Name:117_Swapnil Dhotre_KH

OOPJ Assignment No.3

1) Components of the JDK (Java Development Kit)

The JDK is a complete software development environment used for building Java applications. It consists of:

- **Java Compiler (javac)**: Converts Java source code into bytecode.
- **Java Runtime Environment (JRE)**: Provides the libraries, JVM, and other components required to run Java applications.
- **Java Virtual Machine (JVM)**: Executes Java bytecode on different platforms.
- **Java Debugger (jdb)**: Helps developers debug Java programs.
- **Java API (Application Programming Interface)**: Pre-built libraries and tools for development.
- **Java Documentation Generator (javadoc)**: Generates HTML documentation from Java code.
- **Other Tools**: JAR (Java Archive) tools, utilities for version control, and more.

2) Difference Between JDK, JVM, and JRE

- **JDK (Java Development Kit)**: A superset of JRE that includes tools for developing, compiling, and debugging Java applications. It includes the JVM and the JRE.
- **JVM (Java Virtual Machine)**: A virtual machine that runs Java bytecode. It abstracts the underlying operating system, allowing Java code to be platform-independent.
- **JRE (Java Runtime Environment)**: Includes the JVM and standard libraries required to run Java applications. It lacks development tools like a compiler.

3) Role of JVM in Java & How it Executes Code

- **Role**: The JVM is responsible for running Java applications by converting bytecode into machine code suitable for the host operating system.
- **Execution Process**:
    1. **Class Loading**: The ClassLoader loads the .class files (bytecode) into the JVM.
    2. **Bytecode Verification**: The bytecode verifier checks the code for any security violations or incorrect code.
    3. **Execution**: The JVM executes bytecode using an interpreter or Just-In-Time (JIT) compiler, converting it to native machine code for execution.

4) Memory Management System of the JVM

The JVM has a well-defined memory model:

- **Heap Memory**: Stores objects and instance variables. It's divided into the following regions:
    - **Young Generation**: Newly created objects.
    - **Old Generation**: Long-surviving objects.
    - **Permanent Generation (MetaSpace)**: Stores metadata about classes.
- **Stack Memory**: Stores method frames, local variables, and references to objects. Each thread has its own stack.
- **Garbage Collection**: Automatically removes unused or unreachable objects from heap memory.

5) JIT Compiler and Bytecode Importance in Java

- **JIT Compiler (Just-In-Time)**: Part of the JVM that converts bytecode to native machine code during runtime, improving performance by optimizing frequently executed code.

- **Bytecode**: An intermediate representation of Java code that is platform-independent. It allows Java to be written once and executed anywhere by the JVM.

6) Architecture of the JVM

- **Class Loader Subsystem**: Loads and links Java classes.
- **Memory Area**: Divided into the heap, stack, method area (for class data), and native method stacks.
- **Execution Engine**: Consists of:
  - **Interpreter**: Executes bytecode line by line.
  - **JIT Compiler**: Optimizes and compiles bytecode to native machine code.
  - **Garbage Collector**: Manages memory and deallocates unused objects.
- **Native Method Interface (JNI)**: Allows the execution of native code within Java.
- **Runtime Data Areas**: Includes heap, stack, method area, and more.

7) How Java Achieves Platform Independence via the JVM

Java code is compiled into platform-independent bytecode by the compiler. This bytecode is executed by the JVM, which translates it into platform-specific machine code at runtime. Since every platform has its own JVM implementation, the same bytecode can run on any operating system without modification.

8) Significance of the Class Loader in Java & Garbage Collection Process

- **Class Loader**: A part of the JVM that loads Java classes into memory during runtime. It ensures that classes are loaded only when required and handles dynamic loading of classes.
- **Garbage Collection**: JVM's automatic process of reclaiming memory by deallocating objects that are no longer reachable from any references. It ensures efficient memory management and prevents memory leaks.

9) Four Access Modifiers in Java

- **public**: Accessible from anywhere in the program.
- **protected**: Accessible within the same package and subclasses (even in different packages).
- **default (package-private)**: Accessible only within the same package.
- **private**: Accessible only within the same class.

10) Difference Between Public, Protected, and Default Access Modifiers

- **public**: Visible to all classes, everywhere.
- **protected**: Visible to classes in the same package and subclasses.
- **default (package-private)**: Visible only to classes in the same package.

11) Can You Override a Method with a Different Access Modifier?

Yes, a method can be overridden with a different access modifier, but with constraints. The access level in the subclass must be more permissive than in the superclass:

- **Example**: A protected method in a superclass cannot be overridden by a private method in a subclass, as it would reduce visibility.

12) Difference Between Protected and Default (Package-Private) Access

- **protected**: Accessible within the same package and by subclasses, even if they are in different packages.
- **default (package-private)**: Accessible only within the same package, but not by subclasses in different packages.

13) Is It Possible to Make a Class Private in Java?

You can make inner classes private, but **top-level classes** cannot be private. Inner classes (nested within another class) can be private to restrict their visibility outside the enclosing class.

14) Can a Top-Level Class Be Declared as Protected or Private?

No, a top-level class in Java cannot be declared as **protected** or **private**. Top-level classes can only be declared with **public** or **default** access. This restriction ensures that classes are either visible to all or only to the package.

15) What Happens if You Declare a Variable/Method as Private and Try to Access It from Another Class?

If you declare a variable or method as **private** in a class, it will not be accessible from any other class, even within the same package. You will encounter a compilation error.

16) Concept of "Package-Private" (Default) Access

> **Package-private** access (also known as **default** access) means that members (fields, methods, etc.) are accessible only by classes in the same package. It’s the default access level when no access modifier is specified. This access level restricts visibility to the package without making it fully private.