**JVM**

**Class Loader**

* Loading
* Linking
* Initialization

1. **Loading –**

Loading means reading class files and store corresponding binary data in method area.

For each class file jvm will store corresponding information in method area.

Fully qualified name of class and immediate parent class.

Method, variable, constructors, modifiers, constant pool information and etc;-

After loading .class file JVM will immediately create an object for that loaded class in the heap memory of Type java.lang.Class.

Class Class Object can be used by programmer to get Class Level Information. Like(method, variable , constructor)information etc:-

1. **Linking –**

It consists of Three Activities:

1. Verify
2. Prepare
3. Resolve

**Verifier/Verification** –

It is a process of ensuring that binary representation of a class is Structurally correct or not. That is JVM will check whether the .class is generated by valid compiler or not that is whether .class file is properly formatted or not.

Internally **Byte Code Verifier** is responsible for this activity. Byte Code Verifier is part of class Loader Subsystem.

If verification fails we will get Run time exception saying java.Lang.VerifyError.

**Preparation –**

In this Phase JVM will allocate memory for class level Static variable and assign default values to it.

In Initialization Phase original values will be assigned to the static variables. And here only Default values will be assigned.

**Resolve/Resolution –**

It is the process of replacing symbolic names in our program with original memory references from method area. Symbolic references are basically names of classes, methods and fields referenced from the code. Symbolic references generated by compiler.

**3) Initialization**

In this all static variables are assigned with original values and static blocks will be executed from parent to child or from top to bottom.

**Note –** While loading linking and initializing any exception occurs that Is only called as java.lang.linkageError.

**Types of Class Loader**

* Bootstrap Class Loader
* Extension Class Loader
* Application Class Loader / System Class Loader

**Bootstrap Class Loader –**

It is responsible to load core java API classes. That is classes present in RT.jar

(JDK/JRE/Lib) this location is called bootstrap class Path.

It is by default available with every JVM and it is implemented in native Languages like C/C++ and not implemented in Java. It is not Java Object.

**Extension Class Loader**

It is responsible to load classes from Extension Class Path. (JDK/JRE/Lib/ext/.jar).

It child Class of Bootstrap class Loader.

Extension class loader is implemented in java and corresponding .class file is “sun.misc.launcher$ExtClassLoader.class”

**Application Class Loader**

It is child class of extension of Extension class Loader and this class is responsible to load classes from application class path. It internally uses environment variable class path.  
Application class loader is implemented in java and corresponding .class file is “sun.misc.launcher$AppClassLoader.class”

**Priority -> Bootstrap Class path > Extension Class path > Application Class path**

**Note**

Class Loader follows Delegation Hierarchy principle or Algorithm. Whenever JVM come across a particular Class, First it will check whether the corresponding .class file is already loaded or not. If it is already in **Method Area** then JVM will consider that loaded class. If it is not loaded, Then JVM request class loader Sub-System to load that particular class. Then It will hand over request to application class Loader. Application class loader delegates to extension class Loader which in turn delegates the request to bootstrap class loader. Then Bootstrap loader will search in bootstrap class path if it is available then corresponding .class will be loaded b bootstrap class loader if it is not available then bootstrap class loader delegates the request to extension class loader. Extension class loader will search in extension class path. If it is available then it will be loaded otherwise extension class loader delegates the request to application class loader then application class loader will search in application class path. IF it is available then it will be loaded otherwise we will get Run-Time exception –**NoclassDefFoundError or ClassNotFoundException.**

EG:-

* String.class.getClassLoader() --- Null
* Test.class.getClassLoader() --- sun.misc.launcher$AppClassLoader@hexDecimal
* Customer.class.getClassLoader() -- [sun.misc.launcher$ExtClassLoader@hexDecimal](mailto:sun.misc.launcher$ExtClassLoader@hexDecimal)

Q) What is the need or use of class loader class?

We can use java.lang.classLoader.class to define our own customised class. Every class loader in java should be child class of java.lang.classLoader.class hence this class acts as Base class for all customized class loaders.

**JVM MEMORY AREA:**

1. Method Area
2. Heap Area
3. Stack Area
4. PC Registers
5. Native Method Stacks

Whenever JVM loads and runs Java Program. It needs Memory to Store several things like, Byte Code, Objects, Variables, etc.

* **Method Area**

For every JVM one method area will be available. Method Area will be created at the time of JVM Start up. Inside Method Area class Level Binary data including static variables will be stored. **Constant pools** of a class will be stored inside method area. Method area can access by multiple thread simultaneously. So it’s not Thread safe.

* **Heap Area**

For every JVM one heap Area is available. Heap Area will be created at the time of JVM Start UP. Objects and correspond instance variables will be stored in the heap Area. Every Array is Object only. Hence Arrays also will be stored in the Heap Area. Heap Area can be accessed by multiple Threads. And hence the data stored in the heap memory is not thread safe. Heap Area need not be continuous.

**Note**

Java application can communicate with JVM by using Runtime Object. Runtime Class present in java.lang package and it is a singleton class. We can create Runtime object as Follows –

**Runtime r = Runtime.getRuntime()**

**Methods –**

maxMemory() -> It returns max memory allocated to heap

totalMemory() -> It returns total memory allocated to heap

freeMemory() -> it returns number of free memory allocated in heap

**Stack Memory:**

For Every Thread JVM will create separate stack at the time of thread creation. Each and every method call performed by that thread will be stored in the stack including local variables also. After completing method corresponding entry from the stack will be removed. After completing all method calls, the stack will become empty. And that stack will be destroyed by the Terminating the thread. Each entry in the stack is called **Stack Frame or Activation Record.**

The data stored in stack is available for the corresponding thread and not available to remaining thread hence this data is Thread safe.

**Stack Frame Structure:**

* Local variable array
* Operand stack
* Frame Data

**Local Variable**

It contains all parameter and local variable of the method. Each slot in the array is of four bytes. Values of type int, float and reference occupy one entry in the array. Values of type Double and long occupy two consecutive entries in the arrays. Byte and short will be promoted into int type before storing and occupy one slot. But way of storing Boolean values is varied from JVM to JVM. But most of JVM follows one slot for Boolean values.

**Operand Stack**

JVM uses operand stack as workspace. Some instruction can push values to operand stack and some instructions can pop values from operand Stack and some instructions can perform required operations.

**Frame Data**

It contains all symbolic methods related to that method. It also contains reference to exception table which provides corresponding catch block information in the case of exception.

**PC Registers (program counter Registers)**

For every Thread separate PC register will be created at the time of Thread Creation. PC registers contains the address of current executing instructions. Once instruction execution completes automatically PC Register will Incremented to hold address of next instructions.

**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.

Note

* Method Area, Heap Area and Stack Area are considered as important memory areas with respect to programmer.
* Method Area and Heap Area are per JVM. Whereas Stack Area, PC Register and Native Stack Area are per Thread.
* Static variable will be Stored in method Area.
* Instance Variables will be stored in Heap Area.
* Local Variables will be stored in Stack Area

**Execution Engine**

This is the central component of JVM. It is responsible to execute java class file. Execution Engine mainly contains two components

* 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 interpret is it interpret every time even same method invoked multiple times which reduces performance of the system. To overcome this problem SUN people introduced JIT Compiler in 1.1V.

**JIT Compiler:**

It main purpose is to improve the performance. JIT compiler maintains a separate count for every method. Whenever JVM come across any method call. First 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 value then JIT compiler identifies that method is repeatedly used method. Such methods are called Hotspot. Immediately JIT compiler compiles that method and generates corresponding native code. Next time JVM come across method call then JVM uses native code directly and executes it instead of interpreting once again. So that performance of the system will be improved. The threshold count varied from JVM to JVM.

Some advance JIT compilers will recompile generated native code if count reaches the value second time so that more optimised mission code is generated. Internally **Profiler** which is part of JIT compiler which is responsible to identify Hotspot.

**Java Native interface (JNI)**

It act as mediator for java method calls and corresponding and corresponding native library that is JNI is responsible to provide information about native library to the JVM.

Native method Library provides/holds Native library information.