Virtual Machine:

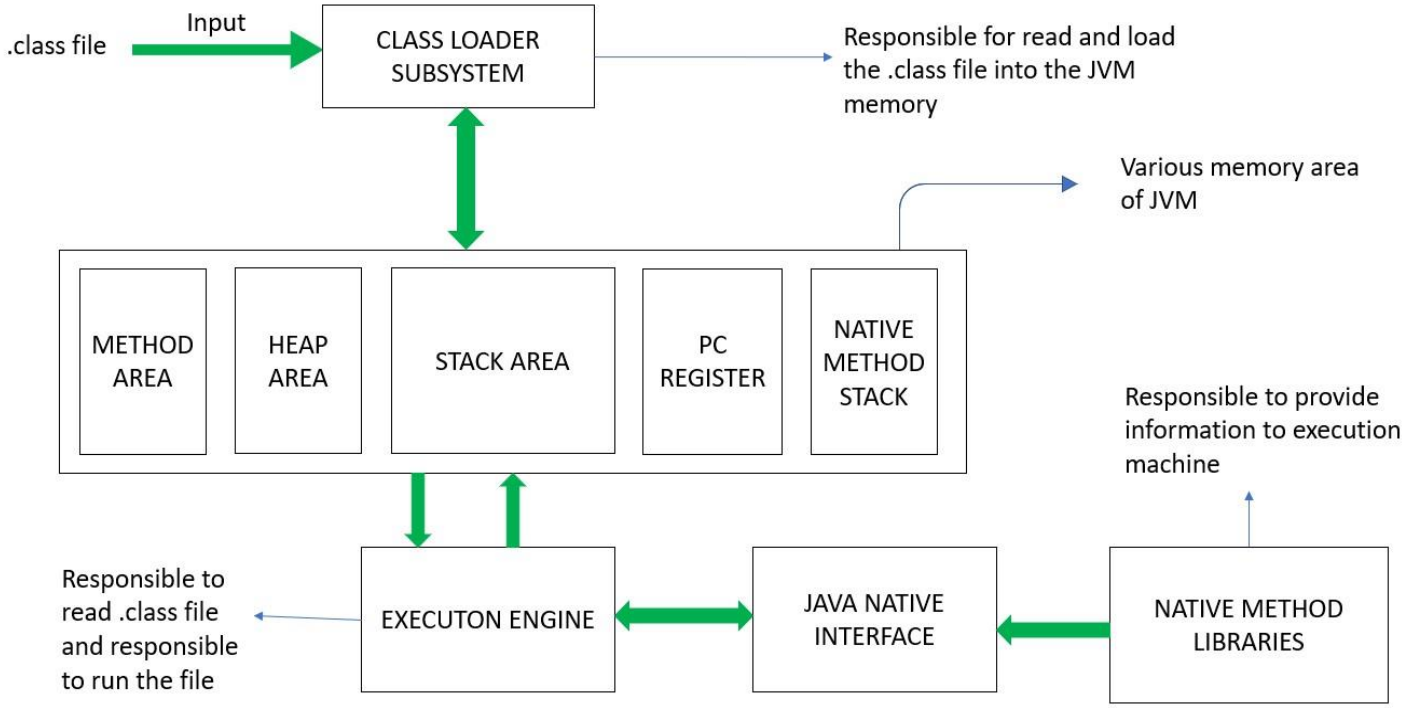
In a physical system when we create more than one logical system to utilize the physical hardware, memory, processor, using a software simulator and we make a logical software simulation of a machine that can operate a physical machine. There are two types of virtual machines are there the

first one is HARDWARE BASED/SYSTEM BASED VIRTUAL MACHINE which provides several logical systems on the same computer with strong isolation from each other. That is on one physical machine we are defining multiple machines. The advantage of this hardware-based virtual machine is hardware resource sharing and improves the utilization of hardware resources. In Linux operating system we use KVM [KERNEL BASED VIRTUAL MACHINE] which is a hardware-based virtual machine with other instances like VMware, cloud computing which is part of this type of virtual machine.

The second one is the APPLICATION BASED/PROCESS BASED VIRTUAL MACHINE which access the runtime engines to run a particular programming language application JAVA VIRTUAL MACHINE(JVM) which acts as runtime engine to run the java based application, similarly, PARROT VIRTUAL MACHINE(PVM) which acts as a runtime engine to run scripting language application and COMMON LANGUAGE RUNTIME(CLR) which acts as a runtime engine to run (.net) based programming language application.

Java Virtual Machine:

It is a virtual machine that is used to run a java programming-based application. The byte code when comes to JVM as an input interprets each statement in a single clock cycle and executes the further operation of the source code which is operated by a user. JVM is stored in a kernel of an operating system. JVM is part of JRE(JAVA RUNTIME ENGINE) and JRE is part of JDK(JAVA DEVELOPMENT KIT) and this is responsible for loading and running the java class file.



Phase Of JVM:

There are three phases in JVM

CLASS LOADER SUBSYSTEM

MEMORY AREA

EXECUTION ENGINE

Class Loader Subsystem:

This phase is responsible for reading the .class file from the hard disk and load in JVM memory. The class loader subsystem is responsible for the following 3 activities

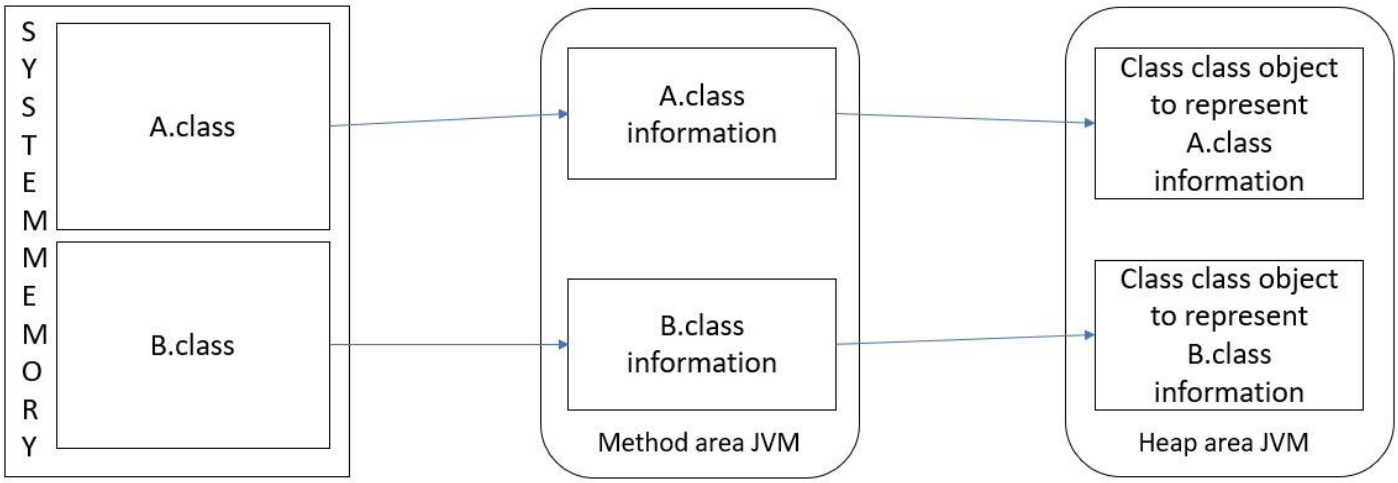
LOADING

LINKING

INITIALIZATION

Loading:

The process of loading is to read the class file and store the corresponding binary data in the method area of JVM. For each class file, JVM will store corresponding information in the method area. After loading the .class file immediately JVM creates an object for that loaded class on the heap memory area of JVM of type “java.lang.class”. JVM is responsible to convert the information to a class object for these class objects can be used by the programmer. A class object can be used by a programmer to get the corresponding information like class level information, method information, variable information, constructor information, etc.



*TEST00.java

```java
import java.lang.reflect.*;
class A
{
    public String getname()
    {
        return null ;
    }
    public int getreg()
    {
        return 12 ;
    }
    void display()
    {
        System.out.println("HELLO WORLD");
    }
}
public class TEST00
{
    public static void main(String args[]) throws Exception
    {
        int num = 0 ;
        Class a = Class.forName("A");       //java.lang.String  //java.lang.Object
        Method [] aa = a.getDeclaredMethods();

        for(Method a1 : aa)
        {
            num++ ;
            System.out.println(a1.getName());
        }
        System.out.println("NUMBER OF METHODS ARE : "+num);
    }
}
/*
 OUTPUT:

getreg
display
getname
NUMBER OF METHODS ARE : 3

 */
```

*TEST00.java

```java
import java.lang.reflect.*;
class A
{
    public String getname()
    {
        return null ;
    }
    public int getreg()
    {
        return 12 ;
    }
    void display()
    {
        System.out.println("HELLO WORLD");
    }
}
public class TEST00
{
    public static void main(String args[]) throws Exception
    {
        A a1 = new A();
        Class c1 = a1.getClass();
        System.out.println("object c1 : "+c1.hashCode());

        A a2 = new A();
        Class c2 = a2.getClass() ;
        System.out.println("object c2 : "+c2.hashCode());

        System.out.println(c1 == c2);
    }
}

/*
OUTPUT :
object c1 : 366712642
object c2 : 366712642
true
 * /

```

Even though we use a specific class multiple times but only one classclass object is created at the beginning.

LINKING:

This phase contains three activities

VERIFICATION/VERIFY: it is the process of ensuring that binary representation of a class is structurally correct or not that is JVM will check whether the (.class) file is generated by a valid compiler or not or the (.class) file is properly formatted or not. Internally byte code verifier is responsible for this activity which is a part of the class loader subsystem. If the verification fails then we will get a runtime exception java.lang.verifyerror.

PREPARATION/PREPARE: in this phase, JVM allocates memory for class-level static variables and assigns default values. In the initialization phase, original values will be assigned to the static variables and here only default values will be assigned.

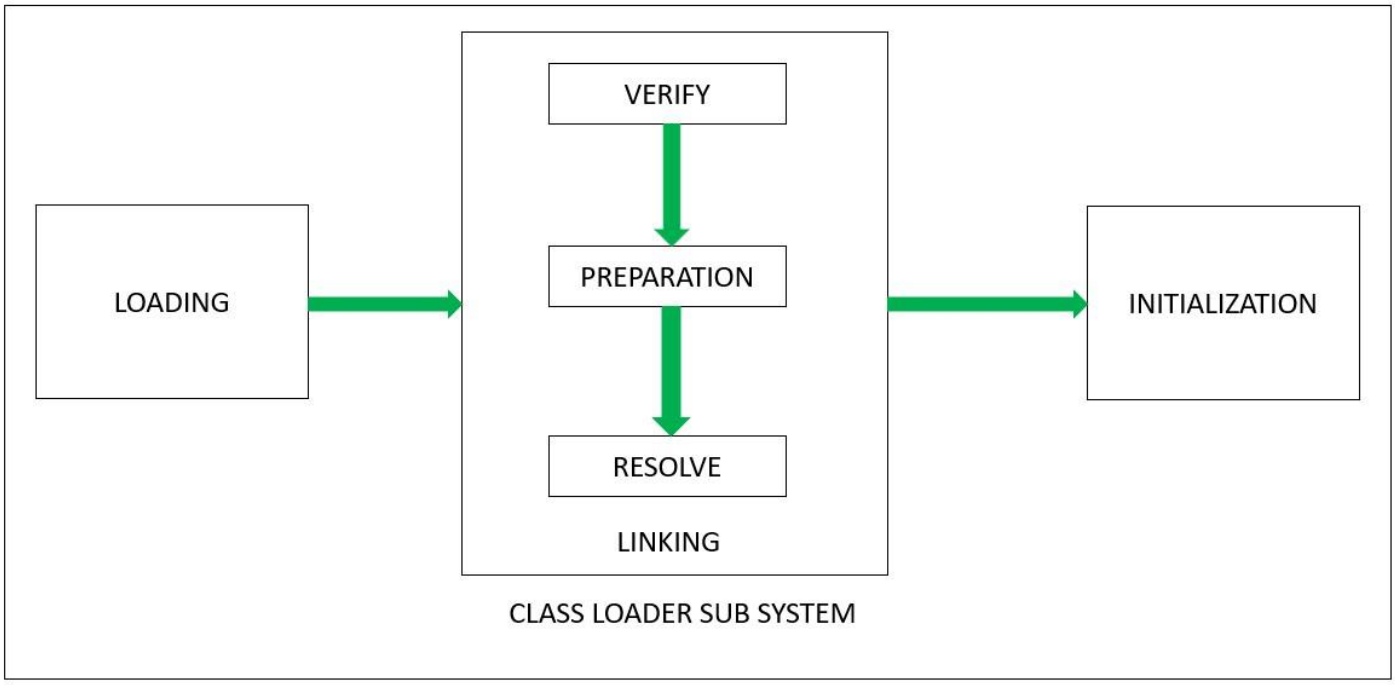
RESOLUTION/RESOLVE: it is the process of replacing symbolic names in your source code with original memory references from the method area. In the below figure classclass loaders load the test00 class, string class, A-class, and object class. The names of these classes are stored in a constant pool of test00 classes. In the resolution phase, these names are replaced with original memory level references from the method area.

*TEST00.java

```
1  import java.lang.reflect.*;
2  class A
3  {
4      public String getname()
5      {
6          return null ;
7      }
8      public int getreg()
9      {
10         return 12 ;
11     }
12     void display()
13     {
14         System.out.println("HELLO WORLD");
15     }
16 }
17 public class TEST00
18 {
19     public static void main(String args[]) throws Exception
20     {
21         String S = new String("ATISH");
22         A a = new A();
23     }
24 }
```

Initialization:

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



Types Of Class Loader System:

BOOTSTRAP CLASS LOADER/PRIMORDIAL CLASS LOADER:

There is a rt.jar file was there where all core java API classes present inside rt.jar, a bootstrap class loader is responsible to load core java API classes that are the classes present in rt.jar

C:\Program Files\Java\jdk1.8.0_251\jre\lib\rt.jar

The above location is called the bootstrap classpath that is the bootstrap class loader is responsible for classes from the bootstrap classpath. A bootstrap class loader is by default available in every JVM. It is implemented in native languages like c/c++ and not implemented in JAVA.

EXTENSION CLASS LOADER:

It is the child class of bootstrap class loader. The extension class loader Is responsible for loading classes from the extension classpath [sun.misc.Launcher$ExtClassLoader.Class]

C:\Program Files\Java\jdk1.8.0_251\jre\lib\ext

Extension class loader is implemented in java and the corresponding (.class) file is

APPLICATION/SYSTEM CLASS LOADER:

Application class loader is the child class of extension class loader. This is responsible for loading classes from the application classpath. It internally uses the environment variable classpath. Application class loader is implemented in java under the corresponding (.class) file name is [sun.misc.Launcher$AppClassLoader.Class]

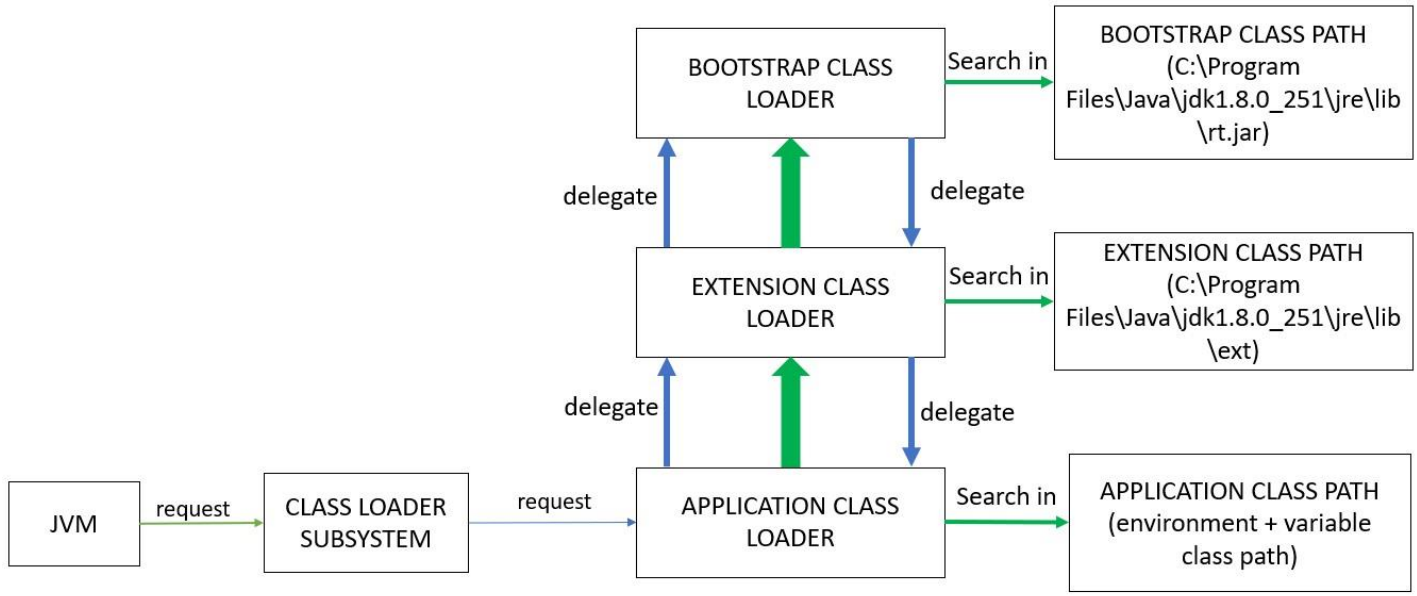
How Class Loader Works:

The class loader subsystem follows the DELEGATION HIERARCHY ALGORITHM

Whenever JVM comes across a particular class first it will check whether the specific (.class) file is already loaded or not. If it is already loaded in the method area then JVM will consider that loaded class. If it is not loaded then JVM requests a class loader subsystem to load that particular class. After sending the request class loader subsystem hand over the request to the application class loader.

Application class loader which the intern delegates the request to extension class loader which intern delegates the request to bootstrap class loader, then bootstrap class loader search in the bootstrap classpath if it is available then the corresponding class will be loaded by a bootstrap class loader. If it is not available then bootstrap class loader delegates request to extension class loader.

Then extension class loader searches in the extension classpath. If it is available then it will be loaded otherwise extension class loader delegates requests to the application class loader. It also searches in the application classpath, if it is available then it will be loaded otherwise we will get runtime exception “NoClassDefFoundError” or “ClassNotFoundException”.




```
public class clss
{

}
```

```
C:\Users\Atish kumar sahu>cd desktop

C:\Users\Atish kumar sahu\desktop>javac clss.java

C:\Users\Atish kumar sahu\desktop>jar -cvf cl.jar clss.class
added manifest
adding: clss.class(in = 182) (out= 153)(deflated 15%)
```

Then add the cl.jar to the location C:\Program Files\Java\jdk1.8.0_251\jre\lib\ext

```
public class clsstest
{
public static void main(String args[])
{
System.out.println(String.class.getClassLoader());
System.out.println(clsstest.class.getClassLoader());
System.out.println(clss.class.getClassLoader());
}}
```

```
C:\Users\Atish kumar sahu\desktop>javac clsstest.java

C:\Users\Atish kumar sahu\desktop>java clsstest
null
sun.misc.Launcher$AppClassLoader@2a139a55
sun.misc.Launcher$ExtClassLoader@70dea4e

C:\Users\Atish kumar sahu\desktop>
```

For ***String.class.getClassLoader***

Bootstrap class loader from the bootstrap classpath

Output: null

Because bootstrap class loader is not for java it is for c and c++. A bootstrap class loader is not a java object hence we got null in output.

For ***clsstest.class.getClassLoader***

Application class loader from application classpath

Output: sun.misc.Launcher$AppClassLoader@2a139a55

For ***clss.class.getClassLoader***

Extension class loader from extension classpath

Output: sum.misc.Launcher$ExtClassLoader@70dea4e

Class loader subsystem will give the highest priority for bootstrap class path and then extension class path followed by application class path.

Need Of Customize Class Loader:

Default class loader will load (.class) file only once even though we are using it 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 is already available in method area). We can resolve this problem by defining our customized class loader. The main advantage of a customized class loader is we can control the class loading mechanism based on our requirements. Ex: we can load the (.class) file separately every time so that the updated version is available to our program.

DEFAULT CLASS LOADER:

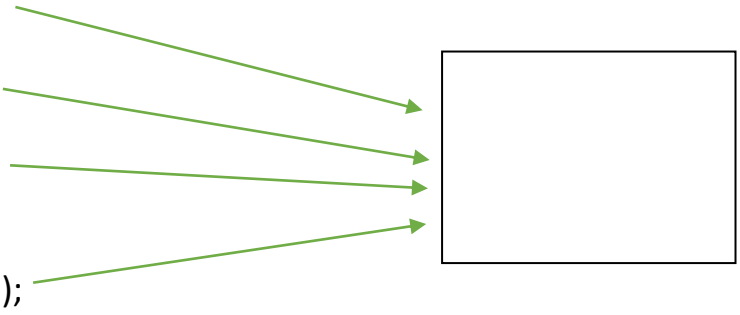
Student s1 = new student();

Student s2 = new student();

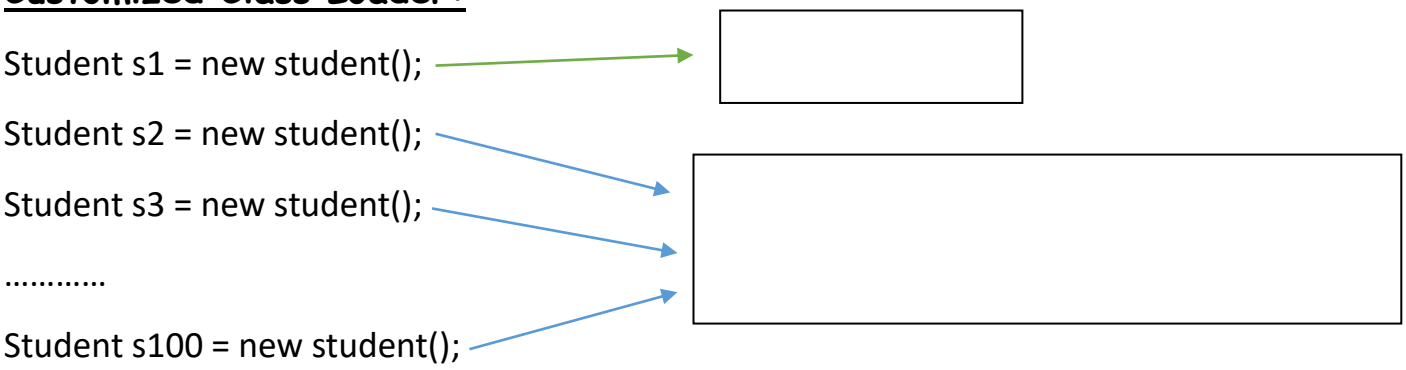
Student s3 = new student();

……..

Student s100 = new student();



Customized Class Loader:



The default class loader will load the .class file only once even though we are using multiple time that class in our program. After loading the .class file it is modified outside then the default class loader won't load the updated version of the class file because the .class file is already available in the method area. We can resolve this problem by defining our customized class loader. The main advantage of a customized class loader is we can control the class loading mechanism based on our requirements. For example, we can load the .class file separately every time an updated version is available to our program.

How To Define Customized Class Loader:

```
Public class CustClassLoader extends ClassLoader
{
Public class LoadClass(String classname) throws ClassNotFoundException
{
Check updates and load .class file and returns corresponding class
}       }
Class client
{
Public static void main(String args[])
{
Dog d1 = new Dog() ;       //default class loader
CustClassLoader cl = new CustClassLoader();
cl.loadClass("dog");
cl.loadclass("dog");
}       }
```

We define our customized class loader by extending java.lang.classloader class. While designing web servers and application servers usually we can go for a customized class loader to customize the class loading mechanism.

Q. What Is the need/use of ClassLoader class?

We can use java.lang.ClassLoader class to define our customized class loaders. Every class loader in java should be a child class of java.lang.ClassLoader class either directly or indirectly hence this class access base class for all customizer class loaders.

Various Memory Areas Of JVM:

Whenever JVM loads and runs the java program it needs memory to store several things like byte code, objects, variables, etc. total JVM memory organized into

#1 METHOD AREA

#2 HEAP AREA

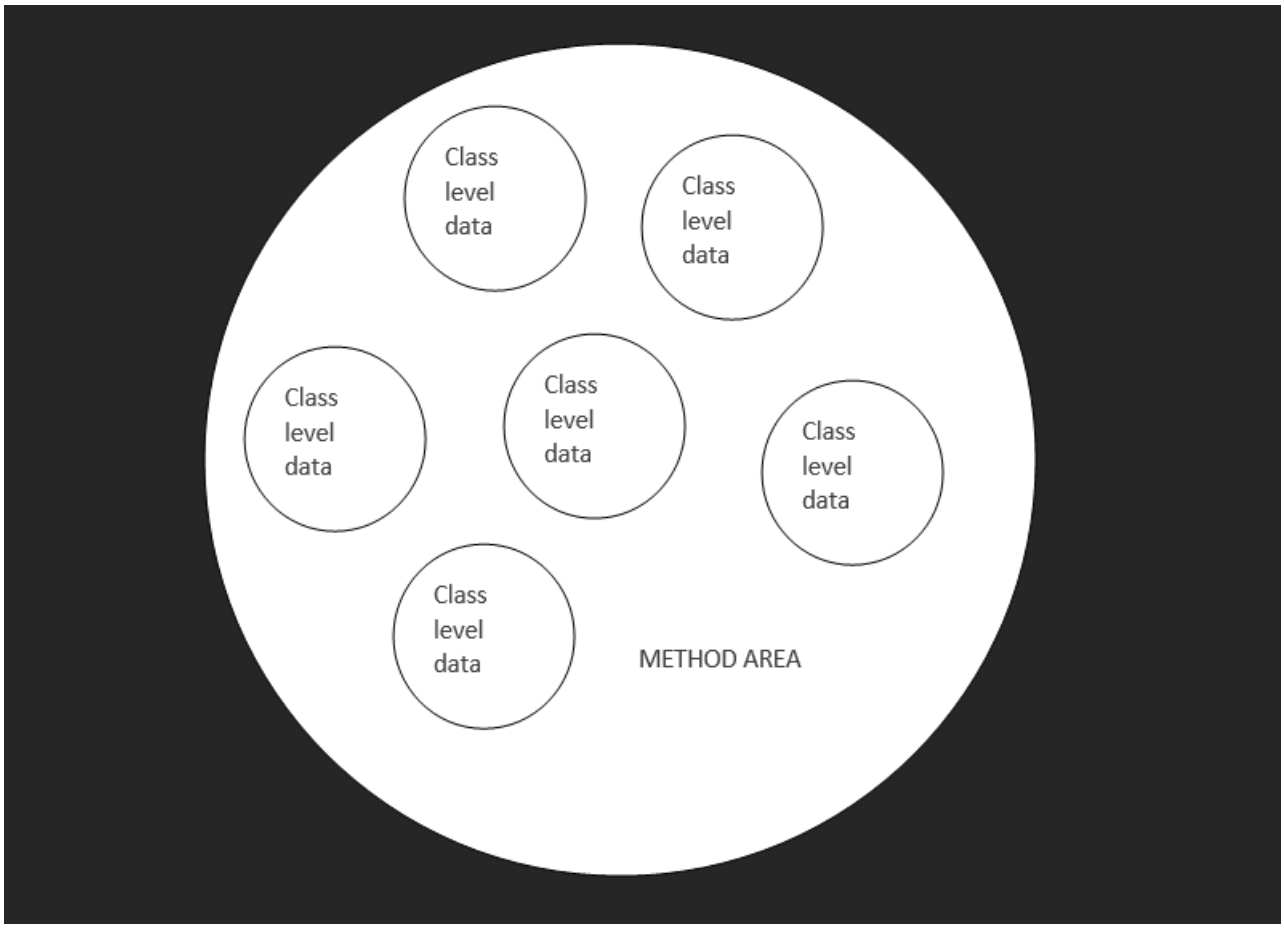
#3 STACK AREA

#4 PC REGISTER

#5 NATIVE METHOD STACKS

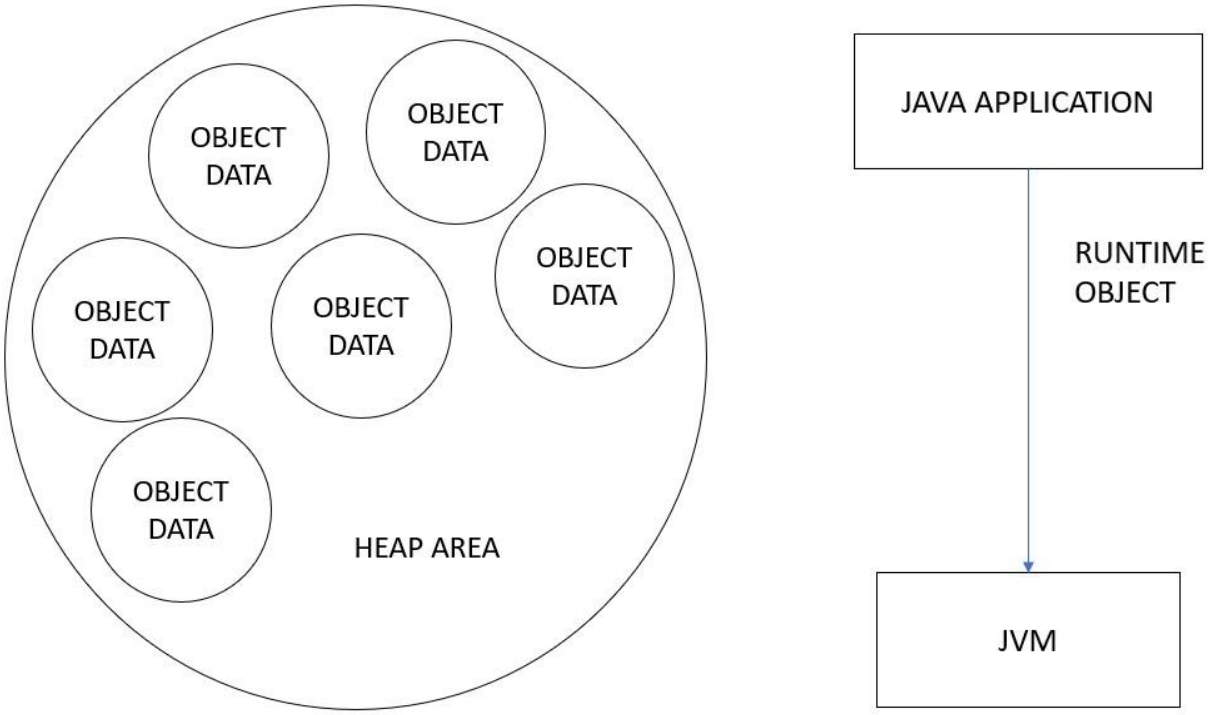
Method Area:

For every JVM one method area is 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. Method area can be accessed by multiple threads simultaneously.



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. Every array in java is object only hence arrays also will be stored in the heap area. The Heap area can be accessed by multiple threads. Heap areas are need not be continuous.



A java application can communicate with JVM by using a runtime object. Runtime class present in java.lang package under it is a single turn class we can create runtime object as follows once we got runtime object we can call the following methods. maxMemory()return the number of bytes of max memory allocated to the heap. totalMemeory()return the number of bytes of the total memory allocated to the heap(initial memory). freeMemory()return the number of bytes of free memory present in the heap.

```
public class HEAP
{
public static void main(String args[])
{
double MB = 1024 * 1024 ;

Runtime RT = Runtime.getRuntime();
System.out.println("MAX MEMORY : "+RT.maxMemory()+ " byte");
System.out.println("TOTAL MEMORY : "+RT.totalMemory()+ " byte");
System.out.println("FREE MEMORY : "+RT.freeMemory()+ " byte");
System.out.println("CONSUMED MEMORY : "+(RT.totalMemory() - RT.freeMemory())+ " byte");

System.out.println("---------------------------------");

System.out.println("MAX MEMORY : "+RT.maxMemory()/MB);
System.out.println("TOTAL MEMORY : "+RT.totalMemory()/MB);
System.out.println("FREE MEMORY : "+RT.freeMemory()/MB);
System.out.println("CONSUMED MEMORY : "+(RT.totalMemory() - RT.freeMemory()) / MB);
}
}
```

```
C:\Users\Atish kumar sahu\desktop>javac HEAP.java

C:\Users\Atish kumar sahu\desktop>java HEAP
MAX MEMORY : 1881145344 byte
TOTAL MEMORY : 128974848 byte
FREE MEMORY : 127611672 byte
CONSUMED MEMORY : 1363176 byte
-----------------------------------
MAX MEMORY : 1794.0
TOTAL MEMORY : 123.0
FREE MEMORY : 121.6999740600586
CONSUMED MEMORY : 1.3000259399414062

C:\Users\Atish kumar sahu\desktop>
```

Q. How To Set Maximum & Minimum Heap Sizes?

Heap memory is a finite memory but based on the environment we can set maximum and minimum heap sizes that we can increase or decrease the heap size based on our requirement. We can use the following flags

-Xmx: to set max heap size which impact to maxMemory()

-Xms: to set min heap size which impacts to totalMemory()

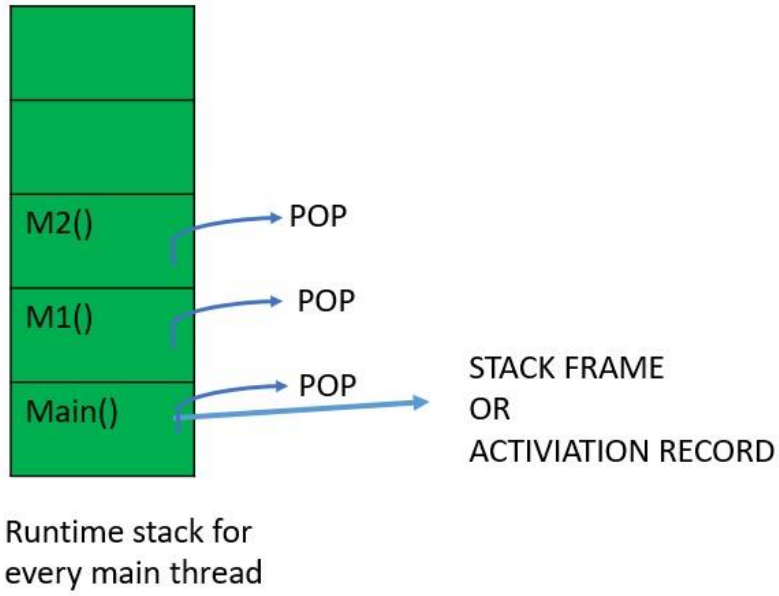
```
C:\Users\Atish kumar sahu\desktop>java -Xmx420m -Xms360m HEAP
MAX MEMORY : 391643136 byte
TOTAL MEMORY : 361758720 byte
FREE MEMORY : 357983816 byte
CONSUMED MEMORY : 3774904 byte
---------------------------------
MAX MEMORY : 373.5
TOTAL MEMORY : 345.0
FREE MEMORY : 341.3999710083008
CONSUMED MEMORY : 3.6000289916992188

C:\Users\Atish kumar sahu\desktop>java HEAP
MAX MEMORY : 1881145344 byte
TOTAL MEMORY : 128974848 byte
FREE MEMORY : 127611672 byte
CONSUMED MEMORY : 1363176 byte
-----------------------------------
MAX MEMORY : 1794.0
TOTAL MEMORY : 123.0
FREE MEMORY : 121.6999740600586
CONSUMED MEMORY : 1.3000259399414062

C:\Users\Atish kumar sahu\desktop>
```

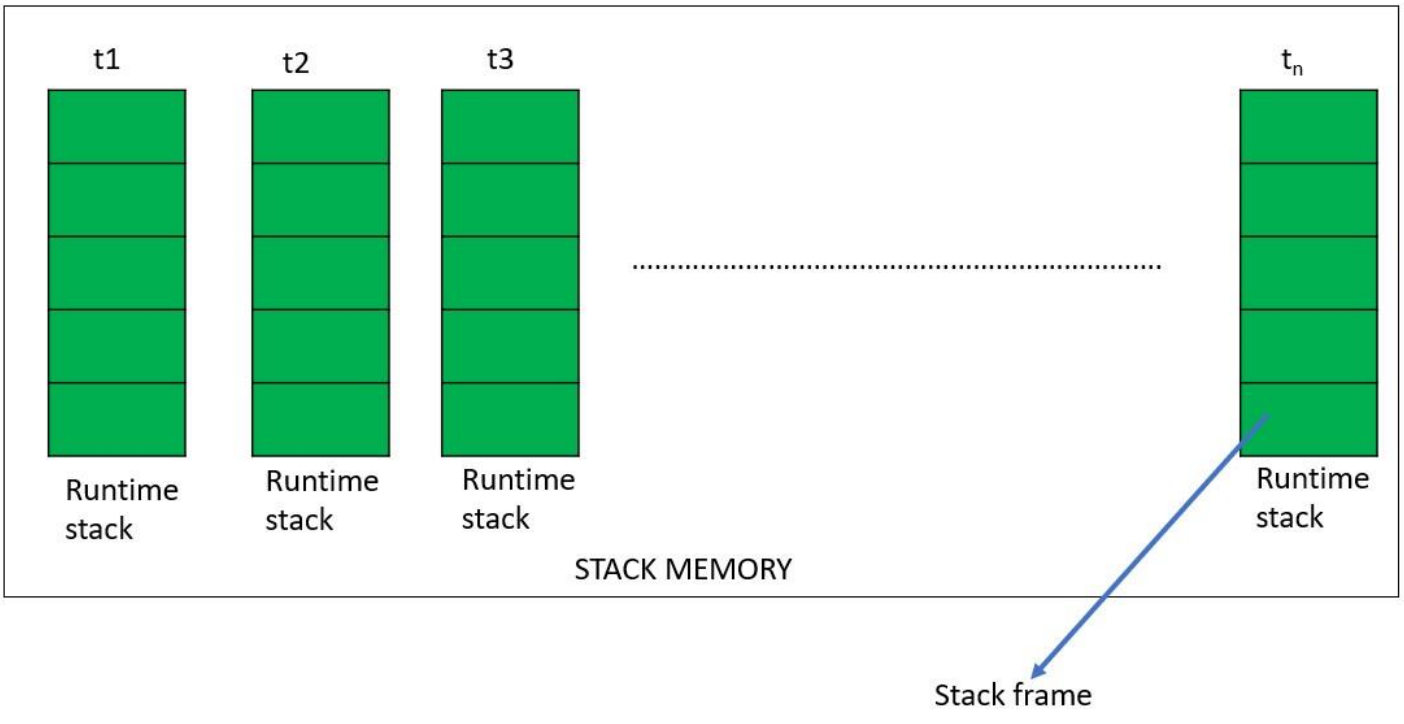
Stack Memory Area:

For every thread JVM will create a separate stack at the time of thread creation each and every method called perform by that thread will be stored in the stack including local variables also. After completely a method the corresponding entry from the stack will be removed after completing all method calls the stack will become empty and the empty stack will be destroyed by JVM just before terminate in the thread. Each entry in the stack is called stack frame or the activation record.

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

```
Class TEST
{
Public static void main(String args[])
{
M1() ;
}
Public static void M1()
{
M2();
}
Public static void M2()
{
}
```





Stack Frame Structure:

Each stack frame contains three parts

LOCAL VARIABLE ARRAY

OPERAND STACK

FRAME DATA

STACK FRAME STRUCTURE

Local Variable Array:

It contains all parameters and local variables of the method each slot in the array is of 4 bytes values of types int, float, reference occupied 1 entry in the array. Values of double, and long occupied 2 consecutive entry in the array. Byte, short and char value will be converted to int type before storing and occupy 1 slot but the storing boolean value is varied from JVM to JVM but most JVM follow 1 slot for boolean values.

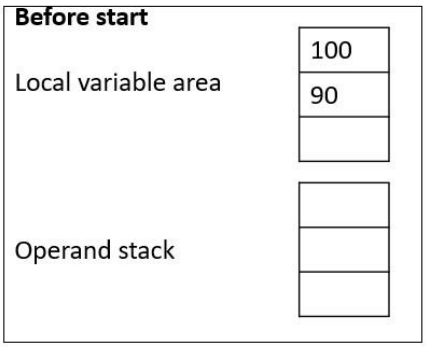
```
Public void m1(int I, double d, object o, float f)
{
Long x ;
}
```

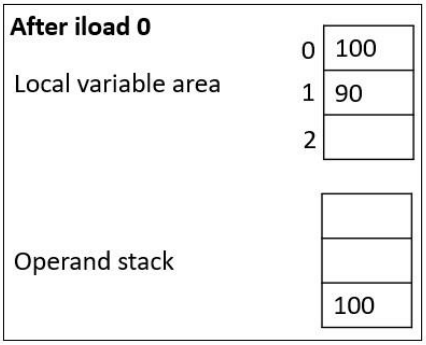
int i	double d	double d	object o	float f	long x	long x
0	1	2	3	4	5	6

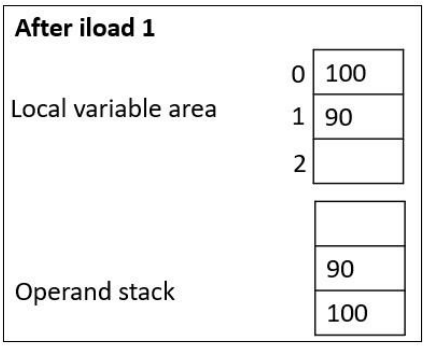
OPERAND STACK:

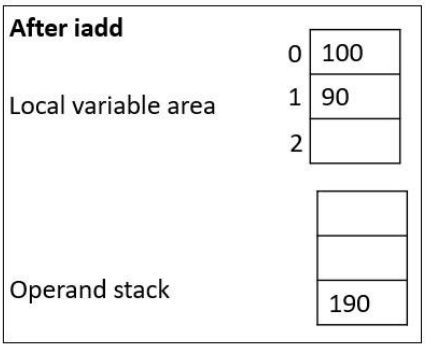
JVM uses operand stack as workspace some instruction push the values to operand stack and some instruction can pop values from operand stack and some instruction can perform required operation.

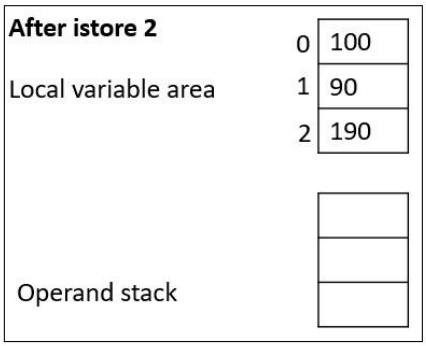
iload 0
iadd
iload 1
istore 2











Frame Data:

Frame data contain all symbolic references related to that method it also contains a references to exception table which provide corresponding catch block information in the case of exception.

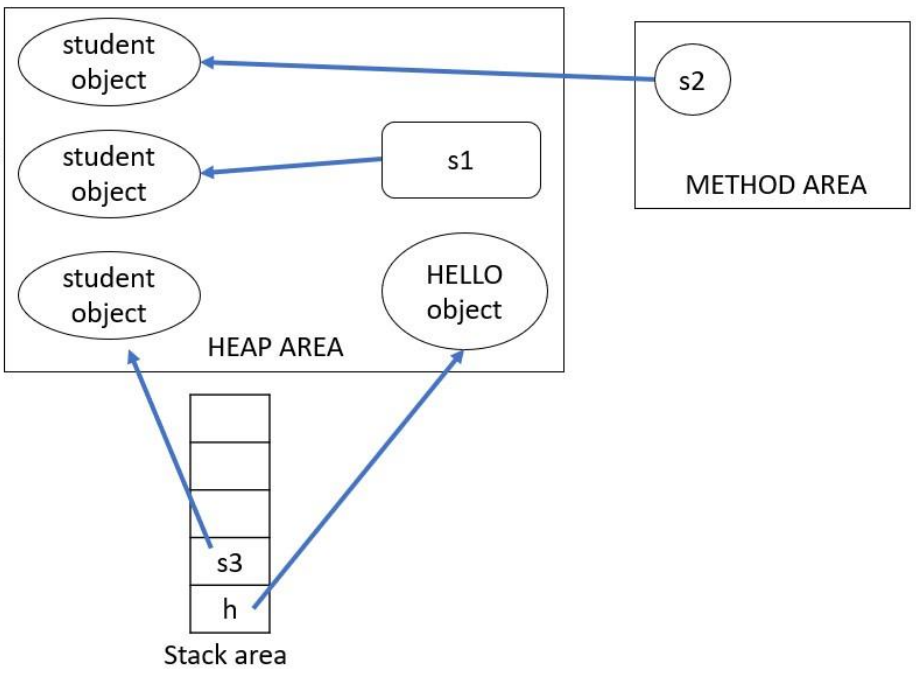
PC register (Program Counter):

Program counter registers for every thread a separate pc register will created at the time of thread creation pc register contains the address of current executing instruction once instruction execution complete automatically pc register will be incremented to hold address of next instruction.

Native Method Stack:

Example: hashcode(), wait(). 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. Method area, heap area, stack area considered as important memory area with respect to programmer. Method area, heap area per JVM. Stack area, pc register, native method stack area are per thread. For every JVM one heap area, one method area. For every thread one stack area, one pc register, one native method stack. Instance variable are stored in heap area, static variable will be stored in method area, local variable will be stored in stack area.

```
Class HELLO
{
student s1 = new student();
static student s2 = new student();
Public static void main(String args[]);
{
HELLO h = new HELLO();
student s3 = new student();
}
}
```



Execution Engine:

This is the central component of JVM. Execution engine is responsible to execute java (.class) files execution machine contain two component.

INTERPRETER # JIT COMPILER (JUST IN TIME)

Interpreter:

Interpreter is responsible to read byte code and interpreter into machine code (native code) and execute the machine code line by line. The problem with interpreter is it interprets every time even same method invoked multiple times which reduces performance of system to overcome this problem such people introduce JIT compiler in 1.1 version.

JIT Compiler(Just In Time):

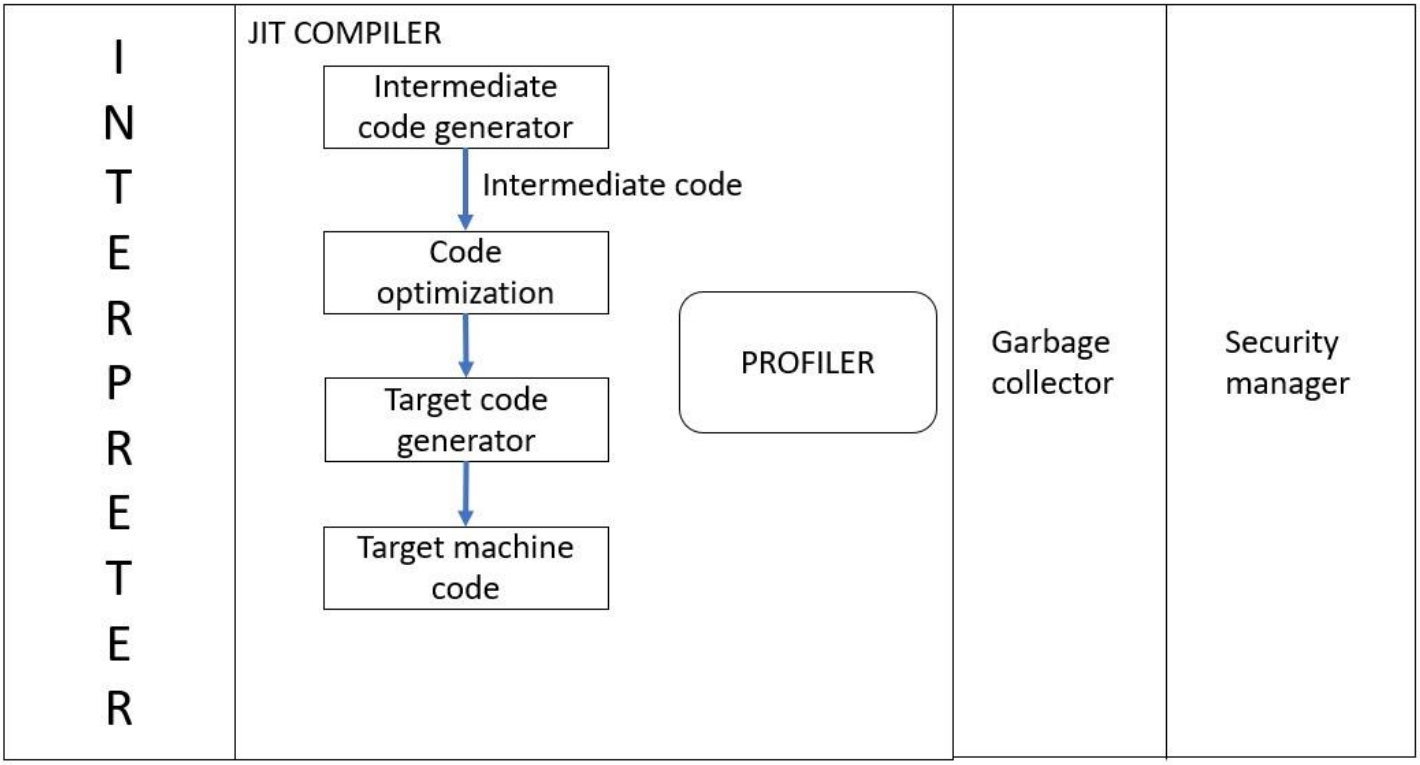
JIT compiler is a compiler which main purpose id to improve performance internally JIT compiler maintain a separate count for every method whenever JVM come across any method call first that method interpreted normally by interpreter and JIT compiler increment the corresponding count variable.

This process will be continued for every method once if any method count reaches threshold value then JIT compilers identify that method is a repeatedly used method such methods called as hotspot.

Immediately JIT compiler compiles that methods and generates correspondence native code next time JVM come across the method called then JVM uses native code directly and executes it instead of interpreting once again so that performance of system will be improved the threshold count is varied from JVM to JVM.

Some advanced JIT compiler will recompile generated native code if count reaches threshold value second time so that more optimized machine code will be generated. Internally profiler which is part of JIT compiler is responsible to identify hotspots.

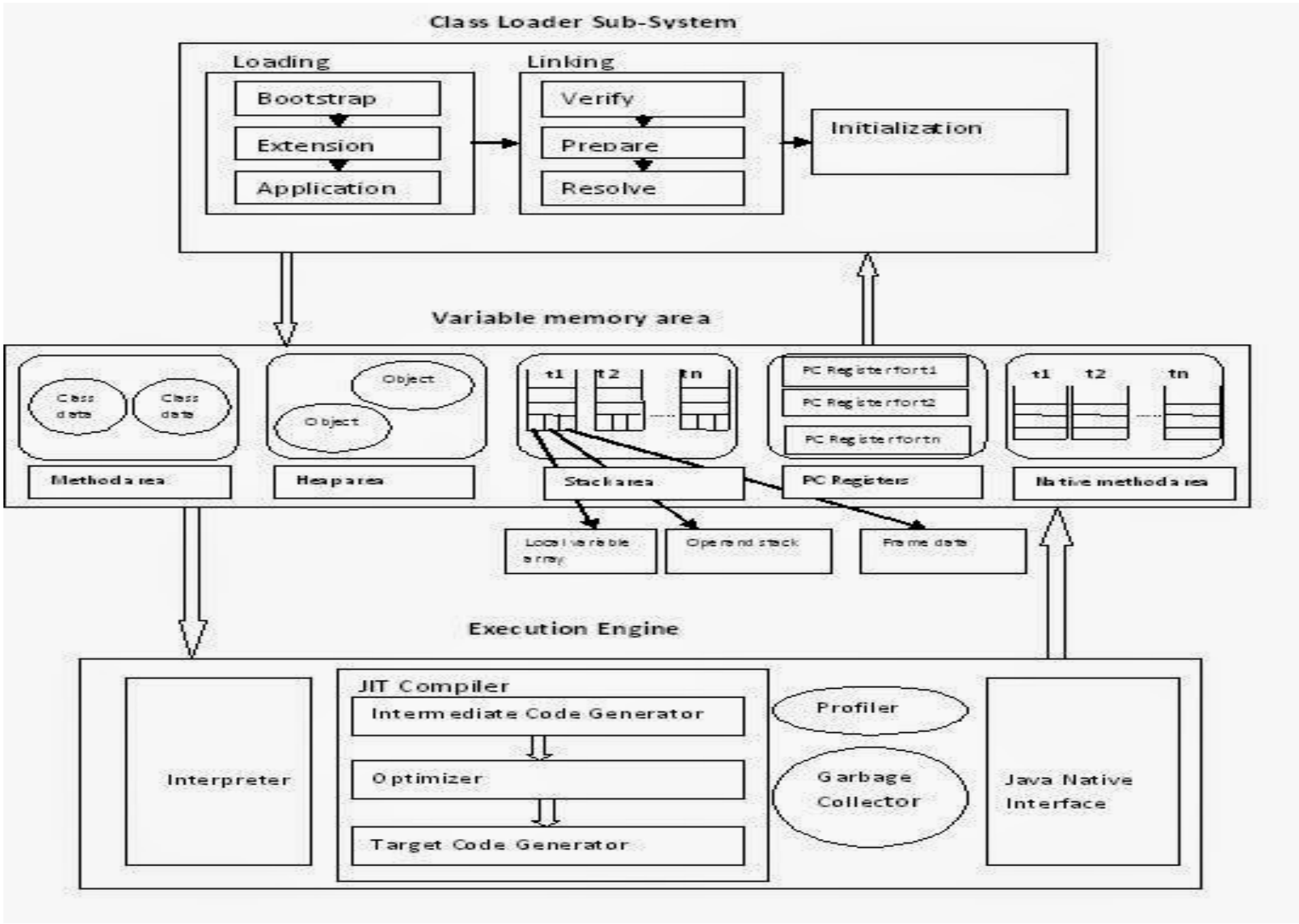
JVM interprets total program at least once. JIT compilation is applicable only for repeatedly required methods not for every method.



Java Native Interface:

It access mediator for java method calls and corresponding native libraries that is JNI is responsible to provide information about native libraries to JVM native method libraries provided and holds native libraries information.

JAVA VIRTUAL MACHINE:



Class File Structure:

Class file

{

magic_numbers ;

minor_version ;

major_version ;

constant_pool_count ;

constant_pool [] ;

access_flags ;

this_class ;

super_class ;

interface_count ;

interface [] ;

fields_count ;

fields [] ;

methods_count ;

methods [] ;

attributes_count ;

attributes [] ;

}

magic_numbers:

The first 4 bytes of class file is magic number. This is a predefined value used by JVM to identify (.class) file generated by valid compilers or not the value should be 0XCAFEBABE. Whenever we execute a java class if JVM unable to find the valid magic number then we will get run time exception saying

java.lang.ClassFormatError.Incompatiblemagicvalue

minor_version and major_version:

These represents (.class) file version. JVM will use this version to identify which version of compiler generates the current (.class) file can be run by higher version JVM but the higher version (.class) file cannot be run by the lower version of JVM. If we trying to run we will get the runtime exception "java.lang.UnsupportedClassVersionError"

Javac 1.6 < = > java 1.7 => correct

Java 1.7 < = > java 1.6 => error

constant_pool_count:

it represents number of constant in constant pool.

constant_pool [] :

it represents information about constant present in constant pool.

access_flags:

it provides information about modifier which are declared to the class.

this_class:

it represents fully qualified name of class.

super_class:

it represent fully qualified name of immediate super class of current class.

interface_count:

it return number of interface implemented by current class.

interface []:

it returns interface information implemented by current class.

fields_count:

it represents the number of fields in current class. Field = static variables

fields []:

it represent fields information in current class.

methods_count:

it represent number of methods in current class.

methods []:

it provide information about all method present in current class.

attributes_count:

it return number of attributes present in current class.

attribute []:

it provide information about all attribute present in current class.

use the command in cmd -> javap -verbose classname.class