Introduction

Each Java developer knows that the JRE (Java Runtime Environment) executes bytecode. But not everyone knows that JRE is the implementation of Java Virtual Machine (JVM), a program that analyzes bytecode, interprets it, and executes it. To write code efficiently, developers need to understand the architecture of the JVM.

Throughout this article, we will learn more about the JVM architecture and its various components.

**What is JVM in Java?**

An implementation of a physical machine in software is called a virtual machine. As Java runs on a virtual machine, it is built using the concept of WORA (Write Once Run Anywhere).

The built-in Java compiler compiles the Java file into a Java .class file, then that .class file is passed to the JVM, so the class can get loaded and executed. The architecture of the JVM is illustrated below.

It is:

* A **specification**that describes how Java Virtual Machines work. However, implementation providers are free to choose which algorithm to implement. Its implementation comes from Oracle and other companies.
* This **implementation**is known as JRE (Java Runtime Environment).
* **Runtime Instance**When you run a java class from the command prompt, a JVM instance gets created.

**Class Loader Subsystem**

In RAM, the JVM is located. The ClassLoader subsystem loads class files into RAM during execution. Its Java feature called dynamic class loading allows classes to be loaded dynamically. A class file (.class) is loaded, linked, and initialized for the first time when a reference is made to a class (not at compile time).

**1) Loading**

A ClassLoader is responsible for loading compiled classes into memory (.class files). In general, class loading begins with loading the main class (the one with a static main() method declaration). Class loading is carried out based on the class references in the already-running classes as follows:

* When bytecode makes a static reference to a class (e.g., System.out)
* When bytecode create a class object (e.g. Person person = new Person("John"))

**2) Linking**

The Linking process consists of verifying and preparing a loaded class or interface, its direct superclasses and superinterfaces, and its element type based on the properties listed below.

* The class or interface must fully load before it can be linked.
* Verifying and preparing a class or interface is required before initializing it (the next step).
* During the linking process, JVM throws an error when the program takes some action requiring linkage to the class or interface involved in the error.

**3) Initialization**

It is the step where JVM performs initialization logic for each loaded class or interface(e.g., calling the class constructor). The JVM is multi-threaded, so a class or interface initialization must be carefully handled and correctly synchronized to prevent another thread from attempting the same class or interface(i.e., making it thread-safe).

Class loading ends with assigning the static variables with their original values in the code, followed by the execution of the static blocks (if any). The code execution is performed in a class hierarchy from top to bottom and from parent to child.



**How Does the JVM Work?**

As shown in the above architecture diagram, the JVM is divided into three main subsystems:

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

**1. ClassLoader Subsystem**

Java's [dynamic class loading](http://www.javainterviewpoint.com/use-of-class-forname-in-java/) 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.

**1.1 Loading**

Classes will be loaded by this component. BootStrap ClassLoader, Extension ClassLoader, and Application ClassLoader are the three ClassLoaders that will help in achieving it.

1. **BootStrap [ClassLoader](http://www.javainterviewpoint.com/" \t "_blank)** – Responsible for loading classes from the bootstrap classpath, nothing but **rt.jar.**Highest priority will be given to this loader.
2. **Extension ClassLoader** – Responsible for loading classes which are inside the ext folder **(jre\lib).**
3. **Application ClassLoader** –Responsible for loading Application Level Classpath, path mentioned Environment Variable, etc.

The above ClassLoaders will follow Delegation Hierarchy Algorithm while loading the class files.

**1.2 Linking**

1. **Verify** – Bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get the verification error.
2. **Prepare** – For all static variables memory will be allocated and assigned with default values.
3. **Resolve** – All symbolic memory references are replaced with the original references from Method Area.

**1.3 Initialization**

This is the final phase of ClassLoading; here, all [static variables](http://www.javainterviewpoint.com/use-of-static-keyword-in-java/) will be assigned with the original values, and the [static block](http://www.javainterviewpoint.com/java-static-import/) will be executed.

**2. 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.
   1. **Intermediate Code Generator** – Produces intermediate code
   2. **Code Optimizer** – Responsible for optimizing the intermediate code generated above
   3. **Target Code Generator** – Responsible for Generating Machine Code or Native Code
   4. **Profiler** – A special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not.
3. **Garbage Collector**: Collects and removes unreferenced objects. Garbage Collection can be triggered by calling System.gc(), but the execution is not guaranteed. Garbage collection of the JVM collects the objects that are created.

**Java Native Interface (JNI)**: JNI will be interacting with the Native Method Libraries and provides the Native Libraries required for the Execution Engine.

**Native Method Libraries**: This is a collection of the Native Libraries, which is required for the Execution Engine.