

Name : Anuj Urunkar

Branch: Kharghar

Assignment No.1

Q.1) Reading Assignment: A Short History of Java

1. Origins (1991):

- Java began as the **Green Project** at Sun Microsystems, led by James Gosling. It was initially intended for programming home appliances.
- The first version was called **Oak** but was later renamed to **Java** in 1995.

2. Internet Revolution (1995):

- Java was launched with the idea of "**Write Once, Run Anywhere**" (WORA), meaning Java programs could run on any device with a Java Virtual Machine (JVM).
- It gained early popularity through **applets**, small programs that ran in web browsers.

3. Enterprise and Mobile Expansion (Late 1990s - 2000s):

- Java became popular for building large-scale business applications with **Java 2 Enterprise Edition (J2EE)**.
- It also became a key language for mobile development with **Java 2 Micro Edition (J2ME)**.

4. Open Source (2006):

- Sun Microsystems open-sourced Java, allowing the global developer community to contribute to its development.

5. Oracle Acquisition (2010):

- Oracle acquired Sun Microsystems and took over Java's development, focusing on improving performance and adding modern features.

6. Modern Java (2010s-Present):

- Java remains widely used, especially in enterprise environments, cloud computing, and Android development, with continuous updates keeping it relevant.

Q2. Reading Assignment: Java Language Features

1. Simple

- Java's syntax is easy to learn, especially if you have prior experience with C or C++. It removes many complexities like pointers, operator overloading, etc., making it simpler to use.

2. Object-Oriented

- Everything in Java revolves around objects and classes. It promotes reusability and modularity by using concepts like inheritance, encapsulation, and polymorphism.

3. Platform-Independent

- Java code is written once and runs anywhere. This is made possible by the Java Virtual Machine (JVM), which converts Java bytecode into machine code that can run on any operating system.

4. Secure

- Java has strong security features like bytecode verification, sandboxing, and a security manager, which protects against malicious software.

5. Robust

- Java is known for its reliability. Features like automatic garbage collection, exception handling, and strong type checking contribute to fewer errors and more stable code.

6. Multithreaded

- Java supports multithreading, allowing multiple threads of execution within a single program. This is useful for performing many tasks simultaneously, like handling multiple client requests in a server application.

7. Architecture-Neutral

- Java's bytecode is architecture-neutral, meaning it does not depend on the architecture of the machine it is running on. This is different from traditional compiled languages, which generate machine code specific to a particular machine.

8. Portable

- Because Java is platform-independent at both the source and binary levels, programs can move easily from one platform to another. This portability is enhanced by the standardization of data types and the JVM.

Q3. Which Version of JDK Should I Use?

1. LTS (Long-Term Support) Versions

- **Use this if:** You need stability and long-term support.
- **Example:** JDK 17 (released in September 2021) is an LTS version.
- **Why:** LTS versions receive updates and support for many years, making them ideal for production environments where you need reliability.

2. Latest Version

- **Use this if:** You want access to the latest features and improvements.
- **Example:** JDK 21 (if it's the latest available).

3. Project Requirements

- **Use this if:** Your project or a third-party library requires a specific JDK version.
- **Why:** Some projects or libraries may only work with certain JDK versions due to compatibility issues.

Q 4. JDK Installation Directory Structure

Typical JDK Installation Directory Structure

1. **bin**

- **Contents:** Executable files (e.g., java, javac, javaw, jar, javadoc).
- **Purpose:** Contains the command-line tools necessary for running and compiling Java applications.

2. **conf**

- **Contents:** Configuration files.
- **Purpose:** Holds configuration files for the JVM and other tools. This is not always present in all JDK distributions.

3. **include**

- **Contents:** Header files (e.g., jni.h).
- **Purpose:** Contains header files used for native programming with Java (e.g., for JNI – Java Native Interface).

4. **lib**

- **Contents:** Libraries and support files (e.g., rt.jar, tools.jar).
- **Purpose:** Includes runtime libraries (like the Java standard library), tools for development, and other necessary resources. For example, rt.jar contains the Java runtime classes.

5. **legal**

- **Contents:** Legal notices and license files.
- **Purpose:** Contains documentation related to the legal aspects of the JDK.

6. **jmods** (Available in JDK 9 and later)

- **Contents:** Modular JAR files (e.g., java.base.jmod, java.logging.jmod).
- **Purpose:** Contains modules for the Java Platform Module System (JPMS).

7. **src.zip**

- **Contents:** Source code for the Java standard library.
- **Purpose:** Provides the source code for the JDK libraries, which can be useful for debugging and learning.

8. **demo**

- **Contents:** Sample code and demonstrations.
- **Purpose:** Includes sample programs and demonstrations to help users understand how to use the JDK features.

9. **man** (Optional, depending on the platform)

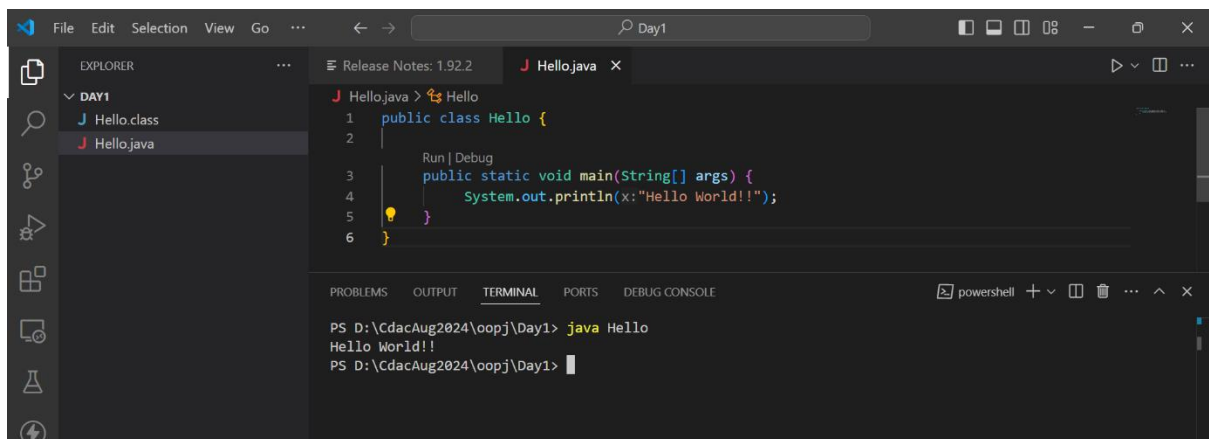
- **Contents:** Manual pages.
- **Purpose:** Provides Unix manual pages for JDK tools.

Q.5 : About Java Technology

Java technology is a combination of tools, libraries, and frameworks that together make it possible to create and run software applications. Java is not just a programming language; it's an entire ecosystem that supports the development of software across different platforms.

Q.6 Coding Assignments

1. **Hello World Program:** Write a Java program that prints "Hello World!!" to the console.



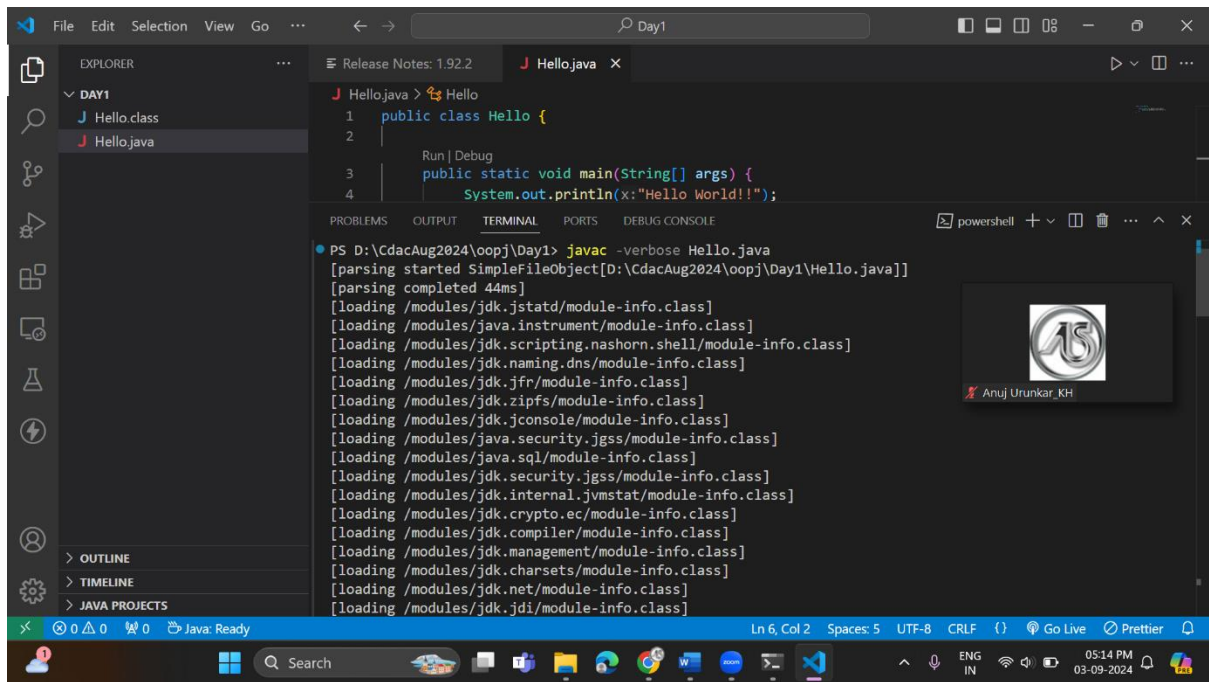
The screenshot shows an IDE with a dark theme. On the left, the Explorer pane shows a project named 'DAY1' containing 'Hello.class' and 'Hello.java'. The main editor displays the source code for 'Hello.java':

```
1 public class Hello {  
2  
3     Run | Debug  
4     public static void main(String[] args) {  
5         System.out.println(x:"Hello World!!");  
6     }  
}
```

Below the editor, the TERMINAL pane shows the command prompt output:

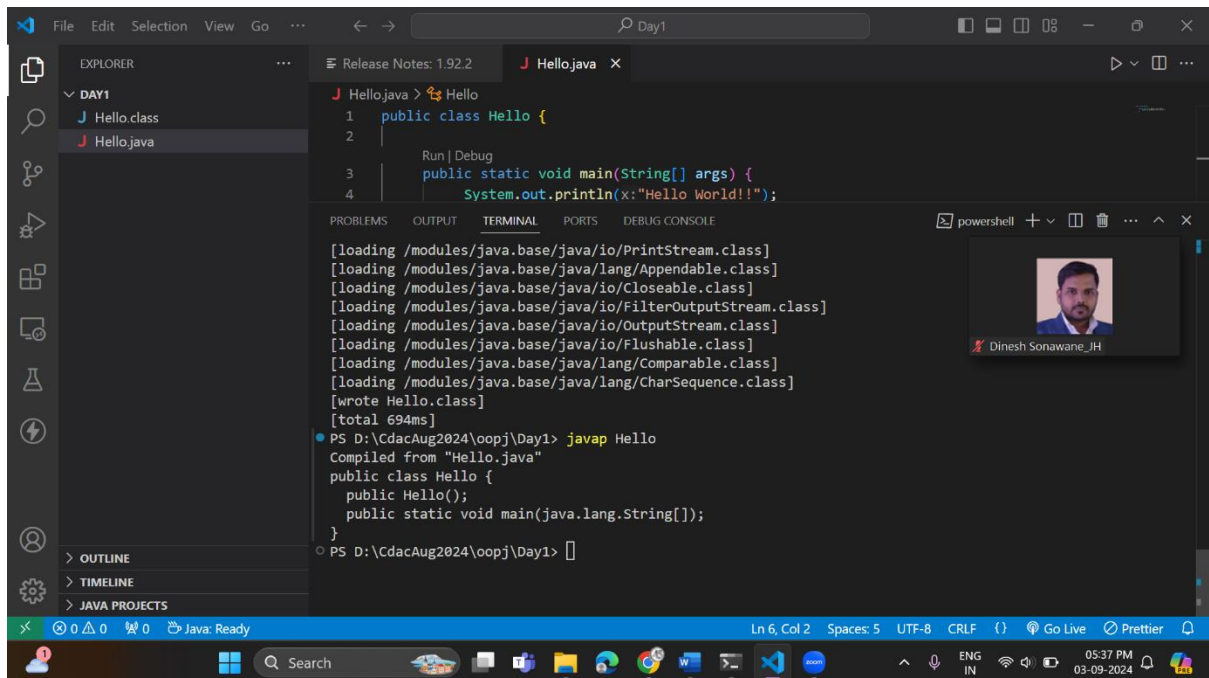
```
PS D:\CdacAug2024\oopj\Day1> java Hello  
Hello World!!  
PS D:\CdacAug2024\oopj\Day1>
```

2. **Compile with Verbose Option:** Compile your Java file using the `-verbose` option with `javac`. Check the output.



```
PS D:\CdacAug2024\oopj\Day1> javac -verbose Hello.java
[parsing started SimpleFileObject[D:\CdacAug2024\oopj\Day1\Hello.java]]
[parsing completed 44ms]
[loading /modules/jdk.jstatd/module-info.class]
[loading /modules/java.instrument/module-info.class]
[loading /modules/jdk.scripting.nashorn.shell/module-info.class]
[loading /modules/jdk.naming.dns/module-info.class]
[loading /modules/jdk.jfr/module-info.class]
[loading /modules/jdk.zipfs/module-info.class]
[loading /modules/jdk.jconsole/module-info.class]
[loading /modules/java.security.jgss/module-info.class]
[loading /modules/java.sql/module-info.class]
[loading /modules/jdk.security.jgss/module-info.class]
[loading /modules/jdk.internal.jvmstat/module-info.class]
[loading /modules/jdk.crypto.ec/module-info.class]
[loading /modules/jdk.compiler/module-info.class]
[loading /modules/jdk.management/module-info.class]
[loading /modules/jdk.charsets/module-info.class]
[loading /modules/jdk.net/module-info.class]
[loading /modules/jdk.jdi/module-info.class]
```

3. **Inspect Bytecode:** Use the javap tool to examine the bytecode of the compiled .class file. Observe the output



```
[loading /modules/java.base/java/io/PrintStream.class]
[loading /modules/java.base/java/lang/Appendable.class]
[loading /modules/java.base/java/io/Closeable.class]
[loading /modules/java.base/java/io/FilterOutputStream.class]
[loading /modules/java.base/java/io/OutputStream.class]
[loading /modules/java.base/java/io/Flushable.class]
[loading /modules/java.base/java/lang/Comparable.class]
[loading /modules/java.base/java/lang/CharSequence.class]
[wrote Hello.class]
[total 694ms]
PS D:\CdacAug2024\oopj\Day1> javap Hello
Compiled from "Hello.java"
public class Hello {
    public Hello();
    public static void main(java.lang.String[]);
}
```

Q.7 The JVM Architecture Explained

Java Virtual Machine (JVM) Architecture

1. Overview of the JVM

- The **Java Virtual Machine (JVM)** is an abstract computing machine that allows Java bytecode to be executed on any device or operating system, ensuring platform independence. It provides the environment in which Java programs run, including managing memory, executing bytecode, and providing other necessary runtime services.

2. Key Components of the JVM

2.1. Class Loader Subsystem

- **Purpose:** The Class Loader Subsystem is responsible for loading class files into memory, verifying their correctness, and preparing them for execution.
- **Main Functions:**
 1. **Loading:** The process of finding and loading class files (.class files) into the JVM memory.
 - **Bootstrap Class Loader:** Loads core Java classes (e.g., java.lang.Object, java.lang.String) from the JDK/JRE libraries.
 - **Extension Class Loader:** Loads classes from the extension libraries, typically found in jre/lib/ext.
 - **Application Class Loader:** Loads classes from the classpath, including user-defined classes and third-party libraries.
 2. **Linking:** Involves verifying, preparing, and optionally resolving the loaded classes.
 - **Verification:** Ensures that the loaded class files are correctly formatted and adhere to the JVM's rules.
 - **Preparation:** Allocates memory for class variables and initializes them to default values.
 - **Resolution:** Converts symbolic references in the class files to direct references (e.g., converting method names to actual memory addresses).
 3. **Initialization:** Executes the static initializers of the class and assigns initial values to static variables.

2.2. Runtime Data Areas

- **Purpose:** These are memory areas allocated by the JVM to manage data during the execution of a Java program.
 1. **Method Area:**

- Stores class-level data such as method code, field data, runtime constant pool, and static variables.
- Shared among all threads.
- Contains information about method and interface code.

2. **Heap:**

- The largest memory area in the JVM, used for dynamic memory allocation.
- Stores all Java objects and arrays.
- Garbage collection occurs here, freeing memory by removing objects that are no longer referenced.

3. **Java Stack:**

- Each thread has its own Java Stack, which holds frames for method invocations.
- **Stack Frame:** Each frame contains local variables, operand stacks (for intermediate calculations), and references to the current method's constant pool.
- Used for managing method execution, storing local variables, and tracking method calls.

4. **Program Counter (PC) Register:**

- A small memory space that holds the address of the current instruction being executed by the thread.
- Each thread has its own PC register.

5. **Native Method Stack:**

- Used for executing native (non-Java) methods, written in languages like C or C++.
- Stores native method calls and any associated data.

2.3. Execution Engine

- **Purpose:** The Execution Engine is responsible for executing the bytecode that has been loaded into the JVM.

1. **Interpreter:**

- Reads and executes bytecode instructions one by one.
- **Pros:** Simple and quick to start.

- **Cons:** Slower than compiled code because each instruction is interpreted in real-time.

2. Just-In-Time (JIT) Compiler:

- Compiles bytecode into native machine code at runtime, which is then executed directly by the CPU.
- **Pros:** Significantly improves performance by compiling hot spots (frequently executed code segments) into native code.
- **Cons:** Initial overhead during the compilation process, but improves speed in the long run.

3. Garbage Collector:

- **Role:** Manages memory automatically by reclaiming space from objects that are no longer in use.
- **Types:** Different garbage collection algorithms exist, such as Serial, Parallel, CMS (Concurrent Mark-Sweep), and G1 (Garbage-First).
- **Process:** Tracks object references and removes objects that are no longer accessible, freeing up memory for new objects.

2.4. Native Method Interface (JNI)

- **Purpose:** Provides the framework that allows Java code to interact with native applications and libraries written in other languages like C and C++.
- **Function:** JNI is used to call functions written in other languages from Java code and to pass data between Java and native methods.

2.5. Native Method Libraries

- **Purpose:** These are libraries required for the execution of native methods, which the JNI calls. They are typically platform-specific (e.g., .dll files on Windows, .so files on Linux).
- **Integration:** The JVM can load these native libraries at runtime as needed.

3. JVM Execution Process

1. **Loading:** The Class Loader Subsystem loads the .class files into the Method Area.
2. **Linking:** The JVM verifies and prepares the loaded classes.
3. **Initialization:** The JVM initializes static variables and executes static blocks.
4. **Execution:** The Execution Engine begins executing the bytecode.

- Bytecode is either interpreted or compiled into native code by the JIT compiler.
 - Memory management is handled by the Garbage Collector.
-

4. Summary for Quick Review

- **Class Loader Subsystem:** Loads, verifies, and initializes classes. It includes Bootstrap, Extension, and Application Class Loaders.
- **Runtime Data Areas:** Includes Method Area (class-level data), Heap (objects), Stack (method calls), PC Register (current instruction), and Native Method Stack.
- **Execution Engine:** Executes bytecode using the Interpreter or JIT Compiler and manages memory with Garbage Collection.
- **Native Interface:** Allows Java to interact with native code.
- **Native Libraries:** Platform-specific libraries required for native method execution.