JAVA

Java is a class-based, **object-oriented programming language** that is designed to have as few implementation dependencies as possible. It is intended to let application developers **Write Once and Run Anywhere (WORA)**, meaning that compiled Java code can run on all platforms that support Java without the need for recompilation.

JDK, JRE and JVM

JDK (Java Development Kit): Provides tools for development (compiler, debugger).

JRE (Java Runtime Environment): Includes libraries and JVM for running Java applications.

JVM (Java Virtual Machine): Converts bytecode into machine code and executes it.

Java applications are compiled to byte code that can run on any Java Virtual Machine

IDENTIFIERS:

An identifier in Java is the name given to Variables, Classes, Methods, Packages, Interfaces, etc. These are the unique names and every Java Variables must be identified with unique names.

Rules For Naming Java Identifiers

- The only allowed characters for identifiers are all alphanumeric characters([**A-Z**],[**a-z**],[**0-9**]), '**$**'(dollar sign) and '**_**' (underscore). For example "@" is not a valid Java identifier as it contains a '@' a special character.
- Identifiers should **not** start with digits(**[0-9]**). For example "123" is not a valid Java identifier.
- Java identifiers are **case-sensitive**.
- There is no limit on the length of the identifier but it is advisable to use an optimum length of 4 – 15 letters only.
- **Reserved Words** can't be used as an identifier. For example "int while = 20;" is an invalid statement as a while is a reserved word. There are **53** reserved words in Java.

KEYWORDS:

In Java, Keywords are the Reserved words in a programming language that are used for some internal process or represent some predefined actions. These words are therefore not allowed to use as variables names or objects.

DATA TYPES

Data types in Java are of different sizes and values that can be stored in the variable that is made as per convenience and circumstances to cover up all test cases. Java has two categories in which data types are segregated

1. **Primitive Data Type:** such as boolean, char, int, short, byte, long, float, and double.

Type	Description	Default	Size	Example Literals	Range of values
boolean	true or false	false	8 bits	true, false	true, false
byte	twos-complement integer	0	8 bits	(none)	-128 to 127
char	Unicode character	\u0000	16 bits	‘a’, ‘\u0041’, ‘\101’, ‘\\’, ‘\’, ‘\n’, ‘β’	characters representation of ASCII values 0 to 255
short	twos-complement integer	0	16 bits	(none)	-32,768 to 32,767
int	twos-complement intger	0	32 bits	-2,-1,0,1,2	-2,147,483,648 to 2,147,483,647
long	twos-complement integer	0	64 bits	-2L,-1L,0L,1L,2L	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Type	Description	Default	Size	Example Literals	Range of values
float	IEEE 754 floating point	0.0	32 bits	1.23e100f , -1.23e-100f , .3f ,3.14F	upto 7 decimal digits
double	IEEE 754 floating point	0.0	64 bits	1.23456e300d , -123456e-300d , 1e1d	upto 16 decimal digits

2.Non-Primitive Data Type or Object Data type: such as String, Array, Class, Interface.

1. C**lasses**:

- A class is a blueprint for objects. It defines the properties (fields) and behaviors (methods) that objects created from the class can have.

```
class Car {
    String model;
    int year;
    Car(String model, int year) {
        this.model = model;
        this.year = year;
}
    void displayInfo() {
        System.out.println("Model: " + model + ", Year: " + year);
    }
}
 public class Main {
    public static void main(String[] args) {
```

```
        Car car1 = new Car("Toyota", 2020); // Creating an object of the class Car
        car1.displayInfo(); // Calling a method
    }
}
```

2. **Interfaces**:

- An interface is a reference type, similar to a class, that is a collection of abstract methods (methods without implementation). Interfaces can be implemented by classes.

```
interface Animal {
    void sound();  // abstract method
}
class Dog implements Animal {
    public void sound() {
        System.out.println("Bark");
    }
}
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();  // Creating an object of Dog
        dog.sound();  // Calling the implemented method
    }
}
```

3. **Arrays**:

- An array is a container object that holds a fixed number of values of a single type. It is an object, and each element is accessed using an index.

```
public class Main {

    public static void main(String[] args) {

        int[] numbers = {1, 2, 3, 4, 5}; // Creating an array of integers

        System.out.println("First element: " + numbers[0]);  // Accessing an array element

        System.out.println("Last element: " + numbers[4]);  // Accessing another array element

    }}
```

4. **Strings**:

- The `String` class in Java is used to represent a sequence of characters. Even though it seems like a primitive type, `String` is actually a class, making it a non-primitive type.
- Example: `String name = "John";`

```
public class Main {

    public static void main(String[] args) {

        String greeting = "Hello, World!"; // Creating a string

        System.out.println(greeting); // Printing the string

    }

}
```

VARIABLES

Variables are the containers for storing the data values or you can also call it a memory location name for the data. Every variable has a:

- **Data Type** – The kind of data that it can hold. For example, int, string, float, char, etc.
- **Variable Name** – To identify the variable uniquely within the scope.
- **Value** – The data assigned to the variable.

int a =10; (Integer variable a having value 10)

Types of variables:

In Java, there are three main types of variables, each serving different purposes based on their scope, lifetime, and where they are declared.

1.Local variable: Local variables are declared inside a method, constructor, or block and are used only within that specific scope.

- **Scope**: They are only accessible within the method, constructor, or block where they are declared.
- **Lifetime**: Local variables are created when the method is invoked and are destroyed when the method execution completes.
- **Access**: Local variables cannot be accessed outside the method/block where they are declared

```
public class Main {
    public static void main(String[] args) {
        // Local variable
        int sum = 10 + 20;
        System.out.println("Sum: " + sum);  // Output: Sum: 30
    }
}
```

2.Instance variable: Instance variables are non-static variables that are defined inside a class but outside any method, constructor, or block.

- **Scope**: Each object created from the class has its own copy of these variables.

- **Lifetime**: They are created when the object is created and destroyed when the object is destroyed (garbage collected).
- **Access**: These variables can be accessed directly using the object reference.

```
class Car {
    // Instance variable
    String model;
    // Constructor to initialize the model
    Car(String model) {
        this.model = model;
    }
    // Method to display the model of the car
    void displayModel() {
        System.out.println("Car model: " + model);
    }
}
public class Main {
    public static void main(String[] args) {
        // Creating two objects of the Car class
        Car car1 = new Car("Toyota");
  Car car2 = new Car("Honda");
        // Displaying their models
        car1.displayModel();  // Output: Car model: Toyota
        car2.displayModel();  // Output: Car model: Honda
    }
}
```

3.Static variable(Class variable): Class variables are static variables, declared with the static keyword inside a class, but outside any method, constructor, or block.

- **Scope**: There is **only one copy** of the class variable for the entire class, shared across all instances (objects) of the class.
- **Lifetime**: Class variables exist as long as the class is loaded into memory.

- **Access**: These variables can be accessed directly using the class name or through an object reference.

```
class Car {
    // Class variable (static variable)
    static int wheels = 4;
    // Method to display the number of wheels
    void displayWheels() {
        System.out.println("Number of wheels: " + wheels);
    }
}

public class Main {
    public static void main(String[] args) {
        // Accessing the static variable through objects
        Car car1 = new Car();
        car1.displayWheels();  // Output: Number of wheels: 4

        Car car2 = new Car();
        car2.displayWheels();  // Output: Number of wheels: 4

        // You can also access static variables via the class name
        System.out.println("Wheels: " + Car.wheels);  // Output: Wheels: 4
    }
}
```

DATA STRUCTURE

A **data structure** is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.
A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed. So we must have good knowledge about data structures. ac

Classification of Data Structure

1. **Linear Data Structure:** Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.
   Example: Array, Stack, Queue, Linked List, etc.
- **Static Data Structure:** Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.
  Example: array.
- **Dynamic Data Structure:** In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.
  Example: Queue, Stack, etc.

2. **Non-Linear Data Structure:** Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can't traverse all the elements in a single run only.
   Examples: Trees and Graphs.

1. Array:

- Array is a linear data structure that stores a collection of elements of the same data type. Elements are allocated contiguous memory, allowing for constant-time access. Each element has a unique index number.

SEARCHING IN ARRAY:

Searching is one of the most common operations performed in an array. Array searching can be defined as the operation of finding a particular element or a group of elements in the array.

There are several searching algorithms. The most commonly used among them are:

1.Linear Search:

Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element or group of elements is found. Otherwise, the search continues till the end of the data set. This has a time complexity of O(N) where 'N' is the length of the array

2.Binary Search:

Binary Search is a searching algorithm used in a sorted array. In this algorithm, the element is found by repeatedly dividing the search interval in half and deciding the next interval to find the element. This searching algorithm has a time complexity of O(log 2 N) where 'N' is the length of the array. The only thing to note is that the array must be sorted in increasing or decreasing order.

3.Ternary Search:

Ternary search is a divide and conquer algorithm that can be used to find an element in an array. It is similar to binary search where we divide the array into two parts but in this algorithm, we divide the given array into three parts and determine which has the key (searched element). This algorithm also has the constraint that the array must be sorted. The time complexity for this algorithm is O(log 3 N) where 'N' is the size of the array.

SORTING IN ARRAY:

Sorting an array means arranging the elements of the array in a certain order. Generally sorting in an array is done to arrange the elements in increasing or decreasing order.

Popular Sorting Algorithms:

1.Bubble Sort: Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large arrays as its average and worst-case time complexity is quite high.

2.Selection Sort: Selection sort is another sorting technique in which we find the minimum element in every iteration and place it in the array beginning from the first index. Thus, a selection sort also gets divided into a sorted and unsorted subarray.

3.Insertion Sort: Insertion sort similarly to the way we sort playing cards in our hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed in the correct position in the sorted part.

4.Merge Sort: It is a sorting algorithm that is based on the Divide and Conquer paradigm. In this algorithm, the array is repeatedly divided into two equal halves and then they are combined in a sorted manner.

5.Quick Sort: This is a sorting algorithm based on the divide and conquer approach where an array is divided into subarrays by selecting a pivot element (element selected from the array).

6.Heap Sort: Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

Complexity of Sorting Algorithms	**Best Case**	**Average Case**	**Worst Case**	**Memory**	**Stable**	**Method Used**
Quick Sort	N * logN	N * logN	N2	N	No	Partitioning

Merge Sort	N * logN	N * logN	N * logN	N	Yes	Merging
Heap Sort	N * logN	N * logN	N * logN	1	No	Selection
Insertion Sort	N	N2	N2	1	Yes	Insertion
Selection Sort	N2	N2	N2	1	No	Selection
Bubble Sort	N	N2	N2	1	Yes	Exchanging

It offers mainly the following advantages over other data structures.

- Random Access : i-th item can be accessed in O(1) Time as we have the base address and every item or reference is of same size.
- Cache Friendliness : Since items / references are stored at contiguous locations, we get the advantage of locality of reference.
- It is not useful in places where we have operations like insert in the middle, delete from middle and search in a unsorted data.
- It is a fundamental and linear data structure using which we build other data structures like Stack Queue, Deque, Graph, Hash Table, etc.
- **Matrix** is a two-dimensional array of elements, arranged in **rows** and **columns**. It is represented as a rectangular grid, with each element at the intersection of a row and column.

2 Queue

- Queue is a linear data structure that follows the First In, First Out (FIFO) principle. Queues play an important role in managing tasks or data in order, scheduling and message handling systems.

Some of the basic operations for Queue in Data Structure are:

- enqueue() – Insertion of elements to the queue.
- dequeue() – Removal of elements from the queue.
- peek() or front()- Acquires the data element available at the front node of the queue without deleting it.
- rear() – This operation returns the element at the rear end without removing it.
- isFull() – Validates if the queue is full.
- isEmpty() – Checks if the queue is empty.
- size(): This operation returns the size of the queue i.e. the total number of elements it contains.

Types of Queues:

There are five different types of queues that are used in different scenarios. They are:

1. Circular Queue: Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called 'Ring Buffer'. This queue is primarily used in the following cases:

- Memory Management: The unused memory locations in the case of ordinary queues can be utilized in circular queues.
- Traffic system: In a computer-controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.
- CPU Scheduling: Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

The time complexity for the circular Queue is O(1).

2. Input restricted Queue: In this type of Queue, the input can be taken from one side only(rear) and deletion of elements can be done from both sides(front and rear). This kind of Queue does not follow FIFO(first in first out). This queue is used in cases where the consumption of the data needs to be in FIFO order but if there is a need to remove the

recently inserted data for some reason and one such case can be irrelevant data, performance issue, etc.

Advantages of Input restricted Queue:

- Prevents overflow and overloading of the queue by limiting the number of items added
- Helps maintain stability and predictable performance of the system

Disadvantages of Input restricted Queue:

- May lead to resource wastage if the restriction is set too low and items are frequently discarded
- May lead to waiting or blocking if the restriction is set too high and the queue is full, preventing new items from being added.

3. Output restricted Queue: In this type of Queue, the input can be taken from both sides(rear and front) and the deletion of the element can be done from only one side(front). This queue is used in the case where the inputs have some priority order to be executed and the input can be placed even in the first place so that it is executed first.

4. Double ended Queue: Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear). That means, we can insert at both front and rear positions and can delete from both front and rear positions. Since Deque supports both stack and queue operations, it can be used as both. The Deque data structure supports clockwise and anticlockwise rotations in O(1) time which can be useful in certain applications. Also, the problems where elements need to be removed and or added both ends can be efficiently solved using Deque.

5. **Priority Queue:** A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. There are two types of Priority Queues. They are:

Ascending Priority Queue: Element can be inserted arbitrarily but only smallest element can be removed. For example, suppose there is an array having elements 4, 2, 8 in the same

order. So, while inserting the elements, the insertion will be in the same sequence but while deleting, the order will be 2, 4, 8.

Descending priority Queue: Element can be inserted arbitrarily but only the largest element can be removed first from the given Queue. For example, suppose there is an array having elements 4, 2, 8 in the same order. So, while inserting the elements, the insertion will be in the same sequence but while deleting, the order will be 8, 4, 2.

The time complexity of the Priority Queue is O(logn).

3. Linked List

- Linked list is a linear data structure that stores data in nodes, which are connected by pointers. Unlike arrays, nodes of linked lists are not stored in contiguous memory locations and can only be accessed sequentially, starting from the head of list.

Types Of Linked Lists:

1. Singly Linked List

Singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.

The node contains a pointer to the next node means that the node stores the address of the next node in the sequence. A single linked list allows the traversal of data only in one way.

2. Doubly Linked List

A doubly linked list or a two-way linked list is a more complex type of linked list that contains a pointer to the next as well as the previous node in sequence.

Therefore, it contains three parts of data, a pointer to the next node, and a pointer to the previous node. This would enable us to traverse the list in the backward direction as well.

3. Circular Linked List

A circular linked list is a type of linked list in which the last node’s next pointer points back to the first node of the list, creating a circular structure. This design allows for continuous traversal of the list, as there is no null to end the list.

While traversing a circular linked list, we can begin at any node and traverse the list in any direction forward and backward until we reach the same node we started. Thus, a circular linked list has no beginning and no end.

4. Tree

- Tree is a non-linear, hierarchical data structure consisting of nodes connected by edges, with a top node called the root and nodes having child nodes. It is widely used in file systems, databases, decision-making algorithms, etc.
- Binary Tree is a non-linear and hierarchical data structure where each node has at most two children referred to as the left child and the right child.
- Binary Search Tree is a type of binary tree in which each node's left subtree contains only values smaller than the node, and each node's right subtree contains only values greater than the node. This property applies recursively, meaning that for every node, its left and right subtrees must also satisfy the conditions of a valid Binary Search Tree.

5. Graph

- Graph is a non-linear data structure consisting of a finite set of vertices(or nodes) and a set of edges(or links)that connect a pair of nodes. Graphs are widely used to represent relationships between entities.

JAVA OPERATORS

Operator Category	Operator	Description	Example
Arithmetic Operators	+	Addition	5 + 3 = 8
	-	Subtraction	5 - 3 = 2
	*	Multiplication	5 * 3 = 15
	/	Division	5 / 3 = 1
	%	Modulus (remainder)	5 % 3 = 2
Relational Operators	==	Equal to	5 == 3 → false
	!=	Not equal to	5 != 3 → true
	>	Greater than	5 > 3 → true
	<	Less than	5 < 3 → false
	>=	Greater than or equal to	5 >= 3 → true
	<=	Less than or equal to	5 <= 3 → false
Logical Operators		Logical AND (true if both conditions are true)	true && false → false
	!	Logical NOT (inverts boolean value)	!true → false
Assignment Operators	=	Assigns the right operand to the left operand	a = 5
	+=	Adds and assigns the result	a += 3 → a = a + 3
	-=	Subtracts and assigns the result	a -= 2 → a = a - 2
	*=	Multiplies and assigns the result	a *= 2 → a = a * 2
	/=	Divides and assigns the result	a /= 2 → a = a / 2
	%=	Modulus and assigns the result	a %= 2 → a = a % 2

Unary Operators	`++`	Increment (adds 1 to the operand)	`a++` or `++a`
	`--`	Decrement (subtracts 1 from the operand)	`a--` or `--a`
	`+`	Unary plus (no change to value)	`+a`
	`-`	Unary minus (negates the value)	`-a`
	`!`	Logical NOT (inverts boolean value)	`!a`
Ternary Operator	`? :`	Conditional (shorthand for `if-else` statement)	`condition ? expr1 : expr2`
Bitwise Operators	`&`	Bitwise AND (performs AND on each bit)	`a & b`
	`` `` ``	Bitwise OR (performs OR on each bit)	
	`^`	Bitwise XOR (performs XOR on each bit)	
	`~`	Bitwise NOT (inverts each bit)	`~a`
	`<<`	Left shift (shifts bits to the left by specified number)	`a << 1`
	`>>`	Right shift (shifts bits to the right by specified number)	`a >> 1`
	`>>>`	Unsigned right shift (shifts bits to the right, fills 0)	`a >>> 1`

JAVA USER INPUT

The most common way to take user input in Java is using **Scanner** class which is part of **java.util package**. The scanner class can read input from various sources like console, files or streams

- Import the Scanner class using ***import java.util.Scanner;***
- **Create the Scanner object** and connect Scanner with **System.in** by passing it as an argument i.e. ***Scanner scn = new Scanner(System.in);***
- Print a message to prompt for user input and you can use the various methods of Scanner class like nextInt(), nextLine(), next(), nextDouble etc according to your need.

Method	Description
nextBoolean()	Used for reading Boolean value.
nextByte()	Used for reading Byte value.
nextDouble()	Used for reading Double value.
nextFloat()	Used for reading Float value.
nextInt()	Used for reading Int value.
nextLine()	Used for reading Line value.
nextLong()	Used for reading Long value.
nextShort()	Used for reading Short value.

FLOW CONTROL

1.DECISION MAKING :

In Java, decision-making statements allow the program to make decisions based on conditions.

Java provides several control flow statements for decision-making:

1. if statement

The if statement allows the program to execute a block of code if the condition is true

```
Ex:   if (number > 5) {
          System.out.println("Number is greater than 5");
      }
```

2. if-else statement

The if-else statement allows you to specify an alternative action if the condition is false.

```
Ex:   int number = 3;
      if (number > 5) {
          System.out.println("Number is greater than 5");
      } else {
          System.out.println("Number is not greater than 5");
      }
```

3. else-if ladder

If you have multiple conditions to check, you can use the else-if ladder. It allows checking multiple conditions in sequence.

```
Ex:   int number = 5;
      if (number > 10) {
```

```
    System.out.println("Number is greater than 10");
} else if (number > 5) {
    System.out.println("Number is greater than 5 but less than or equal to 10");
} else {
    System.out.println("Number is 5 or less");
```

4. switch statement

The switch statement allows you to check multiple conditions for a single variable, and execute code based on its value. It is often used when there are many possible conditions to evaluate.

Ex:

```
int day = 3;
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    case 4:
        System.out.println("Thursday");
        break;
    case 5:
        System.out.println("Friday");
```

```
            break;
        case 6:
            System.out.println("Saturday");
            break;
        case 7:
            System.out.println("Sunday");
            break;
        default:
            System.out.println("Invalid day");
    }
```

2.LOOPS :

In Java, loops are used to repeat a block of code multiple times based on certain conditions. Java provides several types of loops:

1. for loop

The for loop is used when you know the number of iterations in advance. It consists of three parts: initialization, condition, and increment/decrement.

Ex: // Prints numbers from 1 to 5

```
    for (int i = 1; i <= 5; i++) {
        System.out.println(i);
```

- **Initialization**: int i = 1 - Starts the loop counter at 1.
- **Condition**: i <= 5 - Loops will continue as long as this condition is true.
- **Increment**: i++ - Increases i by 1 after each loop.

2. while loop

The while loop is used when you want to repeat a block of code as long as a condition is true. The condition is checked before executing the loop body.

Ex: int i = 1;

```
// Prints numbers from 1 to 5
    while (i <= 5) {
        System.out.println(i);
        i++;  // Increment counter
    }
```

- **Condition**: The loop continues as long as i <= 5 is true.
- **Increment**: The i++ ensures that the counter increases with each iteration.

3. do-while loop

The do-while loop is similar to the while loop, but it checks the condition after executing the loop body. This guarantees that the loop body will be executed at least once.

Ex: int i = 1;

```
// Prints numbers from 1 to 5
    do {
        System.out.println(i);
        i++;  // Increment counter
    } while (i <= 5);
```

Do-While Loop Behavior: The loop will always run at least once, even if the condition is false at the start.

METHODS

Java Methods are blocks of code that perform a specific task. A method allows us to reuse code, improving both efficiency and organization. All **methods in Java** must belong to a **class**. Methods are similar to functions and expose the behavior of objects.

Method Declaration:

- **Modifier:** It specifies the method's access level (e.g., public, private, protected, or default).
- **Return Type**: The type of value returned, or void if no value is returned.
- **Method Name**: It follows Java naming conventions; it should start with a lowercase verb and use camel case for multiple words.
- **Parameters**: A list of input values (optional). Empty parentheses are used if no parameters are needed.
- **Exception List**: The exceptions the method might throw (optional).
- **Method Body**: It contains the logic to be executed (optional in the case of abstract methods).

```
return_type method_name(parameters) {
   // Method body
   // Code to be executed
   return value;  // Optional, only if return_type is not void
}
```

Access Modifier Use Cases:

1. public: Use when you want to make a member accessible from anywhere.
2. private: Use when you want to restrict access to a member within the same class.
3. protected: Use when you want to allow access within the same package and by subclasses.

4. default: Use when you want to allow access only within the same package.

Types of Methods in Java

1. Predefined Method

Predefined methods are the method that is already defined in the Java class libraries. It is also known as the standard library method or built-in method.

Example:

Math.random() // returns random value

Math.PI() // return pi value

2. User-defined Method

The method written by the user or programmer is known as a user-defined method**.** These methods are modified according to the requirement.

Example:

sayHello // user define method created above in the article

Greet()

setName()

Method Calling

Method calling allows to reuse code and organize our program effectively. The method needs to be called for use its functionality. There can be three situations when a method is called:
A method returns to the code that invoked it when:

- It completes all the statements in the method.
- It reaches a return statement.
- Throws an exception.

Example : Calling Methods in Different Ways

```
// Java Program to Illustrate Method Calling

import java.io.*;

// Helper class

class Test {

    public static int i = 0;

    // Constructor to count objects

    Test() {

        i++;

    }

    // Static method to get the

    // number of objects created

    public static int get() {

        return i;

    }

    // Instance method m1 calling

    // another method m2

    public int m1() {

        System.out.println("Inside the method m1");

        this.m2();  // Calling m2 method

        return 1;
```

```
    }

    // Method m2 that prints a message
    public void m2() {
        System.out.println("In method m2");
    }
}

// Main class
class harmans {
    // Main driver method
    public static void main(String[] args) {
        // Creating object of Test class
        Test obj = new Test();
        // Calling m1 method
        int i = obj.m1();
        System.out.println("Control returned after m1: " + i);
        // Get and print the number of objects created
        int o = Test.get();
        System.out.println("No of instances created: " + o);
    }
}
```

Advantages of Methods:

- **Reusability**: Methods allow us to write code once and use it many times.
- **Abstraction**: Methods allow us to abstract away complex logic and provide a simple interface for others to use.
- **Encapsulation:** Allow to encapsulate complex logic and data
- **Modularity**: Methods allow us to break up your code into smaller, more manageable units, improving the modularity of your code.
- **Customization**: Easy to customize for specific tasks.
- **Improved performance**: By organizing your code into well-structured methods, you can improve performance by reducing the amount of code.

Type of method	Description	Example
Void Method	Does not return a value	public void greet()
Method with Params	Takes input parameters	public void add(int a, int b)
Return Method	Returns a value	public int sum(int a, int b)
Overloaded Method	Same method name, different parameters	public int add(int, int) & public int add(int, int, int)
Recursive Method	Calls itself	public int factorial(int n)

Command Line Arguments in Java

Command-line arguments allow you to pass information to a Java program when you run it from the command prompt or terminal. These arguments are passed as strings to the main method in the String[] args array.

Arguments are always passed as strings.You can convert them to other data types (e.g., int, double) if necessary.

Syntax: public static void main(String[] args)

args: An array of strings containing the command-line arguments.

Example:

```
public class CommandLineExample {

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

```
        // Check if arguments are provided
        if (args.length > 0) {
            System.out.println("Command-line arguments:");
            for (String arg : args) {
                System.out.println(arg);
            }
        } else {
            System.out.println("No command-line arguments provided.");
        }
    }
}
```

Run the Java program with arguments:

java CommandLineExample Hello World 123

Output:

Command-line arguments:

Hello

World

123

Variable Arguments in Java (Varargs)

In Java, varargs (variable-length argument lists) allow you to pass a variable number of arguments to a method. Instead of defining a fixed number of parameters, you can use varargs to handle any number of arguments passed to the method.

- type: The data type of the arguments (e.g., int, String).
- varArgs: A special syntax indicating that any number of arguments can be passed.

```
public class VarargsExample {
    // Method that accepts a variable number of integers
```

```
    public static void sum(int... numbers) {
        int sum = 0;
        for (int num : numbers) {
            sum += num;  // Adding each number to sum
        }
        System.out.println("Sum: " + sum);
    }
    public static void main(String[] args) {
        sum(1, 2, 3);          // Passing three arguments
        sum(5, 10, 15, 20);    // Passing four arguments
        sum();                 // Passing no arguments
    }
}
```

Key Points About Varargs:

- Varargs is treated as an array
- Varargs should be the last parameter
- You can pass an array directly to a method with varargs

ARRAY

Concept	Description
Declaration	int[] arr; or int arr[];
Initialization	arr = new int[5]; or int[] arr = {1, 2, 3, 4, 5};
Access Elements	arr[index] (e.g., arr[0] for the first element)
Length of Array	arr.length (gives the number of elements in the array)
Multidimensional Arrays	Arrays of arrays, e.g., int[][] matrix = {{1, 2}, {3, 4}};
Iteration	Use loops (for, foreach) to iterate over elements

Jagged array:

A **Jagged Array** is an array of arrays in which the sub-arrays can have different lengths. Unlike a multidimensional array, where all rows have the same length, jagged arrays are more flexible and allow for varying row sizes.

Characteristics of a Jagged Array:

- It is an array of arrays, where each element is itself an array.
- Sub-arrays can have different sizes.
- Useful for scenarios where data is inherently irregular, such as triangular matrices, sparse matrices, or storing variable-length data.

```
public class JaggedArrayExample {

   public static void main(String[] args) {

  // Declare a jagged array with 3 rows

       int[][] jaggedArray = new int[3][];


       // Initialize each row with different sizes

       jaggedArray[0] = new int[]{1, 2, 3};

       jaggedArray[1] = new int[]{4, 5};
```

```
        jaggedArray[2] = new int[]{6, 7, 8, 9};


        // Print the jagged array
        for (int i = 0; i < jaggedArray.length; i++) {
            for (int j = 0; j < jaggedArray[i].length; j++) {
                System.out.print(jaggedArray[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Applications of Jagged Arrays:

- Storing student grades where different students have different numbers of subjects.
- Representing adjacency lists in graph algorithms.
- Working with hierarchical or sparse data structures.

Array classes:

In Java, the term **Array Classes** typically refers to classes and utilities provided by the Java API to work with arrays. Java's standard arrays (int[], String[], etc.) are simple and fixed in size. However, several classes and utilities enhance the functionality and operations of arrays.

Important Array Classes in Java

1. java.util.Arrays: A utility class that provides static methods for manipulating arrays.

Common Methods:

```
import java.util.Arrays;


public class ArraysDemo {
    public static void main(String[] args) {
        int[] numbers = {5, 3, 8, 1};
```

```
        // Sort an array
        Arrays.sort(numbers);
        System.out.println("Sorted Array: " + Arrays.toString(numbers));

        // Binary search (after sorting)
        int index = Arrays.binarySearch(numbers, 3);
        System.out.println("Index of 3: " + index);

        // Fill array with a specific value
        Arrays.fill(numbers, 9);
        System.out.println("Filled Array: " + Arrays.toString(numbers));

        // Check equality of arrays
        int[] numbersCopy = {9, 9, 9, 9};
        System.out.println("Arrays equal: " + Arrays.equals(numbers, numbersCopy));
    }
}
```

2. java.lang.reflect.Array: This class allows dynamic creation and manipulation of arrays at runtime.

Example: import java.lang.reflect.Array;

```
public class ReflectArrayExample {
    public static void main(String[] args) {
        // Create an array of integers dynamically
        int size = 5;
        Object array = Array.newInstance(int.class, size);
```

```
        // Set values in the array
        Array.set(array, 0, 10);
        Array.set(array, 1, 20);
            // Get and print values
        System.out.println("First Element: " + Array.get(array, 0));
        System.out.println("Second Element: " + Array.get(array, 1));
    }
}
```

3. java.util.ArrayList (Dynamic Array Alternative)

While not technically an array, this class provides a resizable array-like data structure.

Example:

```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<String> names = new ArrayList<>();

        names.add("Alice");
        names.add("Bob");
        System.out.println("ArrayList: " + names);

        names.remove("Bob");
        System.out.println("After removal: " + names);
    }
}
```

STRINGS

In Java, **strings** are objects that represent a sequence of characters. The `String` class is used to create and manipulate strings. Strings in Java are immutable, meaning once a string is created, its value cannot be changed.

1. Creating Strings in Java

There are two main ways to create a string in Java:

a) Using String Literal

When you use a string literal, Java automatically creates a String object and stores it in the **String Pool** for reuse.

String str1 = "Hello"; // String literal

b) Using new Keyword

You can create a string using the new keyword. In this case, the string is created on the heap memory and not in the String Pool.

String str2 = new String("Hello"); // Using new keyword

2. String Methods in Java

The String class in Java provides various built-in methods to work with strings.

Common String Methods:

1. **length()**
    - Returns the length of the string (i.e., the number of characters).

    String str = "Hello";
    System.out.println(str.length()); // Output: 5

2. **charAt(int index)**
    - Returns the character at the specified index.

```
String str = "Hello";
System.out.println(str.charAt(1));  // Output: 'e'
```

3. **substring(int beginIndex) / substring(int beginIndex, int endIndex)**
    - Returns a new string that is a substring of the original string. The beginIndex is inclusive, and endIndex is exclusive.

```
String str = "Hello, World!";
System.out.println(str.substring(7));        // Output: "World!"
System.out.println(str.substring(0, 5));     // Output: "Hello"
```

4. **toUpperCase()**
    - Converts all characters in the string to uppercase.

```
String str = "hello";
System.out.println(str.toUpperCase());  // Output: "HELLO"
```

5. **toLowerCase()**
    - Converts all characters in the string to lowercase.

```
String str = "HELLO";
System.out.println(str.toLowerCase());  // Output: "hello"
```

6. **equals(Object obj)**
    - Compares two strings for equality (case-sensitive).

```
String str1 = "hello";
String str2 = "hello";
System.out.println(str1.equals(str2));  // Output: true
```

7. **equalsIgnoreCase(String other)**
    - Compares two strings for equality, ignoring case differences.

```
String str1 = "Hello";
String str2 = "hello";
System.out.println(str1.equalsIgnoreCase(str2));  // Output: true
```

8. **contains(CharSequence sequence)**
    - Checks if the string contains a specified sequence of characters.

```
String str = "Hello, World!";
System.out.println(str.contains("World"));  // Output: true
```

9. **replace(char oldChar, char newChar)**
    - Replaces all occurrences of a specified character with another character.

```
String str = "Hello, World!";
System.out.println(str.replace('o', '0'));  // Output: "Hell0, W0rld!"
```

10. **trim()**
    - Removes leading and trailing whitespace from the string.

```
String str = "   Hello, World!   ";
System.out.println(str.trim());  // Output: "Hello, World!"
```

11. **split(String regex)**
    - Splits the string into an array of substrings based on a regular expression.

```
String str = "apple,banana,orange";
String[] fruits = str.split(",");
for (String fruit : fruits) {
   System.out.println(fruit);
}
```

Output:

```
apple
banana
orange
```

12. **indexOf(String str)**
    - Returns the index of the first occurrence of the specified substring. If the substring is not found, it returns -1.

```
String str = "Hello, World!";
System.out.println(str.indexOf("World"));  // Output: 7
```

3. String Concatenation

In Java, you can concatenate strings using the + operator or the concat() method.

Using the + Operator:

```
String str1 = "Hello";
String str2 = "World";
String result = str1 + " " + str2;  // Concatenating with space
System.out.println(result);  // Output: "Hello World"
```

Using the concat() Method:

```
String str1 = "Hello";
```

```
String str2 = "World";
String result = str1.concat(" ").concat(str2);  // Concatenating with space
System.out.println(result);  // Output: "Hello World"
```

4. String Immutability

In Java, strings are immutable, which means once a string object is created, its value cannot be changed. If you try to modify a string, a new string object will be created.

Example:

```
String str = "Hello";
str = str + " World";  // A new string object is created
System.out.println(str);  // Output: "Hello World"
```

Even though it seems like we are modifying the string, the original string "Hello" remains unchanged. A new string "Hello World" is created and assigned to str.

5. StringBuilder and StringBuffer

Although strings are immutable in Java, if you need to modify a string multiple times (e.g., in loops), using **StringBuilder** or **StringBuffer** is recommended as they are mutable and more efficient for such operations.

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");
System.out.println(sb);  // Output: "Hello World"
```

- **StringBuilder** is used when thread synchronization is not necessary.
- **StringBuffer** is similar but is thread-safe, i.e., synchronized for use in multi-threaded environments.

Feature	String	StringBuilder	StringBuffer
Introduction	Introduced in JDK 1.0	Introduced in JDK 1.5	Introduced in JDK 1.0
Mutability	Immutable	Mutable	Mutable
Thread Safety	Thread Safe	Not Thread Safe	Thread Safe

Feature	String	StringBuilder	StringBuffer
Memory Efficiency	High	Efficient	Less Efficient
Performance	High(No-Synchronization)	High(No-Synchronization)	Low(Due to Synchronization)
Usage	This is used when we want immutability.	This is used when Thread safety is not required.	This is used when Thread safety is required.

6. String Formatting

You can format strings using the String.format() method, which works similar to printf in C.

Example:

```
int number = 10;
String formattedString = String.format("The value is: %d", number);
System.out.println(formattedString);  // Output: "The value is: 10"
```

You can also use printf() directly:

```
int number = 10;
System.out.printf("The value is: %d%n", number);  // Output: "The value is: 10"
```

7. String Pool in Java

Java maintains a **string pool** to optimize memory usage. When a string literal is created (like "Hello"), it is stored in the string pool. If the same literal is used again in the program, the same object is reused, avoiding the creation of duplicate objects.

Example:

```
String str1 = "Hello";
String str2 = "Hello";
System.out.println(str1 == str2);  // Output: true
```

In the example above, since both strings refer to the same object in the string pool, the comparison str1 == str2 returns true.

OOPS IN JAVA

Object-Oriented Programming or **Java OOPs** concept refers to programming languages that use objects in programming. They use objects as a primary source to implement what is to happen in the code. Objects are seen by the viewer or user, performing tasks you assign. **Object-oriented programming** aims to implement real-world entities like **inheritance**, **hiding**, **polymorphism**, etc. in programming. The main aim of OOPs is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Java Class

A class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. Using classes, you can create multiple objects with the same behavior instead of writing their code multiple times. This includes classes for objects occurring more than once in your code. In general, class declarations can include these components in order:

1. **Modifiers**: A class can be public or have default access
2. **Class name:** The class name should begin with the initial letter capitalized by convention.
3. **Body:** The class body is surrounded by braces, { }.

Java Object

An **Object** is a basic unit of Object-Oriented Programming that represents real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. The objects are what perform your code, they are the part of your code visible to the viewer/user. An object mainly consists of:

1. **State**: It is represented by the attributes of an object. It also reflects the properties of an object.

2. **Behavior**: It is represented by the methods of an object. It also reflects the response of an object to other objects.
3. **Identity**: It is a unique name given to an object that enables it to interact with other objects.
4. **Methods:** A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to **reuse** the code without retyping it, which is why they are considered **time savers**. In Java, every method must be part of some class

```
public class GFG {

    // Properties Declared
    static String Employee_name;
    static float Employee_salary;

    // Methods Declared
    static void set(String n, float p) {
        Employee_name  = n;
        Employee_salary  = p;
    }

    static void get() {
        System.out.println("Employee name is: " +Employee_name );
        System.out.println("Employee CTC is: " + Employee_salary);
    }

    // Main Method
    public static void main(String args[]) {
        GFG.set("Rathod Avinash", 10000.0f);
        GFG.get();
    }
}
```

Constructors in Java

A **constructor** in Java is a special type of method that is called when an object is created. It is used to initialize the state of the object.

Constructors have the following characteristics:

- **Name**: The constructor's name must be the same as the class name.

- **No return type**: Constructors do not have a return type, not even void.
- **Automatically called**: Constructors are automatically invoked when an object is created using the new keyword.
- **Used for initialization**: They are primarily used to set initial values for object attributes.

Types of Constructors in Java

1. **Default Constructor**: A constructor that takes no arguments. If you don't define a constructor, Java provides a default constructor that initializes the object with default values.
2. **Parameterized Constructor**: A constructor that takes arguments. It allows you to set initial values for the object's fields when creating an object.

1. Default Constructor

A **default constructor** is automatically provided by Java if no constructors are explicitly defined in the class. It initializes the object with default values (e.g., `null` for objects, `0` for numeric types).

Example:

```
class Car {
    String model;
    int year;

    // Default constructor
    public Car() {
        model = "Unknown";
        year = 2020;
    }

    void display() {
        System.out.println("Car Model: " + model);
        System.out.println("Car Year: " + year);
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object using the default constructor
        Car myCar = new Car();

        // Displaying the values
        myCar.display();
```

```
    }
}
```

Output:

```
Car Model: Unknown
Car Year: 2020
```

In this example:

- The default constructor sets the model to "Unknown" and year to 2020.
- When the object myCar is created, the default constructor is invoked automatically.

2. Parameterized Constructor

A **parameterized constructor** allows you to pass values to the object when it's created, enabling the object to be initialized with specific values.

Example:

```
class Car {
    String model;
    int year;

    // Parameterized constructor
    public Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    void display() {
        System.out.println("Car Model: " + model);
        System.out.println("Car Year: " + year);
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object using the parameterized constructor
        Car myCar = new Car("Tesla Model S", 2023);

        // Displaying the values
        myCar.display();
    }
}
```

Output:

Car Model: Tesla Model S
Car Year: 2023

In this example:

- The Car class has a **parameterized constructor** that takes two parameters (model and year).
- When the object myCar is created, you pass the values "Tesla Model S" and 2023 to the constructor, initializing the object with those values.

4 Pillars of Java OOPs Concepts

The four main pillars of **Object-Oriented Programming (OOP)** in Java are:

1. **Encapsulation**
2. **Inheritance**
3. **Polymorphism**
4. **Abstraction**

1. Encapsulation

Definition: Encapsulation is the concept of wrapping the data (variables) and methods (functions) that operate on the data into a single unit, i.e., a class. It also restricts direct access to some of the object's components, typically by making fields private and providing public methods (getters and setters) to access and modify those fields.

Why it's important: Encapsulation helps in hiding the internal details of the object and protects the object's state from unauthorized access or modification.

Example of Encapsulation:

```
class Person {
    // Private fields
    private String name;
    private int age;

    // Getter method for 'name'
    public String getName() {
        return name;
```

```
    }

    // Setter method for 'name'
    public void setName(String name) {
        this.name = name;
    }

    // Getter method for 'age'
    public int getAge() {
        return age;
    }

        // Setter method for 'age'
    public void setAge(int age) {
        if (age > 0) {  // Validation
            this.age = age;
        }
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object of Person class
        Person person = new Person();

        // Using setter methods to set values
        person.setName("John");
        person.setAge(25);

        // Using getter methods to access values
        System.out.println("Name: " + person.getName());  // Output: Name: John
        System.out.println("Age: " + person.getAge());    // Output: Age: 25
    }
}
```

In this example:

- The name and age fields are **private** and can't be accessed directly from outside the class.
- **Getter** and **Setter** methods are used to access and modify these private fields.

2. Inheritance

Definition: Inheritance is the mechanism in Java where one class (subclass) acquires the properties and behaviors (fields and methods) of another class (superclass). It promotes code reusability.

Why it's important: Inheritance allows you to define a new class based on an existing class, reducing redundancy in your code.

Example of Inheritance:

```
// Superclass (Parent class)
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}

// Subclass (Child class)
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking");
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object of Dog class
        Dog dog = new Dog();

        // Calling method from the superclass (Animal)
        dog.eat();  // Output: Animal is eating

        // Calling method from the subclass (Dog)
        dog.bark();  // Output: Dog is barking
    }
}
```

In this example:

- The Dog class **inherits** the eat() method from the Animal class, allowing us to reuse that code in the Dog class.
- Dog also has its own method bark().

5. Polymorphism

Definition: Polymorphism means "many forms." It allows an object to behave in multiple ways. In Java, polymorphism can be achieved through **method overloading** (compile-time polymorphism) and **method overriding** (runtime polymorphism).

Why it's important: Polymorphism enables you to use the same method name for different behaviors, making the code more flexible and easier to maintain.

Example of Polymorphism (Method Overloading):

```
class Calculator {
   // Overloaded method to add two integers
   int add(int a, int b) {
      return a + b;
   }

   // Overloaded method to add three integers
   int add(int a, int b, int c) {
      return a + b + c;
   }
}

public class Main {
   public static void main(String[] args) {
      Calculator calculator = new Calculator();

      // Calling the overloaded methods
      System.out.println(calculator.add(5, 10));         // Output: 15
      System.out.println(calculator.add(5, 10, 15));     // Output: 30
   }
}
```

In this example:

- The add() method is **overloaded** to accept different numbers of parameters (2 and 3), demonstrating **compile-time polymorphism**.

Example of Polymorphism (Method Overriding):

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

```
// Subclass (Child class) overrides the method
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        Animal myDog = new Dog();

        // Calling the sound() method on different objects
        myAnimal.sound();  // Output: Animal makes a sound
        myDog.sound();     // Output: Dog barks
    }
}
```

In this example:

- Dog class **overrides** the sound() method of the Animal class, providing its own implementation.
- At **runtime**, the method sound() of the Dog class is called, even though the reference type is Animal. This is **runtime polymorphism**.

6. Abstraction

Definition: Abstraction is the concept of hiding the implementation details and exposing only the essential features of an object. In Java, abstraction can be achieved using **abstract classes** and **interfaces**.

Why it's important: Abstraction allows you to define the structure of a class without worrying about the implementation details. This makes your code more modular and easier to maintain.

Example of Abstraction (Using an Abstract Class):

```
// Abstract class
abstract class Animal {
    // Abstract method (no implementation)
    abstract void sound();

    // Regular method (with implementation)
    void sleep() {
        System.out.println("Animal is sleeping");
```

```
    }
}

// Subclass that provides implementation of the abstract method
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        // Cannot create an object of abstract class
        // Animal animal = new Animal();  // Error: Cannot instantiate the type Animal

        // Creating object of subclass
        Animal dog = new Dog();
        dog.sound();  // Output: Dog barks
        dog.sleep();  // Output: Animal is sleeping
    }
}
```

In this example:

- The Animal class is **abstract**, meaning it can't be instantiated directly.
- The Dog class provides an implementation for the abstract method sound() of the Animal class.
- The sleep() method in Animal is concrete and can be inherited as is.

Summary of the 4 Main Pillars of OOP in Java:

Pillar	Description	Example
Encapsulation	Bundling data and methods, restricting direct access.	Private int age; with getter and setter methods.
Inheritance	A class inherits properties and behaviors from another.	Dog extends Animal
Polymorphism	Same method name, different behavior (overloading/overriding).	Method overloading (same method name, different parameters), Method overriding (same method name, different classes).

Pillar	Description	Example
Abstraction	Hiding implementation details, exposing only essential features.	Abstract class or Interface with abstract methods.

JAVA INTERFACE:

An **Interface in Java** programming language is defined as an abstract type used to specify the behavior of a class. An interface in Java is a blueprint of a behavior. A Java interface contains static constants and abstract methods.

- The interface in Java is *a* mechanism to achieve abstraction.
- By default, variables in an interface are public, static, and final.
- It is used to achieve abstraction and multiple inheritances in Java.
- It is also used to achieve loose coupling.
- In other words, interfaces primarily define methods that other classes must implement.
- An interface in Java defines a set of behaviors that a class can implement, usually representing an IS-A relationship, but not always in every scenario.

```
interface testInterface {

// public, static and final

final int a = 10;

 // public and abstract

void display();

}
```

```
// Class implementing interface

class TestClass implements testInterface {

 // Implementing the capabilities of  Interface

public void display(){

System.out.println("HARMAN");

} }
```

Difference Between Class and Interface

Although Class and Interface seem the same there are certain differences between Classes and Interface. The major differences between a class and an interface are mentioned below:

Class	Interface
In class, you can instantiate variables and create an object.	In an interface, you must initialize variables as they are final but you can't create an object.
A class can contain concrete (with implementation) methods	The interface cannot contain concrete (with implementation) methods.
The access specifiers used with classes are private, protected, and public.	In Interface only one specifier is used- Public.

Multiple Inheritance in Java Using Interface

Multiple Inheritance is an OOPs concept that can't be implemented in Java using classes. But we can use multiple inheritances in Java using Interface. Let us check this with an example.

```
import java.io.*;

// Add interface

interface Add{
```

```
    int add(int a,int b);
}
// Sub interface
interface Sub{
      int sub(int a,int b);
}
// Calculator class implementing
// Add and Sub
class Cal implements Add , Sub
{
      // Method to add two numbers
      public int add(int a,int b){
          return a+b;
    }
      // Method to sub two numbers
      public int sub(int a,int b){
        return a-b;
    }
}

class GFG{
    // Main Method
    public static void main (String[] args)
```

```
    {
        // instance of Cal class
        Cal x = new Cal();

        System.out.println("Addition : " + x.add(2,1));
        System.out.println("Substraction : " + x.sub(2,1));

    }
}
```

New Features Added in Interfaces in JDK 8

There are certain features added to Interfaces in JDK 8 update mentioned below:

1. Default Methods

- Interfaces can define methods with default implementations.
- Useful for adding new methods to interfaces without breaking existing implementations.

2. **Static Methods**

- Interfaces can now include static methods.
- These methods are called directly using the interface name and are not inherited by implementing classes.

Another feature that was added in JDK 8 is that we can now define static methods in interfaces that can be called independently without an object. These methods are not inherited.

Extending Interfaces

One interface can inherit another by the use of keyword extends. When a class implements an interface that inherits another interface, it must provide an implementation for all methods required by the interface inheritance chain.

```
interface A {
```

```
    void method1();
    void method2();
}

// B now includes method1
// and method2
interface B extends A {
    void method3();
}

// the class must implement
// all method of A and B.
class GFG implements B
{
    public void method1() {
        System.out.println("Method 1");
    }

    public void method2() {
        System.out.println("Method 2");
    }
    public void method3() {
        System.out.println("Method 3");
```

```
    }
      public static void main(String[] args){
   // Instance of GFG class created
          GFG x = new GFG();
          // All Methods Called
          x.method1();
          x.method2();
          x.method3();
    }
}
```

New Features Added in Interfaces in JDK 9

From Java 9 onwards, interfaces can contain the following also:

1. Static methods
2. Private methods
3. Private Static methods