Java unit 1

1)a). Define OOP and explain the concepts of OOPs.

b). Explain java features.

1) a) Define OOP and Explain the Concepts of OOP

Definition of OOP:

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects." Objects are instances of classes, which can encapsulate data and methods that operate on that data. OOP aims to improve the modularity, reusability, and maintainability of code by organizing it around objects rather than functions and logic.

Key Concepts of OOP:

1. **Class and Object:**
   - **Class:** A blueprint for creating objects. It defines a set of properties (attributes) and methods (behaviors) that the objects created from the class will have.
   - **Object:** An instance of a class. It is a concrete entity that has states and behaviors defined by the class.

```
// Example in Java
class Car {
  // Attributes
  String color;
  String model;

  // Method
  void drive() {
    System.out.println("The car is driving.");
  }
}

// Creating an object
Car myCar = new Car();
```

2. **Encapsulation:**
   - The practice of bundling the data (attributes) and methods (functions) that operate on the data into a single unit or class. It also involves restricting direct access to some of the object's components, which is a way of preventing unintended interference and misuse of the data.

```
class Car {
  // Private attributes
  private String color;
  private String model;

  // Public methods
  public String getColor() {
```

```
    return color;
  }

  public void setColor(String color) {
    this.color = color;
  }
}
```

3. **Inheritance:**
    - A mechanism where a new class (child/subclass) inherits attributes and methods from an existing class (parent/superclass). This allows for code reuse and the creation of hierarchical relationships between classes.

```
// Parent class
class Vehicle {
  void start() {
    System.out.println("Vehicle started.");
  }
}

// Child class
class Car extends Vehicle {
  void drive() {
    System.out.println("Car is driving.");
  }
}

Car myCar = new Car();
myCar.start(); // Inherited method
myCar.drive(); // Method of Car class
```

4. **Polymorphism:**
    - The ability of a single function or method to work in different ways based on the context. It is mainly achieved through method overriding (runtime polymorphism) and method overloading (compile-time polymorphism).

```
// Method Overloading (Compile-time Polymorphism)
class MathOperations {
  int add(int a, int b) {
    return a + b;
  }

  double add(double a, double b) {
    return a + b;
  }
}

// Method Overriding (Runtime Polymorphism)
class Animal {
  void sound() {
    System.out.println("Animal makes a sound");
  }
}

class Dog extends Animal {
```

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

Animal myDog = new Dog();
myDog.sound(); // Outputs: Dog barks
```

5. **Abstraction:**
    - The concept of hiding the complex implementation details and showing only the necessary features of an object. It can be achieved using abstract classes and interfaces.

```
// Abstract class
abstract class Animal {
    abstract void makeSound();
}

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

Animal myDog = new Dog();
myDog.makeSound(); // Outputs: Dog barks
```

1) b) Explain Java Features

Java is a popular, high-level, class-based, and object-oriented programming language. Here are some of the key features of Java:

1. **Simple:**
    - Java is designed to be easy to learn and use. Its syntax is clean and easy to understand, resembling C and C++ without the complexities of pointers and multiple inheritance.
2. **Object-Oriented:**
    - Java follows the principles of OOP, which include encapsulation, inheritance, and polymorphism. This allows for modular, reusable, and maintainable code.
3. **Platform-Independent:**
    - Java is designed to be write-once, run-anywhere (WORA). Java programs are compiled into bytecode, which can be executed on any platform with a Java Virtual Machine (JVM).
4. **Secure:**
    - Java provides a secure execution environment by implementing several security features, such as bytecode verification, sandboxing, and built-in security APIs.
5. **Robust:**

    - Java emphasizes reliability through strong memory management, exception handling, and type checking. It reduces the chances of system crashes and memory leaks.
6. **Multithreaded:**
    - Java supports multithreading, which allows concurrent execution of two or more threads. This enables efficient use of CPU resources and improves the performance of applications.
7. **Interpreted and Compiled:**
    - Java source code is compiled into bytecode, which is then interpreted by the JVM. This allows for a balance between performance and portability.
8. **High Performance:**
    - While not as fast as native languages like C or C++, Java's performance is enhanced by the Just-In-Time (JIT) compiler, which converts bytecode into native machine code at runtime.
9. **Distributed:**
    - Java has built-in support for network programming and distributed computing, making it suitable for developing large-scale, distributed applications.
10. **Dynamic:**
    - Java is designed to adapt to an evolving environment. It supports dynamic loading of classes, dynamic compilation, and automatic memory management (garbage collection).

Example of Java Features in Action:

```
// Simple Java Program to demonstrate some features
public class Main {
  public static void main(String[] args) {
    // Object-Oriented
    Car myCar = new Car("Red", "Tesla");
    myCar.displayInfo();

    // Multithreaded
    Thread thread = new Thread(() -> System.out.println("Running in a separate thread"));
    thread.start();
  }
}

// Class definition to show Object-Oriented feature
class Car {
  private String color;
  private String model;

  public Car(String color, String model) {
    this.color = color;
    this.model = model;
  }

  public void displayInfo() {
    System.out.println("Car Model: " + model + ", Color: " + color);
  }
}
```

- **Object-Oriented:** The `Car` class encapsulates data and methods.

- **Multithreaded:** The use of a thread demonstrates Java's multithreading capability.
- **Platform-Independent:** The compiled bytecode can run on any platform with a JVM.

2)What is an operator? Explain various types of operators available in Java with an example code for each.

What is an Operator?

An operator is a special symbol in Java that performs specific operations on one, two, or three operands and then returns a result. Operators are used to manipulate data and variables in your programs.

Types of Operators in Java

Java provides several types of operators, which can be grouped into the following categories:

1. **Arithmetic Operators**
2. **Relational Operators**
3. **Logical Operators**
4. **Assignment Operators**
5. **Unary Operators**
6. **Bitwise Operators**
7. **Ternary Operator**

Let's explore each type with examples.

1. Arithmetic Operators

Arithmetic operators are used to perform basic mathematical operations like addition, subtraction, multiplication, division, and finding the remainder.

- **+** (Addition)
- **-** (Subtraction)
- ***** (Multiplication)
- **/** (Division)
- **%** (Modulus)

Example:

```
public class ArithmeticExample {
    public static void main(String[] args) {
        int a = 10;
        int b = 5;

        System.out.println("a + b = " + (a + b)); // Addition
        System.out.println("a - b = " + (a - b)); // Subtraction
        System.out.println("a * b = " + (a * b)); // Multiplication
        System.out.println("a / b = " + (a / b)); // Division
        System.out.println("a % b = " + (a % b)); // Modulus
    }
}
```

2. Relational Operators

Relational operators are used to compare two values. They return a boolean result (`true` or `false`).

- **==** (Equal to)
- **!=** (Not equal to)
- **>** (Greater than)
- **<** (Less than)
- **>=** (Greater than or equal to)
- **<=** (Less than or equal to)

Example:

```
public class RelationalExample {
    public static void main(String[] args) {
        int a = 10;
        int b = 5;

        System.out.println("a == b: " + (a == b)); // Equal to
        System.out.println("a != b: " + (a != b)); // Not equal to
        System.out.println("a > b: " + (a > b)); // Greater than
        System.out.println("a < b: " + (a < b)); // Less than
        System.out.println("a >= b: " + (a >= b)); // Greater than or equal
to
        System.out.println("a <= b: " + (a <= b)); // Less than or equal to
    }
}
```

3. Logical Operators

Logical operators are used to combine multiple boolean expressions.

- **&&** (Logical AND)
- **||** (Logical OR)
- **!** (Logical NOT)

Example:

```
public class LogicalExample {
    public static void main(String[] args) {
        boolean x = true;
        boolean y = false;

        System.out.println("x && y: " + (x && y)); // Logical AND
        System.out.println("x || y: " + (x || y)); // Logical OR
        System.out.println("!x: " + (!x)); // Logical NOT
    }
}
```

4. Assignment Operators

Assignment operators are used to assign values to variables.

- **=** (Simple assignment)
- **+=** (Add and assign)
- **-=** (Subtract and assign)
- ***=** (Multiply and assign)
- **/=** (Divide and assign)
- **%=** (Modulus and assign)

Example:

```
public class AssignmentExample {
    public static void main(String[] args) {
        int a = 10;
        a += 5; // a = a + 5
        System.out.println("a += 5: " + a);

        a -= 3; // a = a - 3
        System.out.println("a -= 3: " + a);

        a *= 2; // a = a * 2
        System.out.println("a *= 2: " + a);

        a /= 4; // a = a / 4
        System.out.println("a /= 4: " + a);

        a %= 3; // a = a % 3
        System.out.println("a %= 3: " + a);
    }
}
```

5. Unary Operators

Unary operators are used with only one operand to perform operations such as incrementing/decrementing a value, negating an expression, or inverting the value of a boolean.

- **+** (Unary plus)
- **-** (Unary minus)
- **++** (Increment)
- **--** (Decrement)
- **!** (Logical complement)

Example:

```
public class UnaryExample {
    public static void main(String[] args) {
        int a = 10;
        boolean b = true;

        System.out.println("+a: " + (+a)); // Unary plus
        System.out.println("-a: " + (-a)); // Unary minus

        a++;
        System.out.println("a++: " + a); // Increment

        a--;
        System.out.println("a--: " + a); // Decrement
```

```
        System.out.println("!b: " + (!b)); // Logical complement
    }
}
```

6. Bitwise Operators

Bitwise operators perform bit-level operations on integer types.

- **&** (Bitwise AND)
- **|** (Bitwise OR)
- **^** (Bitwise XOR)
- **~** (Bitwise complement)
- **<<** (Left shift)
- **>>** (Right shift)
- **>>>** (Unsigned right shift)

Example:

```
public class BitwiseExample {
    public static void main(String[] args) {
        int a = 5; // 0101 in binary
        int b = 3; // 0011 in binary

        System.out.println("a & b: " + (a & b)); // Bitwise AND
        System.out.println("a | b: " + (a | b)); // Bitwise OR
        System.out.println("a ^ b: " + (a ^ b)); // Bitwise XOR
        System.out.println("~a: " + (~a)); // Bitwise complement
        System.out.println("a << 1: " + (a << 1)); // Left shift
        System.out.println("a >> 1: " + (a >> 1)); // Right shift
        System.out.println("a >>> 1: " + (a >>> 1)); // Unsigned right
shift
    }
}
```

7. Ternary Operator

The ternary operator is a shorthand for the `if-else` statement and is used to evaluate a boolean expression.

- **?:** (Ternary conditional)

Example:

```
public class TernaryExample {
    public static void main(String[] args) {
        int a = 10;
        int b = 5;

        int max = (a > b) ? a : b;
        System.out.println("Maximum of a and b is: " + max);
    }
}
```

3)Explain the following with suitable examples,i). Simple if statement, ii). if- else statement,

iii). if-else-if statement, iv).Switch statement

i). Simple `if` Statement

The `if` statement is used to test a condition. If the condition evaluates to true, the block of code inside the `if` statement is executed.

Syntax:

```
java
Copy code
if (condition) {
    // code to be executed if condition is true
}
```

Example:

```
public class IfExample {
    public static void main(String[] args) {
        int age = 20;
        if (age >= 18) {
            System.out.println("You are an adult.");
        }
    }
}
```

ii). `if-else` Statement

The `if-else` statement is used to execute one block of code if the condition is true, and another block of code if the condition is false.

Syntax:

```
if (condition) {
    // code to be executed if condition is true
} else {
    // code to be executed if condition is false
}
```

Example:

```
public class IfElseExample {
    public static void main(String[] args) {
        int age = 16;
        if (age >= 18) {
            System.out.println("You are an adult.");
        } else {
            System.out.println("You are not an adult.");
        }
    }
}
```

iii). `if-else-if` Statement

The `if-else-if` ladder is used to test multiple conditions. If one condition is true, the corresponding block of code is executed, and the rest of the ladder is skipped.

Syntax:

```
if (condition1) {
    // code to be executed if condition1 is true
} else if (condition2) {
    // code to be executed if condition2 is true
} else if (condition3) {
    // code to be executed if condition3 is true
} else {
    // code to be executed if all conditions are false
}
```

Example:

```
public class IfElseIfExample {
    public static void main(String[] args) {
        int score = 85;

        if (score >= 90) {
            System.out.println("Grade: A");
        } else if (score >= 80) {
            System.out.println("Grade: B");
        } else if (score >= 70) {
            System.out.println("Grade: C");
        } else if (score >= 60) {
            System.out.println("Grade: D");
        } else {
            System.out.println("Grade: F");
        }
    }
}
```

iv). `switch` Statement

The `switch` statement is used to select one of many code blocks to be executed. It works with byte, short, char, int, and String data types.

Syntax:

```
switch(expression) {
    case value1:
        // code to be executed if expression equals value1
        break;
    case value2:
        // code to be executed if expression equals value2
        break;
    // you can have any number of case statements
    default:
        // code to be executed if none of the cases are true
}
```

Example:

```
public class SwitchExample {
    public static void main(String[] args) {
        int day = 3;
        String dayName;

        switch (day) {
            case 1:
                dayName = "Monday";
                break;
            case 2:
                dayName = "Tuesday";
                break;
            case 3:
                dayName = "Wednesday";
                break;
            case 4:
                dayName = "Thursday";
                break;
            case 5:
                dayName = "Friday";
                break;
            case 6:
                dayName = "Saturday";
                break;
            case 7:
                dayName = "Sunday";
                break;
            default:
                dayName = "Invalid day";
                break;
        }

        System.out.println("The day is: " + dayName);
    }
}
```

4)a). Illustrate the Iteration Statements in java with suitable examples.

b). Write a program to illustrate the sum of "n" natural numbers using for loop statement.

a). Iteration Statements in Java

Iteration statements, also known as loops, are used to execute a block of code repeatedly as long as a specified condition is true. Java provides several types of iteration statements:

1. **`for` loop**
2. **`while` loop**
3. **`do-while` loop**
4. **Enhanced `for` loop (for-each loop)**

Let's illustrate each type with suitable examples.

1. `for` Loop

The `for` loop is used to execute a block of code a specific number of times.

Syntax:

```
for (initialization; condition; update) {
    // code to be executed
}
```

Example:

```
public class ForLoopExample {
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            System.out.println("Iteration: " + i);
        }
    }
}
```

2. `while` Loop

The `while` loop is used to execute a block of code as long as a specified condition is true.

Syntax:

```
while (condition) {
    // code to be executed
}
```

Example:

```
public class WhileLoopExample {
    public static void main(String[] args) {
        int i = 0;
        while (i < 5) {
            System.out.println("Iteration: " + i);
            i++;
        }
    }
}
```

3. `do-while` Loop

The `do-while` loop is similar to the `while` loop, but it executes the block of code at least once before checking the condition.

Syntax:

```
do {
    // code to be executed
} while (condition);
```

Example:

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

```
        int i = 0;
        do {
            System.out.println("Iteration: " + i);
            i++;
        } while (i < 5);
    }
}
```

4. Enhanced `for` Loop (for-each loop)

The enhanced `for` loop, also known as the for-each loop, is used to iterate over elements in an array or a collection.

Syntax:

```
for (type variable : array) {
    // code to be executed
}
```

Example:

```
public class ForEachLoopExample {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};
        for (int number : numbers) {
            System.out.println("Number: " + number);
        }
    }
}
```

b). Program to Illustrate the Sum of "n" Natural Numbers Using `for` Loop Statement

Let's write a program that calculates the sum of the first `n` natural numbers using a `for` loop.

Example:

```
import java.util.Scanner;

public class SumOfNaturalNumbers {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter a positive integer: ");
        int n = scanner.nextInt();

        int sum = 0;
        for (int i = 1; i <= n; i++) {
            sum += i;
        }

        System.out.println("The sum of the first " + n + " natural numbers
is: " + sum);
    }
}
```

5)a). Write short notes on break, continue statements with examples.

b). Write a java program to Illustratehow the break statement alters control flow within the loop?

5) a). Short Notes on `break` and `continue` Statements with Examples

`break` Statement

The `break` statement is used to exit from the current loop or switch statement before it has completed its normal iteration. When a `break` statement is encountered, control is transferred to the statement immediately following the enclosing loop or switch statement.

Syntax:

```
break;
```

Example:

```
public class BreakExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            if (i == 5) {
                break; // Exit the loop when i is 5
            }
            System.out.println("i: " + i);
        }
        System.out.println("Loop exited.");
    }
}
```

Output:

```
i: 1
i: 2
i: 3
i: 4
Loop exited.
```

`continue` Statement

The `continue` statement is used to skip the current iteration of the loop and continue with the next iteration. When a `continue` statement is encountered, the rest of the loop body is skipped, and control is transferred to the next iteration of the loop.

Syntax:

```
continue;
```

Example:

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

```
        for (int i = 1; i <= 10; i++) {
            if (i == 5) {
                continue; // Skip the iteration when i is 5
            }
            System.out.println("i: " + i);
        }
    }
}
```

Output:

```
i: 1
i: 2
i: 3
i: 4
i: 6
i: 7
i: 8
i: 9
i: 10
```

5) b). Java Program to Illustrate How the `break` Statement Alters Control Flow Within the Loop

Let's write a Java program to demonstrate how the `break` statement alters the control flow within a loop.

Example:

```
import java.util.Scanner;

public class BreakControlFlow {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter a positive integer: ");
        int n = scanner.nextInt();

        int sum = 0;
        for (int i = 1; i <= n; i++) {
            if (i == 5) {
                System.out.println("Breaking the loop at i = " + i);
                break; // Exit the loop when i is 5
            }
            sum += i;
        }

        System.out.println("The sum of numbers until break: " + sum);
    }
}
```

6)a)Explain array concept using Java.

b) Write a java program to sort an array of strings.

6) a) Explain Array Concept Using Java

An array in Java is a collection of variables of the same type that are stored at contiguous memory locations. Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

Characteristics of Arrays:

- **Fixed Size:** Once an array is created, its size cannot be changed.
- **Homogeneous Elements:** All elements in an array are of the same type.
- **Zero-based Indexing:** Array indices start at 0 and go up to (array length - 1).

Array Declaration and Initialization:

Syntax:

```
// Declaration
type[] arrayName;

// Declaration and Initialization
type[] arrayName = new type[size];
```

Example:

```
public class ArrayExample {
    public static void main(String[] args) {
        // Declare and initialize an array of integers
        int[] numbers = new int[5];

        // Assign values to the array elements
        numbers[0] = 10;
        numbers[1] = 20;
        numbers[2] = 30;
        numbers[3] = 40;
        numbers[4] = 50;

        // Access and print array elements
        for (int i = 0; i < numbers.length; i++) {
            System.out.println("Element at index " + i + ": " +
numbers[i]);
        }
    }
}
```

Array Initialization with Values:

Example:

```
public class ArrayExample {
    public static void main(String[] args) {
        // Declare and initialize an array with values
        int[] numbers = {10, 20, 30, 40, 50};

        // Access and print array elements
        for (int i = 0; i < numbers.length; i++) {
            System.out.println("Element at index " + i + ": " +
numbers[i]);
        }
```

```
    }
}
```

6) b) Java Program to Sort an Array of Strings

To sort an array of strings, we can use the `Arrays.sort()` method provided by Java. Here's a simple Java program to demonstrate this:

Example:

```
import java.util.Arrays;

public class SortStringArray {
    public static void main(String[] args) {
        // Declare and initialize an array of strings
        String[] names = {"John", "Alice", "Bob", "Charlie", "David"};

        // Print the original array
        System.out.println("Original array:");
        for (String name : names) {
            System.out.println(name);
        }

        // Sort the array using Arrays.sort() method
        Arrays.sort(names);

        // Print the sorted array
        System.out.println("\nSorted array:");
        for (String name : names) {
            System.out.println(name);
        }
    }
}
```

Explanation:

1. **Import `Arrays` class:** We need the `Arrays` class from `java.util` package to use the `sort()` method.
2. **Declare and initialize array:** We declare and initialize an array of strings `names`.
3. **Print original array:** We print the original array to show the unsorted order.
4. **Sort array:** We use `Arrays.sort(names)` to sort the array of strings in alphabetical order.
5. **Print sorted array:** We print the sorted array to show the new order of elements.

Output:

```
Original array:
John
Alice
Bob
Charlie
David

Sorted array:
Alice
Bob
Charlie
David
```

John

7)What is a Constructor? Explain various types of Constructors in detail with a suitable example.

What is a Constructor?

A constructor in Java is a special method that is called when an object is instantiated. Its main purpose is to initialize the newly created object. The constructor has the same name as the class and does not have a return type, not even `void`.

Types of Constructors

1. **Default Constructor**
2. **Parameterized Constructor**
3. **Copy Constructor** (not natively supported in Java but can be implemented)

1. Default Constructor

A default constructor is a constructor that takes no arguments. If no constructor is explicitly defined in a class, the Java compiler automatically provides a default constructor.

Example:

```
class DefaultConstructorExample {
    int number;
    String text;

    // Default constructor
    DefaultConstructorExample() {
        number = 0;
        text = "Default";
    }

    void display() {
        System.out.println("Number: " + number + ", Text: " + text);
    }

    public static void main(String[] args) {
        DefaultConstructorExample obj = new DefaultConstructorExample();
        obj.display(); // Output: Number: 0, Text: Default
    }
}
```

2. Parameterized Constructor

A parameterized constructor is a constructor that takes one or more arguments. It is used to provide different values to the distinct objects.

Example:

```
class ParameterizedConstructorExample {
    int number;
```

```
    String text;

    // Parameterized constructor
    ParameterizedConstructorExample(int num, String txt) {
        number = num;
        text = txt;
    }

    void display() {
        System.out.println("Number: " + number + ", Text: " + text);
    }

    public static void main(String[] args) {
        ParameterizedConstructorExample obj1 = new
ParameterizedConstructorExample(10, "Hello");
        ParameterizedConstructorExample obj2 = new
ParameterizedConstructorExample(20, "World");

        obj1.display(); // Output: Number: 10, Text: Hello
        obj2.display(); // Output: Number: 20, Text: World
    }
}
```

8)What is Method Overloading? Write a java program to illustrate "Method Overloading".

What is Method Overloading?

Method overloading in Java is a feature that allows a class to have more than one method with the same name, but with different parameter lists. Method overloading can be achieved by varying the number of parameters or the type of parameters in the methods.

Key Points of Method Overloading:

- **Different Parameters:** Methods must differ in the number or type of parameters.
- **Return Type:** The return type of the methods can be different, but it does not play a role in method overloading.
- **Compile Time Polymorphism:** Method overloading is an example of compile-time polymorphism.

Example of Method Overloading

Let's illustrate method overloading with a Java program.

Example:

```
public class MethodOverloadingExample {

    // Method with one parameter
    public void display(int a) {
        System.out.println("Argument: " + a);
    }

    // Overloaded method with two parameters
    public void display(int a, int b) {
```

```
        System.out.println("Arguments: " + a + ", " + b);
    }

    // Overloaded method with different parameter types
    public void display(String a) {
        System.out.println("String Argument: " + a);
    }

    // Overloaded method with different parameter order
    public void display(String a, int b) {
        System.out.println("Arguments: " + a + ", " + b);
    }

    // Overloaded method with different parameter order
    public void display(int a, String b) {
        System.out.println("Arguments: " + a + ", " + b);
    }

    public static void main(String[] args) {
        MethodOverloadingExample example = new MethodOverloadingExample();

        // Call the overloaded methods
        example.display(10);                // Calls display(int a)
        example.display(10, 20);            // Calls display(int a, int b)
        example.display("Hello");           // Calls display(String a)
        example.display("Hello", 20);       // Calls display(String a, int
b)
        example.display(10, "Hello");       // Calls display(int a, String
b)
    }
}
```

Explanation:

1. **Method Signature:** Each `display` method has a different method signature (the combination of method name and parameter list).
2. **Single Parameter:** `display(int a)` method takes a single integer argument.
3. **Two Parameters:** `display(int a, int b)` method takes two integer arguments.
4. **Different Type:** `display(String a)` method takes a single string argument.
5. **Different Parameter Order:**
    - `display(String a, int b)` method takes a string followed by an integer.
    - `display(int a, String b)` method takes an integer followed by a string.

Output:

```
Argument: 10
Arguments: 10, 20
String Argument: Hello
Arguments: Hello, 20
Arguments: 10, Hello
```

9)a)Explain in detail about recursion in java?

b)Write a program to implement the factorial of a number using recursion in java.

9) a) Recursion in Java

Recursion is a programming technique where a method calls itself to solve a problem. In Java, a method that calls itself is known as a recursive method. Recursion provides an elegant and concise way to solve problems that can be broken down into smaller, similar subproblems.

Key Concepts of Recursion:

1. **Base Case:** Every recursive method must have a base case that determines when the recursion should stop. It prevents the method from infinitely calling itself.
2. **Recursive Case:** This is where the method calls itself with a modified parameter to move closer to the base case.
3. **Stack Usage:** Recursive methods use the call stack to store intermediate values and return addresses. Too many recursive calls without reaching the base case can lead to stack overflow.

Example of Recursion

Let's look at an example of calculating the factorial of a number using recursion:

b) Java Program to Implement Factorial Using Recursion

Factorial of a non-negative integer nnn is denoted as n!n!n! and is defined as the product of all positive integers less than or equal to nnn.

Example:

```
public class FactorialExample {

    // Recursive method to calculate factorial
    public static int factorial(int n) {
        // Base case: factorial of 0 or 1 is 1
        if (n == 0 || n == 1) {
            return 1;
        } else {
            // Recursive case: factorial of n is n * factorial(n-1)
            return n * factorial(n - 1);
        }
    }

    public static void main(String[] args) {
        int number = 5;
        int fact = factorial(number);
        System.out.println("Factorial of " + number + " is: " + fact);
    }
}
```

Explanation:

1. **`factorial` Method:**
    - **Base Case:** If `n` is 0 or 1, return 1 because 0!=10! = 10!=1 and 1!=11! = 11!=1.
    - **Recursive Case:** If `n` is greater than 1, recursively call `factorial(n - 1)` and multiply the result by `n`.
2. **Main Method:**

- Initialize `number` with 5 (or any other positive integer).
- Call `factorial(number)` to compute the factorial.
- Print the result.

Output:

```
Factorial of 5 is: 120
```

In this example, `factorial(5)` calls itself with `n = 4`, then `factorial(4)` calls itself with `n = 3`, and so on until reaching `factorial(1)`. Each call returns the result to its caller, eventually computing the factorial of 5 by multiplying the results of smaller factorials.

Benefits and Considerations of Recursion:

- **Simplicity:** Recursion can provide a simpler and more readable solution for problems that have repetitive structures.
- **Memory Usage:** Recursion uses more memory due to multiple function calls stored on the call stack. Excessive recursion can lead to stack overflow errors.
- **Performance:** In some cases, iterative solutions may be more efficient than recursive ones due to the overhead of function calls.

Recursion is a powerful technique in programming but should be used judiciously with careful consideration of base cases and termination conditions to avoid infinite loops and excessive memory usage.

10)What is garbage collection? Explain how to call the garbage collector explicitly in java.

10) What is Garbage Collection?

Garbage collection in Java is the process by which the Java Virtual Machine (JVM) automatically manages memory by reclaiming unused objects that are no longer referenced and making the memory available for future allocations. It is a part of Java's automatic memory management system, which eliminates the need for manual memory management (like in languages such as C++).

Key Points about Garbage Collection:

1. **Automatic Process:** Garbage collection is performed automatically by the JVM without programmer intervention.
2. **Memory Management:** It frees up memory occupied by objects that are no longer referenced by any part of the program, preventing memory leaks.
3. **Performance:** Java's garbage collector is designed to minimize the impact on application performance by running concurrently with the application threads.

Calling Garbage Collector Explicitly in Java

In Java, it is generally not necessary to explicitly call the garbage collector, as the JVM manages this automatically. However, there are methods provided by the `System` and `Runtime` classes that can be used to suggest or request garbage collection:

Using `System.gc()` Method:

The `System` class in Java provides a `gc()` method that suggests the JVM to run the garbage collector. However, it does not guarantee immediate garbage collection.

Example:

```
public class GarbageCollectionExample {
    public static void main(String[] args) {
        // Code that uses objects and then suggests garbage collection
        String str = new String("Hello");
        System.out.println("String created.");

        // Suggest garbage collection
        System.gc();

        System.out.println("End of main method.");
    }
}
```

In the example above, `System.gc()` suggests that the JVM should run the garbage collector. However, the actual execution of garbage collection is up to the JVM's discretion and may not happen immediately.

Using `Runtime.getRuntime().gc()` Method:

The `Runtime` class also provides a `gc()` method that suggests garbage collection, similar to `System.gc()`.

Example:

```
public class GarbageCollectionExample {
    public static void main(String[] args) {
        // Code that uses objects and then suggests garbage collection
        String str = new String("Hello");
        System.out.println("String created.");

        // Suggest garbage collection
        Runtime.getRuntime().gc();

        System.out.println("End of main method.");
    }
}
```

Important Considerations:

- **Effectiveness:** Explicitly calling `System.gc()` or `Runtime.getRuntime().gc()` does not guarantee immediate garbage collection. The JVM decides when to run the garbage collector based on various factors such as available memory, system load, etc.

- **Performance:** Explicitly calling garbage collection unnecessarily can impact performance by introducing unnecessary overhead.
- **Best Practice:** Trust the JVM's garbage collection mechanism to manage memory efficiently. Explicitly calling `gc()` should generally be avoided unless there is a specific and valid reason to do so (e.g., optimizing memory usage in certain rare cases).