SECTION - A

1)Define polymorphism.

Polymorphism in Java refers to the ability of different objects to be treated as instances of the same class, but each behaves differently based on its own implementation. For example, in a banking system, both a SavingsAccount and a CheckingAccount could inherit from an Account superclass. They can each override a common method like `calculateInterest()`, implementing it differently according to their specific account type's rules. This flexibility allows code to handle various types of objects through a unified interface.

2)What do you mean by overloaded constructor?

An overloaded constructor in Java is a constructor within a class that shares the same name as another constructor but differs in terms of the number or types of parameters it accepts. This allows multiple constructors to exist within the same class, each serving different initialization purposes. For example, a class `Car` might have overloaded constructors like `Car(String model)` and `Car(String model, int year)`, allowing objects to be instantiated with different sets of initial attributes based on the constructor used.

3.)How multiple inheritances in implemented in java?

In Java, multiple inheritance (where a class inherits from more than one class) is not directly supported to avoid complexities like the diamond problem. Instead, Java uses interfaces to achieve a form of multiple inheritance. A class can implement multiple interfaces, each defining a set of methods that the class must implement. This way, a class can inherit behavior from multiple sources through interfaces while maintaining a single inheritance hierarchy with its superclass. For example:

```
interface A {
    void methodA();
}

interface B {
    void methodB();
}

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

    public void methodB() {
        // Implementation of methodB
    }
}
```

4)How multiple inheritances in implemented in java?

In Java, multiple inheritance is achieved through interfaces rather than through classes. A class can implement multiple interfaces, thereby inheriting abstract methods from each interface. This approach avoids the complexities associated with multiple inheritance of state (fields) that can lead to the diamond problem in other programming languages. Here’s a concise example:

```
interface A {
    void methodA();
}

interface B {
    void methodB();
}

class MyClass implements A, B {
```

Mahesh

```
    public void methodA() {
        // Implementation of methodA
    }

    public void methodB() {
        // Implementation of methodB
    }
}
```

In this example, `MyClass` implements both interfaces `A` and `B`, inheriting and providing implementations for the methods defined in each interface.

5.)Define exception handling.

Exception handling in Java refers to the mechanism of managing and responding to runtime errors, known as exceptions, in a structured manner. It involves identifying and anticipating potential exceptions that may occur during program execution, and using try-catch blocks to handle these exceptions gracefully. For example, in a file processing application, handling exceptions like `FileNotFoundException` ensures that the program can recover from errors, provide meaningful error messages, and maintain program stability even when unexpected conditions arise during execution.

6.)What do you mean by re-throwing of exception?

Rethrowing an exception in Java refers to the act of catching an exception in a catch block and then throwing it again to be handled by another part of the program or propagated up the call stack. This is often done when additional context or specific handling is needed for a particular type of exception. For example:

```
try {
    // Some code that may throw an exception
    throw new IOException("File not found");
} catch (IOException e) {
    // Perform some handling or logging
    // Then rethrow the exception
    throw e;
}
```

7.)Mention the feature of java.

java features include platform independence, object-oriented principles, strong type checking, automatic memory management (garbage collection), and multi-threading support. It supports robust exception handling, allowing developers to create reliable and fault-tolerant applications. Java's rich standard library provides extensive utility classes for various tasks. It also supports dynamic linking and bytecode that enables secure execution of code over networks. Java's simplicity in syntax and focus on readability makes it widely adopted for enterprise-level applications, web development, mobile applications, and more recently, in cloud-based services and Internet of Things (IoT) devices.

8.)What is java token?

In Java, a token refers to the smallest individual unit of a program that is meaningful to the compiler. Tokens can be classified into several categories, including keywords (like `class`, `public`, `static`), identifiers (names of variables, methods, classes), literals (constant values like numbers or strings), operators (arithmetic, logical operators), separators (such as parentheses, commas), and comments. Each token serves a specific purpose in defining the structure and behavior of Java programs, and the compiler uses them to parse and understand the code during compilation.

9.)Difference between an interface and a class.

Mahesh

An interface in Java is like a blueprint of a class that defines a set of method signatures without implementing them. It serves as a contract for classes that implement it, specifying what methods those classes must implement. Interfaces can't contain instance fields (variables) or constructors.

A class, on the other hand, is a blueprint for creating objects that can have fields (variables) and methods (functions). Classes can implement interfaces and inherit from other classes (except multiple inheritance of classes).

10)What is an exception?

In Java, an exception is an event that disrupts the normal flow of a program's execution. It occurs when a situation arises during runtime that prevents the program from continuing its normal operation. Exceptions can occur due to various reasons such as invalid input, resource unavailability, or logical errors. Examples include `NullPointerException`, `IOException`, and `ArithmeticException`. Exception handling in Java allows programmers to manage and recover from these unexpected situations, ensuring robustness and reliability in their applications.

10)What is oops?

OOPS stands for Object-Oriented Programming System. It is a programming paradigm based on the concept of "objects", which can contain data in the form of fields (attributes or properties) and code in the form of procedures (methods or functions). OOPS allows developers to structure their programs in a way that models real-world entities using classes and objects. Key principles of OOPS include encapsulation, inheritance, polymorphism, and abstraction, which together provide modularity, reusability, and maintainability in software development. Java is an example of an OOPS language.

Mahesh

SECTION B

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

a) Platform Independence and Cross-Platform

Platform independence means that Java code can run on any device or operating system that has the Java Virtual Machine (JVM) installed. This is achieved through Java bytecode, which is compiled from Java source code and interpreted by the JVM, making Java applications "write once, run anywhere" (WORA). Cross-platform capability refers to the ability of software to operate on multiple types of systems and devices. Java's platform independence inherently supports cross-platform functionality, allowing Java applications to be easily transferred and executed on various hardware and software environments without modification.

b) Role of Inheritance in Java

Inheritance in Java allows a new class (subclass) to inherit fields and methods from an existing class (superclass). This promotes code reuse, as common attributes and behaviors can be defined in a superclass and shared among multiple subclasses. Inheritance also supports polymorphism, enabling a subclass to be treated as an instance of its superclass, which allows for flexible and interchangeable use of objects. Inheritance helps in creating hierarchical class structures and simplifies code maintenance and enhancement by enabling the extension and modification of existing classes without altering their original implementation.

12. Describe the JDK, JRE and JVM with a suitable diagram. with real life example

JDK, JRE, and JVM

1. Java Development Kit (JDK)

The JDK is a software development environment used for developing Java applications and applets. It includes the JRE (Java Runtime Environment), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), and other tools needed for Java development.

Real-life example: Think of the JDK as a complete toolkit for a carpenter, containing all the tools needed to build furniture, including the final pieces and the tools to assemble them.

2. Java Runtime Environment (JRE)

The JRE provides the libraries, Java Virtual Machine (JVM), and other components required to run applications written in Java. It does not include development tools like compilers and debuggers.

Real-life example: The JRE is like the finished furniture itself. It's all you need if you just want to use (run) the furniture without building it.

3. Java Virtual Machine (JVM)

The JVM is a part of the JRE. It is an abstract machine that enables your computer to run a Java program. It converts Java bytecode into machine-specific code and is responsible for memory management, garbage collection, and ensuring platform independence.

Real-life example: The JVM can be compared to an assembly line worker who interprets the design plans (bytecode) to create the final product (machine code), ensuring the product works correctly regardless of the factory's location (platform independence).

Diagram

```
+-------------------------------------+
|           Java Development Kit (JDK)          |
|  +--------------------------------+  |
|  |  Java Runtime Environment (JRE)   |  |
|  |  +-----------------------------+  |  |
|  |  |      Java Virtual Machine (JVM)    |  |  |
|  |  +-----------------------------+  |  |
|  +--------------------------------+  |
+-------------------------------------+
```

In summary:

- **JDK**: Complete toolkit for Java development.
- **JRE**: Environment to run Java applications.

Mahesh

- **JVM**: Engine that executes Java bytecode and ensures platform independence.

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

Structure and Function of the `for` Statement in Java

The `for` loop in Java is used to iterate a part of the program multiple times. Its structure consists of three main parts: initialization, condition, and update. The loop executes the block of code repeatedly until the condition is false.

Structure

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

1. **Initialization**: This part is executed only once at the beginning of the loop. It is used to initialize the loop variable.
2. **Condition**: This part is evaluated before each iteration of the loop. If it is true, the loop continues; if false, the loop terminates.
3. **Update**: This part is executed after each iteration. It is typically used to update the loop variable.

Function

- The `for` loop initializes a variable, checks a condition, executes the block of code if the condition is true, and updates the variable after each iteration.
- It provides a concise way of writing the loop structure and is commonly used when the number of iterations is known beforehand.

Example

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

Explanation:

1. **Initialization**: `int i = 1;` sets the starting value of `i` to 1.
2. **Condition**: `i <= 5;` checks if `i` is less than or equal to 5.
3. **Update**: `i++` increments the value of `i` by 1 after each iteration.

Output:

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

Mahesh

In this example, the loop starts with `i` equal to 1 and increments `i` by 1 on each iteration, printing the current value of `i` until `i` exceeds 5.

14. Describe in detail the concept used to implement multiple inheritance in java with a suitable example.150 word limit

In Java, multiple inheritance is implemented using interfaces. Unlike classes, an interface can be implemented by any class, and a class can implement multiple interfaces. This allows a class to inherit behavior from multiple sources, providing a way to achieve multiple inheritance while avoiding the complexities and ambiguities associated with it.

Concept:

1. **Interface Declaration**: Define interfaces with abstract methods.
2. **Class Implementation**: A class implements multiple interfaces, providing concrete implementations for the methods declared in those interfaces.

Example:

```
interface Flyable {
    void fly();
}

interface Swimmable {
    void swim();
}

class Duck implements Flyable, Swimmable {
    public void fly() {
        System.out.println("Duck is flying.");
    }

    public void swim() {
        System.out.println("Duck is swimming.");
    }
}

public class Main {
    public static void main(String[] args) {
        Duck duck = new Duck();
        duck.fly();
        duck.swim();
    }
}
```

Explanation:

- **Flyable and Swimmable**: Two interfaces with abstract methods.
- **Duck Class**: Implements both interfaces, providing definitions for `fly()` and `swim()` methods.
- **Main Class**: Instantiates `Duck` and calls its methods, demonstrating multiple inheritance.

This approach allows `Duck` to inherit and implement behaviors from both `Flyable` and `Swimmable` interfaces, achieving multiple inheritance.

15. Write java program to add two times objects using operator overloading.

Mahesh

In Java, operator overloading is not directly supported as it is in languages like C++. However, we can achieve similar functionality by defining methods that perform operations on objects. To add two `Time` objects, we can create a method that adds the time fields (hours, minutes, and seconds) and returns a new `Time` object.

Here’s how you can implement it:

```
class Time {
    int hours;
    int minutes;
    int seconds;

    // Constructor to initialize time object
    public Time(int hours, int minutes, int seconds) {
        this.hours = hours;
        this.minutes = minutes;
        this.seconds = seconds;
    }

    // Method to add two time objects
    public Time add(Time t) {
        int totalSeconds = this.seconds + t.seconds;
        int totalMinutes = this.minutes + t.minutes + totalSeconds / 60;
        int totalHours = this.hours + t.hours + totalMinutes / 60;

        return new Time(totalHours % 24, totalMinutes % 60, totalSeconds %
60);
    }

    // Method to display time
    public void display() {
        System.out.println(String.format("%02d:%02d:%02d", hours, minutes,
seconds));
    }

    public static void main(String[] args) {
        Time t1 = new Time(10, 45, 30);
        Time t2 = new Time(3, 20, 40);
        Time t3 = t1.add(t2);

        System.out.print("Time 1: ");
        t1.display();
        System.out.print("Time 2: ");
        t2.display();
        System.out.print("Sum of Time: ");
        t3.display();
    }
}
```

Explanation:

1. **Time Class**:
    - `int hours, minutes, seconds`: Fields to store time components.
    - `Time(int hours, int minutes, int seconds)`: Constructor to initialize time objects.
    - `add(Time t)`: Method to add two `Time` objects. It sums up the seconds, minutes, and hours, carrying over the extra values appropriately.
    - `display()`: Method to print the time in `HH:MM:SS` format.
2. **Main Method**:
    - Creates two `Time` objects `t1` and `t2`.

Mahesh

  - Calls `add()` to get the sum of the times as `t3`.
  - Displays `t1`, `t2`, and `t3`.

This approach simulates operator overloading by using a method to add two objects of the `Time` class.

16. Differentiate between method overloading and method overriding.

Method Overloading:

Method overloading occurs when multiple methods in the same class have the same name but different parameters (different type, number, or both). It allows a class to have more than one method with the same name, enhancing readability and usability.

Key Points:

1. **Same Method Name**: Different parameters.
2. **Compile-Time Polymorphism**: Resolved at compile-time.
3. **Example**:

```
java
Copy code
class MathUtil {
    int add(int a, int b) { return a + b; }
    double add(double a, double b) { return a + b; }
}
```

Method Overriding:

Method overriding occurs when a subclass provides a specific implementation for a method already defined in its superclass. It allows a subclass to modify or extend the behavior of a superclass method.

Key Points:

1. **Same Method Signature**: Same name, parameters, and return type.
2. **Runtime Polymorphism**: Resolved at runtime.
3. **Example**:

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

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

Summary:

- **Method Overloading**: Same class, different parameters, compile-time.
- **Method Overriding**: Different classes (inheritance), same signature, runtime.

17. Different between c++ and JAVA .

Differences Between C++ and Java

Mahesh

1. Language Paradigm

- **C++**: Multi-paradigm language supporting both procedural and object-oriented programming.
- **Java**: Purely object-oriented language (everything is an object).

2. Memory Management

- **C++**: Manual memory management using pointers and `new`/`delete` operators.
- **Java**: Automatic garbage collection handles memory management.

3. Platform Independence

- **C++**: Compiled to platform-specific machine code; not inherently platform-independent.
- **Java**: Compiled to bytecode, which runs on the Java Virtual Machine (JVM), making it platform-independent ("write once, run anywhere").

4. Multiple Inheritance

- **C++**: Supports multiple inheritance directly.
- **Java**: Supports multiple inheritance through interfaces only (no direct multiple inheritance of classes).

5. Pointers

- **C++**: Supports direct use of pointers for memory manipulation.
- **Java**: No direct support for pointers; uses references instead, enhancing security and simplicity.

6. Operator Overloading

- **C++**: Supports operator overloading, allowing developers to define custom behavior for operators.
- **Java**: Does not support operator overloading.

7. Exception Handling

- **C++**: Basic support for exception handling with `try`, `catch`, and `throw`.
- **Java**: Robust exception handling with checked and unchecked exceptions, enforcing handling or declaration of exceptions.

8. Standard Library

- **C++**: Standard Template Library (STL) with containers, algorithms, and iterators.
- **Java**: Rich standard library with extensive APIs for data structures, networking, threading, etc.

9. Compilation

- **C++**: Compiled directly to machine code.
- **Java**: Compiled to bytecode, which is interpreted or Just-In-Time (JIT) compiled by the JVM.

Mahesh

Example Code

C++:

```
#include <iostream>
using namespace std;

class HelloWorld {
public:
    void display() {
        cout << "Hello, World!" << endl;
    }
};

int main() {
    HelloWorld hw;
    hw.display();
    return 0;
}
```

Java:

```
public class HelloWorld {
    public void display() {
        System.out.println("Hello, World!");
    }

    public static void main(String[] args) {
        HelloWorld hw = new HelloWorld();
        hw.display();
    }
}
```

Summary:

- **C++**: Offers more control and flexibility with features like pointers and operator overloading but requires manual memory management and lacks inherent platform independence.
- **Java**: Provides a simpler and more secure programming model with automatic memory management and platform independence, but does not support pointers and operator overloading.

SECTION C

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

```
import java.util.Scanner;

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

        // Prompt user for input
        System.out.print("Enter a number: ");
        int number = scanner.nextInt();

        // Check if the number is a palindrome
        if (isPalindrome(number)) {
            System.out.println(number + " is a palindrome.");
        } else {
```

Mahesh

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

        scanner.close();
    }

    // Method to check if a number is a palindrome
    public static boolean isPalindrome(int number) {
        int originalNumber = number;
        int reversedNumber = 0;

        // Reverse the number
        while (number != 0) {
            int digit = number % 10;
            reversedNumber = reversedNumber * 10 + digit;
            number /= 10;
        }

        // Check if original number is equal to reversed number
        return originalNumber == reversedNumber;
    }
}
```

Explanation

1. **Input**:
    - The program prompts the user to enter a number.
    - The number is read using a `Scanner`.
2. **Palindrome Check**:
    - The `isPalindrome` method reverses the given number.
    - It compares the reversed number with the original number.
3. **Output**:
    - If the original number is equal to the reversed number, it prints that the number is a palindrome.
    - Otherwise, it prints that the number is not a palindrome.

Example Run

```
Enter a number: 121
121 is a palindrome.

Enter a number: 123
123 is not a palindrome.
```

This program efficiently checks if a given number is a palindrome by reversing the number and comparing it to the original.

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

```
public class SumOfOddEven {
    public static void main(String[] args) {
        int[] array = { 3, 7, 12, 5, 8, 9, 4, 6, 11 };

        int sumOdd = 0;
        int sumEven = 0;

        for (int num : array) {
            if (num % 2 == 0) {
                sumEven += num; // Adding even numbers to sumEven
```

Mahesh

```
            } else {
                sumOdd += num; // Adding odd numbers to sumOdd
            }
        }

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

Explanation:

1. **Array Initialization**:
    - An array `array` is initialized with some integers.
2. **Variables Initialization**:
    - `sumOdd` and `sumEven` are initialized to zero to store the sums of odd and even numbers, respectively.
3. **Loop through Array**:
    - Using an enhanced for loop (`for-each` loop), each element `num` in the array `array` is checked:
        - If `num % 2 == 0`, it means `num` is even, so add `num` to `sumEven`.
        - Otherwise, add `num` to `sumOdd` because it's odd.
4. **Output**:
    - Finally, print the sums of odd and even numbers using `System.out.println()`.

Example Output:

For the array `{ 3, 7, 12, 5, 8, 9, 4, 6, 11 }`, the output will be:

```
mathematica
Copy code
Sum of odd numbers: 35
Sum of even numbers: 30
```

This program efficiently computes the sum of odd and even numbers separately in an array using a simple loop and conditional statements.

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

```
class ComplexNumber {
    private double real;
    private double imaginary;

    // Default constructor
    public ComplexNumber() {
        this.real = 0.0;
        this.imaginary = 0.0;
    }

    // Parameterized constructor to initialize real and imaginary parts
    public ComplexNumber(double real, double imaginary) {
        this.real = real;
        this.imaginary = imaginary;
    }

    // Method to add two complex numbers
```

Mahesh

```
    public ComplexNumber add(ComplexNumber other) {
        double newReal = this.real + other.real;
        double newImaginary = this.imaginary + other.imaginary;
        return new ComplexNumber(newReal, newImaginary);
    }

    // Method to display complex number in a+bi format
    public void display() {
        System.out.println(real + " + " + imaginary + "i");
    }

    public static void main(String[] args) {
        // Create two complex numbers using different constructors
        ComplexNumber c1 = new ComplexNumber(2.5, 3.0);
        ComplexNumber c2 = new ComplexNumber(1.5, 2.5);

        // Add c1 and c2
        ComplexNumber sum = c1.add(c2);

        // Display the result
        System.out.print("Sum of complex numbers: ");
        sum.display();
    }
}
```

Explanation:

1. **ComplexNumber Class**:
    - `private double real;` and `private double imaginary;` are instance variables to store real and imaginary parts of complex numbers.
2. **Constructors**:
    - `ComplexNumber()`: Default constructor initializes the complex number to `0 + 0i`.
    - `ComplexNumber(double real, double imaginary)`: Parameterized constructor initializes the complex number with given real and imaginary parts.
3. **add Method**:
    - `add(ComplexNumber other)`: Method adds the current complex number (`this`) with another complex number (`other`) and returns a new `ComplexNumber` object representing the sum.
4. **display Method**:
    - `display()`: Method prints the complex number in the format `a + bi`.
5. **Main Method**:
    - Creates two complex numbers `c1` and `c2` using different constructors.
    - Calls `add()` method to compute the sum of `c1` and `c2`.
    - Displays the result using `display()` method.

Example Output:

For `c1 = 2.5 + 3.0i` and `c2 = 1.5 + 2.5i`, the output will be:

```
Sum of complex numbers: 4.0 + 5.5i
```

This program demonstrates constructor overloading in Java to initialize complex numbers and add them using instance methods.

Mahesh

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

Here's a Java program to implement the Fibonacci series using a `while` loop:

```
java
Copy code
public class FibonacciSeries {
    public static void main(String[] args) {
        int n = 10; // Number of terms in Fibonacci series
        int firstTerm = 0, secondTerm = 1;
        int count = 0;

        System.out.println("Fibonacci Series up to " + n + " terms:");

        // Print the first two terms
        System.out.print(firstTerm + " " + secondTerm + " ");

        // Initialize the loop to generate the Fibonacci sequence
        while (count < n - 2) {
            int nextTerm = firstTerm + secondTerm;
            System.out.print(nextTerm + " ");
            firstTerm = secondTerm;
            secondTerm = nextTerm;
            count++;
        }
    }
}
```

Explanation:

1. **Initialization**:
    - `n`: Number of terms to generate in the Fibonacci series.
    - `firstTerm` and `secondTerm`: Initial terms of the Fibonacci sequence.
    - `count`: Variable to count the number of terms generated.
2. **Printing Initial Terms**:
    - Prints the first two terms (`firstTerm` and `secondTerm`) of the Fibonacci series.
3. **While Loop**:
    - Executes until `count` is less than `n - 2` (because we've already printed the first two terms).
    - Calculates the next Fibonacci term (`nextTerm = firstTerm + secondTerm`).
    - Prints `nextTerm`.
    - Updates `firstTerm` and `secondTerm` for the next iteration.
    - Increments `count` to track the number of terms printed.

Example Output:

For `n = 10`, the Fibonacci series will be generated and printed as:

```
mathematica
Copy code
Fibonacci Series up to 10 terms:
0 1 1 2 3 5 8 13 21 34
```

This program uses a `while` loop to generate and print the Fibonacci series up to `n` terms.

21. Write a program to implement Fibonacci series using while control structure.22. What are the various data type available in java? Explain

Mahesh

It seems like you repeated the request for implementing the Fibonacci series using a `while` loop. Here’s the response for the second part of your query about the various data types available in Java:

Data Types in Java

Java supports two categories of data types: primitive data types and reference data types.

1. Primitive Data Types

Primitive data types represent basic types of data that are directly supported by the Java language. They are divided into four groups: integers, floating-point numbers, characters, and boolean values.

- **Integers**: Represent whole numbers without fractional parts.
    - `byte`: 8-bit signed integer (range: -128 to 127)
    - `short`: 16-bit signed integer (range: -32,768 to 32,767)
    - `int`: 32-bit signed integer (range: -2^31 to 2^31 - 1)
    - `long`: 64-bit signed integer (range: -2^63 to 2^63 - 1)
- **Floating-Point Numbers**: Represent numbers with fractional parts.
    - `float`: 32-bit IEEE 754 floating-point (range: approximately ±3.40282347E+38F)
    - `double`: 64-bit IEEE 754 floating-point (range: approximately ±1.79769313486231570E+308)
- **Characters**: Represent single 16-bit Unicode characters.
    - `char`: 16-bit Unicode character (range: '\u0000' to '\uffff')
- **Boolean**: Represent true/false values.
    - `boolean`: Represents true or false values (default value: false)

2. Reference Data Types

Reference data types are used to refer objects. They include classes, interfaces, arrays, and enumerations.

- **Class Types**: User-defined data types.
    - Example: `String`, `Integer`, `ArrayList`, etc.
- **Interface Types**: Similar to classes but cannot be instantiated on their own.
    - Example: `Runnable`, `Comparable`, etc.
- **Array Types**: Collections of similar types of data.
    - Example: `int[]`, `String[]`, etc.
- **Enumeration Types**: Special data types that enable a variable to be a set of predefined constants.
    - Example: `enum Days { MONDAY, TUESDAY, WEDNESDAY, ... }`

Example Usage:

```
java
Copy code
public class DataTypesExample {
    public static void main(String[] args) {
        // Primitive data types
        byte age = 30;
        int salary = 50000;
        double height = 5.8;
```

Mahesh

```
        char gender = 'M';
        boolean isEmployed = true;

        // Reference data types
        String name = "John Doe";
        int[] numbers = { 1, 2, 3, 4, 5 };
        ArrayList<String> cities = new ArrayList<>();
        cities.add("New York");
        cities.add("London");

        System.out.println(name + " is " + age + " years old.");
        System.out.println("Cities: " + cities);
    }
}
```

Summary:

Java provides a rich set of data types to accommodate various types of data, allowing developers to efficiently manipulate and store different kinds of information in their programs. Understanding these data types is essential for effective programming in Java.

Mahesh