Advanced Programming Lab - II

(CS215/ 18B17CI573)

Name - Rishabh Jain

Er.No. - 231B264

Batch - BY(B8)

- Study the levels of programming languages & determine where Java fits in this hierarchy ?

Levels of Programming Languages :

Machine Language (Low-Level) :

- **Definition :** The lowest level instructions are written in binary that the computer CPU directly understands.
- **Pros :** Runs very fast because no translation is needed.
- **Cons :** Very difficult to read, write, debug.

Assembly Language (Low-Level) :

- **Definition :** Uses short codes instead of pure binary. Each assembly instruction corresponds directly to a machine code instruction.
- **Pros :** Easier than machine language, still gives direct hardware control.
- **Cons :** CPU-specific, not portable, still complex for large programs.

High-Level Languages :

- **Definition :** Use human-readable syntax. A **compiler** or **interpreter** translates them into machine code.
- **Examples :** C, C++, Java, Python.
- **Pros :** Easy to learn, portable, faster development.
- **Cons :** Slightly slower than low-level because of translation steps.

Very High-Level / Domain-Specific Languages :

- **Definition :** Designed for specific purposes and even more abstracted from hardware.
- **Examples :** SQL, HTML, MATLAB.
- **Pros :** Very easy for specific tasks.
- **Cons :** Limited use outside their domain.

Java Fits :

- Java is a High-Level Language**.**
- It uses English like syntax.
- Java uses *bytecode* it is first compiled into intermediate bytecode, then the *Java Virtual Machine (JVM)* interprets or compiles it into machine code at runtime.
- **Advantages:**
  - **Platform Independent :** "Write Once, Run Anywhere" due to JVM.
  - **Object-Oriented :** Focuses on classes and objects.
  - **Portable & Secure :** Runs in a controlled JVM env. .

- Study key Java features of Java ?

Object-Oriented :

- Java is built around the concept of *objects* & *classes*.
- Core principles :
  - **Encapsulation :** Data & behavior are bundled together.
  - **Inheritance :** Classes can inherit from other classes.
  - **Polymorphism :** One interface, multiple behaviors.
  - **Abstraction :** Hiding complex implementation details behind simple interfaces.

Platform Independence :

- Java code compiles to **bytecode**, not native machine code.
- Bytecode runs on any system with a *Java Virtual Machine (JVM)*.
- Makes Java applications portable across operating systems without recompilation.

Simple & Familiar Syntax :

- Java syntax is clean & resembles C/C++ but it removes complex or unsafe features (like pointer arithmetic).
- Designed to be easy to learn & use.

Robustness :

- Java emphasizes early error checking (compile-time) & runtime checking.
- Strong type checking, exception handling, memory management reduce common bugs.

Automatic Memory Management (Garbage Collection) :

- Java automatically reclaims memory that is no longer in use via the *Garbage Collector*.
- Developers don't manually free memory, reducing memory leaks & pointer errors.

Multithreading :

- Built in support for **concurrent execution** using threads.
- Enables writing high-performance, responsive applications that can do multiple tasks simultaneously.

Secure :

- Java has several built-in security features:
    - *Bytecode verifier* ensures code adheres to language rules.
    - *Security manager* can restrict what code can do (file access, network access).
    - No direct pointer access prevents many memory related vulnerabilities.

High Performance :

- Java uses *JIT (Just-In-Time)* **compilation** inside the JVM to compile frequently used bytecode to native machine code at runtime, improving speed.
- Modern JVMs optimize execution dynamically.

Distributed Computing Support :

- Java was designed with distributed environments in mind.
- Features like *RMI (Remote Method Invocation)* & built in networking libraries make it easier to work over networks.

Dynamic & Extensible :

- Java supports *reflection*, allowing inspection & modification of classes/objects at runtime.
- Can load classes dynamically, making frameworks, plugins, serialization easier.

Architecture Neutral :

- Bytecode has a well defined format that not tied to any specific CPU arch. .
- Ensures consistent behavior across platforms.

Rich Standard Library (Java API)

- Provides a large set of prewritten classes for:
    - Collections, I/O, networking, concurrency, utilities, GUI (Swing/JavaFX), XML parsing, etc.
- Speeds development because common tasks are already implemented.

Exception Handling

- Structured mechanism to handle runtime errors gracefully using `try`, `catch`, `finally`, custom exceptions.
- Separates error handling code from regular logic.

Strong Typing

- Types are checked at compile time; reduces errors & makes code more predictable.

Community & Ecosystem

- **Vast ecosystem** : build tools (Maven/Gradle), frameworks (Spring, Jakarta EE), libraries, large community support.

- Investigate the Java architecture (JVM, JRE, JDK, bytecode, class loader, etc) ?

Java architecture is the design that enables Java programs to be platform independent, secure, efficient. It involves how **source code** is written, compiled into **bytecode**, executed on any device with a Java Virtual Machine (JVM).

Main Components :

JDK (Java Development Kit) :

- **Purpose :** For developers to *write, compile, run* Java programs.
- **Includes :**
  - **JRE** (Java Runtime Environment)
  - **Java Compiler (`javac`)** Converts (`.java`) source files into `.class` bytecode files.
  - Development tools like `javadoc`, `jar`, debugging tools.

JRE (Java Runtime Environment) :

- **Purpose :** Provides the libraries, Java Class Loader, and JVM required to run Java applications.
- **Includes :**
  - Java API (standard libraries)
  - JVM (Java Virtual Machine)
- **Note :** JRE can run Java programs but cannot compile them.

C. JVM (Java Virtual Machine) :

- **Purpose :** Executes the **bytecode** on the host machine.
- **Functions :**
    - Loads code (via Class Loader)
    - Verifies code (ensures security & correctness)
    - Executes code (interprets or compiles to native machine code)
    - Manages memory (Garbage Collection)
- **Platform Specific :** Each OS has its own JVM implementation but all run the same bytecode.

Key Internal Components :

1. Bytecode :

- **Definition :** Intermediate, platform independent code generated after compilation.
- Stored in (`.class`) files.
- Runs on any machine with a JVM.

Class Loader :

- **Purpose :** Dynamically loads `.class` files into JVM at runtime.
- **Types :**
  - **Bootstrap Class Loader :** Loads core Java classes (`java.lang.*`, `java.util.*`).
  - **Extension Class Loader :** Loads classes from `ext` directory.
  - **Application Class Loader :** Loads classes from the application's classpath.

Execution Engine :

- **Purpose :** Executes bytecode inside the JVM.
- **Parts :**
  - **Interpreter :** Reads and executes bytecode line-by-line.
  - **JIT Compiler :** Converts frequently used bytecode into native machine code for better performance.
  - **Garbage Collector :** Automatically removes unused objects to free memory.

Java API & Libraries :

- Pre-built classes & packages (like `java.util`, `java.io`, `java.net`) that provide functionality for networking, collections, file I/O, etc.

- Write a Java program to print "Hello World". Compile & execute it ?

```
public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello World"); }

}
```

- Save the above code in a file named *HelloWorld.java*
- Run : *javac HelloWorld.java*
- It create a *HelloWorld.class* file.
- Execute : *java HelloWorld*
- Output : `Hello World`

- Analyze the "Hello World" program, explaining the purpose of each keyword and the role of System.out.println ?

`public class HelloWorld` → Defines the class.

`public static void main(String[] args)`

- **`public`** → JVM can access this method from anywhere.
- **`static`** → Method can run without creating an object of the class.
  - JVM calls `main()` directly.
- **`void`** → Method doesn't return any value.
- **`main`** → The **entry point** of every Java program; JVM looks for this method first.
- **`String[] args`** → Array of strings for command line arguments.

  Ex : Running `java HelloWorld Hello` stores `"Hello"` in `args[0]`.

`System.out.println("Hello World");`

- **`System`** → A final class in `java.lang` package containing useful fields & methods.
- **`out`** → A **static** member of `System` class; an object of type `PrintStream` used to send output to the console.
- **`println()`** → Method of `PrintStream` that prints text & moves to a new line.
- **`"Hello World"`** → String literal to be displayed.

- From your first Java program, identify & list all the keywords, identifiers, class name, method name used ?

Keywords :

- `class`
- `public`
- `static`
- `void`

Identifiers :

- `HelloWorld`
- `main`
- `String`
- `args`
- `System`
- `out`
- `println`

Class Name :

- `HelloWorld`

Method Name :

- `main`
- `println`

- User Input in Java ?

The `Scanner` class can be used to obtain data from the keyboard, files, strings.

`nextBoolean()`	Reads a `boolean` value from the user.
`nextByte()`	Reads a `byte` value from the user.
`nextDouble()`	Reads a `double` value from the user.
`nextFloat()`	Reads a `float` value from the user.
`nextInt()`	Reads a `int` value from the user.
`nextLine()`	Reads a `String` value from the user.
`nextLong()`	Reads a `long` value from the user.
`nextShort()`	Reads a `short` value from the user.

Example :

```
import java.util.Scanner;

class input {

 public static void main(String[] args) {

  Scanner A= new Scanner(System.in);

  System.out.println("Enter Name, Age, Character");

  String S = A.nextLine();

  int N = A.nextInt();

  char C = A.next().charAt(0);

  System.out.println("Name: "+S);

  System.out.println("Age: "+N);

  System.out.println("Character: "+C);}

}
```

- Command Line Arguments in Java ?

It is used because it allows us to provide input at runtime without modifying the whole program.

It helps to run programs automatically by giving them the needed information from outside.

Example :

```
class CLA{
public static void main(String[] a){
System.out.println(a.length);
System.out.println(a[0]);
System.out.println(a[1]);
System.out.println(a[2]);
System.out.println(a[0]+a[1]+a[2]);}
}
```

- Java Access Specifiers ?

Java provides four access specifiers also known as access modifiers that control the visibility & accessibility of classes, methods, variables :

Public :

Members declared as `public` are accessible from any other class or package within the Java program.

Private :

Members declared as `private` are accessible only within the class in which they are defined.

Protected :

Members declared as `protected` are accessible within the same package & by subclasses in any package.

Default or Package-Private :

If no access modifier is explicitly specified, the member has default or package-private access meaning it is accessible only within the same package.