JVM

Virtual Machine: JVM

It is a Software Simulation of a Machine which can Perform Operations Like a Physical Machine.

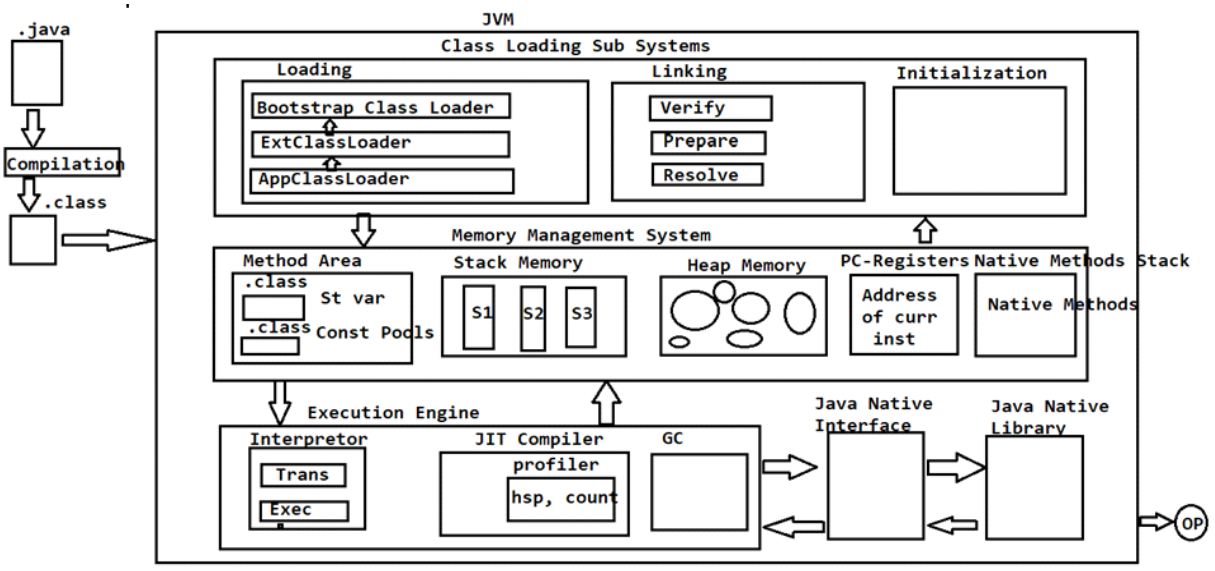
EX:

- **JVM Acts as Runtime Engine to Run Java Applications**
- **PVM (Parrot VM) Acts as Runtime Engine to Run Scripting Languages Like PEARL.**
- **CLR (Common Language Runtime) Acts as Runtime Engine to Run .Net Based Application**
- **KVM (Kernel Based Virtual Machine) for Linux Systems**

JVM

- **JVM is the Part of JRE.**
- **JVM is Responsible to Load and Run Java Applications.**
- **JVM Runs Java Byte Code by creating 5 Identical Runtime Areas to execute Class Members.**

- **Class Loader Sub System**
- **Memory Management System**
- **Execution Engine**
- **PC-Registers**
- **Native Methods Stack**



ClassLoader Sub System:

ClassLoader Sub System is Responsible for the following 3 Activities.

- **Loading**
- **Linking**
- **Initialization**

• Loading:

- **Loading Means Reading Class Files and Store Corresponding Binary Data in Method Area.**
- **For Each Class File JVM will Store the following Information in Method Area.**
  - **Fully Qualified Name of the Loaded Class OR Interface OR enum.**
  - **Fully Qualified Name of its Immediate Parent Class OR Interface OR enum.**
  - **Whether .class File is related to Class OR Interface OR enum.**
  - **The Modifiers Information**
  - **Variable OR Fields Information**
  - **Method Information**
  - **Constant Pool Information and so on.**

After loading .class File Immediately JVM will Creates an Object of the Type class Class to Represent Class Level Binary Information on the Heap Memory.

The Class Object used by Programmer to got Class Level Information Like Fully Qualified Name of the Class, Parent Name, Method and Variable Information Etc.

NOTE: For Every Loader Time Only One Class Object will be Created Even though we are using Class Multiple Times in Our Application.

• Linking:

Linking Consists of 3 Activities

- **Verification**
- **Preparation**
- **Resolution**

• Verification:

- **It is the Process of ensuring that Binary Representation of a Class is structurally Correct OR Not.**
- **That is JVM will Check whether .class File generated by Valid Compiler OR Not and whether .class File is Properly Formatted OR Not.**
- **Internally Byte Code Verifier which is Part of ClassLoader Sub System is Responsible for this Activity.**
- **If Verification Fails then we will get Runtime Exception Saying java.lang.VerifyError.**

- **Preparation:**

  In this Phase JVM will Allocate Memory for the Class Level Static Variables and Assign Default Values (But Not Original Values Assign to the Variable).

  NOTE: Original Values will be assigned in Initialization Phase.

- **Resolution:**
  - **It is the Process of Replaced Symbolic References used by the Loaded Type with Original References.**
  - **Symbolic References are Resolved into Direct References by searching through Method Area to Locate the Referenced Entity.**

- **Initialization:**

  In this Phase All Static Variables Assignments with Original Values and Static Block Execution will be performed from Parent Class to Child Class.

  NOTE: While Loading, Linking and Initialization if any Error Occurs then we will get Runtime Exception Saying LinkageError.

Types of ClassLoaders:

Every ClassLoader Sub System contains the following 3 ClassLoaders.

- **Bootstrap ClassLoader OR Primordial ClassLoader**
- **Extension ClassLoader**
- **Application ClassLoader OR System ClassLoader**

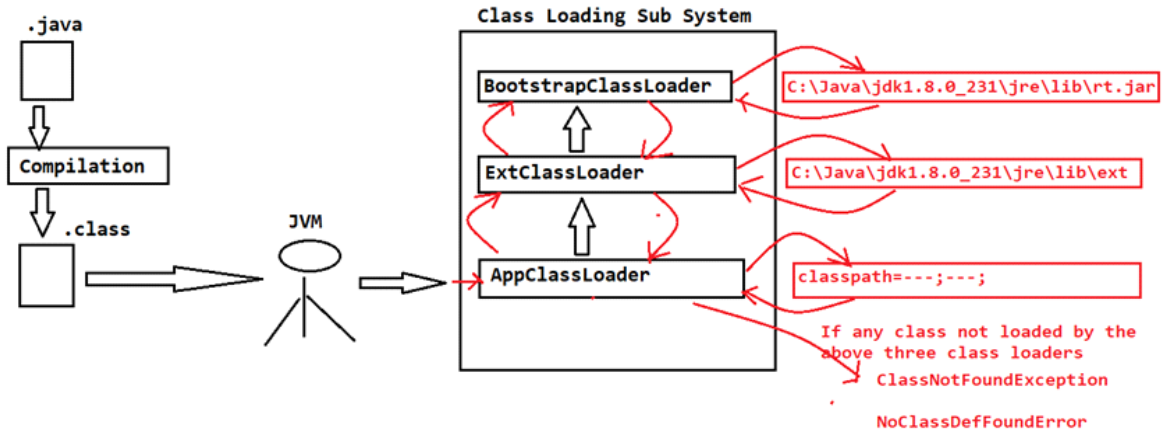
- **Bootstrap ClassLoader:**
  - **This ClassLoader is Responsible for loading 4 Java API Classes.**
  - **That is the Classes Present in rt.jar (runtime.jar).**
    Location: %JAVA_HOME%\jre\lib\rt.jar
  - **This Location is Called Bootstrap Class Path.**
  - **That is Bootstrap ClassLoader is Responsible to Load Classes from Bootstrap Class Path.**
    Bootstrap ClassLoader is by Default Available with the JVM.
  - **It is implemented in Native Languages Like C and C++.**

- **Extension ClassLoader:**
  - **It is the Child of Bootstrap ClassLoader.**
  - **This ClassLoader is Responsible to Load Classes from Extension Class Path.**
    Location: %JAVA_HOME\jre\lib\ext
  - **This ClassLoader is implemented in Java and the corresponding .class File Name is sun.misc.Launcher$extClassLoader.class**

- **Application ClassLoader OR System ClassLoader:**
  - It is the Child of Extension ClassLoader.
  - This ClassLoader is Responsible to Load Classes from Application Class Path (Current Working Directory).
  - It Internally Uses Environment Variable Class Path.
  - Application ClassLoader is implemented in Java and the corresponding .class File Name is sun.misc.Launcher$appClassLoader.class

How ClassLoader will Work?

- ClassLoader follows Delegation Hierarchy Principle.
- Whenever JVM Come Across a Particular Class 1st it will Check whether the corresponding Class is Already Loaded OR Not.
- If it is Already Loaded in Method Area then JVM will Use that Loaded Class.
- If it is Not Already Loaded then JVM Requests ClassLoader Sub System to Load that Particular Class.
- Then ClassLoader Sub System Handovers the Request to Application ClassLoader.
- Application ClassLoader Delegates that Request to Extension ClassLoader and ExtenstionClassLoader in-turn Delegates that Request to Bootstrap ClassLoader.
- Bootstrap ClassLoader Searches in Bootstrap Class Path for the required .class File (jdk/jre/lib)
- If the required .class is Available, then it will be Loaded. Otherwise Bootstrap ClassLoader Delegates that Request to Extension ClassLoader.
- Extension ClassLoader will Search in Extension Class Path (jdk/jre/lib/ext). If the required .class File is Available then it will be Loaded, Otherwise it Delegates that Request to Application ClassLoader.
- Application ClassLoader will Search in Application Class Path (Current Working Directory). If the specified .class is Already Available, then it will be Loaded Otherwise we will get Runtime Exception Saying ClassNotFoundException OR NoClassDefFoundError.



Memory Management System:

- While Loading and Running a Java Program JVM required Memory to Store Several Things Like Byte Code, Objects, Variables, Etc.
- Total JVM Memory organized in the following 5 Categories:
    - Method Area
    - Heap Area OR Heap Memory
    - Java Stacks Area
    - PC Registers Area
    - Native Method Stacks Area

• Method Area:

- Method Area will be Created at the Time of JVM Start - Up.
- It will be Shared by All Threads (Global Memory).
- This Memory Area Need Not be Continuous.
- Method area shows runtime constant pool.
- Total Class Level Binary Information including Static Variables Stored in Method Area.

• Heap Area:

- Programmer Point of View Heap Area is Consider as Important Memory Area.
- Heap Area will be Created at the Time of JVM Start - Up.
- Heap Area can be accessed by All Threads (Global OR Sharable Memory).
- Heap Area Nee not be Continuous.
- All Objects and corresponding Instance Variables will be stored in the Heap Area.
- Every Array in Java is an Object and Hence Arrays also will be stored in Heap Memory Only.

- We are able to get Heap memory calculations by using java.lang.Runtime class.
- Runtime Class Present in java.lang Package and it is a Singleton Class.
- We can Create Runtime Object by using
  Runtime r = Runtime.getRuntime();
- Once we got Runtime Object we can call the following Methods on that Object.
    - maxMemory(): Returns Number of Bytes of Max Memory allocated to the Heap.
    - totalMemory(): Returns Number of Bytes of Total (Initial) Memory allocated to the Heap.
    - freeMemory(): Returns Number of Bytes of Free Memory Present in Heap.

EX:

```
1)  class A
2)  {
3)  }
4)  class Test
5)  {
6)     public static void main(String[] args)throws Exception
7)     {
8)        A a1=new A();
9)        A a2=new A();
10)       A a3=new A();
11)
12)       Runtime rt=Runtime.getRuntime();
13)       System.out.println(rt.maxMemory());
14)       System.out.println(rt.totalMemory());
15)       System.out.println(rt.freeMemory());
16)       System.out.println(rt.totalMemory()-rt.freeMemory());
17)    }
18) }
```

Stack Memory:

- For Every Thread JVM will Create a Separate Runtime Stack.
- Runtime Stack will be Created Automatically at the Time of Thread Creation.
- All Method Calls and corresponding Local Variables, Intermediate Results will be stored in the Stack.
- For Every Method Call a Separate Entry will be Added to the Stack and that Entry is Called Stack Frame OR Activation Record.
- After completing that Method Call the corresponding Entry from the Stack will be Removed.
- After completing All Method Calls, Just Before terminating the Thread, the Runtime Stack will be destroyed by the JVM.
- The Data stored in the Stack can be accessed by Only the corresponding Thread and it is Not Available to Other Threads.

PC (Program Counter) Registers Area:

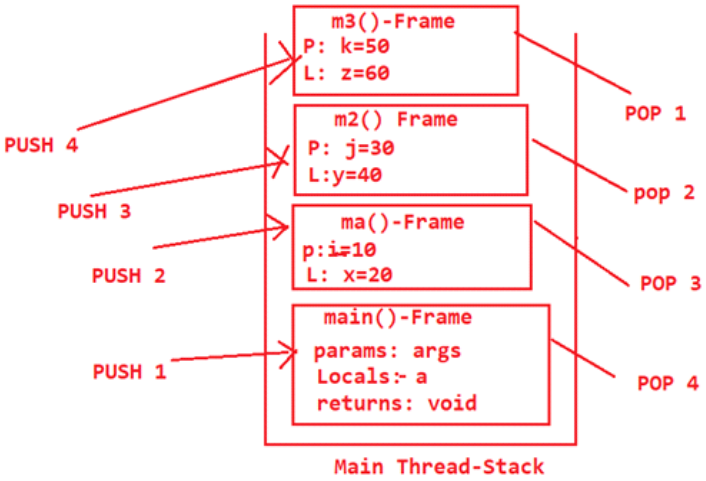
- For Every Thread a Separate PC Register will be Created at the Time of Thread Creation.
- PC Registers contains Address of Current executing Instruction.
- Once Instruction Execution Completes Automatically PC Register will be incremented to Hold Address of Next Instruction.

- ## Native Method Stacks:
  - For Every Thread JVM will Create a Separate Native Method Stack.
  - All Native Method Calls invoked by the Thread will be stored in the corresponding Native Method Stack.

```
class A
{
   void m1(int i){
      int x = 20;
      m2(30);
   }
   void m2(int j){
      int y = 40;
      m3(50);
   }
   void m3(int k){
      int z = 60;
   }
}
class Test{
  public static void main(S[] args){
     A a = new A();
     a.m1();
  }
}
```

Main Thread

10



NOTE:

Method Area, Heap Area and Stack Area are considered as Major Memory Areas with Respect to Programmers Point of View.

Method Area and Heap Area are for JVM. Whereas Stack Area, PC Registers Area and Native Method Stack Area are for Thread. That is

- **One Separate Heap for Every JVM**
- **One Separate Method Area for Every JVM**
- **One Separate Stack for Every Thread**
- **One Separate PC Register for Every Thread**
- **One Separate Native Method Stack for Every Thread**

Static Variables will be stored in Method Area whereas Instance Variables will be stored in Heap Area and Local Variables will be stored in Stack Area.

Execution Engine:

- **This is the Central Component of JVM.**
- **Execution Engine is Responsible to Execute Java Class Files.**
- **Execution Engine contains 2 Components for executing Java Classes.**
  - **Interpreter**
  - **JIT Compiler**

- **Interpreter:**
  - It is Responsible to Read Byte Code and Interpret (Convert) into Machine Code (Native Code) and Execute that Machine Code Line by Line.
  - The Problem with Interpreter is it Interpreters Every Time Even the Same Method Multiple Times. Which Reduces Performance of the System.
  - To Overcome this Problem SUN People Introduced JIT Compilers in 1.1 Version.

- **JIT Compiler:**
  - The Main Purpose of JIT Compiler is to Improve Performance.
  - Internally JIT Compiler Maintains a Separate Count for Every Method whenever JVM Come Across any Method Call.
  - First that Method will be interpreted normally by the Interpreter and JIT Compiler Increments the corresponding Count Variable.
  - This Process will be continued for Every Method.
  - Once if any Method Count Reaches Threshold (The Starting Point for a New State) Value, then JIT Compiler Identifies that Method Repeatedly used Method (HOT SPOT).
  - Immediately JIT Compiler Compiles that Method and Generates the corresponding Native Code. Next Time JVM Come Across that Method Call then JVM Directly Use Native Code and Executes it Instead of interpreting Once Again. So that Performance of the System will be Improved.
  - The Threshold Count Value varied from JVM to JVM.
  - Profiler which is the Part of JIT Compiler is Responsible to Identify HOT SPOTS.

NOTE:
- JVM Interprets Total Program Line by Line at least Once.
- JIT Compilation is Applicable Only for Repeatedly invoked Methods. But Not for Every Method.

Java Native Interface (JNI):

JNI Acts as Bridge (Mediator) between Java Method Calls and corresponding Native Libraries.

Java Native Library:

Java Native Library is the collection of Native methods which are required in java.
Native method is a method declared in java, but, implemented in non java programming languages like C , C++,...