https://tinyurl.com/corejava6pmnotes

JVM ARch:
—--------
Virtual machine: It is a software simulation of the H/W , it is able to perform all the physical activities like H/W components.

EX:
JVM[Java Virtual Machine] : To execute Java applications.
PVM[Python Virtual Machine] : To execute Python applications.
KVM[Kernel Virtual Machine] : To execute Linux Commands.
CLR[COmmon Language Runner] : To execute .Net applications
—--
—--

IN Java , when we compile a java file , automatically .class files are generated , where JVM must execute the java application by executing main class and main() method .

To execute Java applications, JVM has the following components internally.

1. Classes Loading Subsystem
2. Memory Management System
3. Execution Engine
4. Java Native Interface
5. Java Native Library

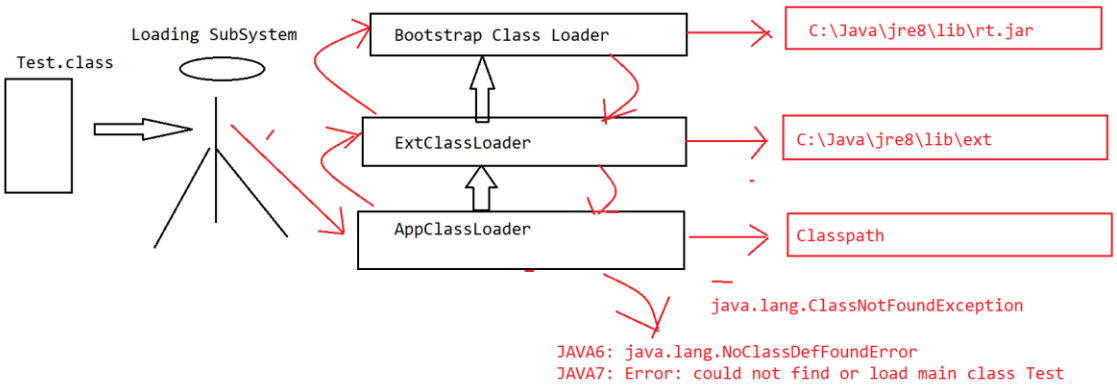
Classes Loading Subsystem
—-------------------------
The main purpose of the classes loading sub system is to load the classes bytecode to the memory and perform the activities at the time of loading the classes bytecode.

There are three phases in Classes Loading Subsystem.
1. Loading
2. Linking
3. Initialization

Loading:
—-------
The main purpose of the Loading phase is to load the classes bytecode to the memory[Method Area].

To load the classes bytecode to the memory , JVM uses Class Loaders.
There are three types of classloaders inside the JVM.

1. Bootstrap Class Loader
2. Extension Class Loader
3. Application Class Loader



When we execute a Java application, the Classes loading subsystem will perform the following actions.

1. Classes Loading Subsystem will give classes loading responsibility to the AppClassLoader, where AppClassLoader will bypass classes loading responsibility to the ExtClassLoader, where the ExtClassLoader will bypass the classes loading responsibility to to the BootstrapClassLoader.
2. BootstrapClassLoader was designed by using some native language , no java class is representing BootstrapClassLoader, it is a parent to the ExtClassLoader, it is able to load all the predefined library to the memory by getting the predefined libraries .class files from “C:\Java\jre8\lib\rt.jar”.
3. After the BootstrapClassLoader work, BootstrapClassLoader handover classes loading responsibility to the ExtClassLoader.
4. ExtClassLoader was designed by using Java programing language, it was represented in the form of a predefined class like “sun.misc.Launcher$ExtClassLoader” , It is able to load the libraries which are available at “C:\java\jre8\lib\ext” location and which are used in the java application.

5. In general, some third party libraries like Hibernate libraries , Spring libraries,... are used to provide in the ext location.
6. After the ExtClassLoader work, ExtClassLoader hands over the classes loading responsibility to AppClassLoader.
7. AppClassLoader was provided by using java programming language, it was represented in the form of “sun.misc.Launcher$AppClassLoader”, it is able to load the libraries which are available at the “classpath” environment variable and which are used in the Java applications.
8. In general, in Jdbc applications, we use Driver class through the “classpath” environment variable only, in this case the required driver class is loaded by AppClassLoader.
9. After the AppClassLoader , if any class is remaining to load then JVM will provide an exception like java.lang.ClassNotFoundException, if the main class is not loaded then JVMwill raise the following exceptions.
   JAVA6: java.lang.NoClassDefFoundError: Test
   JAVA7: Error: could not find or load main class Test.

Note: Classes Loading SUB Systems follows a design principle internally to load the classes that is “Delegation Hierarchy Principle”.

Linking:
—-------
The main purpose of the Linking phase is to provide the links between the classes which are loaded at method Area as per the application.

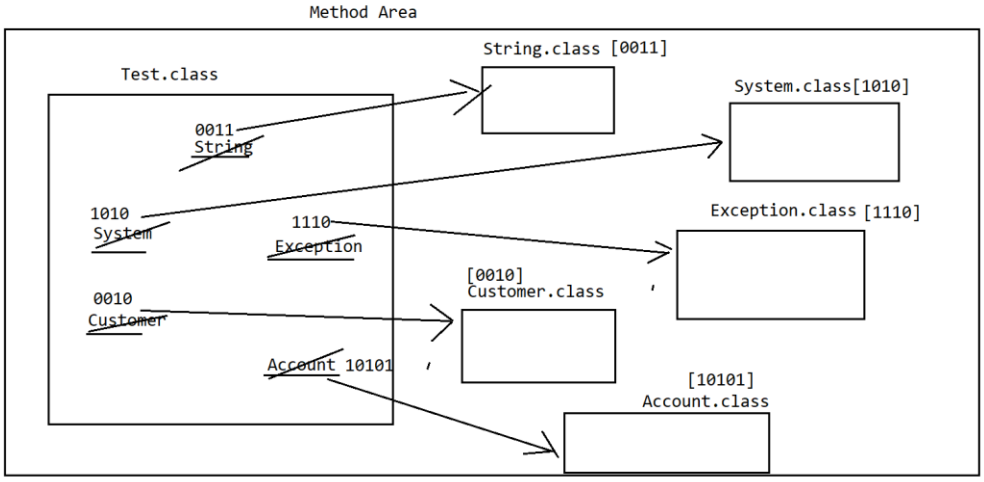
Linking Phase has the following phases mainly.

1. Verify
2. Prepare
3. Resolve

In the Verify phase, Classes Loading Subsystem is able to check whether all the .class files are generated by the right compilers or not.

In the Prepare phase, Classes Loading Subsystem is able to assign memory for the static variables with the default values.

In the Resolve phase, the Classes loading subsystem is able to replace all the naming symbols with their actual references.



Initialization:
—--------------
The main purpose of the initialization phase is to provide the actual initializations to the static variables which are created in the Prepare phase and all the executions like static blocks, static methods along with static variables,.... which are done at the class's loading time.

Memory Management System:
—------------------------
Memory Management System is able to provide memory elements to store classes bytecode, Objects,.....

JVM has mainly the following five types of memories.

1. Method Area
2. Stack Memory
3. Heap Memory
4. PC-Register
5. Native Method Stack

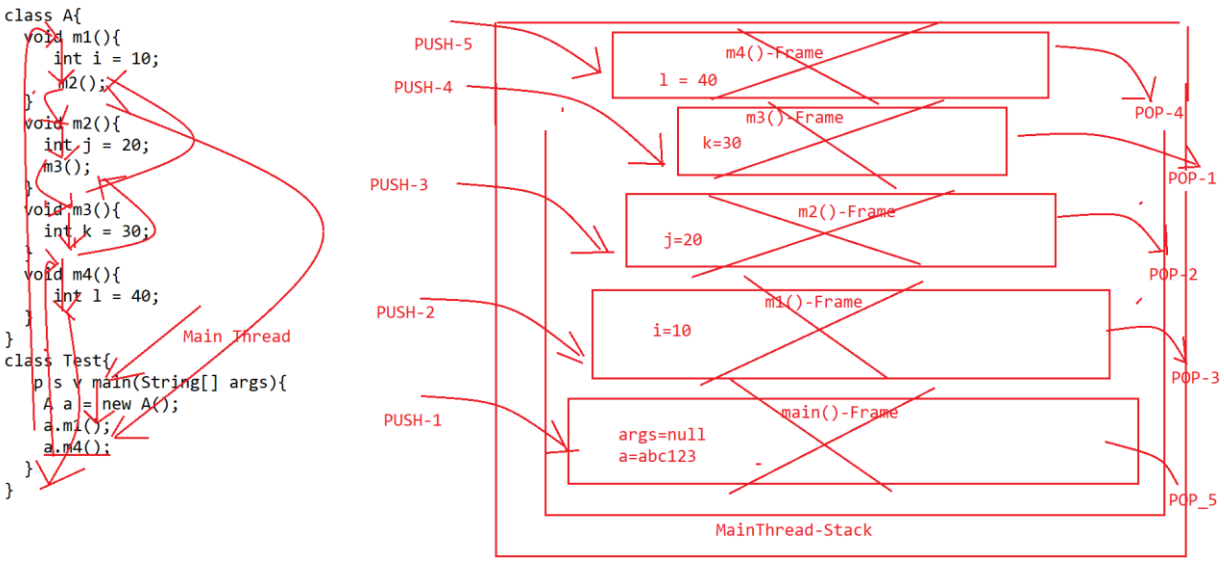
Method Area:

1. It is able to store all the classes bytecode to the memory.
2. It is able to provide constant pool areas to store constant values including String constant objects.
3. It is able to store all the static variables data.
4. When a class bytecode is loaded in the method area , automatically JVM will create a java.lang.Class object at Heap memory with the metadata of the loaded class .
5. Method Area will be created at the time of JVM startup.
6. All the threads which are created in the Java application are able to access method area data.

Stack Memory:

—----------

1. In Java applications, Stack memory is able to create a separate stack for each and every thread which are created in the java application.
2. If any thread accesses any method then a Frame or Activation Record will be created w.r.t the method and it will be pushed to the Thread respective stack.
3. When the Thread completes its execution over the method then the method's respective Activation record or frame will be popped out from the stack.
4. In general, Activation Records or Frames are able to manage the methods data like parameter variables and their values, local variables and their values, return values,....
5. This memory will be created when a thread is created in java applications.
6. This memory is not shareable to all the threads, it is able to provide a separate stack for each and every thread when they are created and these stacks are accessed by the respective threads only, one thread’s stack is not accessible to the other threads.



Heap Memory:

---

1. This memory is the heart in the memory management system.
2. It is able to store all the objects which are created by using the "new" keyword.
3. This memory is able to manage all the objects of the user defined classes , java.lang.Class objects of the classes which are loaded at method area.,...
4. This memory is a shared memory to all the threads which are running in the present java application.
5. This memory is created at the time of JVM startup.

In Java applications, we are able to represent Heap memory and we are able to get all the Heap memory statistics.

To represent Heap memory , JAVA has provided a predefined class in the form of java.lang.Runtime class.

To get a Runtime object we have to use the following method.

public static Runtime getRuntime()

To get all the statistics of Heap memory we have to use the following methods from Runtime class.

To get the Max Heap memory:
public long maxMemory()

To get the initial memory:
public long totalMemory()

To get the free memory:
public long freeMemory()

Consumed Memory = totalMemory - freeMemory

EX:

```
import java.util.Date;

public class Main {
public static void main(String[] args) {
Date[] dates = new Date[10000];
for(int i = 0; i < 10000; i++){
dates[i] = new Date();
}

Runtime runtime = Runtime.getRuntime();
System.out.println("Max Heap Memory : "+runtime.maxMemory());
System.out.println("Total Initial Memory : "+runtime.totalMemory());
System.out.println("Free Memory : "+runtime.freeMemory());
System.out.println("Consumed Memory : "+ (runtime.totalMemory() -
runtime.freeMemory()));

}
}
```

```
Max Heap Memory  : 5318377472
Total Initial Memory : 335544320
Free Memory   : 332186384
Consumed Memory  : 3357936
```

PC-Register:
The main purpose of the PC-Register is to store the current executable instruction address location. If the current instruction execution is completed then Flow of execution will come to the next instruction, so PC-Register will store the next instructions address location.

Native Methods Stack:
Native Methods Stack is like stack memory, but Native method stack is providing its service to the Native Methods Executions.

3. Execution Engine:
—--------------------
The main purpose of the Execution Engine is to execute the java program.

To execute the java application, Execution Engine contains the following elements.

1. Interpreter
2. JIT Compiler
3. Garbage Collector

1. Interpreter
The main purpose of the interpreter is ,
   a. Converting bytecode[Neutral Code] to the Native Code[Executable Code]
   b. Executing the Native Code.

There is a problem with the interpreter. When we access a method repeatedly, the interpreter converts the bytecode to the native code every time unnecessarily, it will reduce the JVM performance.

To address the above problem, SUN Microsystems has given a solution in the form of “JITCompiler” in its JDK1.1 version.

JIT Compiler:
The main purpose of the JIT Compiler is to improve JVM performance.
JIT COmpiler will maintain a separate count variable and its Threshold value for each and every method, when a method call is identified by the interpreter JIT Compiler will increment the method respective count variable value, When the method respective count variable value reached to the threshold value of that method JIT Compiler will supply directly native code to the interpreter to execute the method, so here the interpreter is not required to convert the method from bytecode to the native code , it can execute the method directly by getting the native from JITCompiler.

In the above context, the spot where the method Count variable value reached to the Threshold value is called “Hotspot”.

Inside the JITCompiler, all the count variables w.r.t the methods , threshold values and their Hotspots are maintained by a separate component inside the JITCompiler called “Profiler”.

Garbage Collector:
—----------------
The main purpose of the Garbage Collector is to destroy the objects which are eligible for the Garbage Collection, that is the objects which are not having references.

Java Native Interface and Java Native library:
—---
Java native Library is the collection of native methods , where the Java native method is a method that was declared in the Java programming language and implemented in the Non Java programming language like C, C++, Pascal, Python,....

Java Native Interface is an interface existing between Java application and Java native library, it is responsible for mapping java methods representations to the Native Language methods representations and vice versa.