

JVMis an application-based VM, which is a part of the Java Run Time Environment(JRE) and is responsible to load and run java class files. JVM is the one which calls the **main**method of a java program.

The JVM is divided into three main subsystems:

1. ClassLoader Subsystem
2. Memory Area Subsystem
3. Execution Engine Subsystem

**1. ClassLoader Subsystem:**It is responsible for Loading, Linking, Initialization.

**i) Loading:**

In the loading phase, the class loader reads each .class file and store corresponding binary data in the **Method area**.

**Types of class loaders:**  
Classloader subsystem contains 3 types of subsystems and JVM follows the “Delegation-Hierarchy principle” to load classes.  
**1. Bootstrap class loader   
2. Extension class loader  
3. Application class loader**

**1.Bootstrap class loader:**This is responsible to load core java API classes(such as string class, string buffer etc.)

**2. Extension class loader:**This is the child class of Bootstrap class loader.  
This class loader is responsible to load classes from extension class path (jdk/jre/lib/ext/ .jar) etc.

**3. Application class loader/ System class loader:**This is the child class of extension class loader. This class loader is responsible to load classes from the application classpath. It internally uses the environment variable classpath.

**ii) Linking:**Linking consists of 3 activities Verify, Prepare, Resolve.  
**a) Verify/ Verification:**It is the process of ensuring that, the binary representation of a class is structurally correct or not, i.e inside JVM, the **Byte-code verifier** will check whether the .class file is generated by a valid compiler or not, whether .class file is properly formatted or not.

**b) Prepare/Preparation:**In this phase, JVM will allocate memory for class level static variables and assign default values.

**c) Resolve/Resolution:**It is the process of replacing symbolic memory references/names in our program with the original memory references from the Method area.

**iii) Initialization:**  
In this, all static variables are assigned with original values and static blocks will be executed from parent to child and from top to bottom.

## 2. Memory Area Subsystem:

Whenever JVM loads and runs a java program it needs memory to store several things such as byte code, objects, variables etc. Total JVM memory is organized into the following 5 categories.

1. **Method area:** For every JVM, one method area will be available. Method area will be created at the time of JVM startup. Inside the method area, class level binary data including static variables will be stored. Constant pools of a class will be stored inside the method area.
2. **Heap Area:**For every JVM, one heap area is available. Heap area will be created at the time of JVM startup. Objects and corresponding instance variables will be stored in the heap area.

**iii. Stack Area:**For every thread, JVM will create a 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 too. Each stack frame contains Local Variable Arrays, Operand Stack, Framed Data.

* 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** act 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.

1. **PC registers:**For every thread, separate PC(Program counter) registers will be created at the time of thread creation. PC registers contain the address of the currently executing instruction, once instruction execution completes automatically PC register will be incremented to hold the address of the next instruction.`
2. **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. Why will Java use native method?

Native methods allow you to use code from other languages such as C or C++ in your java code. You use them when java doesn't provide the functionality that you need.

## 3. Execution Engine:

This is the central component of JVM and is responsible to execute java class files. It mainly contains three components:

1. **Interpreter:** This is responsible for reading byte code and interpret it into sample machine code(native code) and executes the machine code line by line. **The drawback of the interpreter is, it interprets every time, even if the same method is invoked multiple times, which reduces the performance of the system.**
2. **JIT(Just In Time) Compiler:** The main purpose of the JIT compiler is to improve performance. **Next time JVM comes across that same method call, then JVM uses native code directly and executes it instead of interpreting it once again** so that performance of the system will be improved.. JIT compilation is applicable only for repeatedly required methods and native libraries.
3. **Garbage Collector**: GC Collects and removes unreferenced objects.

**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:** It is a Collection of the Native Libraries which is required for the Execution Engine.