JVM (Java Virtual Machine) Architecture

Virtual Machine :—

→ It is a software simulation of a machine which can perform operations like a physical machine.

→ There are 2 types of virtual machines.

1. Hardware based (or) System based virtual machines
2. Application based (or) Process based virtual machines

1. Hardware based (or) System based Virtual machines :—

→ It provides several logical systems on the same computer with strong isolation from each other.

Ex: KVM (Kernel based virtual Machice for LINUX systems)
VMWare
Xen
Cloud computing etc.

2. Application based (or) Process based Virtual Machines :—

→ These virtual machines acts as runtime engines to run a particular programming language applications.

Ex ①: JVM acts as Runtime Engine to run Java applications.

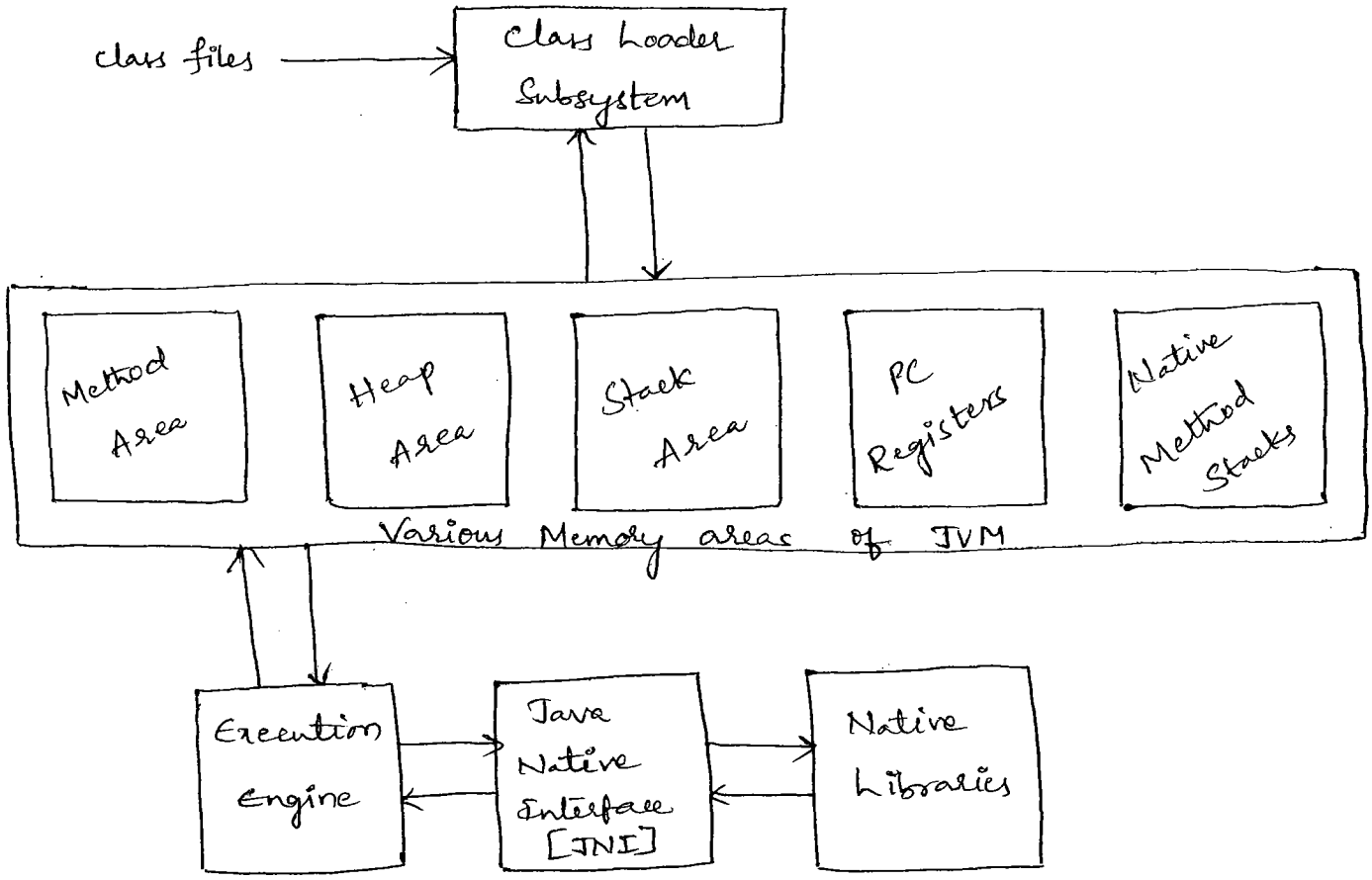
②. Parrat Virtual Machine acts as Runtime Engine to run scripting language applications like Perl.

③. CLR (Common Language Runtime) acts as Runtime Engine to run .Net applications.

JVM :—

→ It is the part of JRE (Java Runtime Environment).

→ JVM is responsible to load & run Java applications.



1) Class Loader Sub system:-

→ Class loader sub system is responsible for the following 3 activities.

1. Loading
2. Linking
3. Initialization

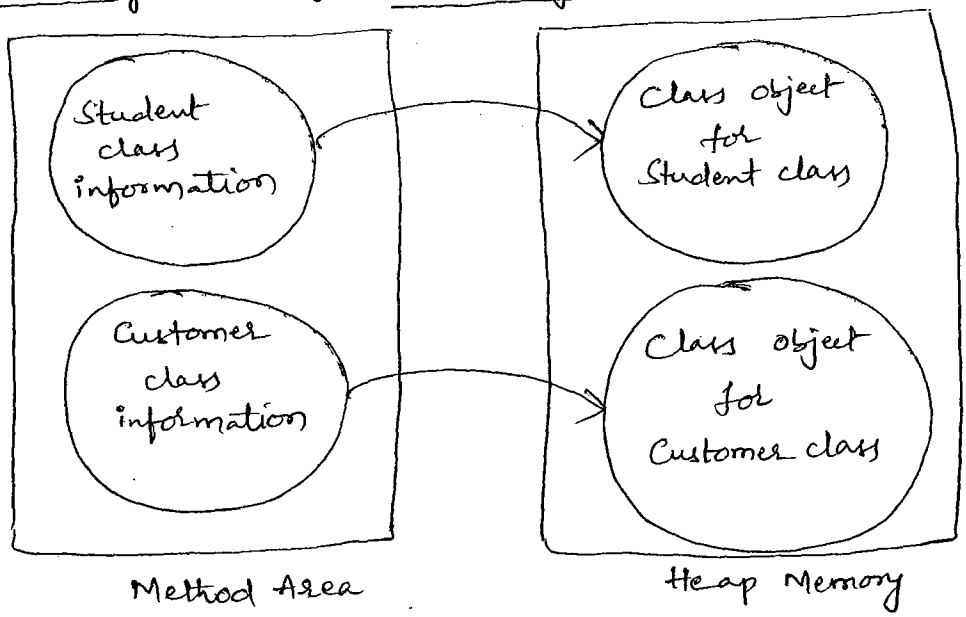
① Loading:-

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

→ For each class, file, JVM will store the following information on Method Area.

1. Fully qualified name of component (class / interface / enum).
2. Fully qualified name of parent (class / interface / enum).
3. Is .class file related to class / interface / enum?
4. Modifiers information.
5. Methods information.
6. Variables / Fields information.
7. Constant Pool.

→ After loading .class file JVM creates an object for that loaded class on the Heap memory with type java.lang.Class.



→ By using this class object programmer can get corresponding class information like its name, its parent name, constructors information, methods information, fields information etc.

Ex: String s = new String("durga");
S.o.p(s.getClass().getName());
O/P : java.lang.String.

Note:- For every loaded type only one Class object will be created, eventhough we are using that class multiple times in our application.

② Linking:-

→ Linking consists of the following 3 activities.

1. Verification
2. Preparation
3. Resolution.

1. Verification:-

→ It is the process of ensuring that binary representation of a class is structurally correct or not i.e., JVM will check whether the class file generated by valid compiler / not and whether class file properly formatted or not.

→ Internally Bytecode verifier is responsible for this activity.

→ Bytecode Verifier is the part of Class loader sub systems.

→ If verification fails then we will get RE saying, java.lang.VerifyError

2. Preparation:-

→ In this phase, JVM will allocate memory for class level static variables and assign default values (but not original values assigned to that variables).

Note:- Original values won't be assigned until initialization phase.

3. Resolution:-

→ It is the process of replacing symbolic names used by loaded type with original references.

→ Symbolic references are resolved into direct references by searching through Method Area to locate the referenee entity.

Ex:
```
class Test
{
    p s v m(String[] args)
    {
        String s1 = new String("durga");
    } } Student s = new Student();
```

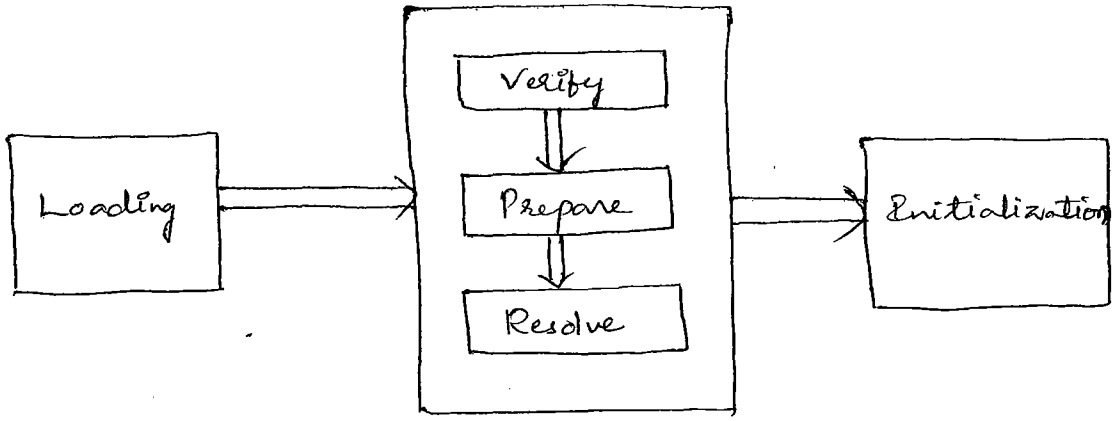
→ For the above class, class loader loads Test.class, String.class, Student.class and Object.class.

→ The names of these classes are stored in Constant pool of Test class.

→ In Resolution phase, these names are replaced with actual references from Method area.

③ Initialization :—

→ In this phase, all static variables will be assigned with original values & static blocks will be executed from parent to child, from top to bottom.



Loading of Java class

Note:- While Loading, Linking & Initialization if any error occurs then we will get RE saying, java.lang.LinkageError.

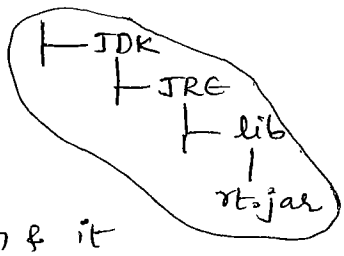
Types of class loaders :—

→ Every class loader subsystem contains the following 3 class loaders.

1. Bootstrap / Primordial class Loader
2. Extension class Loader
3. Application / System class Loader.

1. Bootstrap / Primordial class Loader :—

→ This class loader is responsible for loading Core Java API classes i.e., the classes present in rt.jar.



→ This location is called Bootstrap class path i.e., Bootstrap class loader is responsible for to load classes from Bootstrap class path.

→ Bootstrap class loader by default available in JVM & it is implemented in native languages like C, C++.

2. Extension Class Loader :—

→ This class loader is the child of Bootstrap class loader.

→ This class loader is responsible for to load class from extension class path (JDK\JRE\lib\ext).

→ This class loader is implemented in Java & the corresponding .class file name is sun.misc.Launcher$ExtClassLoader.class.

3. Application / System class loader :—

→ It is the child of Extension class loader.

→ It is responsible for to load classes from Application classpath.

→ It internally uses environment variable classpath.

→ It is internally implemented in Java by SUN people & the corresponding .class file name is sun.misc.Launcher$AppClassLoader.class.

How Class Loader Works ?

→ Class Loader follows Delegation Hierarchy principle.

→ Whenever JVM come across a particular class first it will check 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 requires class loader subsystem to load the particular class then class Loader subsystem handovers the request to Application Class Loader.

→ Application class Loader deligates request to Extension class Loader & Extension class Loader intern delegates that request to Bootstrap class Loader.

→ Bootstrap class Loader searches in Bootstrap class path (JDK\JRE\lib\rt.jar).

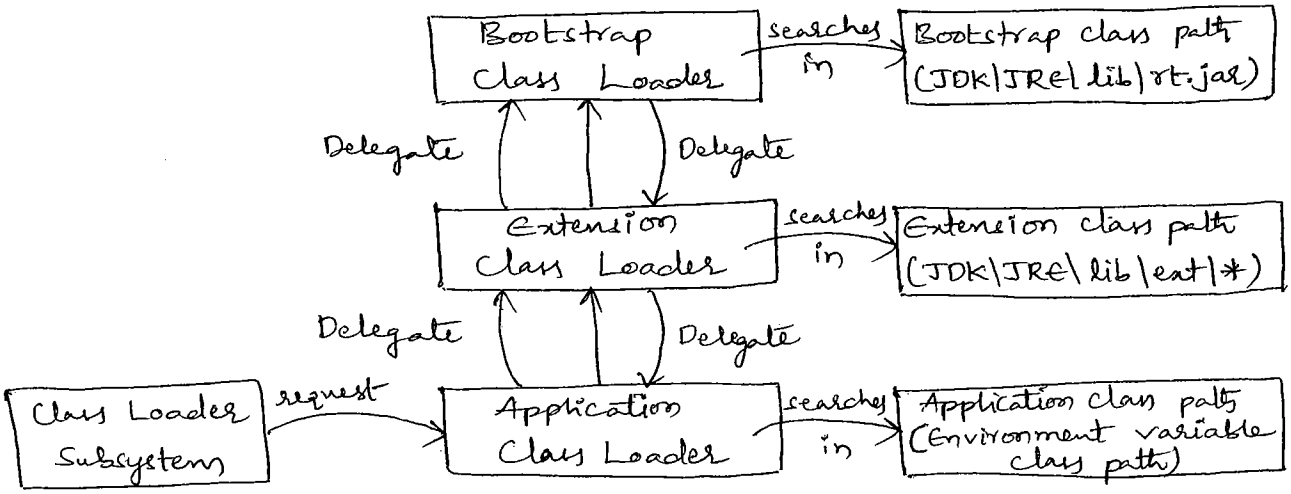
→ If the specified class is available then it will be loaded, o.w. Bootstrap class Loader delegates the request to Extension class Loader.

→ Extension class Loader will search in Extension class path (JDK\JRE\lib\ext*).

→ If the required class is available then it will be loaded, o.w. Extension class Loader delegates the request to Application class Loader.

→ Application class Loader will search in Application class path for the required .class file.

→ If the specified class is available then it will be loaded, o.w. we will get RE saying, ClassNotFoundException (or) NoClassDefFoundError.



Ex:
```
class Test
{
  p s v m(-)
  {
    S.o.p(String.class.getClassLoader());
    S.o.p (Student.class.getClassLoader());
    S.o.p (Test.class.getClassLoader());
  }
}
```

(Assume Student.class present both Extension & Application class path where as Test.class present only in Application class path)

For String.class:—

→ From Bootstrap class path by Bootstrap class Loader.

Olp : null.

For Student.class,

→ From Extension class path by Extension class Loader.

Olp : sun.misc.Launcher$ExtClassLoader@1234

For Test.class,

→ From Application class path by Application Class Loader.

Olp : sun.misc.Launcher$AppClassLoader@3567

Note: ① :- Bootstrap class Loader is not Java object and hence we got for the first S.o.p is null, but Extension & Application Class Loaders are Java objects & hence we are getting the proper o/p (ClassName@hexadecimal String of hashcode).

② Class Loader Subsystem will give highest priority for Bootstrap class path & then Extension class path followed by Application class path.

What is the need of Customized Class Loader?

→ Default Class Loaders will load .class file only once even though we are using multiple times that class in our program.

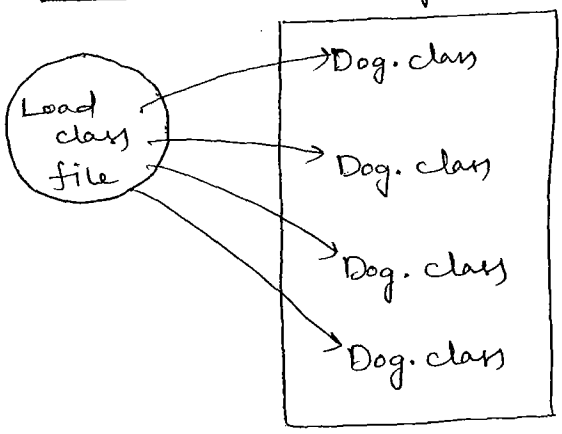
→ After loading .class file if it is modified outside then default Class Loader won't load updated version of class file (because .class file already there in method area).

→ We can resolve this problems by defining our own class loader.

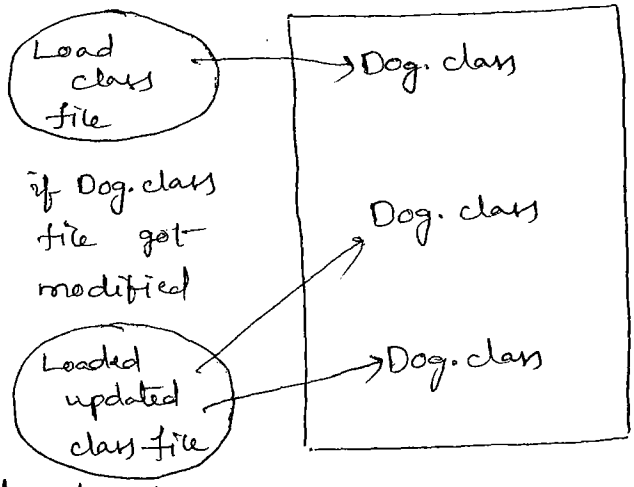
→ The main advantage of customized class loader is we can control class loading mechanism based on our requirement.

Ex: We can load class file separately every time so that updated version available.

Default Class Loading



Customized Class Loading



How to develop our own Customized class Loader:—

→ We can define our own Customized Class Loader while to customize class loading mechanism.

→ We can define our own customized class loader by extending java.lang.ClassLoader class.

Ex:
```
public class CustomizedClassLoader extends ClassLoader
{
    :
    public Class loadClass(String cname) throws ClassNotFoundException
    {
        || Read updated class file and returns it
    }
}
```

```
class Client
{
  P S V m(-) {
  Dog d1 = new Dog();
    |
  CustomizedClassLoader c = new CustomizedClassLoader();
  c.loadClass("Dog");
    |
  c.loadClass("Dog");
    |
  }
}
```

Note:- Usually we can define our own customized class loader while developing web servers and application servers.

***Q: What is the purpose of java.lang.ClassLoader class?

Ans: This class acts as base class for designing class loaders.

Every customized class loader class should extend java.lang.ClassLoader either directly or indirectly.

2. Various Memory areas of JVM:—

→ Whenever JVM runs a program it needs memory to store several things like byte code, objects, variables etc.

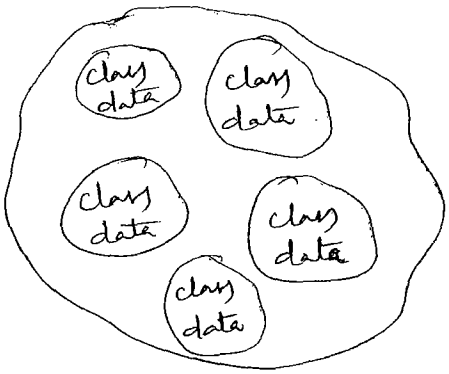
→ Total JVM memory organized in the following 5 categories.

1. Method Area
2. Heap Area
3. Stack Area
4. PC Registers (Program Counter)
5. Native Method Stacks

1. Method Area:—

→ Method Area stores Runtime constant pools, variables & methods information, static variables, byte code of classes & interfaces loaded by JVM.

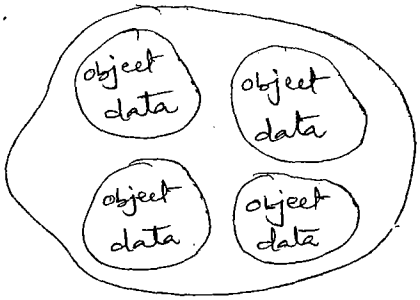
→ Method Area will be created at the time of JVM start up.
→ This memory area will be shared by all threads (shared / global memory).
→ This memory area need not be continuous.



Method area

2. Heap Area :—

→ It is the main important memory area to the programmer.
→ Heap Area will be created at the time of JVM start up.
→ Heap Area need not be continuous.
→ It will be shared by all threads (global / shared memory).
→ All objects & corresponding instance variables will be stored in the Heap memory.
→ Every Array in Java is an object and hence Arrays will be stored in the Heap memory.



Heap Memory

// program to display Heap Memory Statistics:

→ A Java application can communicate with JVM by using Runtime object.
→ Runtime class is a Singleton class present in java.lang package.
→ We can create Runtime object by using getRuntime() method.

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

→ Once we get Runtime object we can call the following methods on that object.

1. maxMemory();

It returns no. of bytes of max. memory allocated to the Heap.

2. totalMemory();

It returns no. of bytes of total memory allocated to the Heap (initial memory).

3. freeMemory();

It returns the no. of bytes of free memory present in the Heap.

```
class HeapDemo
{
  p s v m(-)
  {
    long mb = 1024*1024;
    Runtime r = Runtime.getRuntime();
    S.o.p ("Max Memory :"+r.maxMemory()/mb);
    S.o.p ("Total Memory :"+r.totalMemory()/mb);
    S.o.p ("Free Memory :"+r.freeMemory()/mb);
    S.o.p ("Consumed Memory :"+(r.totalMemory()-r.freeMemory())/mb);
  }
}
```

O/P (in terms of bytes): Max Memory : 66650112
total Memory : 5177344
Free Memory : 4995960
Consumed Memory : 181384

O/P (interms of MB's): Max Memory : 63
Total Memory : 4
Free Memory : 4
Consumed Memory : 0

Note:- Default Heap size 64 MB.

***
Q: How to set max. and min. Heap size?

Ans: Heap memory is final memory & based on our requirement we can increase & decrease Heap size.

We can use the following flags with Java command.

① -Xmx : To set maximum Heap size i.e., maxMemory().

Ex: java -Xmx128m HeapDemo↵

→ This cmd will set 128 MB as max. Heap size.

O/P : Max Memory : 127
Total Memory : 4
Free Memory : 4
Consumed Memory : 0

② -Xms : To set minimum Heap size i.e., totalMemory().

Ex: java -Xms64m HeapDemo↵

→ This cmd will set min. Heap size as 64 MB.

Ex: java -Xmx128 -Xms64m HeapDemo↵

O/P : Max Memory : 127
Total Memory : 63
Free Memory : 63
Consumed Memory : 0

3. Stack Memory:-

→ For every thread JVM will create a separate stack. Runtime stack will be created automatically at the time of thread creation.

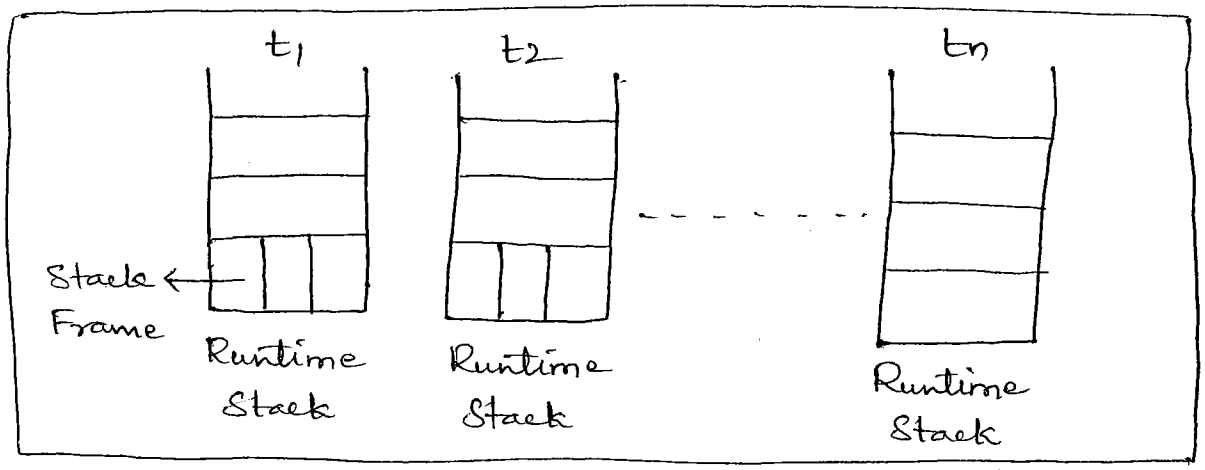
→ All method calls and corresponding local variables, intermediate results will be stored in the stack.

→ For every method call a separate entry will be added to the stack and the entry is called Stack Frame.

→ After completing that method the corresponding entry from the stack will be removed.

→ After completing all method calls just before terminating the thread runtime stack will be destroyed by the JVM.

→ The data stored in the stack is private to the corresponding thread.

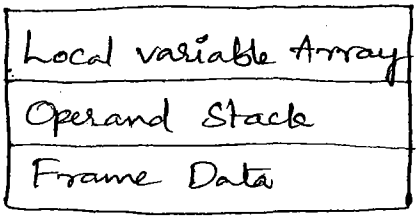


Stack Memory

Stack Frame Structure :—

→ Each Stack Frame contains 3 parts.

1. Local variable Array
2. Operand Stack
3. Frame Data



Stack Frame

1) Local variable Array :—

→ It contains all parameters and local variables of the method.

→ Each slot in the array is of 4 bytes.

→ Values of type int, float and reference occupy one entry in

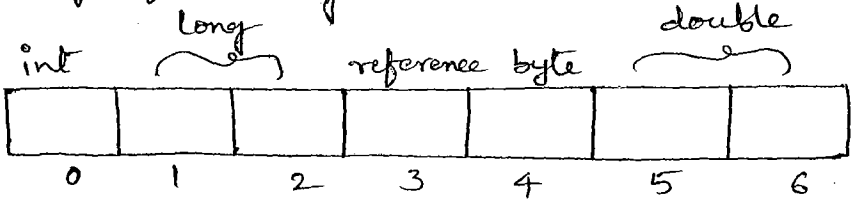
the array.

→ Values of type long and double occupy 2 consecutive entries in array.

→ byte, short and char values will be converted to int type before storing and occupy one slot.

→ But the way of storing boolean values is varied from JVM to JVM.

Eg:




```
public void m1(int i, long l, Object o, byte b, double d)
{
==
}
```

→ But most of the JVM follow one slot for boolean value.

2) Operand Stack :—

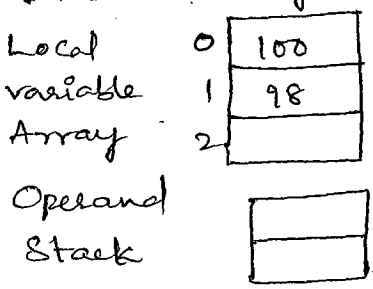
→ JVM uses operand stack as workspace.

→ Some instructions can push the values to the operand stack and some instructions pop the values from operand stack and store result once again to the operand stack.

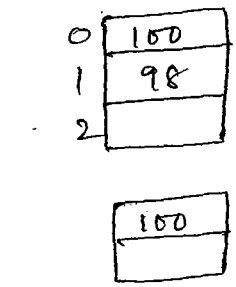
Program

```
iload-0
iload-1
iadd
istore-2
```

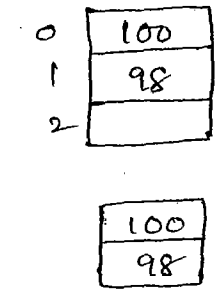
Before Starting



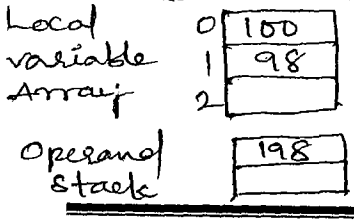
After iload-0



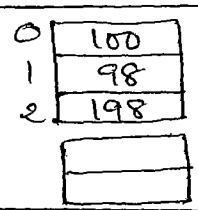
After iload-1



After iadd



After istore-2



3) Frame Data :—

→ Frame Data contains all symbolic references (constant pool) related to that method

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

4. PC (Program Counter) Registers :—

→ For every thread a separate pc register will be created at the time of thread creation.

→ PC registers contain address of current executing instruction. Once instruction execution completes automatically PC register will be incremented to hold address of next instruction.

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 ① :— Method Area, heap & stack are considered as major memory areas w.r.t. programmer's view.

② Method area and heap area are for JVM whereas stack, PC registers and native method stack are for thread i.e., one separate heap for JVM, one separate method area for every JVM, one stack for every thread, one separate pc register for every thread and one separate native method stack for every thread.

③ 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.

3. Execution Engine :—

→ This is central component of JVM.

→ Execution Engine is responsible to execute Java class files.

→ Execution engine mainly contains 2 components for executing Java classes.

1. Interpreter
2. JIT compiler

1) Interpreter :—

→ It is responsible to read byte code and interpret into machine code (Native code) and 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 SUN people introduced JIT compiler in 1.1 version.

2) 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 value then JIT compiler identifies that method is 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.

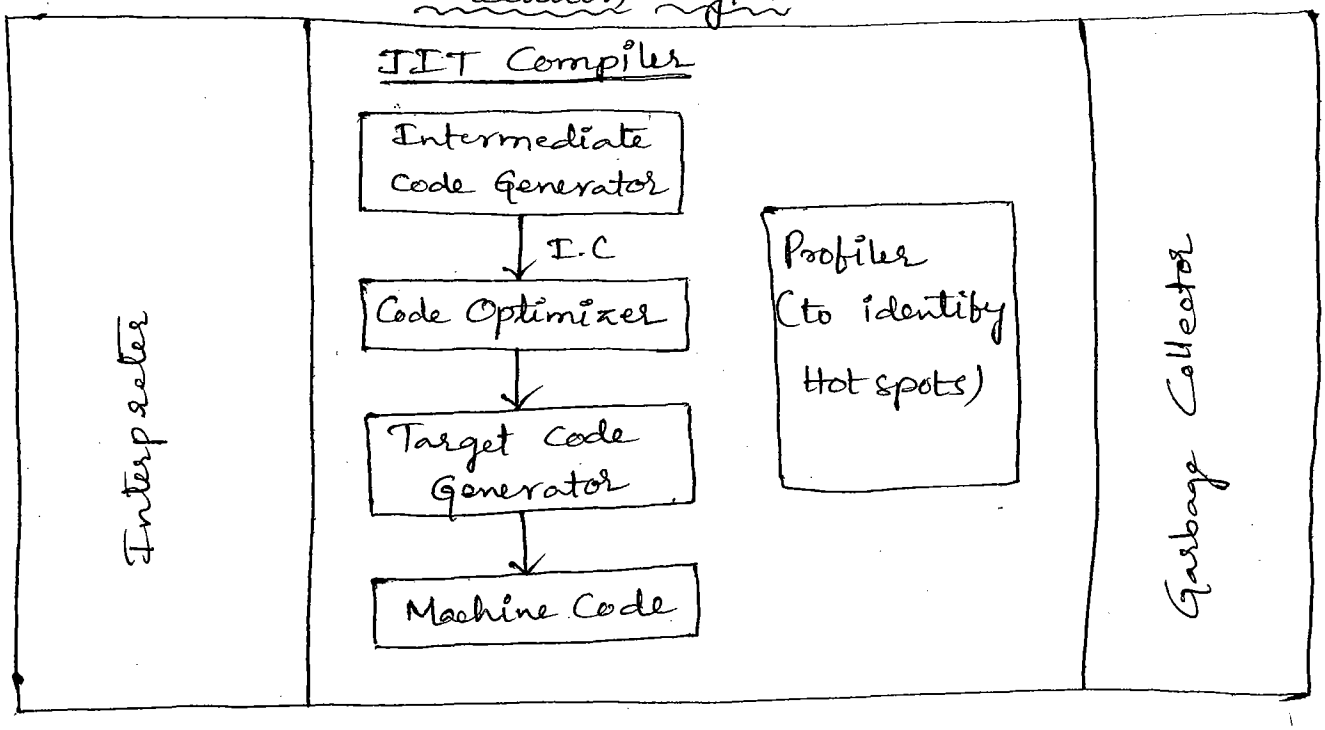
→ The Threshold count varying from JVM to JVM. Some advanced JIT compilers will recompile generated native code if count reaches threshold value second time. So that more optimized machine code will be generated.

→ Profiler which is the part of JIT compiler is responsible to identify Hot-spots.

→ JVM interprets total program line by line atleast once.

→ JIT compiler is applicable only for repeatedly invoked methods but not for every method.

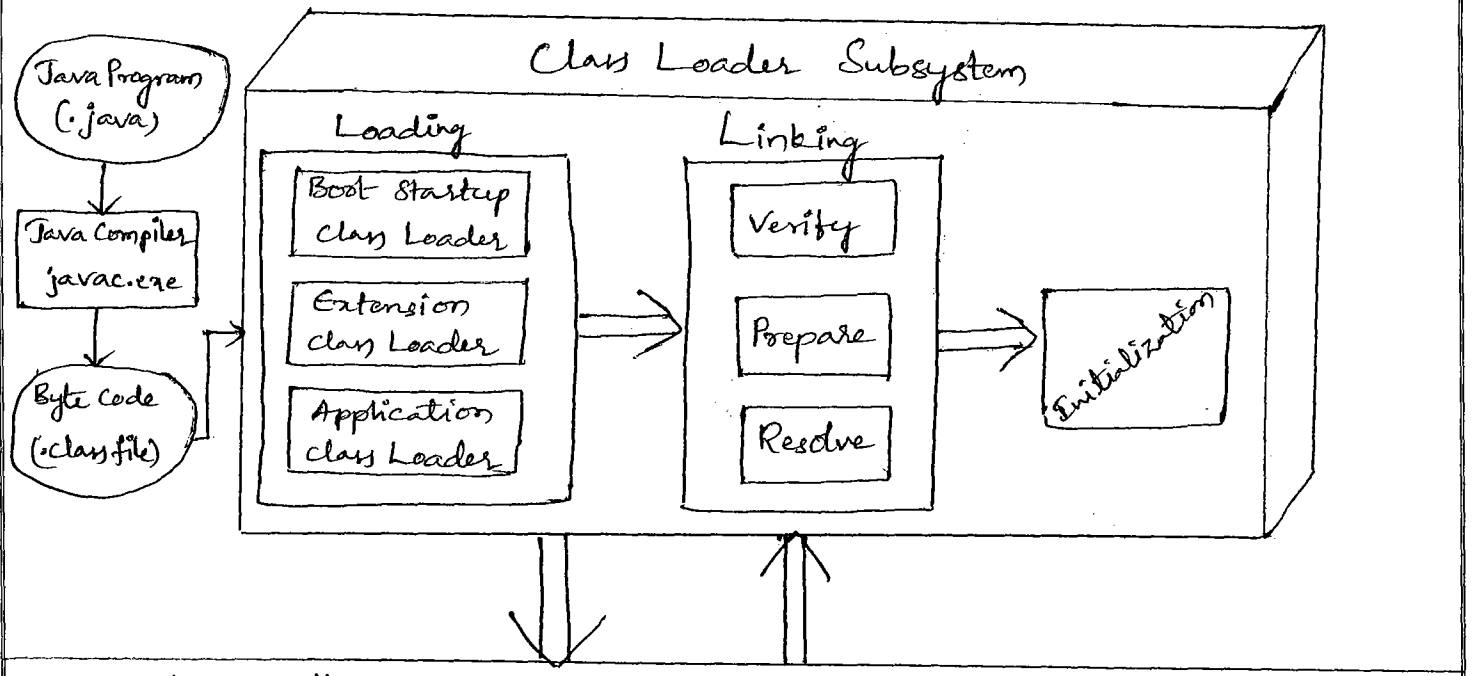
Execution Engine

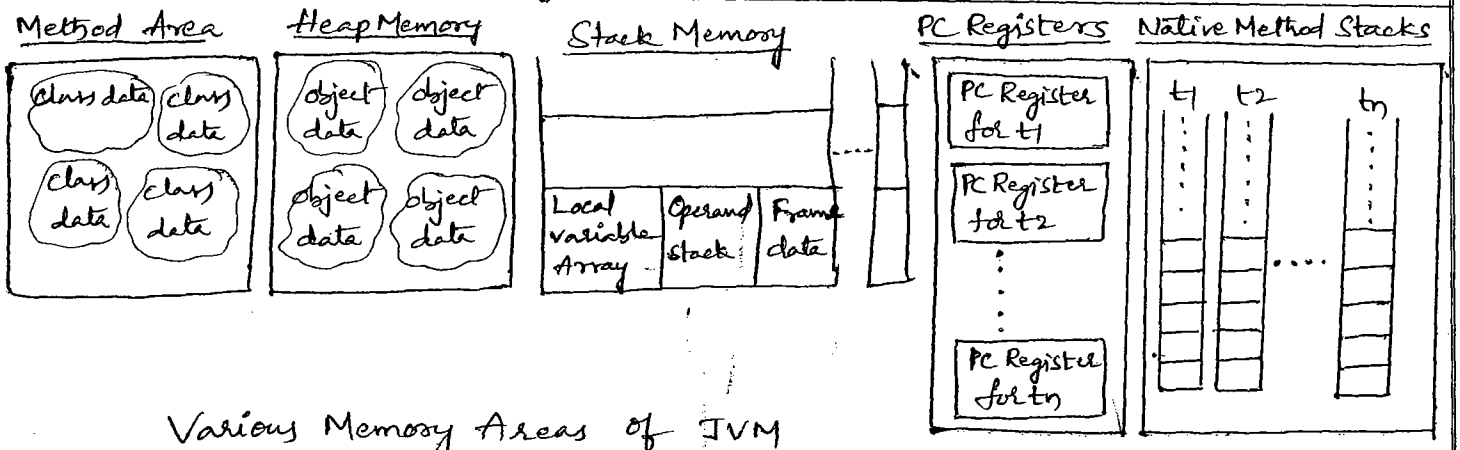


JNI (Java Native Interface):-

→ JNI acts as bridge (mediator) for Java method call and corresponding native libraries.

JVM Architecture





Various Memory Areas of JVM