1. Introduction to Java

1.1 What is Java?

Java is a high-level, object-oriented programming language developed by **Sun Microsystems** in **1995**, and later acquired by **Oracle Corporation**.
It is designed to be **portable, secure, and efficient**, with a philosophy of **"Write Once, Run Anywhere" (WORA)**, meaning that Java applications can run on any platform that has a **Java Virtual Machine (JVM)**.
Java is widely used for building **enterprise applications, mobile apps (Android), web applications, games**, and **big data technologies**.

1.2 Features of Java

- **Platform Independence**:
  Java code is compiled into **bytecode**, which can run on any system with a **JVM**, regardless of the underlying operating system.
- **Object-Oriented**:
  Everything in Java is considered an **object**, promoting better organization, code reuse, and modularity.
  Core principles: **Encapsulation, Inheritance, Polymorphism, and Abstraction**.
- **Simple and Familiar**:
  Java's syntax is similar to **C/C++**, making it easier to learn for developers familiar with those languages.
  Complex features like **pointers and manual memory management** are eliminated.
- **Secure**:
  Provides built-in security features like **bytecode verification**, and **automatic memory management (Garbage Collection)**, which helps prevent memory leaks and errors.
- **Multithreaded**:
  Java has built-in support for **multithreading**, allowing for concurrent execution of two or more parts of a program for maximum utilization of the CPU.
- **Robust and Reliable**:
  Strong **memory management** and handling of **runtime errors** through **exception handling**.
  Type-checking at both **compile-time** and **runtime** ensures program stability.
- **Distributed**:
  Java is designed to accommodate the **networking capabilities** needed for distributed computing.
  Comes with an extensive set of libraries for **network communication** (e.g., **RMI, EJB**).
- **High Performance**:
  While interpreted languages are generally slower, Java uses techniques like **Just-In-Time (JIT) compilation** to optimize performance.
- **Dynamic**:
  The language supports **dynamic memory allocation**, which reduces memory wastage and improves program performance.

1.3 History of Java

- **James Gosling** and his team, known as the **Green Team**, initiated the Java project in the **early 1990s**.
- It was initially called **"Oak"**, after an oak tree outside Gosling's office, but was later renamed to **Java**, inspired by **Java coffee**.
- The first version, **Java 1.0**, was released in **1995**, and it became one of the most popular programming languages.

1.4 Why Use Java?

- **Wide Industry Usage**:
  Popular in **Android development**, **enterprise applications**, **financial services**, **scientific computing**, and **big data technologies** (e.g., **Hadoop**).
- **Extensive Community and Ecosystem**:
  One of the largest programming communities, with a vast number of **libraries, frameworks**, and **tools** to support development.
- **Backward Compatibility**:
  New versions of Java maintain compatibility with older versions, ensuring long-term support for projects.

In [ ]:

2. Overview of Java

2.1 Java Architecture

- **Java Architecture** consists of three main components: **JVM (Java Virtual Machine)**, **JRE (Java Runtime Environment)**, and **JDK (Java Development Kit)**.

- **JVM**: Executes Java bytecode on any platform, providing platform independence.
- **JRE**: Provides libraries and JVM to run Java applications.
- **JDK**: Includes JRE and development tools (compiler, debugger) for writing and running Java programs.

2.2 Java Development Process

- **Write**: Create source code (`.java` files).
- **Compile**: Convert source code to **bytecode** (`.class` files) using the Java compiler (`javac`).
- **Execute**: Run the bytecode using the **JVM** (`java` command).

2.3 Java Programming Paradigms

- **Object-Oriented Programming (OOP)**: Organizes code around objects and classes.
- **Platform Independence**: Supports **WORA (Write Once, Run Anywhere)** through bytecode.
- **Automatic Memory Management**: Uses **Garbage Collection** to free up memory automatically.

2.4 Basic Structure of a Java Program

- **Class Declaration**: Every Java program contains at least one **class**.
- **Main Method**: The entry point of a Java program: `public static void main(String[] args)` .
- **Comments**: Used for documentation (`//` for single-line, `/* */` for multi-line).

Example Program

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!"); // Prints a message to the console
    }
}
```

In [ ]:

=

```
public class DEMO {

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

1. `public class DEMO`

- **`public`** : This is an access modifier. It indicates that the class `DEMO` is accessible from anywhere in the program, meaning it can be used by other classes and packages.
- **`class`** : This keyword is used to define a new class in Java. A class is a blueprint for creating objects, which are instances of that class.
- **`DEMO`** : The name of the class. In Java, the class name should start with an uppercase letter by convention. The file name should also match the class name if it's a public class, so the file name should be `DEMO.java` in this case.

2. `public static void main(String[] args)`

This line defines the main method, which is the entry point for any Java application. Let's break it down:

- **`public`** : This access modifier indicates that the `main` method can be called from anywhere, including outside the class. It must be `public` because the Java Virtual Machine (JVM) needs to call it to start the program.
- **`static`** : This keyword means that the method belongs to the class rather than an instance of the class. The `main` method is static because it can be called without creating an object of the class. The JVM calls the `main` method directly without needing to create an instance of the class.
- **`void`** : This indicates that the method does not return any value. The `main` method does not need to return anything, as its purpose is to execute the program.
- **`main`** : The name of the method. This is a special method name in Java that is recognized as the starting point for program execution.
- **`String[] args`** : This is the parameter for the `main` method. It represents an array of `String` objects, which can be used to pass command-line arguments to the program. If no arguments are passed, it will be an empty array.

Example

The full code:

```
public class DEMO {

    public static void main(String[] args) {
        System.out.println("Hello, World!");
```

```
    }
}
```

- This code defines a class named `DEMO` with a `main` method.
- When you run this program, the `main` method is executed, and it will print "Hello, World!" to the console.

Why is `main` Method Special?

The `main` method is the entry point of a Java application. When you run a Java program, the JVM looks for this specific method signature to start the execution. If it's not present, the program will not run.

Summary

- `public class DEMO` : Defines a public class named `DEMO`.
- `public static void main(String[] args)` : Defines the main method, which is public, static, returns nothing (void), and takes a `String` array as an argument.

In [ ]:

3. Data Types in Java

3.1 Primitive Data Types

Java has 8 primitive data types used for basic data manipulation:

- **byte**
  - Size: 1 byte (8 bits)
  - Range: -128 to 127
  - Usage: Saves memory in large arrays

```
byte b = 100;
System.out.println("Byte value: " + b);
```

- **short**
  - Size: 2 bytes (16 bits)
  - Range: -32,768 to 32,767
  - Usage: Useful when memory is a concern

```
short s = 30000;
System.out.println("Short value: " + s);
```

- **int**
  - Size: 4 bytes (32 bits)
  - Range: -2,147,483,648 to 2,147,483,647
  - Default type for integer values

```
int i = 100000;
System.out.println("Integer value: " + i);
```

- **long**
  - Size: 8 bytes (64 bits)
  - Range: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
  - Used for large integer values, add `L` or `l` for literals

```
long l = 10000000000L;
System.out.println("Long value: " + l);
```

- **float**
  - Size: 4 bytes (32 bits)
  - Precision: Approximately 7 decimal digits
  - Used for fractional numbers, add `f` or `F` for literals

```
float f = 3.14f;
System.out.println("Float value: " + f);
```

- **double**
  - Size: 8 bytes (64 bits)
  - Precision: Approximately 15 decimal digits
  - Default type for floating-point numbers

```
double d = 3.141592653589793;
System.out.println("Double value: " + d);
```

- **char**
  - Size: 2 bytes (16 bits)
  - Represents a single character using Unicode

```
char c = 'A';
System.out.println("Char value: " + c);
```

- **boolean**
  - Values: `true` or `false`
  - Used for logical conditions

```
boolean isJavaFun = true;
System.out.println("Is Java fun? " + isJavaFun);
```

3.2 Non-Primitive Data Types

Non-primitive types are created by the programmer:

- **String**
  - Represents a sequence of characters

```
String greeting = "Hello, Java!";
System.out.println(greeting);
```

- **Arrays**
  - Holds multiple values of the same type

```
int[] numbers = {1, 2, 3, 4, 5};
System.out.println("First element: " + numbers[0]);
```

- **Classes**
  - Blueprint for creating objects

```
class Car {
    String model = "Tesla";
    int year = 2024;
}

Car myCar = new Car();
System.out.println("Car model: " + myCar.model);
```

3.3 Type Conversion

Java supports two types of conversions:

- **Widening Conversion (Automatic)**
  - Converts a smaller type to a larger type

```
int num = 100;
double largeNum = num; // int to double
System.out.println("Widened value: " + largeNum);
```

- **Narrowing Conversion (Explicit)**
  - Converts a larger type to a smaller type (requires casting)

```
double decimal = 9.8;
int integerPart = (int) decimal; // double to int
System.out.println("Narrowed value: " + integerPart);
```

Example: Combining Data Types

```
int age = 25;
float height = 5.9f;
String name = "John";
boolean isStudent = true;

System.out.println("Name: " + name);
System.out.println("Age: " + age);
System.out.println("Height: " + height + " feet");
System.out.println("Is a student? " + isStudent);
```

In [ ]:

4. Variables and Arrays in Java

4.1 Variables

Variables are containers for storing data values. In Java, every variable must be declared before it can be used. A variable's type determines what kind of data it can hold.

Types of Variables

- **Local Variables**
  - Declared inside a method or block.
  - Accessible only within that method or block.

```
public void myMethod() {
    int localVar = 10; // Local variable
    System.out.println("Local variable: " + localVar);
}
```

- **Instance Variables**
  - Declared inside a class but outside any method.
  - Associated with an instance of the class and can be accessed by all methods.

```
class MyClass {
    int instanceVar = 20; // Instance variable

    public void display() {
        System.out.println("Instance variable: " + instanceVar);
    }
}
```

- **Static Variables**
  - Declared with the `static` keyword inside a class.
  - Shared among all instances of the class.

```
class MyClass {
    static int staticVar = 30; // Static variable

    public void display() {
        System.out.println("Static variable: " + staticVar);
    }
}
```

4.2 Arrays

Arrays are used to store multiple values of the same type in a single variable. They are fixed in size, meaning you cannot change the length after creation.

Declaring and Initializing Arrays

- **Declaration**

```
int[] myArray; // Declare an array
```

- **Initialization**

```
myArray = new int[5]; // Initialize an array of size 5
```

- **Declaration and Initialization in One Line**

```
int[] myArray = new int[5]; // Declare and initialize in one line
```

- **Array Initialization with Values**

```
int[] myArray = {1, 2, 3, 4, 5}; // Initialize with values
```

Accessing Array Elements

Array elements are accessed using their index (0-based).

```
int[] myArray = {1, 2, 3, 4, 5};
System.out.println("First element: " + myArray[0]); // Accessing the first element
```

Example: Using Variables and Arrays

```
class Example {
    public static void main(String[] args) {
        // Local variable
        int number = 5;

        // Instance variable
        Example obj = new Example();
        obj.instanceVar = 10;

        // Static variable
        MyClass.staticVar = 20;

        // Array
```

```
        int[] numbers = {10, 20, 30, 40, 50};

        System.out.println("Local variable: " + number);
        System.out.println("Instance variable: " + obj.instanceVar);
        System.out.println("Static variable: " + MyClass.staticVar);
        System.out.println("Array element: " + numbers[2]); // Accessing third element
    }

    int instanceVar; // Instance variable
}
```

In [ ]: