JAVA ARCHITECTURE

1. Java Architecture

JDK :

- It is a software development environment which helps to develop softwares.
- It includes predefined classes.

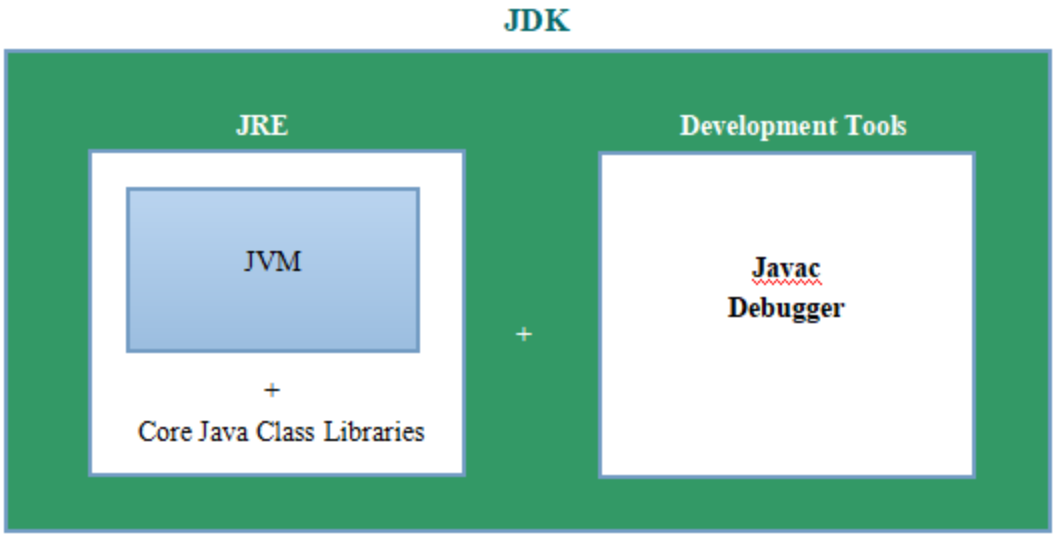
JRE :

- Provides the libraries, JVM and other components to run Java applications.
- Does not include development tools.

JVM :

- Core part of Java Architecture that runs Java Bytecode.
- Abstract machine that provides runtime environment.
- JVM is platform dependent(Separate implementation for each OS).

NOTE : Java Language is platform independent but JVM is platform dependent.



2. Java Compilation and Execution Flow :

Source Code (.java)

|

| Compiled by javac

|

Bytecode (.class)

|

| Executed by JVM

|

Machine Code (Native OS Instructions)

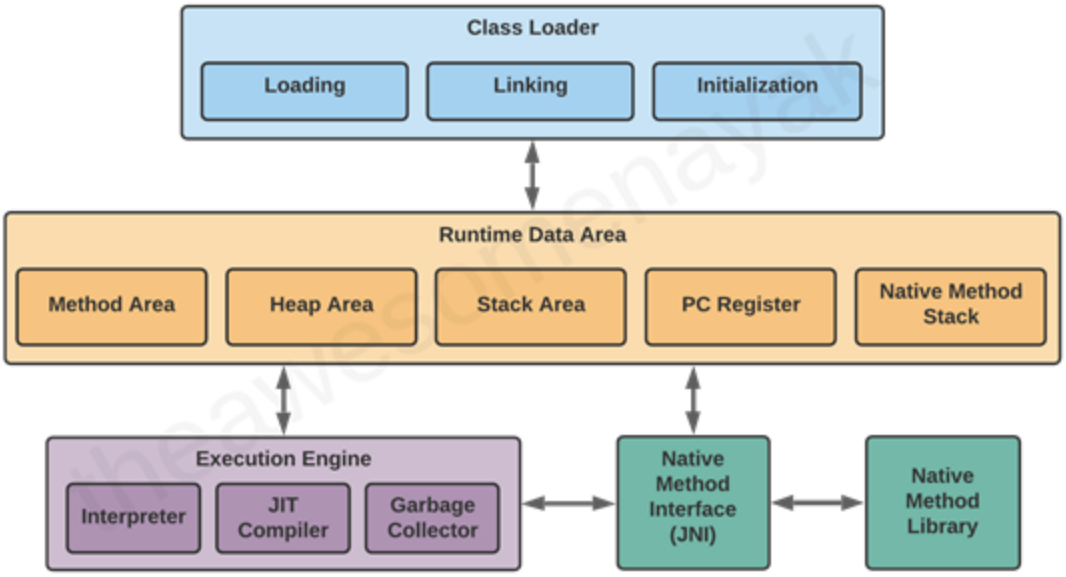
I. Write code in .java file

II. Compile using javac. Generates .class bytecode files.

III. JVM loads the bytecode and execute it using either :

- Interpreter (Line-By-Line Excution)
- JIT Compiler (Compiles hot code into native machine code for faster execution)

3. Internal Architecture of JVM :



JVM consists of 3 distinct components as follows :

A. **Class Loader**

B. **Runtime Data Area**

C. **Execution Engine**

> A. Class Loader

Demo.java file

|

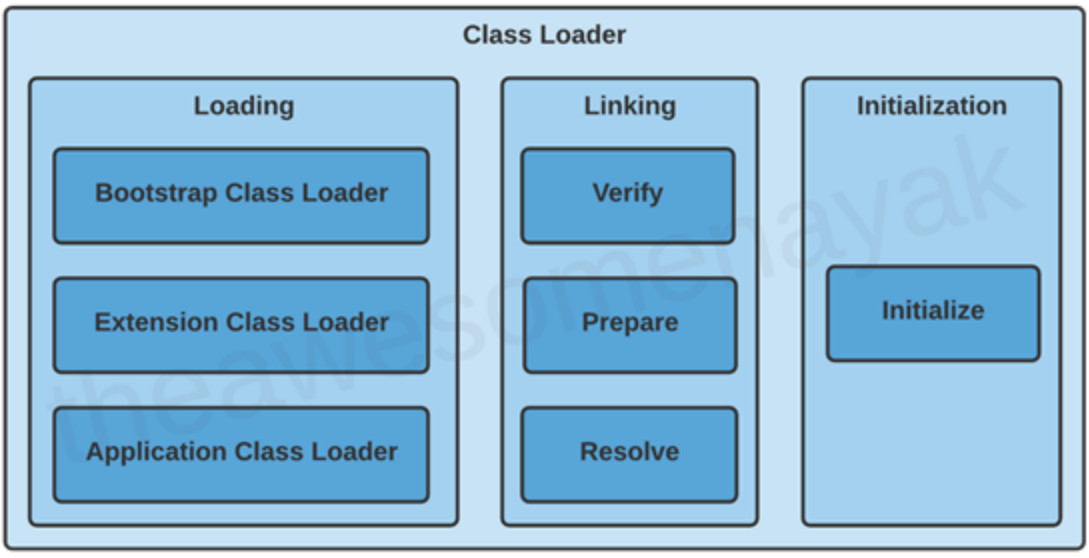
Demo.class (Converted using javac)

|

Class Loader

- When we try to use this .class file in our program, the class loader loads it into the main memory.
- The first class to be loaded into memory is usually the class that contains the main method.

There are three phases in the class loading process : Loading, Linking and Initialization.



Loading :

- Loading involves taking the binary representation (bytecode) of a class or interface with a particular name, and generate the original class and interface

from from that.

There are built-in three class loader in Java :

Bootstrap Class Loader :

- This is a root class loader
- It is a super class of Extension Class Loader
- Loads the standard packages like java.lang, java.net (http://java.net), java.util, java.io (http://java.io), and so on. This packages are present inside the rt.jar and other core libraries present in the $JAVA_HOME/jre/lib directory.

Extension Class Loader :

- Sub class of the Bootstrap Class Loader and Super Class of the Application Class Loader.
- Loads the extensions of standard java libraries which are present in the $JAVA_HOME/jre/lib/ext directory.

Application Class Loader :

- This is the final class loader and sub class of Extension class loader.
- It loads the files present on the classpath.
- By default, the classpath is set to the current directory of the application.
- The classpath can be also modified by adding the -classpath or -cp in command line options.

JVM uses the ClassLoader.loadClass() method for loading the class into memory. It tries to load the class based on a fully qualified name.

Linking :

After class loaded into memory, It undergoes the Linking process.

Linking a class or interface involves combining the different elements and dependencies of the program together.

Linking includes the following steps :

Verification :

This phase checks the structural correctness of the .class file by checking it against a set of constraints or rules. If verification fails for some reason, we a VerifyException.

e.g. : If the code has been built using java 11, but is being run on a system that has java 8 installed. The verification phase will fail.

Preparation :

In this phase, the JVM allocates memory for the static fields of a class or interface, and initialize them with a default values.

Resolution :

In this phase, symbolic references are replaced with the direct references present in the runtime constant pool.

e.g. : if you have references to other classes or constant variables present in the other classes, they are resolved in this phase and replaced with their actual references.

Initialization :

- Initialization involves executing the initialization method of the class or interface (known as <clinit>).
- This can include calling the class's constructor, executing the static block, and assigning values to all the static variables. This is the final stage of class loading.

NOTE : JVM is multi-threaded.

Runtime Data Area :

There are five components inside the runtime data area:

Method Area :

- All the class level data such as the run-time constant pool, field, and method data, and the code for methods and constructors are stored here.
- If the memory available in the method area is not sufficient for the program startup, the JVM throws an OutOfMemoryError.
- The method area is created on the virtual machine start-up, and there is only one method area per JVM.

Heap Area :

- All the objects and their corresponding instance variables are stored here.
- This is the run-time data area from which memory for all class instances and arrays is allocated.
- The heap is created on the virtual machine start-up, and there is only one heap area per JVM.

NOTE: Since the method and heap areas share the same memory for multiple threads, the data stored here is not thread safe.

Stack Area :

- Whenever a new thread is created in the JVM, a separate runtime stack is also created at the same time.
- All local variables, method calls, and partial results are stored in the stack area.
- If the processing being done in a thread requires a larger stack size than what's available, the JVM throws a StackOverflowError.

For every method call, one entry is made in the stack memory which is called the Stack Frame. When the method call is complete, the Stack Frame is destroyed.

The Stack Frame is divided into three sub-parts :

I . Local variables :

Each frame contains an array of variables known as its local variables. All local variables and their values are stored here. The length of this array is determined at compile-time.

II. Operand Stack :

Each Frame contains a last-in-first-out (LIFO) stcak known as its operand stack. This acts as a runtime workspace to perform any intermediate operations. The maximum depth of this stack is determined at compile-time.

III. Frame Data :

All symbols corresponding to the method are stored here. This also stored the catch block information in case of exceptions.

Program Counter (PC) Resisters :

- The JVM supports multiple threads at same time.
- Each thread has it’s own PC register to hold the address of the currently executing JVM instruction.
- Once the instruction is executed, the PC register is updated with the next instruction.

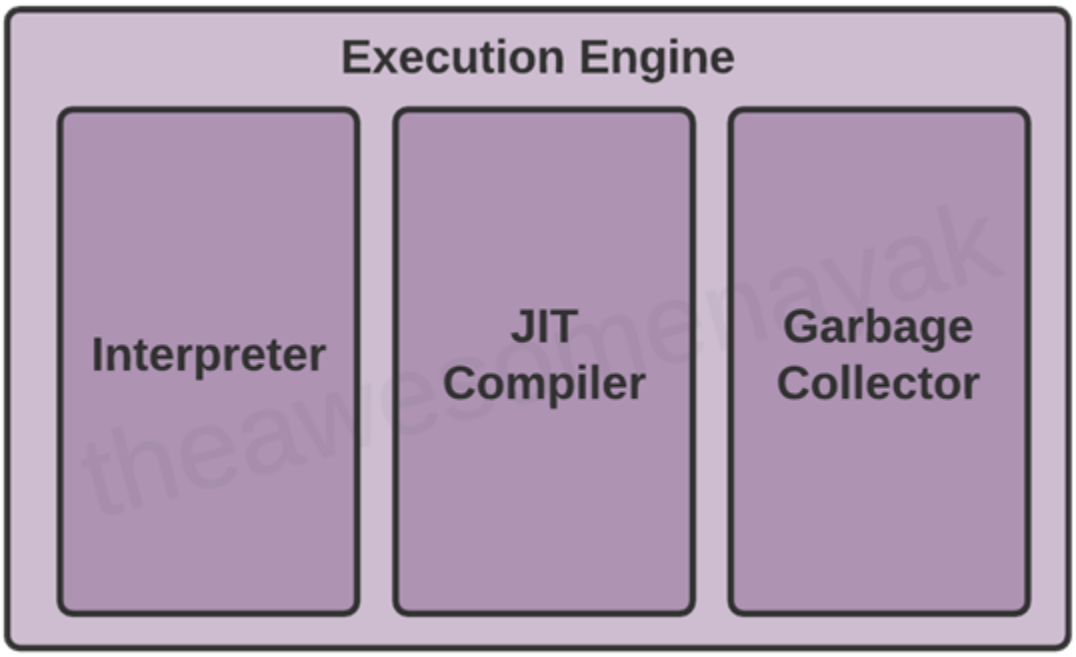
Native Method Stack :

- The JVM contains stacks that support native methods.
- This methods are written in a language other than the Java, such as C and C++.
- For every new thread , a separate native method stack is also allocated.

Execution Engine :

Once the bytecode has been loaded into the main memory, and details are available in the runtime data area, the next step is to run the program. The execution engine handles this by executing the code present in each class.

However, before executing the program, the bytecode needs to be converted into machine language instructions. The JVM can use an interpreter or a JIT compiler for execution engine.



Interpreter :

- The interpreter reads and executes the bytecode instructions line by line. Due to the line by line execution, the interpreter is comparatively slower.
- Another disadvantage of the interpreter is that when a method is called multiple times, every time a new interpretation is required.

JIT Compiler :

- The JIT Compiler overcomes the disadvantage of the interpreter. The Execution Engine first uses the interpreter to execute the bytecode, but when it finds some repeated code, it uses the JIT compiler.
- The JIT compiler then compiles the entire bytecode and changes it to native machine code. This native machine code is used directly for repeated method calls, which improves the performance of the system.

The JIT Compiler has the following components :

Intermediate Code Generator :

Generates the intermediate code

Code Optimizer :

Optimizes the intermediate code for better performance

Target Code Generator :

Converts intermediate code to native machine code.

Profiler :

Finds the hotspots (code that is executed repeatedly)

Garbage Collector :

The Garbage Collector (GC) collects and removes unreferenced objects from the heap area. It is the process of reclaiming the runtime unused memory automatically by destroying them.

Garbage collection makes Java memory efficient because it removes the unreferenced objects from heap memory and makes free space for new objects. It involves two phases:

Mark : In this step, The GC identifies the unused objects in memory
Sweep : In this step, The GC removes the objects identified during the previous phase.

Garbage Collections is done automatically by the JVM at regular intervals and does not need to be handled separately. It can also be triggered by calling System.gc(), but the execution is not guaranteed.

The JVM contains 3 different types of the garbage collectors :

Serial GC :

- This is the simplest implementation of GC, and is designed for small applications running on single-threaded environments.
- It uses a single thread for garbage collection.

- When it runs, it leads to a “stop the world” event where the entire application is paused.
- The JVM argument to use Serial Garbage Collector is -XX:+UseSerialGC

Parallel GC :

- This is the default implementation of GC in the JVM, and is also known as Throughput Collector.
- It uses multiple threads for garbage collection, but still pauses the application when running.
- The JVM argument to use Parallel Garbage Collector is -XX:+UseParallelGC

Garbage First (G1) GC :

- G1GC was designed for multi-threaded applications that have a large heap size available (more than 4GB).
- It partitions the heap into a set of equal size regions, and uses multiple threads to scan them.
- G1GC identifies the regions with the most garbage and performs garbage collection on that region first.
- The JVM argument to use G! Garbage Collector is -XX:+UseG1GC

Note : There is another type of garbge collector called Concurrent Mark Sweep (CMS) GC. However, it has been deprecated since Java 9 and completely removed in Java 14 in favour of G1GC.

Java Native Interface (JNI) :

- At times, it is necessary to use native (non-java) code (for example C/C++). This can be in cases where we need to interact with hardware, or to overcome the memory management and performance constraints in Java. Java supports the execution of native code via the Java Native Interface (JNI).

- JNI acts as a bridge for permitting the supporting packages for other programming languages such as C, C++, and so on. This especially helpful in cases where you need to write code that is not entirely supported by Java, like

some platform specific features that can only be written in C.

- You can use the native keyword to indicate that the method implementation will be provided by a native library. You will also need to invoke System.loadLibrary() to load the shared native library into memory, and make its functions available to Java.

Native Method Libraries :

- Native Method Libraries are libraries that are written in other programming languages, such as C, C++ and assembly. These libraries are usually present in the form of .dll or .so files. This native libraries can be loaded through JNI.

INTERVIEW QUESTIONS :

1. Can you explain the architecture of the JVM? What are its main components?

- The Java Virtual Machine (JVM) is the runtime environment that runs Java bytecode. Its key components are:

Class Loader: Loads .class files into memory.

Runtime Data Areas: Includes Heap, Stack, Method Area (Metaspace in Java 8+), PC Register, and Native Method Stack.

Execution Engine: Executes bytecode; includes interpreter and JIT compiler.

Garbage Collector: Manages memory automatically by reclaiming unused objects.

Native Interface: Connects with native libraries via JNI.

2. What is the role of the Class Loader in the JVM? How does it work?

- The Class Loader loads .class files into the JVM at runtime. It follows a parent delegation model:

Bootstrap ClassLoader: Loads core Java classes from rt.jar.

Extension ClassLoader: Loads classes from ext directory.

Application ClassLoader: Loads classes from the application classpath.

Custom class loaders can also be created for advanced needs like class reloading or sandboxing.

3. What are the different memory areas managed by the JVM? Can you explain each (Heap, Stack, Metaspace, etc.)?

- The JVM memory areas include:

Heap: Stores objects and class instances. Managed by the garbage collector.

Stack: Stores method frames, local variables, and references. Each thread has its own stack.

Method Area (Metaspace in Java 8+): Stores class metadata, static variables.

Program Counter (PC) Register: Holds the address of the current instruction for each thread.

Native Method Stack: Supports native method calls via JNI.

4. What is the difference between Heap and Stack memory in JVM? What goes where?

Aspect	Heap	Stack
Scope	Shared across all threads	Thread-specific
Stores	Objects and class instances	Method calls, local variables
Lifespan	Managed by GC	Destroyed when method exits
Size	Larger	Smaller
Performance	Slower	Faster

5. How does garbage collection work in Java? What are the different types of garbage collectors available in the JVM?

- Garbage Collection (GC) automatically removes unused objects from the Heap. JVM divides the Heap into:

Young Generation: New objects.

Old Generation: Long-living objects.

Permanent Generation / Metaspace: Class metadata.

Common GC types:

Serial GC: Single-threaded, best for small apps.

Parallel GC: Multi-threaded, good throughput.

CMS (Concurrent Mark Sweep): Low pause, concurrent collection.

G1 (Garbage First): Balanced collector with predictable pause times.

ZGC / Shenandoah (Java 11+): Low-latency collectors for large heaps.

6. What happens when you run a .java file? Explain the compilation and execution process step-by-step.

A. Writing the Code:

- You write Java code in a file with a .java extension.

B. Compilation (javac):

- You compile the .java file using the javac compiler.

- Example: javac HelloWorld.java

- This generates a .class file (bytecode) that the JVM understands.

C. Class Loading:

- The JVM loads the .class file using a ClassLoader.

- It verifies the bytecode to ensure it's safe and doesn't violate Java rules.

D. Bytecode Verification:

- Performed by the Bytecode Verifier.
- Checks for illegal code that could corrupt memory or violate access rules.

E. Execution (java):

- You run the program using java HelloWorld.
- The JVM starts and uses the Execution Engine to run the bytecode.
- The main() method is the entry point.

F. JIT Compilation (Optional):

- The Just-In-Time compiler may compile parts of the bytecode into native machine code for performance.

7. Can you explain the lifecycle of a Java class in the JVM — from loading to unloading?

A. Loading:

- The class is loaded from disk by the ClassLoader.
- It could be from the local file system, a JAR, or a network source.

B. Linking:

- Verification: Ensures bytecode is valid.
- Preparation: Allocates memory for static fields and sets default values.
- Resolution: Replaces symbolic references with direct references (e.g., method addresses).

C. Initialization:

- Static initializers (static {} blocks) and static variables are initialized.

- This happens only once per class.

D. Usage:

- Objects are created, methods are called, etc.

E. Unloading:

- When a class is no longer used and its ClassLoader is garbage collected, the class may be unloaded.

- This typically happens in dynamic systems like app servers or plugins, not in basic Java apps.

8. What is JIT (Just-In-Time) Compilation in JVM? How does it improve performance?

- JIT is a part of the JVM that improves performance by compiling bytecode into native machine code at runtime.

- Instead of interpreting bytecode line-by-line, JIT compiles frequently executed code paths ("hot spots") into faster native code.

Benefits:

- Speed: Native code runs much faster than interpreted bytecode.

- Optimization: JIT uses runtime profiling to apply aggressive optimizations (like inlining and loop unrolling).

Types of JIT Compilers:

- Client Compiler (C1): Quick startup, less optimized.

- Server Compiler (C2): More optimization, suitable for long-running applications.

9. What is the difference between interpreted and compiled code in the context of JVM?

Feature	Interpreted Code	Compiled Code (via JIT)
Execution	Bytecode is read and executed line-by-line	Bytecode is converted into native machine code
Performance	Slower	Faster
Flexibility	High (easy to debug)	Lower flexibility, optimized for speed
Optimization	Minimal	Advanced optimizations possible
Example	JVM without JIT	JVM with JIT enabled