Assignment:1

1. Reading Assignment: A Short History of Java

- **Task**: Read about the history and development of Java.
- **Link**: http://sunsite.uakom.sk/sunworldonline/swol-07-1995/swol-07-java.html

Origins: Back in 1991, engineers at Sun Microsystems wanted to create a simple computer language for devices like cable TV boxes. They needed a language that was small, efficient, and could work on different types of devices.

The Green Project: They started a project called "Green" and were inspired by earlier attempts with Pascal, a computer language designed for portability. They used a similar approach of creating a virtual machine, which could run code on any device with the right interpreter.

Development of Oak: Instead of Pascal, they based their language on C++ and made it object-oriented. The lead engineer, James Gosling, named the language "Oak" because of an oak tree outside his window. But later, they changed the name to Java.

Early Attempts: In 1992, they created their first product called "*7," a smart remote control. Unfortunately, it didn't attract much interest. They tried to market their technology to other companies but didn't find success.

Internet Growth: Meanwhile, the internet was expanding rapidly, and browsers were becoming crucial. In 1994, a browser called Mosaic was popular, but there was room for innovation.

The Birth of HotJava: Realizing the potential, the Java team decided to create their own browser, called HotJava. It was not only a browser but also capable of running small programs called applets directly in web pages.

Sun Releases Java: The success of HotJava led Sun to release the first version of Java in 1996.

Oracle Corporation: Sun Microsystems was the original creator of Java. In 2010, It was acquired by Oracle Corporation.

"Write Once, Run Anywhere(WORA)": This slogan highlights Java's platform independence, meaning that Java programs can run on any device or operating system that has a Java Virtual Machine (JVM) installed

2. Reading Assignment: Java Language Features

- **Task**: Learn about the main features of Java.
- **Link**: https://javaalmanac.io/features/

- ****Java SE 1.0 (1996):** ** The first official Java release, featuring core language features, AWT, and basic networking.

- ****Java SE 1.1 (1997):** ** Introduced inner classes, JDBC, RMI, and reflection.

- ****Java SE 1.2 (1998): **** Also known as Java 2, added Swing, Collections API, JIT compiler, and JavaBeans.

- **- **Java SE 1.3 (2000):**** Enhanced performance, added HotSpot JVM, JNDI, JSSE, and JavaSound.

- **- **Java SE 1.4 (2002):**** Introduced regular expressions, non-blocking I/O, assertion, and IPv6 support.

- **- **Java SE 5 (2004):**** Major release with generics, annotations, enhanced for loop, autoboxing/unboxing, enums, and varargs.

- **- **Java SE 6 (2006):**** Improved performance, added scripting support, JDBC 4.0, and pluggable annotations.

- **- **Java SE 7 (2011):**** Introduced try-with-resources, switch on strings, multi-catch, binary literals, diamond operator, and NIO.2.

- **- **Java SE 8 (2014):**** Major release with lambda expressions, streams API, default methods in interfaces, functional interfaces, and Date/Time API.

- **- **Java SE 9 (2017):**** Modular system (Project Jigsaw), JShell REPL, multi-release JARs, HTTP/2 client, and G1 as default GC.

- **- **Java SE 10 (2018):**** Local variable type inference (var), garbage collector interface, application class-data sharing, and experimental Java-based JIT compiler.

- **- **Java SE 11 (2018):**** LTS release with HTTP client (standard), ZGC (experimental), flight recorder, and Epsilon GC.

- **- **Java SE 12 (2019):**** Switch expressions (preview), raw string literals, Shenandoah GC (production), Microbenchmark Suite, and JVM constants API.

- **- **Java SE 13 (2019):**** Text blocks (preview), dynamic class-file constants, switch expressions (standard), ZGC on macOS, and legacy socket API removal.

- **- **Java SE 14 (2020):**** Pattern matching for instanceof (preview), records (preview), packaging tool, foreign-memory access API (incubator), and switch expressions (standard).

- **- **Java SE 15 (2020):**** Sealed classes (preview), text blocks (standard), hidden classes, multi-line strings, records (standard), and deprecated applet API removal.

- **- **Java SE 16 (2021):**** Pattern matching for instanceof (standard), records (standard), foreign-memory access API (preview), vector API (incubator), and ZGC on Windows/AArch64.

- **- **Java SE 17 (2021):**** LTS release with pattern matching for instanceof, records, sealed classes (preview), Unix-domain socket channels, foreign linker API (incubator), and macOS/AArch64 port.

- **- **Java SE 18 (2022):**** Simple web server, UTF-8 by default, vector API (second incubator), foreign function & memory API (incubator), and macOS/AArch64 port.

- **- **Java SE 19 (2022):**** Value-based classes, virtual threads, switch pattern matching, and Linux/RISC-V port.

- **- **Java SE 20 (2023):**** Annotations on Java types, generic specialization, runtime class-file patching, enhanced pseudorandom number generators, vector API (third incubator), and Linux/RISC-V port.

- **- **Java SE 21 (2023):**** Virtual threads (standard), vector API (fourth incubator), and macOS/AArch64 port.

- ****Upcoming Versions (Tentative):****

- **- **Java SE 22 (March 2024):**** Potential features include pattern matching for switch (standard), foreign function & memory API (standard), and more.

- **- **Java SE 23 (September 2024):**** Potential features include further advancements in pattern matching, value-based classes, and performance enhancements.

3. Reading Assignment: Which Version of JDK Should I Use?

- **Task**: Find out which JDK version is right for you.
- **Link**: https://whichjdk.com/

4. Reading Assignment: JDK Installation Directory Structure

- **Task**: Understand the folder structure and files in the JDK installation.

- **Link**: https://docs.oracle.com/javase/8/docs/technotes/tools/windows/jdkfiles.html

5. Reading Assignment: About Java Technology

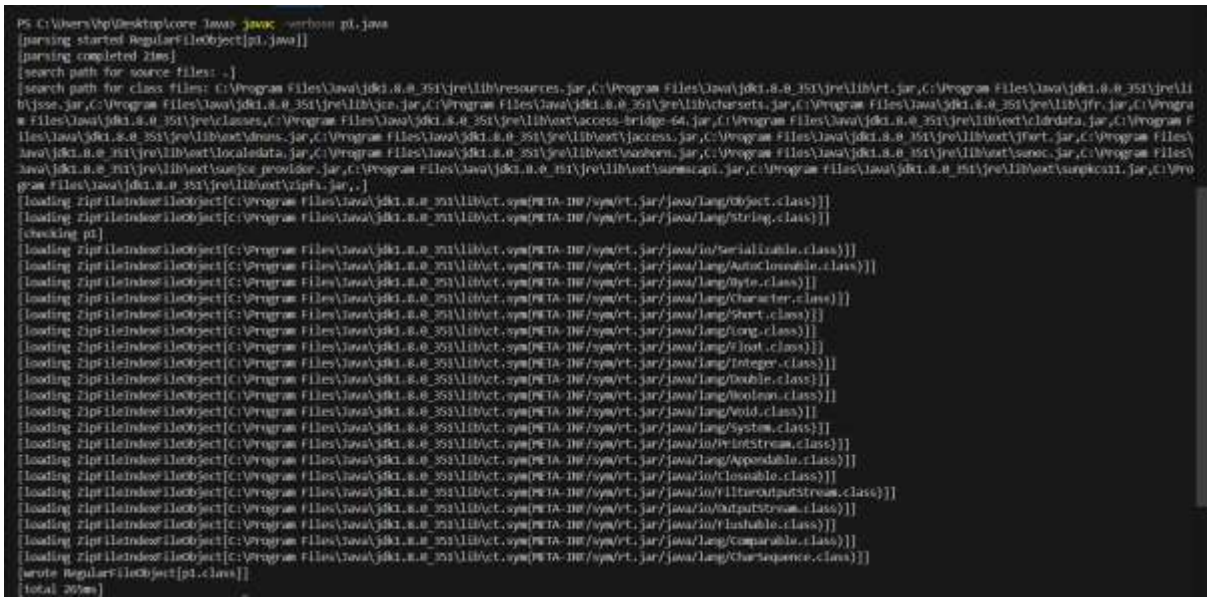
- **Task**: Read about the basics of Java technology and its components.
- **Link**: https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html

6. Coding Assignments

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

```
2. import java.util.*;
3. 
4. public class p1 {
5.     public static void main(String[] args) {
6.         System.out.println("Hello");
7.     }
8. 
9. }
```

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



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

```
PS C:\Users\hp\Desktop\core Java>
javac p1.java
PS C:\Users\hp\Desktop\core Java> javap p1
Compiled from "p1.java"
public class p1 {
  public p1();
  public static void main(java.lang.String[]);
}
```

7. Reading Assignment: The JVM Architecture Explained

- **Task**: Learn about how the Java Virtual Machine (JVM) works.
- **Link**: https://dzone.com/articles/jvm-architecture-explained

bytecode will be executed by the **JRE** (Java Runtime Environment). But the fact that JRE is the implementation of **Java Virtual Machine** (JVM), which analyzes the bytecode, interprets the code, and executes it.

A **Virtual Machine** is a software implementation of a physical machine. Java was developed with the concept of **WORA (*Write Once Run Anywhere*)**, which runs on a **VM**. The **compiler** compiles the Java file into a Java **.class** file, then that .class file is input into the JVM, which loads and executes the class file.

the JVM is divided into three main subsystems:

1. ClassLoader Subsystem
2. Runtime Data Area
3. Execution Engine

ClassLoader Subsystem

Java's dynamic class loading functionality is handled by the ClassLoader subsystem. It loads, links. and initializes the class file when it refers to a class for the first time at runtime, not compile time.

Runtime Data Area

The Runtime Data Area is divided into five major components:

1. **Method Area** – All the class-level data will be stored here, including static variables. There is only one method area per JVM, and it is a shared resource.
2. **Heap Area** – All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe.

3 **Stack Area**– For every thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called Stack Frame. All local variables will be created in the stack memory. The stack area is thread-safe since it is not a shared resource. The Stack Frame is divided into three subentities:

    1. **Local Variable Array** – Related to the method how many local variables are involved and the corresponding values will be stored here.
    2. **Operand stack** – If any intermediate operation is required to perform, operand stack acts as runtime workspace to perform the operation.

3. **Frame data** – All symbols corresponding to the method is stored here. In the case of any **exception**, the catch block information will be maintained in the frame data.

4 PC Registers – Each thread will have separate PC Registers, to hold the address of current executing instruction once the instruction is executed the PC register will be updated with the next instruction.

5 Native Method stacks – Native Method Stack holds native method information. For every thread, a separate native method stack will be created.

3. Execution Engine

The bytecode, which is assigned to the **Runtime Data Area,** will be executed by the Execution Engine. The Execution Engine reads the bytecode and executes it piece by piece.

1. **Interpreter** – The interpreter interprets the bytecode faster but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.
2. **JIT Compiler**– The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code. This native code will be used directly for repeated method calls, which improve the performance of the system.
3. **Java Native Interface (JNI)**: JNI will be interacting with the Native Method Libraries and provides the Native Libraries required for the Execution Engine.
4. **Native Method Libraries**: This is a collection of the Native Libraries, which is required for the Execution Engine.

8. Reading Assignment: The Java Language Environment: Contents

- **Task**: Explore the content and features of the Java language environment.
- **Link**: https://www.oracle.com/java/technologies/language-environment.html