JAVA Assignment 1

Introduction to Java

1. What is Java? Explain its significance in modern software development.

Java is a high-level, object-oriented, and platform-independent programming language developed by Sun Microsystems (now owned by Oracle) in 1995. It follows the principle of "Write Once, Run Anywhere" (WORA), meaning that Java programs can run on any platform with a Java Virtual Machine (JVM).

Significance in Modern Software Development:

- **Cross-Platform Compatibility:** Java's platform independence allows applications to run on various devices without modification.
- **Robust and Secure:** Java offers strong memory management, exception handling, and security features.
- **Scalability:** Java supports multi-threading and distributed computing, making it ideal for enterprise applications.
- **Large Ecosystem:** Java has a vast set of libraries, frameworks (Spring, Hibernate), and community support, making it a dominant language in software development.
- **Used in Various Domains:** Java is widely used in web development, mobile applications (Android), enterprise applications, and cloud computing.

---

2. Key Features of Java

1. **Object-Oriented** – Everything in Java is based on objects and classes, promoting modular and reusable code.
2. **Platform-Independent** – Java programs run on any operating system with a JVM.
3. **Simple & Easy to Learn** – Java eliminates complexities like pointers and direct memory management.
4. **Secure** – Java has built-in security features, such as bytecode verification and runtime checking.
5. **Robust** – Strong memory management, exception handling, and automatic garbage collection reduce runtime errors.
6. **Multi-threaded** – Java allows concurrent execution of multiple threads, improving performance in applications.
7. **Portable** – Java code can be executed on different platforms without modification.
8. **High Performance** – Although Java is interpreted, Just-In-Time (JIT) compilation optimizes performance.

9. **Dynamic & Extensible** – Java can dynamically link new libraries and classes during runtime.
10. **Distributed Computing Support** – Java provides APIs (like RMI and EJB) to build distributed applications.

---

3. Compiled vs. Interpreted Languages & Java's Position

- **Compiled Language:** Converts the entire source code into machine code before execution (e.g., C, C++).
- **Interpreted Language:** Translates the code line-by-line at runtime (e.g., Python, JavaScript).

Where Does Java Fit?
Java is a **hybrid language** because:

1. Java source code is **compiled** into an intermediate form called **bytecode** (.class file).
2. The bytecode is **interpreted** and executed by the JVM, making Java platform-independent.
3. To optimize performance, Java uses **JIT (Just-In-Time) Compilation**, which translates bytecode into native machine code at runtime.

---

4. Platform Independence in Java

Java achieves platform independence through its **JVM-based architecture**:

1. **Compilation to Bytecode:** Java source code (.java) is compiled into bytecode (.class) by the Java Compiler.
2. **Execution on Any OS:** The bytecode runs on the JVM, which interprets it for the underlying operating system (Windows, macOS, Linux, etc.).
3. **JVM-Specific Implementations:** Different OSes have their own JVM implementations, ensuring Java applications run consistently across platforms.

Thus, Java adheres to the **"Write Once, Run Anywhere"** (WORA) principle.

---

5. Real-World Applications of Java

1. **Web Applications** – Java frameworks like Spring and Hibernate are used for scalable web applications.

2. **Enterprise Applications** – Java EE powers large-scale business software like banking and ERP systems.
3. **Mobile Development** – Android apps are primarily built using Java (via Android SDK).
4. **Cloud Computing** – Java is widely used in cloud-based applications and microservices.
5. **Big Data & Analytics** – Java tools like Apache Hadoop and Apache Spark process large datasets.
6. **Embedded Systems** – Java is used in IoT devices, smart cards, and automation systems.
7. **Gaming Applications** – Popular games and engines (like Minecraft and LibGDX) use Java.
8. **Scientific Computing** – Java is used in simulations, data analysis, and research applications.

Java's versatility, reliability, and performance make it a fundamental language for modern software development across various domains.

History of Java

1. Who Developed Java and When Was It Introduced?

Java was developed by **James Gosling** and his team at **Sun Microsystems** in **1991**. It was officially released in **1995** as Java 1.0. The goal was to create a language that could run on various devices and platforms without modification.

---

2. What Was Java Initially Called? Why Was Its Name Changed?

Java was initially called **"Oak"**, named after an oak tree outside James Gosling's office. However, the name had to be changed because **"Oak" was already trademarked** by another company. The team later chose **"Java"**, inspired by Java coffee, reflecting the language's dynamism and energy.

---

3. Evolution of Java Versions

Since its launch, Java has undergone multiple updates, bringing improvements in performance, security, and functionality.

Version	Year	Key Features
Java 1.0	1995	Initial release with applets and WORA (Write Once, Run Anywhere).

Version	Year	Key Features
Java 1.2 (Java 2)	1998	Introduced Swing, Collections Framework, and improved performance.
Java 5	2004	Added generics, enhanced for-loop, and autoboxing/unboxing.
Java 6	2006	Performance improvements and scripting API support.
Java 7	2011	Introduced try-with-resources, improved exception handling, and NIO.2 API.
Java 8	2014	Introduced **Lambdas**, **Streams**, and the **Date & Time API**.
Java 9	2017	Introduced **Modules (JPMS)**, JShell (REPL), and HTTP/2 support.
Java 10	2018	Added **var** keyword for local variable type inference.
Java 11	2018	LTS release; introduced new HTTP Client API and String enhancements.
Java 14	2020	Introduced **Records**, **Pattern Matching**, and better NullPointerException messages.
Java 17	2021	LTS release; introduced **sealed classes**, enhanced switch expressions.
Java 21	2023	LTS release with **Virtual Threads (Project Loom)** and Record Patterns.

---

4. Major Improvements in Recent Java Versions

Recent versions of Java have focused on **performance, security, and developer productivity**. Some key improvements include:

1. **Virtual Threads (Java 21)** – Enhances concurrency performance using lightweight threads.
2. **Pattern Matching (Java 14-21)** – Simplifies conditional logic and improves readability.
3. **Sealed Classes (Java 17)** – Controls class inheritance, improving security and design.
4. **Records (Java 14-16)** – Simplifies data classes, reducing boilerplate code.
5. **Garbage Collection Enhancements** – Improved memory management with **ZGC** and **Shenandoah GC**.

6. **New String Methods** – Methods like stripIndent(), transform(), and text blocks simplify string manipulation.
7. **Performance Improvements** – Continuous optimizations in JIT compilation and memory usage.

---

5. Java vs. C++ vs. Python: Evolution and Usability

Feature	**Java**	**C++**	**Python**
Paradigm	Object-Oriented, Multi-paradigm	Procedural & Object-Oriented	Multi-paradigm
Memory Management	Automatic (Garbage Collection)	Manual (requires explicit deallocation)	Automatic (Garbage Collection)
Platform Independence	Yes (JVM)	No (Compiled to native code)	Yes (Interpreted)
Performance	High (JIT Compiler)	Very High (Compiled)	Slower than Java & C++
Ease of Learning	Moderate	Complex (pointers, manual memory management)	Easiest (concise syntax)
Concurrency	Strong (Virtual Threads, Multithreading)	Complex (requires explicit management)	Limited (GIL in CPython)
Use Cases	Web, Mobile, Enterprise Apps	Game Development, Systems Programming	Data Science, AI, Web Dev

Summary:

- **Java** is **platform-independent**, secure, and widely used in enterprise and Android development.
- **C++** is **faster** but requires **manual memory management**, making it complex for beginners.
- **Python** is **easier to learn** and great for **AI, data science, and scripting**, but it's slower than Java and C++.

Data Types in Java

1. Importance of Data Types in Java

Data types in Java are crucial because they define the **size, type, and operations** that can be performed on a variable. Their importance includes:

- **Memory Efficiency** – Allocates appropriate memory for variables.
- **Type Safety** – Prevents unintended operations (e.g., adding a number to a string).
- **Code Readability & Maintainability** – Clearly specifies the kind of data stored.
- **Performance Optimization** – Allows the compiler to optimize execution based on known types.

---

2. Primitive vs. Non-Primitive Data Types

Feature	**Primitive Data Types**	**Non-Primitive Data Types**
Definition	Basic data types directly supported by Java.	User-defined or complex types that store multiple values.
Stored in	Stack memory	Heap memory (reference stored in stack)
Examples	int, char, double	String, Array, Class, Interface
Operations	Directly manipulated	Accessed via methods and references
Memory Usage	Lightweight	Heavier due to references and methods
Nullability	Cannot be null	Can be assigned null

---

3. The Eight Primitive Data Types in Java

Data Type	**Size**	**Default Value**	**Description**
byte	1 byte	0	Stores small integers (-128 to 127).
short	2 bytes	0	Stores medium-range integers (-32,768 to 32,767).
int	4 bytes	0	Stores standard integers (-2^{31} to 2^{31}-1).
long	8 bytes	0L	Stores large integers (-2^{63} to 2^{63}-1).
float	4 bytes	0.0f	Stores decimal numbers (single precision).
double	8 bytes	0.0d	Stores decimal numbers (double precision).

Data Type	Size	Default Value	Description
char	2 bytes	'\u0000'	Stores a single character (Unicode).
boolean	1 bit	false	Stores true or false.

4. Examples of Declaring and Initializing Data Types

```
// Primitive Data Types
byte age = 25;
short year = 2024;
int population = 1000000;
long distance = 1234567890123L;
float temperature = 36.5f;
double price = 99.99;
char grade = 'A';
boolean isJavaFun = true;
// Non-Primitive Data Types
String name = "Java Programming";
int[] numbers = {1, 2, 3, 4, 5};
```

5. Type Casting in Java

Type casting is the process of converting one data type into another.

Types of Type Casting:

1. **Implicit (Widening Casting):** Smaller to larger type (automatically done).
2. **Explicit (Narrowing Casting):** Larger to smaller type (requires manual conversion).

Example:

```
public class TypeCastingExample {
  public static void main(String[] args) {
    // Implicit Casting (Widening)
    int num = 10;
    double doubleNum = num; // Automatic conversion
```

```
        // Explicit Casting (Narrowing)
        double price = 99.99;
        int intPrice = (int) price; // Manual conversion
        System.out.println("Widened Value: " + doubleNum);
        System.out.println("Narrowed Value: " + intPrice);
    }
}
```

Output:

```
Widened Value: 10.0
Narrowed Value: 99
```

6. Wrapper Classes and Their Usage

Wrapper classes are **object representations** of primitive data types. They are used to:

- Store primitive values as objects (needed for collections like ArrayList).
- Provide utility methods for data type conversions.
- Enable features like null values for primitive types.

Primitive Type	**Wrapper Class**
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

```
public class WrapperExample {
    public static void main(String[] args) {
        int num = 10;
        Integer obj = num; // Autoboxing
        int newNum = obj; // Unboxing

        System.out.println("Integer Object: " + obj);
        System.out.println("Primitive Value: " + newNum);
    }
}
```

7. Static vs. Dynamic Typing & Java's Position

Feature	**Static Typing**	**Dynamic Typing**
Definition	Data types are checked at compile time.	Data types are determined at runtime.
Examples	Java, C, C++	Python, JavaScript, Ruby
Error Detection	Errors caught at compile time.	Errors appear at runtime.
Flexibility	Less flexible but safer.	More flexible but riskier.
Performance	Faster (optimized at compile time).	Slower (type checking at runtime).

Where Does Java Stand?
Java is a **statically typed language** because:

- Variables must be declared with a data type before use.
- Type errors are detected at **compile time**, reducing runtime failures.
- Provides **better performance** and **type safety** compared to dynamically typed languages.

Coding Questions on Data Types:

1. Java Program to Declare and Initialize All Eight Primitive Data Types

```
public class PrimitiveDataTypes {
    public static void main(String[] args) {
        byte b = 10;
        short s = 100;
        int i = 1000;
        long l = 100000L;
        float f = 10.5f;
        double d = 99.99;
        char c = 'A';
        boolean bool = true;
        System.out.println("Byte: " + b);
        System.out.println("Short: " + s);
        System.out.println("Int: " + i);
        System.out.println("Long: " + l);
        System.out.println("Float: " + f);
        System.out.println("Double: " + d);
        System.out.println("Char: " + c);
        System.out.println("Boolean: " + bool);
    }
}
```

2. Java Program for Arithmetic Operations

```
import java.util.Scanner;

public class ArithmeticOperations {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter first number: ");
        int num1 = scanner.nextInt()
        System.out.print("Enter second number: ");
        int num2 = scanner.nextInt();
        System.out.println("Addition: " + (num1 + num2));
        System.out.println("Subtraction: " + (num1 - num2));
        System.out.println("Multiplication: " + (num1 * num2));
        System.out.println("Division: " + (num1 / num2));
        System.out.println("Modulus: " + (num1 % num2));
        scanner.close();
    }
}
```

3. Java Program to Demonstrate Implicit and Explicit Type Casting

```
public class TypeCastingDemo {
    public static void main(String[] args) {
        // Implicit Type Casting (Widening)
        int num = 100;
        double doubleNum = num;  // Automatic conversion
        System.out.println("Implicit Casting (int to double): " + doubleNum);

        // Explicit Type Casting (Narrowing)
        double price = 99.99;
        int intPrice = (int) price;  // Manual conversion
        System.out.println("Explicit Casting (double to int): " + intPrice);
    }
}
```

3. Java Program to Convert Integer to Double and Vice Versa Using Wrapper Classes

```
public class WrapperConversion {
    public static void main(String[] args) {
        // Integer to Double
        Integer intValue = 100;
        Double doubleValue = intValue.doubleValue();
        System.out.println("Integer to Double: " + doubleValue);
        // Double to Integer
        Double doubleNum = 55.55;
        Integer intNum = doubleNum.intValue();
        System.out.println("Double to Integer: " + intNum);
    }
}
```

4. Java Program to Convert Integer to Double and Vice Versa Using Wrapper Classes

java

```
public class WrapperConversion {
    public static void main(String[] args) {
        // Integer to Double
        Integer intValue = 100;
        Double doubleValue = intValue.doubleValue();
        System.out.println("Integer to Double: " + doubleValue)
        // Double to Integer
        Double doubleNum = 55.55;
        Integer intNum = doubleNum.intValue();
        System.out.println("Double to Integer: " + intNum);
    }
}
```

5. Java Program to Swap Two Numbers

```
import java.util.Scanner;
```

Using a Temporary Variable

```
public class SwapWithTemp {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter first number: ");
        int a = scanner.nextInt();
        System.out.print("Enter second number: ");
        int b = scanner.nextInt();
        int temp = a;
        a = b;
        b = temp;
        System.out.println("After swapping: a = " + a + ", b = " + b);
        scanner.close();
    }
}
```

Without Using a Temporary Variable

```
import java.util.Scanner;
public class SwapWithoutTemp {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter first number: ");
        int a = scanner.nextInt();
        System.out.print("Enter second number: ");
        int b = scanner.nextInt();
        a = a + b;
        b = a - b;
        a = a - b;
        System.out.println("After swapping: a = " + a + ", b = " + b);
        scanner.close();
    }
}
```

6. Java Program to Check if a Character is a Vowel or Consonant

```
import java.util.Scanner;
public class VowelOrConsonant {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a character: ");
        char ch = scanner.next().toLowerCase().charAt(0);
        if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u') {
            System.out.println(ch + " is a vowel.");
        } else if ((ch >= 'a' && ch <= 'z')) {
            System.out.println(ch + " is a consonant.");
        } else {
            System.out.println("Invalid input! Please enter a letter.");
        }
        scanner.close();
    }
}
```

7. Java Program to Check Even or Odd Using Command-Line Arguments

```
public class EvenOddChecker {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Please provide a number as a command-line argument.");
            return;
        }
        int num = Integer.parseInt(args[0]);
        if (num % 2 == 0) {
            System.out.println(num + " is even.");
        } else {
            System.out.println(num + " is odd.");
        }
    }
}
```

How to Run This Program from the Command Line

```
javac EvenOddChecker.java
java EvenOddChecker 7
```

Output:

```
7 is odd.
```

Java Development Kit (JDK)

1. What is JDK? How Does It Differ from JRE and JVM?

JDK (Java Development Kit)

- JDK is a software development kit required to develop Java applications.
- It includes the **JRE (Java Runtime Environment)** and **development tools** like the compiler (javac), debugger, and libraries.
- It is used by developers to **write, compile, and debug** Java programs.

JRE (Java Runtime Environment)

- JRE is **only for running Java applications**; it does not include development tools.
- It consists of the **JVM (Java Virtual Machine)** and libraries necessary for execution.

JVM (Java Virtual Machine)

- JVM is an **abstract machine** that executes Java bytecode.
- It makes Java **platform-independent** by interpreting bytecode into machine-specific instructions.

Key Differences:

Feature	JDK	JRE	JVM
Purpose	Development (compile & run Java code)	Running Java applications	Executes Java bytecode
Includes	JRE + development tools	JVM + libraries	Just the execution engine
Used By	Developers	End users	Internally by JRE

2. Main Components of JDK

The JDK consists of several tools and libraries:

1. **Java Compiler (javac)** – Converts Java source code into bytecode.
2. **Java Virtual Machine (java)** – Executes compiled Java programs.
3. **Java Standard Library** – Provides essential classes and APIs.
4. **Debugger (jdb)** – Helps debug Java applications.
5. **JAR Tool (jar)** – Packages multiple Java files into a .jar archive.
6. **JavaDoc (javadoc)** – Generates documentation from source code comments.
7. **Keytool (keytool)** – Manages security certificates.

3. Steps to Install JDK and Configure Java

Windows Installation:

1. **Download JDK** from Oracle or OpenJDK.
2. Run the installer and follow the on-screen instructions.
3. Set up the **environment variables**:
   - **PATH**: Add C:\Program Files\Java\jdk-XX\bin to PATH.
   - **CLASSPATH**: Include . (current directory) and required .jar files.

Verify installation by running

java -version

javac -version

Linux/macOS Installation:

1. Install via package manager:

```
sudo apt install openjdk-XX-jdk   # Ubuntu/Debian
brew install openjdk              # macOS
```

Configure **environment variables** in .bashrc or .zshrc

```
export JAVA_HOME=/usr/lib/jvm/java-XX-openjdk
export PATH=$JAVA_HOME/bin:$PATH
```

Verify with java -version and javac -version.

4. Simple Java Program to Print "Hello, World!"

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

Program Structure Explanation:

- **public class HelloWorld** – Defines a class named HelloWorld.
- **public static void main(String[] args)** – The entry point of a Java application.
- **System.out.println("Hello, World!");** – Prints output to the console.

5. Significance of PATH and CLASSPATH in Java

Variable	**Purpose**
PATH	Tells the OS where to find Java executables like java and javac.
CLASSPATH	Specifies where Java should look for class files and libraries.

Example (Windows):

```
set PATH=C:\Program Files\Java\jdk-XX\bin;%PATH%
set CLASSPATH=.;C:\myLibs\library.jar
```

6. Differences Between OpenJDK and Oracle JDK

Feature	OpenJDK	Oracle JDK
License	Open-source (GPL)	Commercial, requires subscription for production use
Updates	Community-driven	Official Oracle updates
Performance	Similar to Oracle JDK	May include proprietary optimizations
Support	Community support	Paid professional support available

7. How Java Programs Are Compiled and Executed

Compilation Steps:

1. **Write Code** – Create a .java file.
2. **Compile** – Use javac to compile

```
javac HelloWorld.java
```

This produces HelloWorld.class (bytecode).

Execute – Run using java

```
java HelloWorld
```

The JVM interprets and runs the bytecode.

8. What is Just-In-Time (JIT) Compilation?

- JIT **compiles bytecode into native machine code at runtime**, improving performance.
- Located inside the JVM, it speeds up execution by avoiding repeated interpretation.
- Uses **hotspot analysis** to optimize frequently used code.

Example:

- **Without JIT**: The JVM interprets each instruction repeatedly.
- **With JIT**: Frequently used methods are compiled into native code, reducing execution time.

9. Role of Java Virtual Machine (JVM) in Program Execution

JVM Responsibilities:

1. **Loads Bytecode** – Reads .class files.

2. **Verifies Code –** Ensures bytecode follows Java's security rules.
3. **Interprets or Compiles Bytecode –** Uses an interpreter or JIT for execution.
4. **Manages Memory –** Handles garbage collection.
5. **Ensures Platform Independence –** Converts bytecode into machine-specific instructions.

JVM Architecture:

Component	**Function**
Class Loader	Loads Java class files into memory.
Bytecode Verifier	Checks for security and correctness.
Interpreter	Translates bytecode into machine code line by line.
JIT Compiler	Compiles frequently used code into native machine code for faster execution.
Garbage Collector	Manages automatic memory deallocation.