SECTION A

1. **Define polymorphism.and types in 80 word and small piece of code**

Polymorphism in Java

- **Definition**: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It provides a way to perform a single action in different forms.
- **Types**:
  - **Compile-time (Static) Polymorphism**: Achieved through method overloading.
  - **Run-time (Dynamic) Polymorphism**: Achieved through method overriding.

Code Example

```
java

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

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

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog(); // Upcasting
        myDog.sound(); // Calls the overridden method
    }
}
```

- **Output**: "Dog barks"

4o

2. What do you mean by overloaded constructor?80 word and code small

Overloaded Constructor in Java

- **Definition**: Overloaded constructors are multiple constructors in a class with the same name but different parameters. They allow creating objects in different ways, providing flexibility.

Code Example

```
java

class Box {
    double width, height, depth;

    // Constructor with no parameters
    Box() {
        width = height = depth = 0;
    }
```

```
    // Constructor with three parameters
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
}

public class Main {
    public static void main(String[] args) {
        Box box1 = new Box(); // Calls no-arg constructor
        Box box2 = new Box(10, 20, 30); // Calls parameterized constructor
    }
}
```

- **Usage**: Allows creating `Box` objects with or without initial dimensions.

4o

3. How multiple inheritances in implemented in java?code also in 80 word

Multiple Inheritance in Java

- **Implementation**: Java does not support multiple inheritance through classes to avoid ambiguity (diamond problem). However, it supports multiple inheritance through interfaces, where a class can implement multiple interfaces.

Code Example

```
java

interface A {
    void methodA();
}

interface B {
    void methodB();
}

class MyClass implements A, B {
    @Override
    public void methodA() {
        // Implementation of methodA
    }

    @Override
    public void methodB() {
        // Implementation of methodB
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.methodA();
        obj.methodB();
    }
}
```

- **Usage**: `MyClass` implements both interface `A` and `B`, allowing it to inherit behaviors from multiple sources.

4. Define exception handling.in 50 word small piece of code
3/3

Exception Handling in Java

- **Definition**: Exception handling is a mechanism to handle runtime errors, such as divide-by-zero, file not found, etc., gracefully to prevent program termination.

Code Example

```
java

public class Main {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; // Division by zero
        } catch (ArithmeticException e) {
            System.out.println("Error: Division by zero");
        }
    }
}
```

- **Explanation**: The `try` block contains code that might throw an exception. If an exception occurs (here, `ArithmeticException`), it is caught and handled in the `catch` block, preventing the program from crashing.

5. What do you mean by re-throwing of exception?80 word limit small piece of code
3/3

Rethrowing Exceptions in Java

- **Definition**: Rethrowing an exception allows catching an exception in one method and throwing it again for another method or higher-level handler to handle.

Code Example

```
java

public class Main {
    public static void main(String[] args) {
        try {
            method1();
        } catch (Exception e) {
            System.out.println("Exception caught in main: " +
e.getMessage());
        }
    }

    static void method1() throws Exception {
        try {
            method2();
        } catch (Exception e) {
            System.out.println("Exception caught in method1: " +
e.getMessage());
            throw e; // Rethrow the exception
        }
    }

    static void method2() throws Exception {
```

```
        // Simulate an exception
        throw new Exception("Exception in method2");
    }
}
```

- **Explanation**: In `method1`, an exception is caught, handled, and then rethrown using `throw e;`. This allows the exception to be propagated to the calling method (`main`), where it can be further handled or reported.

6. Mention the feature of java. 80 word limit

Features of Java

- **Object-Oriented**: Java supports the principles of encapsulation, inheritance, polymorphism, and abstraction.
- **Platform-Independent**: Java programs can run on any platform with the Java Virtual Machine (JVM), making it highly portable.
- **Simple and Familiar**: Syntax is derived from C/C++ to ease adoption for developers.
- **Robust and Secure**: Built-in memory management, exception handling, and security features ensure reliable code execution.
- **Multithreaded**: Java supports concurrent programming, allowing multiple tasks to run simultaneously for improved performance.
- **Dynamic**: Java applications can adapt dynamically by loading classes on demand and supporting dynamic memory allocation through garbage collection.

7. What is java token? in 80 word limit

Java Token

- **Definition**: In Java, a token is the smallest individual unit in a program that is meaningful to the compiler. Tokens include keywords, identifiers, constants, operators, separators, and literals.
- **Example**: In `int x = 10;`, the tokens are `int`, `x`, `=`, `10`, and `;`. Each token serves a specific role in the syntax and semantics of the Java programming language.

8. Difference between an interface and a class.80 word limit

Difference Between Interface and Class

- **Definition**:
  - **Class**: Represents a blueprint for objects, defining attributes (fields) and behaviors (methods).
  - **Interface**: Defines a contract specifying methods that a class must implement without specifying the implementation details.
- **Inheritance**:
  - **Class**: Supports single inheritance, meaning a class can extend only one superclass.
  - **Interface**: Supports multiple inheritance, allowing a class to implement multiple interfaces.

- **Implementation**:
  - **Class**: Can contain fields, constructors, and both concrete and abstract methods.
  - **Interface**: Only contains constant fields and abstract method declarations.
- **Example**:
  - A `Car` class may have methods like `start()` and `stop()`.
  - An `Animal` interface may declare methods like `eat()` and `sleep()`, which different animals can implement differently.

9. What is an exception? 80 word limit with code

Exception in Java

- **Definition**: An exception is an event that disrupts the normal flow of a program during execution, often due to errors or unexpected conditions.

Code Example

```
java

public class Main {
    public static void main(String[] args) {
        try {
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[5]); // Accessing an index out of
bounds
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

- **Explanation**: In the example, trying to access `numbers[5]` throws an `ArrayIndexOutOfBoundsException`, which is caught and handled in the `catch` block to prevent program termination.

10. What is oops? 80 word limit

Object-Oriented Programming (OOP)

- **Definition**: OOP is a programming paradigm that uses objects and classes to design and implement applications. It emphasizes concepts like encapsulation, inheritance, polymorphism, and abstraction to structure code for reusability, maintainability, and modularity.
- **Key Concepts**: Classes define blueprints for objects, encapsulating data (attributes) and behaviors (methods). Inheritance allows classes to inherit properties and behaviors from other classes. Polymorphism enables objects to be treated as instances of their superclass.
- **Usage**: OOP promotes modular, scalable software design, facilitating code organization and flexibility in complex applications.

SECTION B

11. Describe the following concept : a) Platform independence and cross platform b) Role of inheritance in java

Concepts in Java

a) Platform Independence and Cross-Platform

- **Platform Independence**: Java programs are compiled into bytecode, which can run on any device with a Java Virtual Machine (JVM), making Java platform-independent.
- **Cross-Platform**: Applications developed in Java can run on different operating systems (Windows, macOS, Linux) without modification, thanks to the JVM's ability to interpret bytecode.

b) Role of Inheritance in Java

- **Inheritance**: Allows a class (subclass) to inherit properties and behaviors from another class (superclass). It promotes code reusability, enabling subclasses to extend and specialize the functionality of their superclass.
- **Benefits**: Reduces code duplication, enhances software design by organizing classes into hierarchies, and facilitates polymorphism, where objects of different classes can be treated as instances of their superclass.

12. Describe the JDK, JRE and JVM with a suitable diagram.in 120 words

JDK, JRE, and JVM in Java

- **Java Development Kit (JDK)**: Includes tools necessary for developing and compiling Java applications, such as `javac` (compiler), `java` (interpreter), and `jar` (archiver). It includes JRE and development tools.
- **Java Runtime Environment (JRE)**: Provides the runtime environment for executing Java applications. It includes the JVM and core libraries (`rt.jar`) required for running Java programs.
- **Java Virtual Machine (JVM)**: Executes Java bytecode, serving as an abstract machine that translates bytecode into machine-specific code. It manages memory, provides runtime environment, and facilitates platform independence.

Diagram:

```
scss

+----------------+        +----------------+        +-----------------+
|   Source Code  |  ->    |   Bytecode     |  ->    |   Machine Code  |
|  (.java files) |  javac |  (.class files)|  JVM   |   (Platform-Specific)
+----------------+        +----------------+        +-----------------+
              |                  |                       |
          JDK Tools           JRE Libraries           Hardware
```

- **Explanation**: JDK compiles source code to bytecode, which is interpreted by the JVM to execute on different hardware platforms using the JRE's libraries.

13. Discuss the structure and function for statement in java with example.120

Structure and Function of the `for` Statement in Java

Structure:

The `for` statement in Java allows iterative execution of a block of code based on a condition. It consists of three parts:

1. **Initialization**: Executed only once at the beginning, initializing the loop control variable.
2. **Condition**: Checked before each iteration. If true, the loop continues; if false, the loop terminates.
3. **Update**: Executed after each iteration, typically incrementing or decrementing the loop control variable.

Example:

```
java

public class ForLoopExample {
    public static void main(String[] args) {
        // Example 1: Simple loop to print numbers 1 to 5
        System.out.println("Example 1:");
        for (int i = 1; i <= 5; i++) {
            System.out.println(i);
        }

        // Example 2: Loop to iterate over elements in an array
        System.out.println("\nExample 2:");
        int[] numbers = {1, 2, 3, 4, 5};
        for (int num : numbers) {
            System.out.println(num);
        }

        // Example 3: Nested loop for generating a pattern
        System.out.println("\nExample 3:");
        for (int row = 1; row <= 5; row++) {
            for (int col = 1; col <= row; col++) {
                System.out.print("* ");
            }
            System.out.println();
        }
    }
}
```

Function:

- **Initialization**: `int i = 1;` initializes `i` to 1 before the loop starts.
- **Condition**: `i <= 5;` checks if `i` is less than or equal to 5 before each iteration.
- **Update**: `i++` increments `i` by 1 after each iteration.

Usage:

- The `for` statement efficiently manages loop control, making it suitable for iterating over arrays, generating patterns, and performing repetitive tasks until a specified condition is met.

14. Describe in detail the concept used to implement multiple inheritance in java with a suitable example.

Implementing Multiple Inheritance in Java

Java does not support multiple inheritance through classes to avoid the diamond problem, where ambiguity can arise if multiple superclasses have methods or fields with the same name. However, Java supports multiple inheritance through interfaces, allowing a class to implement multiple interfaces.

Concept:

- **Interface**: An interface in Java defines a contract that classes can implement. It only contains method signatures (without method bodies) and constants (static final fields).
- **Class Implementation**: A class can implement multiple interfaces, inheriting the method signatures from each interface. The class must provide implementations for all methods declared in the interfaces it implements.

Example:

```
java

// Interface 1
interface A {
    void methodA();
}

// Interface 2
interface B {
    void methodB();
}

// Class implementing both interfaces
class MyClass implements A, B {
    @Override
    public void methodA() {
        System.out.println("Implementation of methodA");
    }

    @Override
    public void methodB() {
        System.out.println("Implementation of methodB");
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.methodA();  // Output: Implementation of methodA
        obj.methodB();  // Output: Implementation of methodB
    }
}
```

Explanation:

1. **Interfaces**: `A` and `B` declare methods `methodA()` and `methodB()` respectively, which `MyClass` implements.
2. **Implementation**: `MyClass` provides concrete implementations for `methodA()` and `methodB()`. By implementing both interfaces, `MyClass` effectively inherits behaviors from both `A` and `B`.
3. **Usage**: Through interfaces, Java achieves a form of multiple inheritance, allowing classes to inherit behaviors from multiple sources while avoiding the complexities and

ambiguities of multiple inheritance with classes. This approach promotes code reusability and design flexibility in Java programs.

15. Write java program to add two times objects using operator overloading.120 words wirh code

In Java, operator overloading, as commonly understood in languages like C++, is not directly supported. However, you can achieve a similar effect using method overloading and designing your classes to handle addition operations in a meaningful way. Here's an example where we create a `Time` class to represent time and overload the `+` operator (in a method named `add`) to add two `Time` objects:

```
java

class Time {
    private int hours;
    private int minutes;

    public Time(int hours, int minutes) {
        this.hours = hours;
        this.minutes = minutes;
    }

    public Time add(Time t) {
        int totalMinutes = this.hours * 60 + this.minutes + t.hours * 60 +
t.minutes;
        int h = totalMinutes / 60;
        int m = totalMinutes % 60;
        return new Time(h, m);
    }

    public void display() {
        System.out.println("Time: " + hours + " hours and " + minutes + "
minutes");
    }

    public static void main(String[] args) {
        Time time1 = new Time(2, 45);
        Time time2 = new Time(1, 30);

        Time sum = time1.add(time2);

        System.out.print("Sum of ");
        time1.display();
        System.out.print("and ");
        time2.display();
        System.out.print("is ");
        sum.display();
    }
}
```

Explanation:

- The `Time` class encapsulates hours and minutes as private fields and provides a constructor to initialize them.
- The `add` method calculates the sum of two `Time` objects (`this` and `t`) and returns a new `Time` object representing the sum.
- In the `main` method, we create two `Time` objects (`time1` and `time2`), add them using the `add` method, and display the result.

While Java does not support traditional operator overloading, this approach mimics its functionality by providing a method (`add`) that performs addition according to the logic defined in the `Time` class.

3/3

16. Differentiate between method overloading and method overriding.120 words in table form

Here's a comparison between method overloading and method overriding in Java presented in a table format:

Feature	Method Overloading	Method Overriding
Definition	Multiple methods with the same name but different parameters within the same class.	Redefining a method in the subclass that has the same signature (name, parameters, and return type) as a method in its superclass.
Inheritance Requirement	Occurs within the same class.	Requires inheritance; occurs between a superclass and its subclass.
Purpose	Provides multiple methods with the same name but different behaviors based on different parameter types or number of parameters.	Allows a subclass to provide a specific implementation of a method defined in its superclass.
Compile-time Resolution	Resolved at compile time based on the method signature (name and parameters).	Resolved at runtime based on the object type (dynamic polymorphism).
Return Type	May or may not have the same return type as the overloaded methods can differ in return type.	Must have the same return type or a covariant return type (subtype of the declared return type).
Example	```java void display(int num) { // Code to display an integer } void display(String str) { // Code to display a string } public static void main(String[] args) { MyClass obj = new MyClass(); obj.display(10); obj.display("Hello"); }	```java class Animal { void sound() { System.out.println("Animal makes a sound"); } } class Dog extends Animal { @Override void sound() { System.out.println("Dog barks"); } }

17. Different between c++ and JAVA .120 word in table form

Here's a comparison between C++ and Java presented in a table format:

Feature	C++	Java
Primary Focus	Emphasizes manual memory management and performance.	Focuses on simplicity, portability, and security.
Platform	Compiled to native machine code; platform-dependent.	Compiled to bytecode; platform-independent (runs on JVM).
Object-Oriented	Supports classes and objects with features like multiple inheritance and operator overloading.	Supports classes and objects but avoids multiple inheritance and operator overloading.
Memory Management	Uses explicit memory allocation and deallocation (new/delete).	Relies on automatic garbage collection; no explicit deallocation.
Exception Handling	Uses try-catch blocks for exception handling.	Uses try-catch-finally blocks for exception handling.
Pointers	Supports pointers with direct memory access.	Does not support pointers to prevent direct memory access, enhancing security.
Threads	Provides support for multi-threading using native threads.	Multi-threading is supported using Java threads (java.lang.Thread).
Standard Libraries	Standard Template Library (STL) for data structures and algorithms.	Java Standard Library (java.util, java.io) with built-in support for collections and I/O operations.
Compilation	Compiled directly to machine code by a compiler.	Compiled to bytecode by a Java compiler, then interpreted or compiled by JVM at runtime.
Namespace	Uses namespaces to organize code and avoid naming conflicts.	Uses packages to organize code and provide namespace-like functionality.

This table summarizes key differences in their approach to memory management, exception handling, threading, compilation, and other core features that distinguish C++ and Java as programming languages.

18. Write a program to find out given number is palindrome or not.

To determine whether a given number is a palindrome or not in Java, follow these steps:

1. **Palindrome Definition**: A number is a palindrome if it reads the same backward as forward.
2. **Approach**: Convert the number to a string, reverse the string, and check if the reversed string is equal to the original string representation of the number.

Here's a Java program to check if a number is a palindrome:

```
java

public class PalindromeNumber {
    public static void main(String[] args) {
        int number = 12321; // Change this number to test different cases

        if (isPalindrome(number)) {
            System.out.println(number + " is a palindrome.");
        } else {
            System.out.println(number + " is not a palindrome.");
        }
    }
```

```
    // Function to check if a number is palindrome
    public static boolean isPalindrome(int num) {
        // Convert number to string
        String original = Integer.toString(num);

        // Reverse the string
        String reversed = new StringBuilder(original).reverse().toString();

        // Check if original and reversed strings are equal
        return original.equals(reversed);
    }
}
```

Explanation:

- **`isPalindrome` Method**:
  - Converts the integer `num` to a string `original`.
  - Reverses the string `original` using `StringBuilder`'s `reverse()` method.
  - Compares the original string with the reversed string using `equals()` method.
  - Returns `true` if they are equal (palindrome), otherwise `false`.
- **Main Method**:
  - Initializes `number` with a test value (12321 in this example).
  - Calls `isPalindrome` method and prints whether the number is a palindrome or not based on the returned boolean value.

This program effectively checks whether the given number is a palindrome by leveraging string manipulation techniques in Java.

19. Write a java program to find the sum of odd number and even numbers separately in an array.

To find the sum of odd numbers and even numbers separately in an array using Java, you can iterate through the array elements, check each number for its parity (odd or even), and accumulate the sums accordingly. Here's how you can implement this:

```
java

public class SumOddEvenNumbers {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // Example array

        int sumEven = 0;
        int sumOdd = 0;

        for (int num : numbers) {
            if (num % 2 == 0) {
                sumEven += num; // Add to sumEven if the number is even
            } else {
                sumOdd += num; // Add to sumOdd if the number is odd
            }
        }

        System.out.println("Sum of even numbers: " + sumEven);
        System.out.println("Sum of odd numbers: " + sumOdd);
    }
}
```

Explanation:

1. **Array Initialization**: `numbers` array is initialized with example values.

2. **Variables Initialization**: `sumEven` and `sumOdd` are initialized to zero to store the sums of even and odd numbers respectively.
3. **Loop through Array**: The `for-each` loop iterates through each element (`num`) in the `numbers` array.
4. **Check Even or Odd**:
    - If `num % 2 == 0`, it means `num` is even, so add `num` to `sumEven`.
    - Otherwise (`num % 2 != 0`), it means `num` is odd, so add `num` to `sumOdd`.
5. **Print Results**: Finally, print the sums of even and odd numbers using `System.out.println()`.

This program efficiently computes the sum of even and odd numbers separately in the array using straightforward logic and basic arithmetic operations. Adjust the `numbers` array to test with different sets of values as needed.

20. Write a program to add two complex number using the concept of constructor overloading.

To add two complex numbers using constructor overloading in Java, you can define a `ComplexNumber` class that supports addition of complex numbers through different constructors. Here's how you can implement this:

```
java

class ComplexNumber {
    private double real;
    private double imaginary;

    // Constructor to initialize complex number with real and imaginary
parts
    public ComplexNumber(double real, double imaginary) {
        this.real = real;
        this.imaginary = imaginary;
    }

    // Constructor to initialize complex number with real part (imaginary
part defaults to 0)
    public ComplexNumber(double real) {
        this(real, 0);
    }

    // Method to add two complex numbers
    public ComplexNumber add(ComplexNumber other) {
        double newReal = this.real + other.real;
        double newImaginary = this.imaginary + other.imaginary;
        return new ComplexNumber(newReal, newImaginary);
    }

    // Method to display complex number
    public void display() {
        if (imaginary >= 0) {
            System.out.println(real + " + " + imaginary + "i");
        } else {
            System.out.println(real + " - " + (-imaginary) + "i");
        }
    }

    public static void main(String[] args) {
        // Creating two complex numbers
        ComplexNumber num1 = new ComplexNumber(2, 3);
        ComplexNumber num2 = new ComplexNumber(1, -1);
```

```
        // Adding num1 and num2
        ComplexNumber sum = num1.add(num2);

        // Displaying the result
        System.out.print("Sum of ");
        num1.display();
        System.out.print("and ");
        num2.display();
        System.out.print("is ");
        sum.display();
    }
}
```

Explanation:

- **ComplexNumber Class**:
  - Private variables `real` and `imaginary` represent the real and imaginary parts of a complex number.
  - Two constructors are defined:
    - `ComplexNumber(double real, double imaginary)`: Initializes a complex number with specified real and imaginary parts.
    - `ComplexNumber(double real)`: Initializes a complex number with only the real part (imaginary part defaults to 0).
  - `add` method performs addition of two `ComplexNumber` objects and returns a new `ComplexNumber` object representing their sum.
  - `display` method prints the complex number in the format `a + bi` or `a - bi` depending on the sign of the imaginary part.
- **Main Method**:
  - Creates two `ComplexNumber` objects (`num1` and `num2`) with different constructor invocations.
  - Calls the `add` method on `num1` to add `num2` and stores the result in `sum`.
  - Prints the original numbers and their sum using the `display` method.

This program demonstrates how constructor overloading can be used to initialize complex numbers and perform arithmetic operations on them effectively in Java. Adjust the real and imaginary parts in `num1` and `num2` to test with different complex numbers.

21. Write a program to implement Fibonacci series using while control structure.

To implement the Fibonacci series using a `while` loop in Java, follow this approach where the series starts with 0 and 1, and each subsequent number is the sum of the previous two numbers:

```
java

public class FibonacciWhile {
    public static void main(String[] args) {
        int n = 10; // Number of Fibonacci numbers to generate
        int firstNum = 0, secondNum = 1;
        int count = 2; // Start count from 2 because we already have two
numbers

        // Print the first two Fibonacci numbers
        System.out.print("Fibonacci Series: " + firstNum + ", " +
secondNum);

        // Generate Fibonacci numbers using while loop
        while (count < n) {
            int nextNum = firstNum + secondNum;
```

```
            System.out.print(", " + nextNum);

            // Update numbers for the next iteration
            firstNum = secondNum;
            secondNum = nextNum;
            count++;
        }
    }
}
```

Explanation:

1. **Initialization**:
    - `firstNum` and `secondNum` are initialized to 0 and 1 respectively, which are the first two Fibonacci numbers.
    - `count` is initialized to 2 because we already printed two numbers (`firstNum` and `secondNum`).
2. **Print Initial Numbers**:
    - Prints the first two Fibonacci numbers (`0, 1`).
3. **Generate Fibonacci Numbers**:
    - Enters a `while` loop that continues until `count` reaches `n`.
    - Calculates the next Fibonacci number (`nextNum`) as the sum of `firstNum` and `secondNum`.
    - Prints `nextNum`.
    - Updates `firstNum` to `secondNum` and `secondNum` to `nextNum` for the next iteration.
    - Increments `count` after each iteration.
4. **Output**:
    - Prints the Fibonacci series up to the specified number `n`.

This program effectively generates and prints the Fibonacci series using a `while` loop in Java. Adjust the value of `n` to generate more or fewer Fibonacci numbers as needed.

21. Write a program to implement Fibonacci series using while control structure.

Certainly! Here's a Java program that implements the Fibonacci series using a `while` loop:

```
java

public class FibonacciWhile {
    public static void main(String[] args) {
        int n = 10; // Number of Fibonacci numbers to generate
        int firstNum = 0, secondNum = 1;
        int count = 0; // Initialize count for loop

        // Print the Fibonacci Series up to n numbers
        System.out.print("Fibonacci Series: ");

        while (count < n) {
            System.out.print(firstNum + " ");

            // Compute the next Fibonacci number
            int nextNum = firstNum + secondNum;
            firstNum = secondNum;
            secondNum = nextNum;

            count++;
        }
    }
}
```

Explanation:

1. **Initialization**:
   - `firstNum` and `secondNum` are initialized to 0 and 1 respectively, which are the first two Fibonacci numbers.
   - `count` is initialized to 0 to control the number of Fibonacci numbers generated.
2. **Print Initial Number**:
   - Prints `"Fibonacci Series: "` as the prefix for the series.
3. **Generate Fibonacci Numbers**:
   - Enters a `while` loop that continues until `count` reaches `n`.
   - Prints the current `firstNum`, which starts with 0.
   - Computes the next Fibonacci number (`nextNum`) by adding `firstNum` and `secondNum`.
   - Updates `firstNum` to `secondNum` and `secondNum` to `nextNum` for the next iteration.
   - Increments `count` after each iteration.
4. **Output**:
   - Prints the Fibonacci series up to `n` numbers in a single line.

This program efficiently generates and prints the Fibonacci series using a `while` loop in Java. Adjust the value of `n` to generate more or fewer Fibonacci numbers as needed.