JVM (Java Virtual Machine) Architecture

What is JVM (Java Virtual Machine)?

The **JVM (Java Virtual Machine)** is a virtual machine, an abstract computer that has its own ISA, memory, stack, heap, etc. It runs on the host OS and places its demands for resources on it.

The **JVM (Java Virtual Machine)** is a specification and can have different implementations, as long as they adhere to the specs. The specs can be found in the below link – https://docs.oracle.com

Oracle has its own JVM implementation (called the HotSpot JVM), the IBM has its own (the J9 JVM, for example).

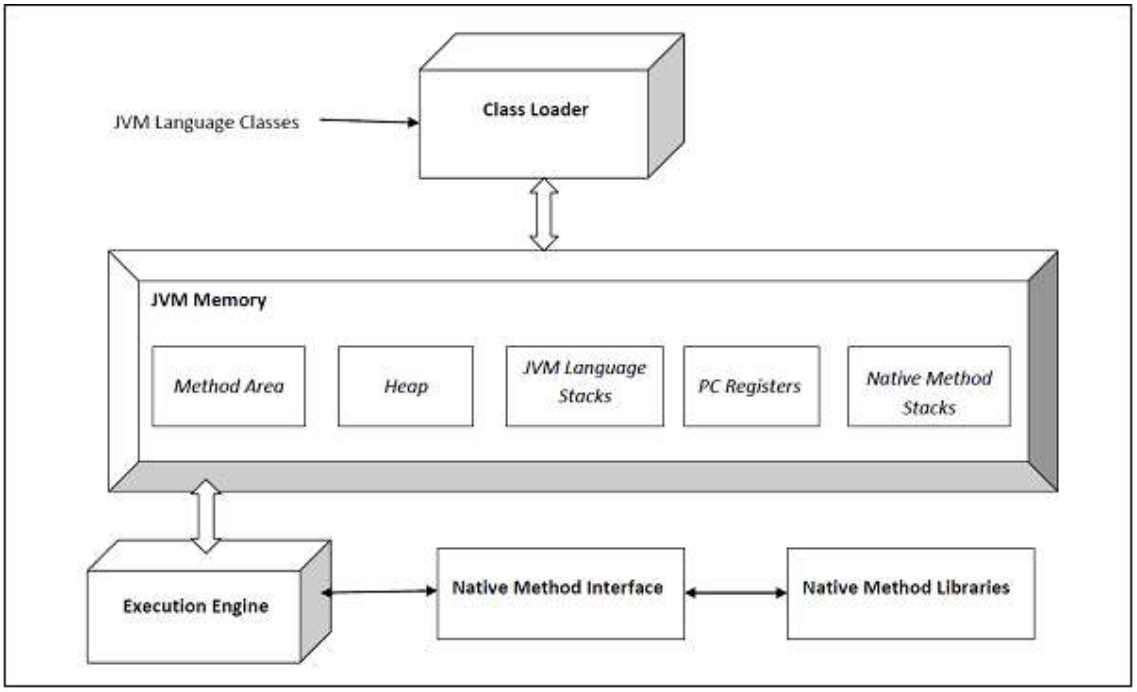
The operations defined inside the spec are given below (source – Oracle JVM Specs)

- The 'class' file format
- Data types
- Primitive types and values
- Reference types and values
- Run-time data areas
- Frames
- Representation of objects
- Floating-point arithmetic
- Special methods
- Exceptions
- Instruction set summary
- Class libraries

- Public design, private implementation

JVM (Java Virtual Machine) Architecture

The architecture of the HotSpot JVM 3 is shown below –



The execution engine comprises the garbage collector and the JIT compiler. The JVM comes in two flavors – **client** and **server**. Both of these share the same runtime code but differ in what JIT is used. We shall learn more about this later. The user can control what flavor to use by specifying the JVM flags -client or -server. The server JVM has been designed for long-running Java applications on servers.

The JVM comes in 32b and 64b versions. The user can specify what version to use by using -d32 or -d64 in the VM arguments. The 32b version could only address up to 4G of memory. With critical applications maintaining large datasets in memory, the 64b version meets that need.

Components of JVM (Java Virtual Machine) Architecture

The following are the main components of JVM (Java Virtual Machine) architecture:

1. Class Loader

The JVM manages the process of loading, linking and initializing classes and interfaces in a dynamic manner. During the loading process, the **JVM finds the binary representation of a class and creates it.**

During the linking process, the **loaded classes are combined into the run-time state of the JVM so that they can be executed during the initialization phase**. The JVM basically uses the symbol table stored in the run-time constant pool for the linking process. Initialization consists of actually **executing the linked classes**.

The following are the types of class loaders:

- **BootStrap class loader**: This class loader is on the top of the class loader hierarchy. It loads the standard JDK classes in the JRE's lib directory.
- **Extension class loader**: This class loader is in the middle of the class loader hierarchy and is the immediate child of the bootstrap class loader and loads the classes in the JRE's lib\ext directory.
- **Application class loader**: This class loader is at the bottom of the class loader hierarchy and is the immediate child of the application class loader. It loads the jars and classes specified by the **CLASSPATH ENV** variable.

2. Linking and Initialization

The linking process consists of the following three steps –

- **Verification** – This is done by the Bytecode verifier to ensure that the generated .class files (the Bytecode) are valid. If not, an error is thrown and the linking process comes to a halt.
- **Preparation** – Memory is allocated to all static variables of a class and they are initialized with the default values.
- **Resolution** – All symbolic memory references are replaced with the original references. To accomplish this, the symbol table in the run-time constant memory of the method area of the class is used.

Initialization is the final phase of the class-loading process. Static variables are assigned original values and static blocks are executed.

3. Runtime Data Areas

The JVM spec defines certain run-time data areas that are needed during the execution of the program. Some of them are created while the JVM starts up. Others are local to threads and are created only when a thread is created (and destroyed when the thread is destroyed). These are listed below –

PC (Program Counter) Register

It is local to each thread and contains the address of the JVM instruction that the thread is currently executing.

Stack

It is local to each thread and stores parameters, local variables and return addresses during method calls. A StackOverflow error can occur if a thread demands more stack space than is permitted. If the stack is dynamically expandable, it can still throw OutOfMemoryError.

Heap

It is shared among all the threads and contains objects, classes' metadata, arrays, etc., that are created during run-time. It is created when the JVM starts and is destroyed when the JVM shuts down. You can control the amount of heap your JVM demands from the OS using certain flags (more on this later). Care has to be taken not to demand too less or too much of the memory, as it has important performance implications. Further, the GC manages this space and continually removes dead objects to free up the space.

Method Area

This run-time area is common to all threads and is created when the JVM starts up. It stores per-class structures such as the constant pool (more on this later), the code for constructors and methods, method data, etc. The JLS does not specify if this area needs to be garbage collected, and hence, implementations of the JVM may choose to ignore GC. Further, this may or may not expand as per the application's needs. The JLS does not mandate anything with regard to this.

Run-Time Constant Pool

The JVM maintains a per-class/per-type data structure that acts as the symbol table (one of its many roles) while linking the loaded classes.

Native Method Stacks

When a thread invokes a native method, it enters a new world in which the structures and security restrictions of the Java virtual machine no longer hamper its freedom. A native method can likely access the runtime data areas of the virtual machine (it depends upon the native method interface), but can also do anything else it wants.

4. Execution Engine

The execution engine is responsible for executing the bytecode, it has three different components:

Garbage Collection

The JVM manages the entire lifecycle of objects in Java. Once an object is created, the developer need not worry about it anymore. In case the object becomes dead (that is, there is no reference to it anymore), it is ejected from the heap by the GC using one of the many algorithms serial GC, CMS, G1, etc.

Read also: Garbage Collection in Java

During the GC process, objects are moved in memory. Hence, those objects are not usable while the process is going on. The entire application has to be stopped for the duration of the process. Such pauses are called 'stop-the-world' pauses and are a huge overhead. GC algorithms aim primarily to reduce this time.

Interpreter

The interpreter Interprets the bytecode. It interprets the code fast but it's slow in execution.

JIT Complier

The JIT stands for Just-In-Time. The JIT compiler is a main part of the Java runtime environment and it compiles bytecodes to machine code at runtime.

5. Java Native Interface (JNI)

Java Native Interface (JNI) interacts with the native method libraries which are essential for the execution.

6. Native Method Libraries

Native method libraries are the collection of C and C++ libraries (native libraries) which are essential for the execution.