JVM (Java Virtual Machine)

Virtual Machine (VM)

> Software stimulation of a machine which can perform operations like physical machine is called Virtual Machine.
> 2 Categories of VM with respect to Programming:

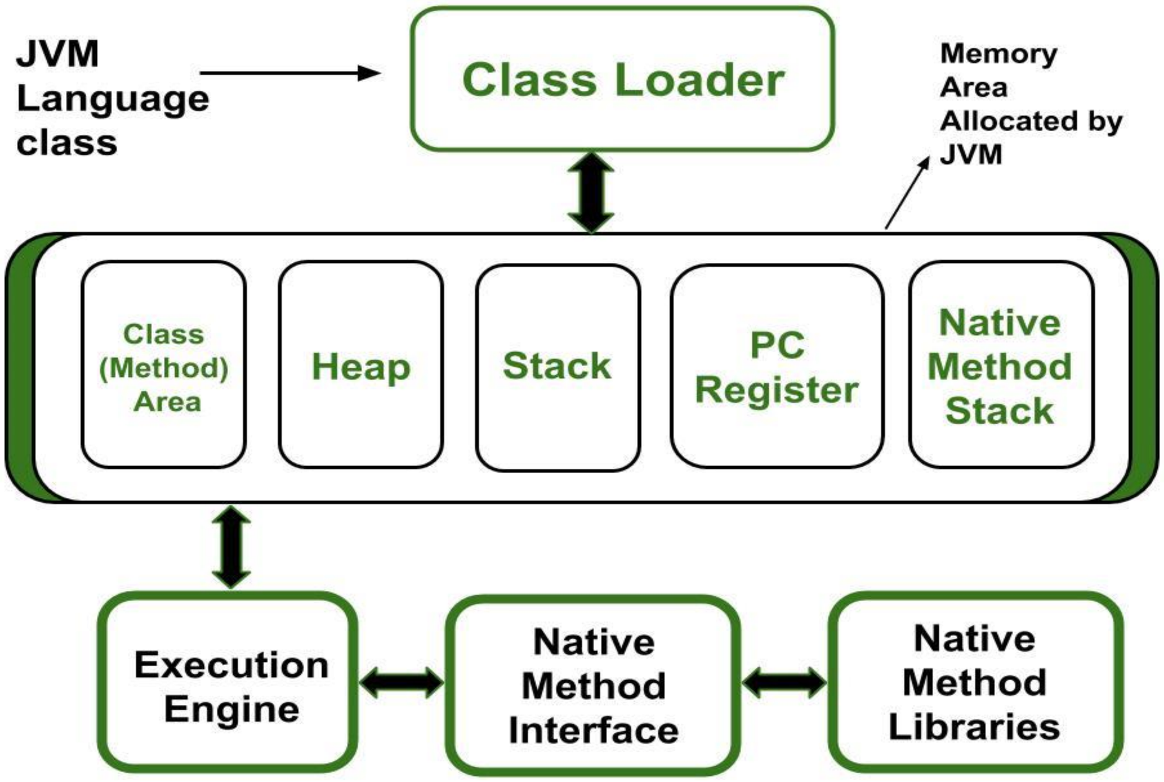
1. Hardware based VM/ System based VM
2. Application based VM/ Process based VM

1. **Hardware based VM**: - It provides several logical systems on the same computer with the strong isolation from each other i.e. on one physical machine; we are defining multiple logical machines.
> Main Advantage of H/w based VM is H/w resources utilization & to improve utilization of H/w resources.
For E.g. KVM (Kernel based VM) for Linux system, VMWARE, Cloud Computing etc.

2**. Application based VM**: - These VM machines act as runtime engines to run a particular programming language application.
For E.g. JVM acts as runtime engine to run Java based applications.
PVM acts as runtime engine to run Perl based applications.
CLR acts as runtime engine to run .NET based applications.

> *JVM is a part of JRE (part of JDK)*

.class file **Class Loader Subsystem**



Class loader subsystem reads .class file & store it inside JVM Memory. **Execution Engine** is responsible for reading the .class file & executing it which results in displaying the corresponding output to the console.

Class Loader Subsystem
> It is responsible for 3 activities

1. Loading
2. Linking
3. Initialization

1. **Loading**: - Loading means reading class file & store the corresponding binary data in Method Area.
> For each class file, JVM will store corresponding information in the method area like Fully Qualified name of class, immediate parent class, Method information, variable information, constructors info, modifiers info, constant pool info etc.
> After loading .class file, JVM immediately creates an object for that loader class on the heap memory of type **java.lang.Class** (object called as Class class object).
> This Class class object can be used by Programmer to get class level information like method info, variable info or constructor info etc.
> Refer programs from repository.

2. **Linking**: - It mainly contain 3 activities

a) **Verify**: - It is the process of ensuring that binary representation of a class is structurally correct or not (formatted or not) i.e. JVM will check whether the .class file generated by a valid compiler or not.
> Internally, **Bytecode verifier** is responsible for this activity & it is a part of class loader subsystem.
> If verification fails, then we will get runtime exception saying **java.lang.verifyError.**
> **This is why Java is Secure.**

b) **Prepare**: - In this process, JVM will allocate memory to the class level static variables & assign default values (Not original value).

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

3. **Initialization**: - In this activity, all static variables are assigned with original values & static blocks will be executed from parent to child.

Note:
> While loading, linking & initialization if any error occurs, we will get runtime exception saying **java.lang.linkageError (java.lang.verifyError is child of linkageError).**

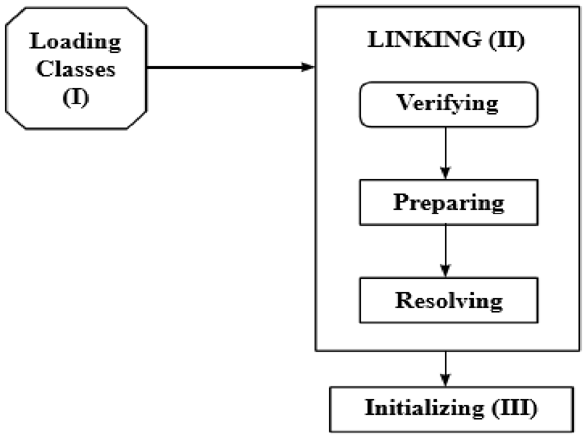


Fig: Class Loader Subsystem

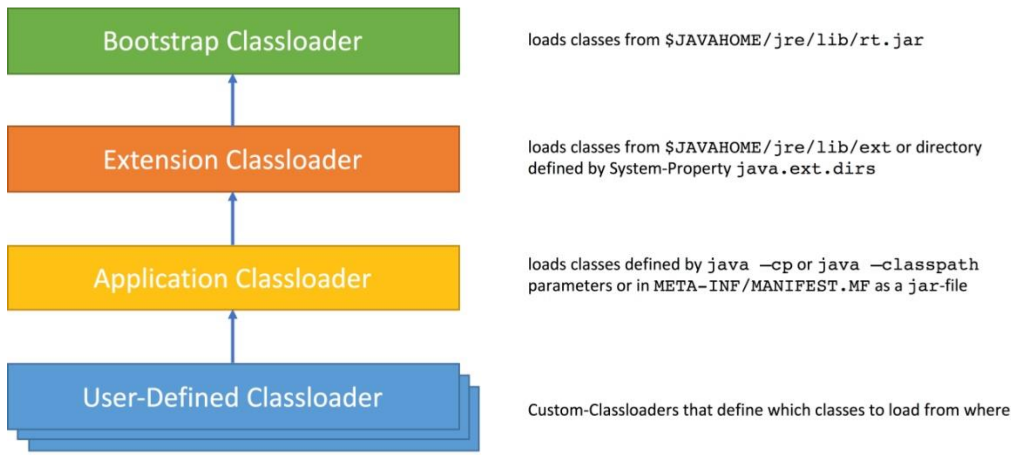
Types of Class Loader Subsystem

1. **Bootstrap class loader**: - It is responsible for loading core java API classes i.e. classes present in **rt.jar** from bootstrap classpath i.e. JDK -> JRE -> lib -> rt.jar
> By default bootstrap class loader is available with every JVM.
> Not implemented in Java (implemented in native language like C/C++)

2. **Extension class loader**: - It is the child class of Bootstrap class loader. It is responsible for loading classes from extension classpath i.e. JDK -> JRE -> lib -> ext -> *.jar
> Implemented in Java & corresponding .class file is **sun.misc.Laucher$ExtClassLoader.class**

Note: If in a class filename $ symbol is present i.e. **ExtClassLoader** is an inner class of **Laucher** outer class.

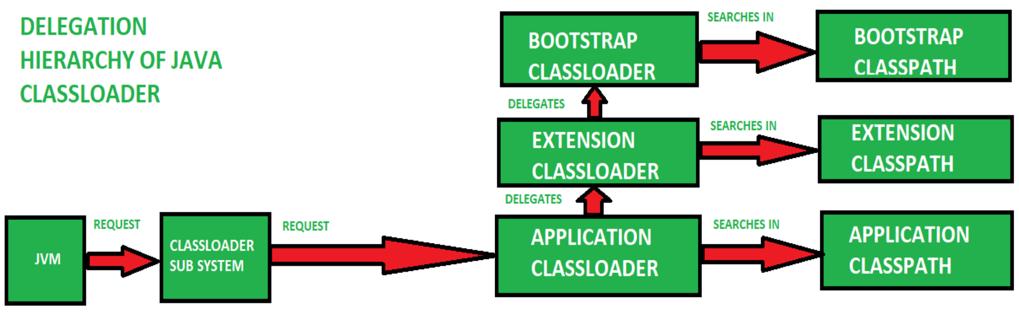
3. **Application class loader/ System class loader**: - It is the child class of Extension class loader. It is responsible to load classes from application classpath (internally uses environment variable classpath).
> Implemented in Java & corresponding .class file is **sun.misc.Laucher$AppClassLoader.class**


Bootstrap Classloader
loads classes from $JAVAHOME/jre/lib/rt.jar
Extension Classloader
loads classes from $JAVAHOME/jre/lib/ext or directory defined by System-Property java.ext.dirs
Application Classloader
loads classes defined by java –cp or java –classpath parameters or in META-INF/MANIFEST.MF as a jar-file
User-Defined Classloader
Custom-Classloaders that define which classes to load from where

Q. How class Loader works?

> Class loader follows Delegation Hierarchy principle(algorithm).
> Whenever JVM come across a particular class, first it will check whether corresponding .class file is already loaded or not.

IF (loaded in method Area) -> JVM will consider the loaded class.
ELSE JVM requests class Loader subsystem to load that particular class
> Class loader subsystem hand over the request to
Application class loader ---- delegates the request to ------ **Extension class loader** ---- delegates the request to ------**Bootstrap class loader**.



> The Bootstrap class loader will search in bootstrap classpath. If it is found, then corresponding .class file will be loaded by it else the bootstrap class loader will delegate the request to the Extension class loader & so on delegation goes.
> If the class is not found in Application class loader, we will get runtime exception saying **NoClassdefFoundError** or **ClassNotFoundException**.

Q. Need for customizer class loader?

> Default class loader will load .class file only once even though we're using multiple time that class in our program. After loading .class file, if it is modified outside then default class loader won't load updated version of class file as it is already available in method area.

>To resolve this problem, we can define our customizer class loader.
> The main advantage of Customizer class loader is we can control class loading mechanism based on our requirement.
> To define customizer class loader
-> Extend **java.lang.ClassLoader** class (base class for all customizer class loader) & override **loadClass** method.

> While designing/developing web servers & application servers usually, we go for customizer classloader to customize class loading mechanism.

Various Memory Areas in JVM

> Whenever JVM loads & runs a Java program, it needs memory to store several things like bytecode, objects, variables etc.
> Total JVM memory is organized into following 5 categories :-

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

1. Method Area
> For every JVM, one method area will be available.
> Method area will be created at the time of JVM startup.
> 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 be accessed by multiple threads simultaneously; hence data stored in the method area is not thread safe.
> Method area needs not to be continuous.

2. Heap Area
> For every JVM, one heap area is available.
> Heap area will be created at the time of JVM startup.
> Objects & the corresponding instance variables will be stored in the heap area.
> Every array in Java is object only; hence array also will be stored in the heap area.
> Heap area can be accessed by multiple threads simultaneously; hence data stored in the method area is not thread safe.
> Heap area needs not to be continuous.
> Heap memory is finite/fixed memory but based on our requirement we can set maximum & minimum heapsize.

-> We can use the following flags with java command

a) –**Xmx** to set maximum heap size.

E.g. java –Xmx512m className

-> This command will set maximum heapsize as 512 mb

b) –**Xmx** to set minimum heap size

E.g. java –Xms64m className

-> This command will set minimum heapsize (total Memory) as 64 mb

Note: A java application can communicate with JVM by using **Runtime** class object. Runtime class is present in java.lang package & it is a singleton class. We can create a Runtime object

```
Runtime r = Runtime.getRuntime ();
```

> Refer program for "displaying heap memory statistics" by using Runtime class object.

3. Stack memory

> For every thread, JVM will create a separate stack at the time of thread creation.
> Each & every method calls performed by the thread will be stored in the stack including local variables too.
> After completing a method, the corresponding entry from the stack will be removed. After completing all method calls, the stack will become empty & that empty stack will be destroyed by the JVM just before terminating the thread.
> Each entry in the stack is called stack frame or Activation record.
> The data stored in the stack is available only for the corresponding thread & not available to the remaining threads. Hence this data is thread safe.

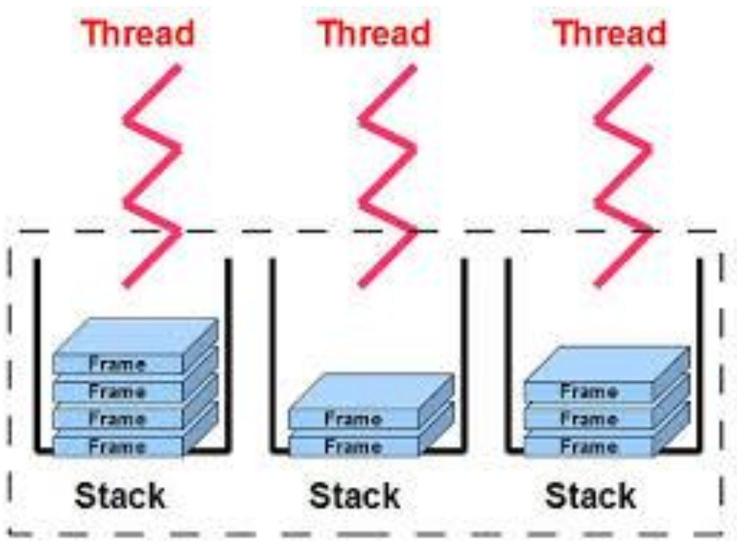


Fig : Stack Memory

> **Stack Frame also called Activation Record.**
> Each activation Record contains 3 parts

Local Variable Array
Operand Stack
Frame Data

Fig: Stack Frame/Activation Record

a) Local Variable Array

> It contains all the parameters & local variables of the method.
> Each slot in the array is of 4 bytes.
> Values of type int, float & reference occupy 1 entry/slot in the array.
> Values of double, long occupy 2 consecutive entries in the array.
> Byte, short & char values will be converted to int type before storing & occupy 1 slot.
> But the way of storing Boolean values is varied from JVM to JVM. But most of the JVM follow 1 slot for Boolean values.

E.g.

```
public void method1 (int i, double d, Object o, float f) {
        long x;
        …..
}
```

i	d	d	o	f	x	x
0	1	2	3	4	5	6

b) Operand Stack

> JVM uses operand stack as workspace.

> Some instructions can push values to the operand stack & some instructions can pop values from operand stack & some instructions can perform required operations.

c) Frame data

> Frame data contains all symbolic references related to that method (constant pool).

> It also contains a reference to exception table which provides corresponding catch block information in case of exceptions.

4. PC registers (Program counter registers)

> For every thread, a separate PC register will be created at the time of thread creation.

> PC registers contains the address of current executing instruction.

> Once instruction execution completes, automatically PC registers will be incremented to hold address of next instruction.

> Internally it is used by JVM.

5. 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: For every thread, one normal stack & one native method stack will be created. If it's a normal java method then it will be stored in normal runtime stack & if it's a Native method call, then it will be stored inside Native method stack.

> Native methods are those methods which are implemented in non Java language.

For E.g. hashCode ();

> Internally it is used by JVM.

Conclusions

1. Method area, Heap area & Stack area are considered as important memory areas with respect to programmer.
2. Method area & Heap area are as per JVM whereas Stack area, PC registers & Native method stack are as per thread.
3. > Static variables will be stored in Method area.
   > Instance variables will be stored in Heap area.
   > Local variables will be stored in Stack area.

E.g. (Very Important Example for understanding point of view)

```
Class Test {
      Student s1 = new Student ();   // instance variable (created when object is created)
      static Student s2 = new Student ();
      public static void main (String[] args) {
            Test t = new Test ();
            Student s3 = new Student ();
      }
}
```

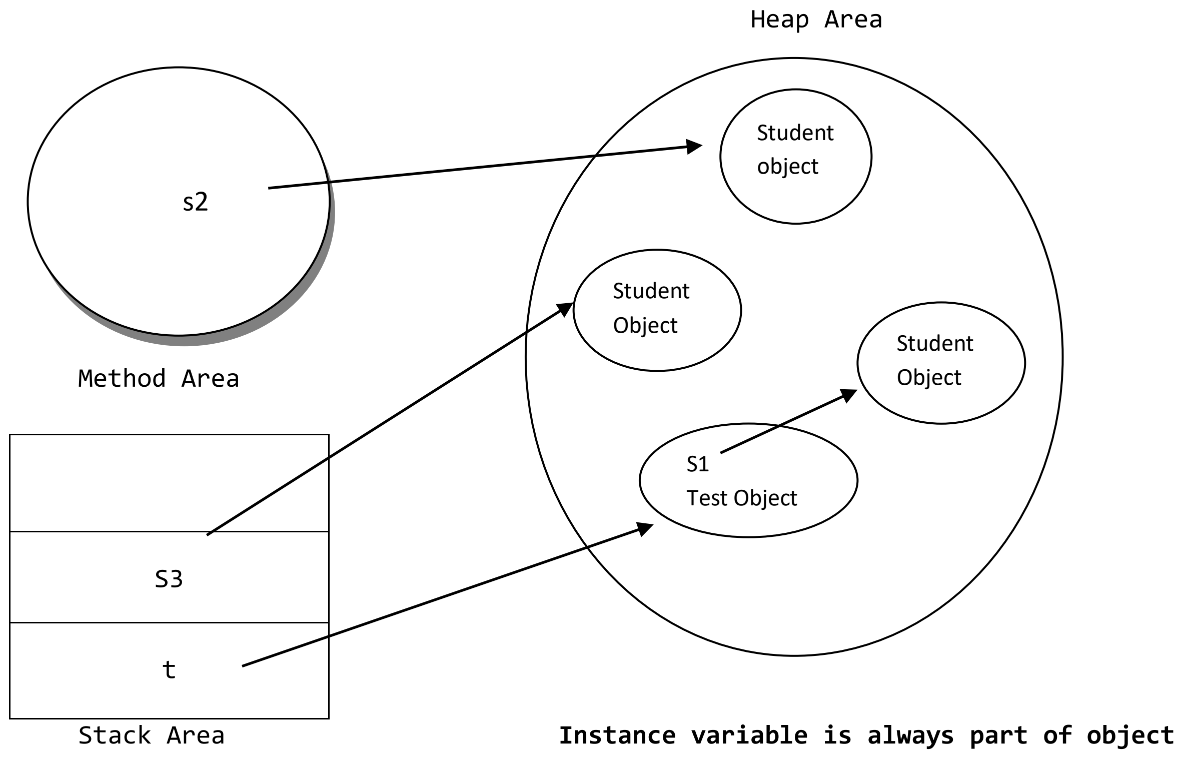


Fig: Instance creation & references

Execution Engine

> This is the central component of JVM.
> Execution engine is responsible to execute Java class file.
> It has mainly 2 components

a) Interpreter
b) JIT Compiler

a) **Interpreter**
> It is responsible for 3 activities:

First: reading bytecode.
Second: interpret that bytecode into machine code (native code).
Third: execute that machine code line by line.

> The problem with interpreter is it interprets every time even same method invoked multiple times which reduces performance of the system.
> To overcome this problem, JIT compiler was introduced.

b) **JIT Compiler (Just In time)**
> 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, first that method will be interpreted normally by the interpreter & JIT compiler increment corresponding count variable. This process will be continued for every method.
> Once if any method count reaches threshold value (threshold count varies from JVM to JVM) then JIT compiler identifies that method is a repetitively used method (also called **Hotspot method**). JIT compiler immediately compiles that method & generates its native code.
Next time if JVM comes across same method call, then JVM uses native code directly & executes it instead of interpreting once again & that's how JIT compiler improves the system's performance.

> Some advance JIT compilers will recompile generated native code if count reaches threshold value 2nd time, so that more optimized machine code will be generated.
> Internally Profiler, which is a part of JIT compiler, is responsible to identify hotspots.

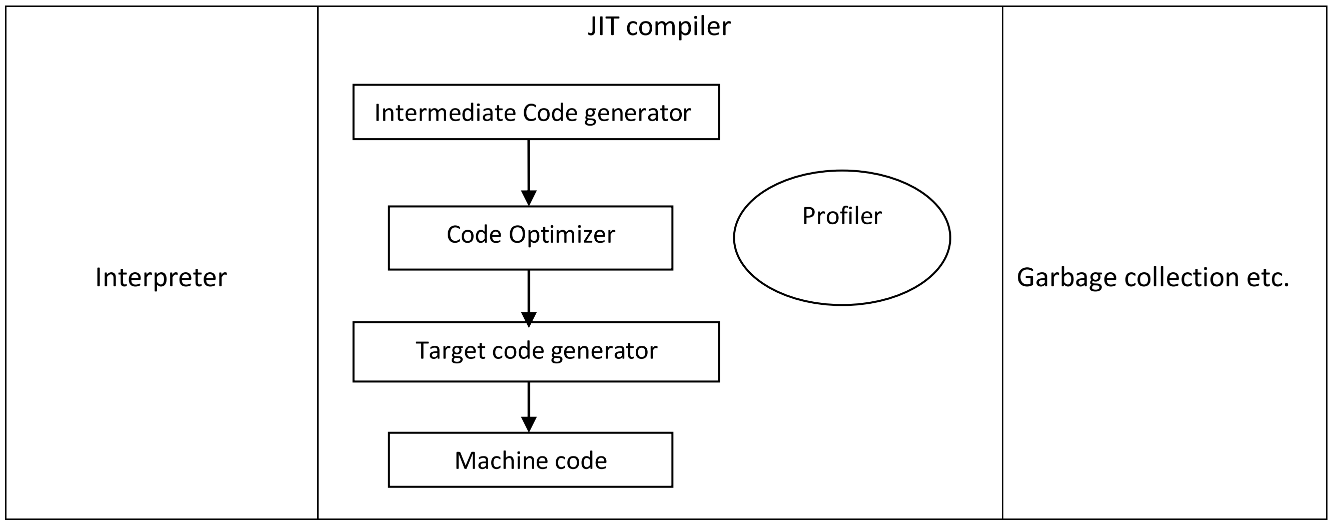


Fig: Execution Engine

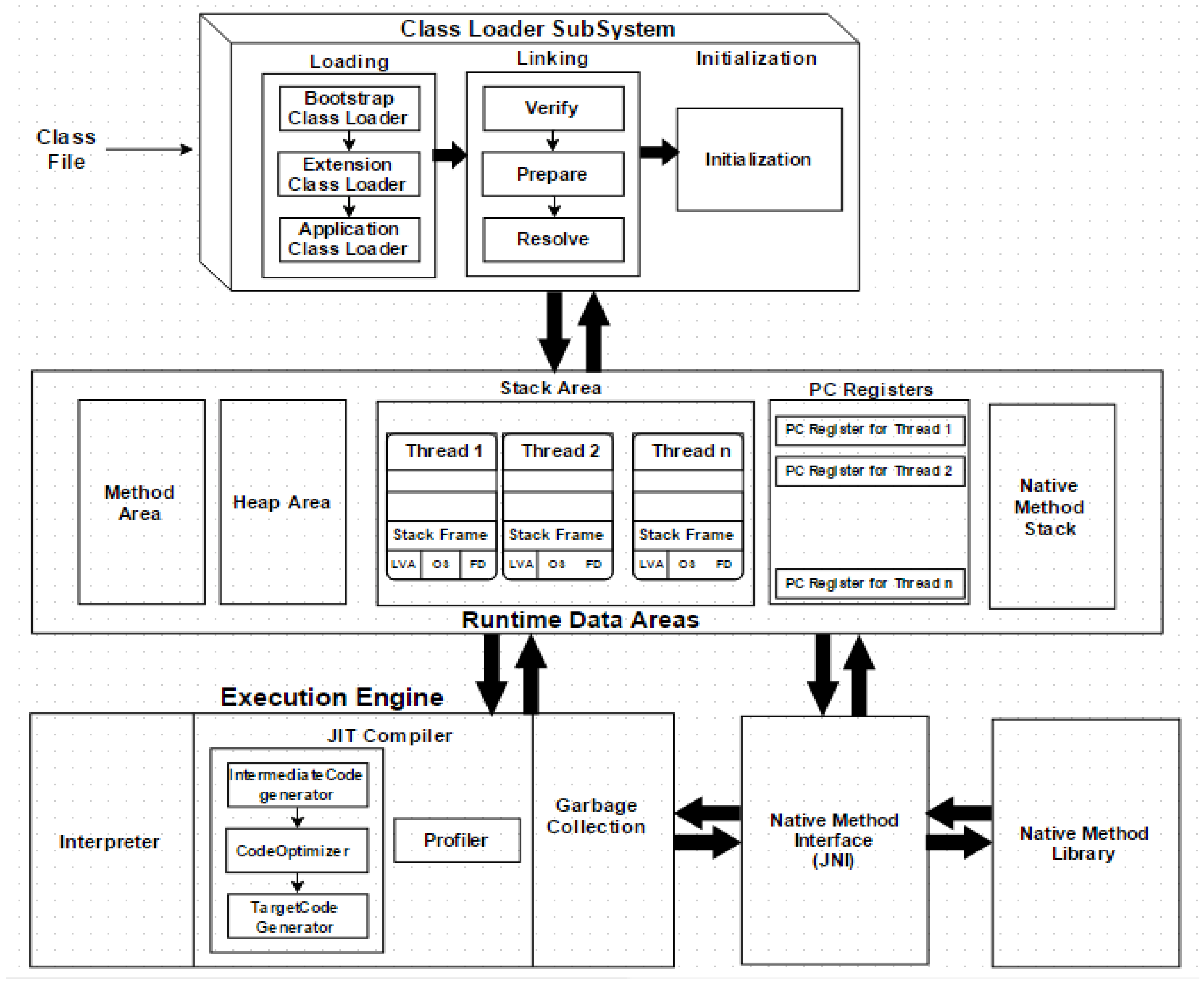
Java Native Interface (JNI)

> JNI acts as a mediator for java method calls & corresponding native libraries i.e. JNI is responsible to provide information about native libraries to the JVM.

> Native Method library provides/holds native libraries information.

$$\begin{tikzcd}
\text{Execution Engine} \arrow[r, shift left] & \text{Java Native Interface (JNI)} \arrow[l, shift left] & \text{Native Method library} \arrow[l]
\end{tikzcd}$$

Complete JVM Architecture



Class Loader SubSystem
Class File
Loading
Bootstrap Class Loader
Extension Class Loader
Application Class Loader
Linking
Verify
Prepare
Resolve
Initialization
Initialization
Stack Area
Method Area
Heap Area
Thread 1
Thread 2
Thread n
Stack Frame
LVA
OS
FD
PC Registers
PC Register for Thread 1
PC Register for Thread 2
PC Register for Thread n
Native Method Stack
Runtime Data Areas
Execution Engine
Interpreter
JIT Compiler
Intermediate Code generator
Code Optimizer
Target Code Generator
Profiler
Garbage Collection
Native Method Interface (JNI)
Native Method Library