Puppy p = new Puppy();
      p.sound(); // Inherited from Animal
      p.bark();  // Inherited from Dog
      p.play();  // Defined in Puppy
   }
}
```

Explanation: The Puppy class extends Dog, and Dog extends Animal. Thus, Puppy indirectly inherits from Animal.

3. **Hierarchical Inheritance**: In hierarchical inheritance, multiple classes can inherit from a single superclass.

Example:

```
class Animal {
   void sound() {
      System.out.println("Animal makes a sound");
   }
}

class Dog extends Animal {
   void bark() {
      System.out.println("Dog barks");
   }
}
```

```
class Cat extends Animal {
   void meow() {
      System.out.println("Cat meows");
   }
}


public class Test {
   public static void main(String[] args) {
      Dog d = new Dog();
      d.sound(); // Inherited from Animal
      d.bark();  // Method of Dog class


      Cat c = new Cat();
      c.sound(); // Inherited from Animal
      c.meow();  // Method of Cat class
   }
}
```

Explanation: Both Dog and Cat classes inherit from Animal. Thus, they both inherit the sound() method.

4. **Hybrid Inheritance**: Hybrid inheritance is a combination of two or more types of inheritance, such as single, multilevel, and hierarchical inheritance. Java does not directly support hybrid inheritance because it can lead to ambiguity.

Example:

```
class Animal {
   void sound() {
      System.out.println("Animal makes a sound");
   }
}
```

```
class Dog extends Animal {
   void bark() {
      System.out.println("Dog barks");
   }
}

class Cat extends Animal {
   void meow() {
      System.out.println("Cat meows");
   }
}

class Pet extends Dog, Cat { // Invalid in Java
   void play() {
      System.out.println("Pet plays");
   }
}
```

Explanation: In the above example, the Pet class is trying to inherit from both Dog and Cat, which is not allowed in Java.

Why Multiple Inheritance is Not Supported by Java

Multiple inheritance is not supported in Java for the following reasons:

1. **Diamond Problem**: In multiple inheritance, a class can inherit from more than one class, which may lead to ambiguity. The most common issue is the **diamond problem**, where a subclass inherits from two classes that have a method with the same signature. The subclass doesn't know which superclass method to use, leading to confusion and potential bugs.

Example:

```
class A {
   void show() {
      System.out.println("A's show");
   }
}
```

```
class B extends A {
   void show() {
      System.out.println("B's show");
   }
}

class C extends A {
   void show() {
      System.out.println("C's show");
   }
}

class D extends B, C {  // Not allowed in Java
   void display() {
      System.out.println("D's display");
   }
}
```

Explanation: In the above code, if class D extends both B and C, it would inherit two show() methods, one from B and one from C, causing ambiguity. Java does not allow this to avoid such problems.

Q5. Explain Static nested Classes? What is the difference between an Inner class and a Sub-Class?

Ans=> **Static Nested Classes**

A **static nested class** is a class defined within another class, marked with the static keyword. Static nested classes do not have access to the instance variables and methods of the outer class. They can only access the static members (variables and methods) of the outer class.

Key Characteristics of Static Nested Classes:

1. **No reference to the outer class's instance**: Unlike inner (non-static) classes, a static nested class does not require an instance of the outer class to be created.
2. **Access to static members**: It can access static fields and methods of the outer class.
3. **Instance creation**: A static nested class can be instantiated without the need for an outer class instance. It can be created using the outer class name, like OuterClass.NestedClass.

Example of Static Nested Class:

```
class OuterClass {
    static int outerStaticVar = 10;

    static class StaticNestedClass {
        void display() {
            // Accessing static variable of outer class
            System.out.println("Static variable from outer class: " + outerStaticVar);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        // Creating an instance of the static nested class
        OuterClass.StaticNestedClass nestedObj = new OuterClass.StaticNestedClass();
        nestedObj.display();
    }
}
```

Difference between an Inner Class and a Sub-Class:

1. **Inner Class**:
    - An **Inner Class** is a class defined inside another class, which can be either static or non-static.
    - **Non-static Inner Classes** have a reference to an instance of the outer class and can access both static and non-static members of the outer class.
    - **Access**: An inner class has access to the instance variables and methods of the outer class.
    - **Instantiating**: To create an instance of a non-static inner class, you must create an instance of the outer class first.

Example of an Inner Class:

```
class OuterClass {
   int instanceVar = 20;
   class InnerClass {
void display() {
        System.out.println("Inner class can access instance member of outer class: " + instanceVar);
      }
   }
}
public class Main {
   public static void main(String[] args) {
      OuterClass outer = new OuterClass();
      OuterClass.InnerClass inner = outer.new InnerClass();
      inner.display();
   }
}
```

2. **Sub-Class**:
    - A **Sub-Class** is a class that is derived from a parent (superclass) using inheritance. It extends the behavior and properties of the parent class.
    - A sub-class can override the methods and inherit the instance variables of the parent class, enabling polymorphism.
    - **Access**: A sub-class has access to the public and protected members of the parent class, but not its private members unless they are accessed via public/protected methods.
    - **Instantiating**: A subclass can be instantiated independently, but it inherits the members of the parent class.

Example of a Sub-Class:

```
class ParentClass {
   void display() {
      System.out.println("This is a method in the parent class.");
   }
}
```

```
class SubClass extends ParentClass {
    @Override
    void display() {
        System.out.println("This is the overridden method in the sub-class.");
    }
}
public class Main {
    public static void main(String[] args) {
        SubClass sub = new SubClass();
        sub.display();
    }
}
```