Assignment 3

04/09/24

Q1. Components of JDK. Explain

A1. JDK is a software development environment used for developing Java applications. It contains several components:

1. Java Compiler (javac)
   - Converts Java source code into bytecode that the Java Virtual Machine (JVM) can execute.
   - The bytecode is platform-independent, allowing Java programs to run on any system with a compatible JVM.
2. Java Runtime Environment (JRE)
   - A subset of the JDK that provides the necessary environment to run Java applications.
   - Includes the JVM, core libraries & supporting files but lacks development tools like compilers.
3. Java Virtual Machine (JVM)
   - The engine that runs Java bytecode. Provide platform independence by abstracting the underlying operating system. Handles memory management, garbage collection, & other low-level operations.
4. Java Debugger - tool used to debug Java applications, allows developers to inspect variables, set breakpoints & step through code to diagnose issues.

5. Java Archive Tool (Jar) - Used to package Java ~~applications~~ classes & resources into a single compressed file, usually with a .jar extension. Facilates distribution & deployment of java applic & libraries.

6. Java Documentation Tool (javadoc) - Generates HTML documentation from java source code comments. Helps in creating professional documentation for ~~Java~~ Java classes, methods & fields.

7. Applet Viewer: tool for running & debugging Java applets without need for a web browser.

8. Java API Documentation - a collection of standard documentation that explains the Java standard Library & other APIs.

9. Java Class loader - ensures that the right classes r loaded & prevents duplicate loading. ~~Is~~ Part of JVM that dynamically loads Java classes into memory as needed during runtime.

10. Java Standard library: Set of pre-written classes & interfaces that provides common functions like data structure, n/king, & file I/O.

11. Java Integrated Develop Envir (IDE) Plugins: can enhance development productivity. They integrate with JDK to provide features like code completions, refactoring & debugging.

Q2. Difference between ~~both~~ JDK, JRE & JVM.

A2.

	JDK	JRE	JVM
1.	A Software deve-lopment kit used 2 develop applica-tion in Java.	A software bundle that provides Java class libraries with necessary components to run Java code.	An abstract machine that provides Java runtime environment & execute java bytecode. code.
2.	It contains tools like for develop, debug. & monitor java code.	Contains class library & other supporting files that the JVM requires to execute prgm.	Contains Software development tools are not included in the JVM.
3.	Superset (includes JRE & dev tools)	Subset of JDK (focus on runtime enviroment)	Core part of both JDK & JRE
4.	It stands for Java dev kit.	It stands for java runtime environment.	It stands java Virtual machine
5.	Platform Indep-dence through JVM.	Platform Independen-dence through JVM.	Provides platform ~~indep~~ ind-endence.
6.	Needed for java development	Needed for running java applications.	Bundled with JRE/JDK, not installed separ

Q3.

A3. The JVM allows Java programs to run on any device or OS that has a com JVM installed. This is foundation of J "Write Once, Run Anywhere" capability.

- Bytecode Execution: JVM executes Java bytecode, which is the intermediate representation of Java source code compiled by java compiler (javac). Bytecode is platform-independent, & the JVM translates it into a machine code specific to the host system.
- JVM provides built-in support 4 multithread allowing java appli. 2 perform multiple tasks efficiently utilizing CPU resources.
- JVM handles garbage collection, preventing memory leaks through clean up of unused objects in memory, freeing up resources.
- JVM enforces strict security policies, including bytecode verification, to prevent the execution of malicious code.

JVM executes java code in 3 main steps:

- Class loading: JVM loads the complied .class files containing java bytecode into memory using the class loader, preparing them to exec
- Execution engine interprets the bytecode or (JIT) compilation 2 convert bytecode into native code, which is then executed directly by the host CPU.
- Memory & Thread Management: manages memory allocation 4 objects, performs garbage collect 2 free up unused memory & handles multithread to ensure efficiently execution of program.

4.

A4. JVM's memory management is designed 2 ensure that Java applications run smoothly & without errors.

- Heap is where JVM allocates memory 4 all objects & class instances. It is divided into generations to optimize garbage collections. The Young Generation is where new objects are created & old generation is where long-lived objects are stored.
- Stack Memory: each thread in a java applica has its own stacks memory, which stores method call frames, local variables & partial results. The stacks operates in a last in first out (lifo) manner. & memory is automatically managed when methods are invoked & exited.
- JVM automatically reclaims memory used by objects that are no longer reachable. Garbage collectors run periodically to clean up unused objects from the heap, freeing memory & preventing memory leaks.
- Method Area or Metaspace stores class-level data such as class definitions, method data & constants. Size of metaspace is not fixed & can grow dynamically, allowing for more efficient use of memory when loading large no. of classes.

5.

A5. JIT (Just In Time) Compiler: A component of the JVM that compiles bytecode into native machine code at runtime rather than before execution.

Role in JVM: Converts frequently executed bytecode into optimized machine code to speed up execution.

Caching: Compiled code is cached, so future executions of the same code are faster.

Runtime Compilation: Allows dynamic optimizations based on actual execution profile of application.

Bytecode: An intermediate, platform-independent code generated by java compiler frm java source code. It is contained in .class files.

Importance: Bytecode is verified by JVM for security before execution, helping to protect against malicious code.

Portability: It can be executed on any platform with a compatible JVM, supporting Java's "Write Once, Run Anywhere" philosophy.

Optimization: Allows the JVM to perform runtime optimizations tailored to specific execution environment.

6. Architecture of JVM

A6. i) Class Loader Subsystem: Loads Classes - handles loading, linking & initializing classes & interfaces.

Class Loaders: Includes Bootstrapping, Extension & System / Application class loaders.

ii) Runtime Data Area:
- Heap - Stores objects & class instances.
- Stack - Stores method call frames, local variables & partial results.
- Method Area: Stores class-level data such as method information, field data & constants.
- Program Counter (PC) Register: Keep tracks of the current instruction being executed.
- Native Method Stack: Stores state for native method calls.

iii) Execution Engine:
- Interpreter: executes bytecode instructions directly.
- JIT Compiler: compiles bytecode into native machine code at runtime for improved performance.

iv) Native Interface:
- JNI (Java Native Interface): Allows Java code to interact with native applications & libraries written in other languages like C or C++.

v) Garbage Collector: &
- Automatic Memory Management: Reclaims memory used by objects that are no longer reachable, helping to manage & optimize memory usage.

vi) Java API:
- Standard Libraries: Provides a wide range of built-in classes & methods for tasks such as data manipulation, I/O operations & networking.

7.

A7. i. Class Loading Mechanism: Dynamic Loading:
- The JVM dynamically loads classes at runtime, which means it can adapt to different runtime environments & configurations without requiring recompilation of the bytecode.

(ii) Platform-Independent Code - Java source code is compiled into bytecode (.class files), which is an intermediate, platform-independent representation of program.

(iii) Abstract Layer: JVM acts as an abstract layer between the bytecode & the underlying hardware & OS. It interprets or compiles the bytecode into native machine code specific to the host system.

(iv) Uniform Execution Environment: JVM provides a consistent execution environment regardless of underlying platform, ensuring that the same bytecode behaves the same way on different systems.

8.

A8. Significance of Class loader in java -

1. Dynamic loading: Loads classes at runtime
- Class Loader is responsible for loading Java classes into the JVM during runtime, as needed.

2. Efficient Memory Usage: Only loads classes when they are required, which helps in optimizing memory usage.

3. Isolates Class Namespaces: Different class loaders can load classes with the same name but from different locations, allowing for versioning & modularity.

4. Ensures Safety loading: by verifying classes before loading them into the JVM, protecting against malicious code.

Types of Class loaders & their differences:

1. Bootstrap Class Loader:
- Loads Core Java Libraries: loads classes from JRE & is part of JVM.
- Implements in native code: it is in native code & doesn't extend java.lang.ClassLoader.

2. Extension Class Loader:
- Delegation: Delegates to the Bootstrap class loader before attempting to load classes.
- Loads Extension libraries: loads classes from the jre/lib/ext directory or any other location specified by the java.ext.dirs system.

3. Key Differences:-
- Hierarchy: Class loaders follow a parent-child hierarchy, where each loader delegates loading tasks to its parent before attempting to load a class itself.

- Loading Source: Each class loader loads classes from different sources.
- While Bootstrap Class Loader is implemented in native code, the other class loaders are implemented in java & extend java.lang.ClassLoader.

9.

A9. Key Difference: Scope of Access:-

- Public: Widest scope; accessible everywhere.
- Protected: Limited to package + subclasses.
- Default (Package-Private): limited to the package.
- Private: Narrowest scope; limited to class.
- ~~(Code)~~ Inheritance:
- ~~Protected~~ Private: Not accessible in subclasses.
- Protected: Accessible in subclass, even if they are in different packages.
- Package:
- Default & Protected: Accessible within the same package, but Protected extends access to subclass outside the package.

1. Public - Accessible from ~~everywhere~~ anywhere.

Use case: Used 4 classes, methods, and variables that should be universally accesible.

2. Protected

- Accessible within the same package and by subclasses

Usecases: Typically used for methods and variables that should be inherited and used by subclasses but not exposed to the world.

3. Default: Accessible only within the same package. No keyword is used

Usecase: Useful for package level encapsulation, where classes, methods or variable should be accessible only within the same package.

4. Private: Accessible only within the same class.

Use case: Used for methods and variables that should be hidden from all other classes, including subclasses.

Q11.

A11. No, we cannot override a protected method in a superclass with a private method in a subclass because it restricts access, violating polymorphism.

- You can override a protected method with a public method, which increases accessibility.
- Access modifiers in ~~one~~ overriding must be the same or less restrictive than the superclass method.
- Restricting access will cause a compilation errors.

12.

A12.- Same Package: Both protected & default access levels allows access within the same package.
- Different Package: Protected allows access in subclasses even if they are in different packages, while default does not.
- Non-subclass Access: Neither protected nor default allows access from non-subclass classes in different packages.
- Subclass Access: Protected can be accessed in subclasses outside the package, but default is limited to same package only.

A 13. Yes, it is possible to make a class private in Java, but with limitations

1. Private Classes: A class can be private only if it is an inner class
2. Scope of Access: A private inner class is accessible only within the outer class where it is defined.
3. Limitation: You cannot make a top-level class private; top-level classes can only be public or default.
4. Use Case: Private inner classes are often used to encapsulate helper classes or implementation details that should not be exposed outside the outer class.

A.14 1. No, a top level class in Java cannot be declared as protected or private.
2. Access Modifiers: Top-level classes can only be declared as public or with default access.
3. Reason: The protected and private modifiers are meant to control member access within classes, not at the package or global level, which is why they cannot be applied to top-level classes.
4. Package-Level Access: Top-level classes are intended to be accessed at the package level, so restricting them with protected or private would conflict with this design principle.

A.15 1. Access Denied: If a variable or method is declared as private in a class, it cannot be accessed from another class.

2. Compile-Time Error: Attempting to access a private member from another class will result in a compile-time error.
3. Encapsulation: Private access is the most restrictive, ensuring the member is only accessible within the class it is defined in.
4. Workaround: To allow access, you can use public, protected, or package-private access modifiers, or provide getter and setter methods.

A.16 1. Package-Private Access: In Java, if no access modifier is specified, the member or class has package-private access.

2. Visibility Within the Same Package: Package-private members are accessible by all classes within the same package.

3. No Access Across Packages: Package-private members are not accessible from classes in different packages, even if those classes are subclasses.

4. Use Case: This access level is typically used to allow collaboration among classes within the same package while preventing access from outside the package.