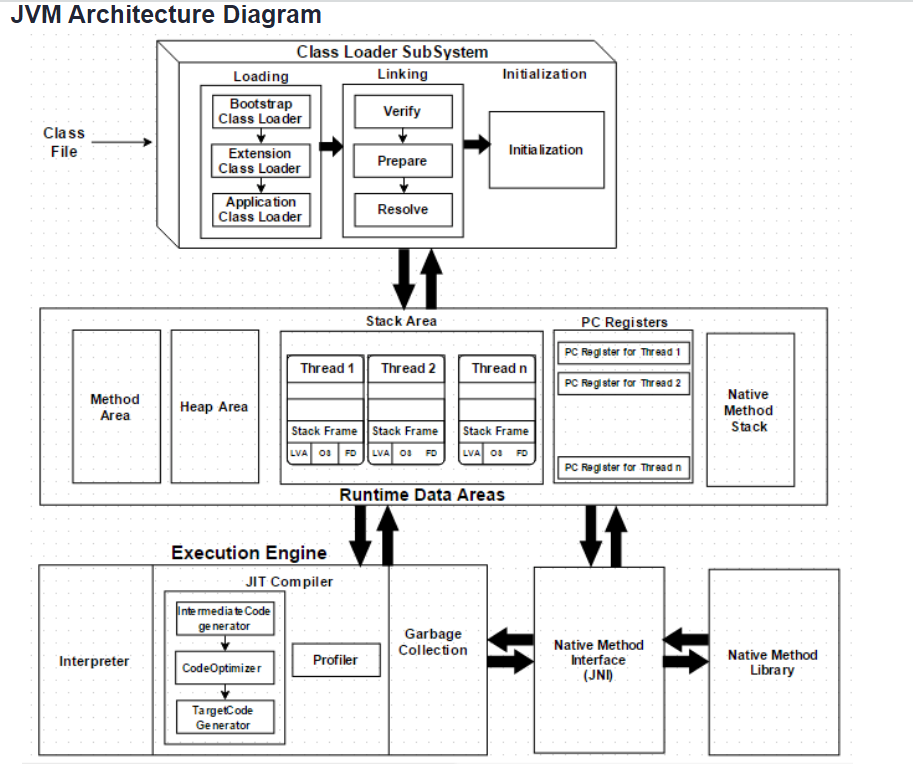
* **JVM architectures :**



**Virtual Machine:**

* It is software simulation of machine which can perform operation like physical machine.
* There are two type of virtual machine.

1. Hardware based virtual machine
2. Software base virtual machine

* It provides several logical systems on the same computer with strong isolation from each other. i.s on one physical machine we are defining multiple logical machines.
* The main advantage of hardware based virtual machine is hardware resource sharing and improves utilization of hardware resources.

e.g 1) KVM (kernel based virtual machine for linux system)

2) VmWare, Zen, cloud computing etc.

**Application/software based virtual machine :**

* These vitual machine act as runtime engines to run a particular programing language applications.

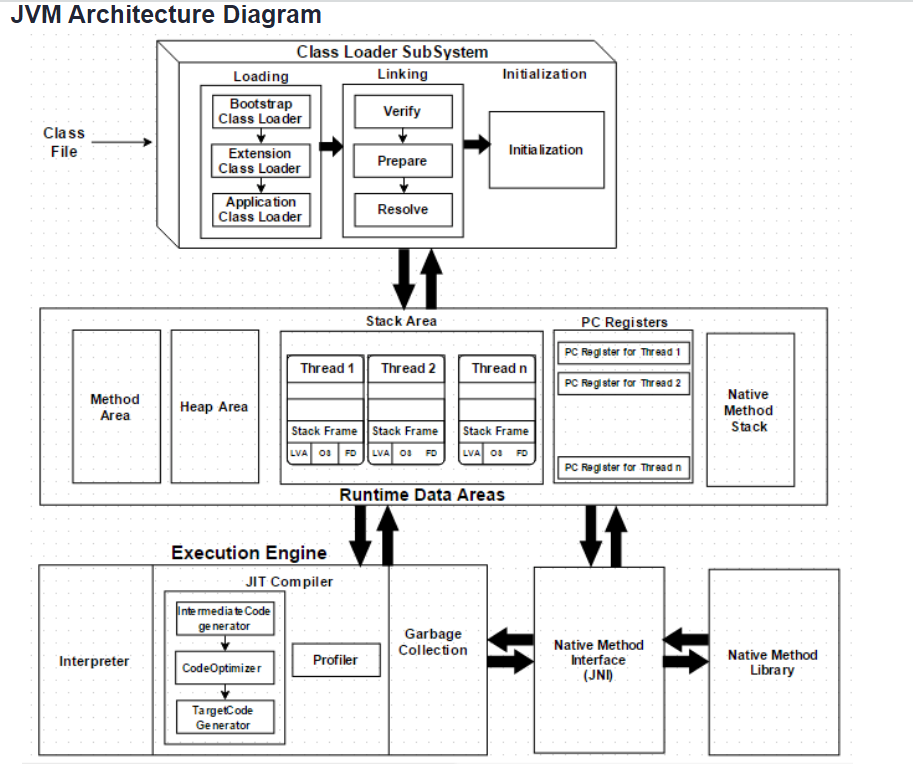
e.g 1) JVM (Java virtual machine) acts as runtime engine to run java based applications

2) PVM (Paret Virtual machine) acts as runtime engine to run Perl based applications

3) CLR (common language Runtime) acts as runtime engine to run .Net based applications

* JVM is the part of JRE and it is responsible to load and run .class file

**Basic architecture of JVM**



**Class Loader subsystem :**

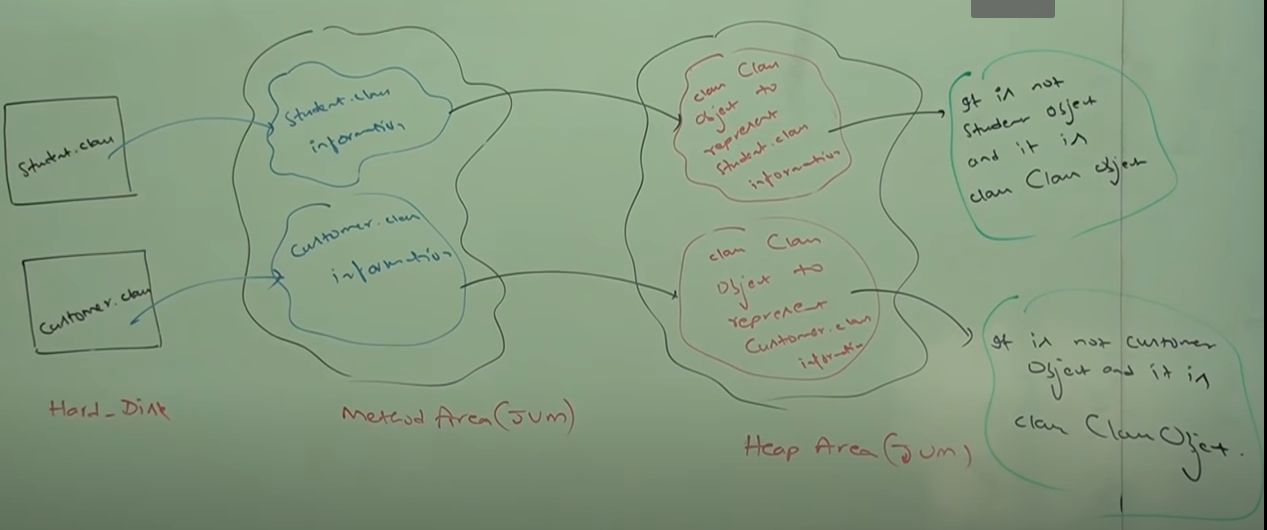
* Class loader subsystem is responsible for the following three activity

1. Loading
2. Linking
3. Initialization

* **Loading :**
* Loading meance reading class files and store corresponding binary data in method area.
* For each class file JVM will store corresponding information in method area.

1. Fully qualified name of class
2. Fully qualified name of the parent class
3. Methods information
4. Variables information
5. Constructor information
6. Modifiers information
7. Constant pool information and ext

* After loading .class file immediately JVM creates an object for that loaded class in the heap memory of type java.lang.class.



* The class class object can be used by programmer to get class level information like methods information or variables information, constructor information etc.
* For every loaded type only one class object will be created even though we are using that multiple time in our program.
* **Linking :**
* Linking consist of three activity.

1. Verification
2. Prepare
3. Resolve
4. **Verify** :

* It is the process of ensuring the binary repartition of a class is structurally correct or not i.e whether the class file is generated by valid compiler or i.e whether class file is properly formatted or not.
* Internally bytecode verifier is responsible for this activity.
* Bytecode verifier is the par of class loader subsystem.
* If verification fails then we will get runtime exception saying java.lang.verify error.

1. **Prepare :**

* In this phase JVM will allocate memory for class level static variables and assign default values.

**Note :** in initialization phase original value will be assigned to the static variable and here only default values will be assigned.

1. **Resolve :**

* It is the process of replacing symbolic names in our program with original memory references from method areas.

e.g class test{

public static void main(String[] args){

String s = new String(“Durga”);

Student s1 = new Strund();

}

}

* For the above class class loader loads test.class, string.class, student.class and object.class
* The names of these class are stored in constant pool of test class.
* In resolution phase these name are replaced with original memory level references from method area.
* **Initialization :**
* In this phase all static variables are assigned with original values and static blocks will be executed from parent to child and from top to bottom.

**Note :** while loading, linking and initialization if any error occurs then we will get runtime exception saying java.lang.linkage error.

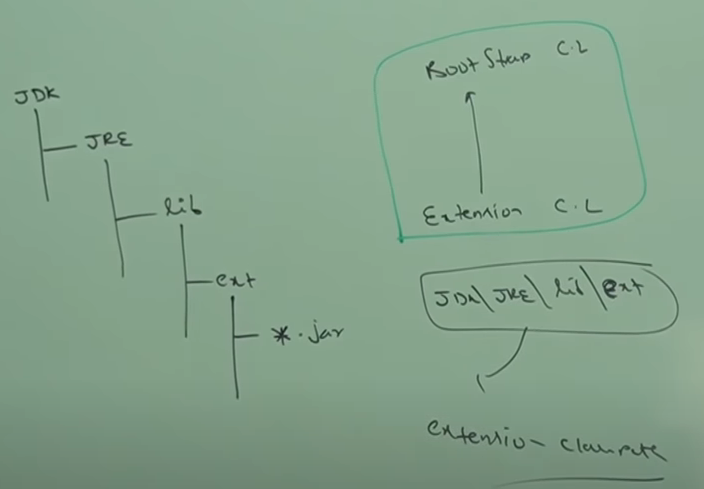
* **Types of class loaders:**
* Class loader subsystem contains the following three types of class loaders.

1. Bootstrap class loaders/ premordial class loader
2. Extension class loader
3. Application class loader/ system class loader
4. **Bootstrap class loader :**

* Bootstrap class loader is responsible to load core java api classes i.e the classes present in rt.jar. (jdk -> jre -> lib -> rt.jar)
* This location is called bootstrap class path i.e bootstrap class loader is responsible to load classes from bootstrap class path
* Bootstrap class loader is by default available with every JVM.
* It is implemented in native languages like C/C++ and implemented in java.

1. **Extensions class loader :**

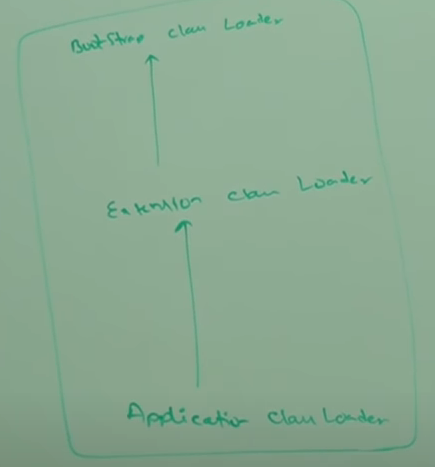
* Extension class loader is the child class of bootstrap class loader.



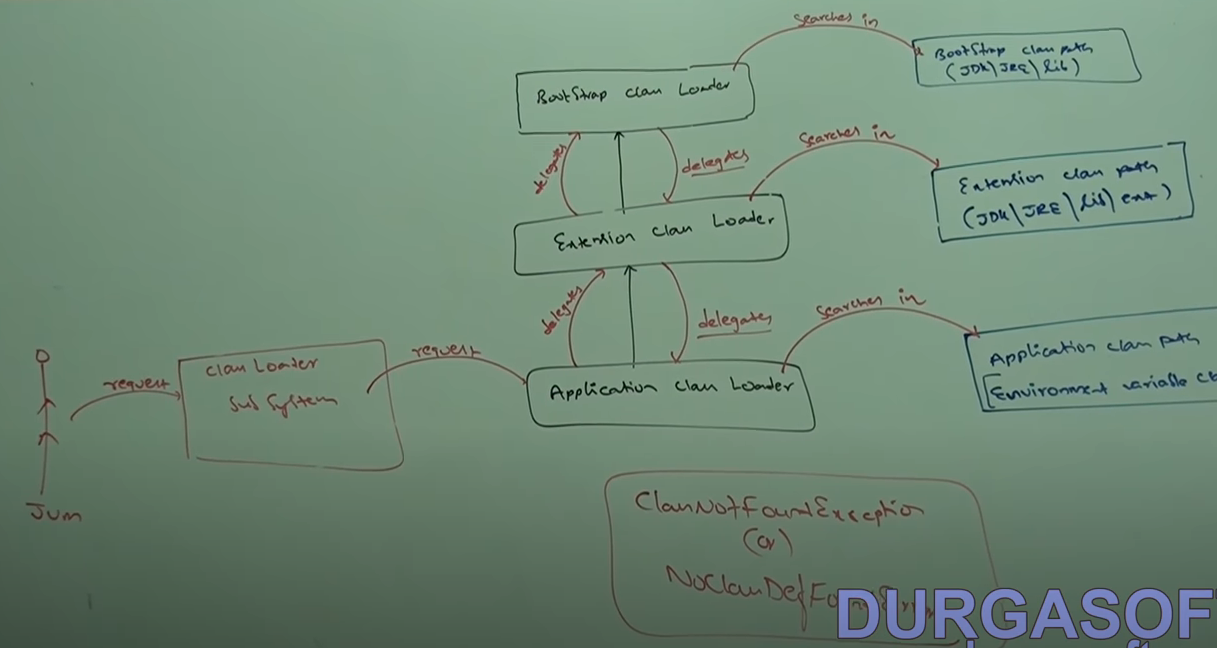
* extension class loader is responsible to load classes from extension class path (jdk -> jre -> lib -> ext)
* extension class loader is implemented in java and the corresponding .class file is sun.misc.Luancher$ExtClassLoader.class

1. **Application class loader/ system class loader :**

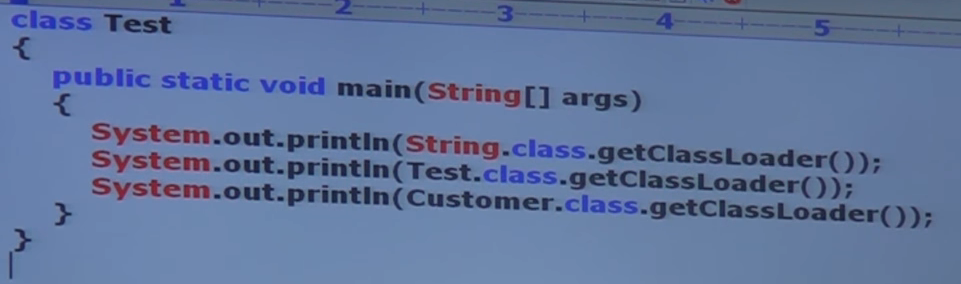
* Application class loader is the child class of extension class loader.
* This class loader is responsible to load classes from application class path.
* It internally uses environment variable class path.
* Application class loader is implemented in java and the corresponding .class file name is [sun.misc.launcher@AppClassLoader.class](mailto:sun.misc.launcher@AppClassLoader.class)

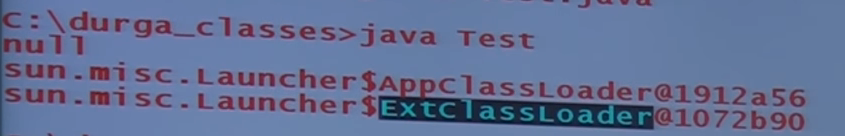


* **How class loader works :**



* Class loader follows delegation hierarchy principle/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 loaded in method area then JVM will consider that loaded class.
* If it is not loaded then JVM request class loader subsystem to load that particular class.
* Then class loader subsystem handover the request to application class loader.
* Application class loader delegates request to extension class loader which inter delegates the request to bootstrap class loader.
* Then bootstrap class will search in bootstrap class path if it is available then the corresponding .class will be loaded by bootstrap class loader. If it is not available then bootstrap class loader delegates the request to extension class loader again.
* 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.
* Application class loader will search in application class path.
* If it is available then it will be loaded otherwise we will get runtime exception saying NoClassDefFoundError or ClassNotFoundException.



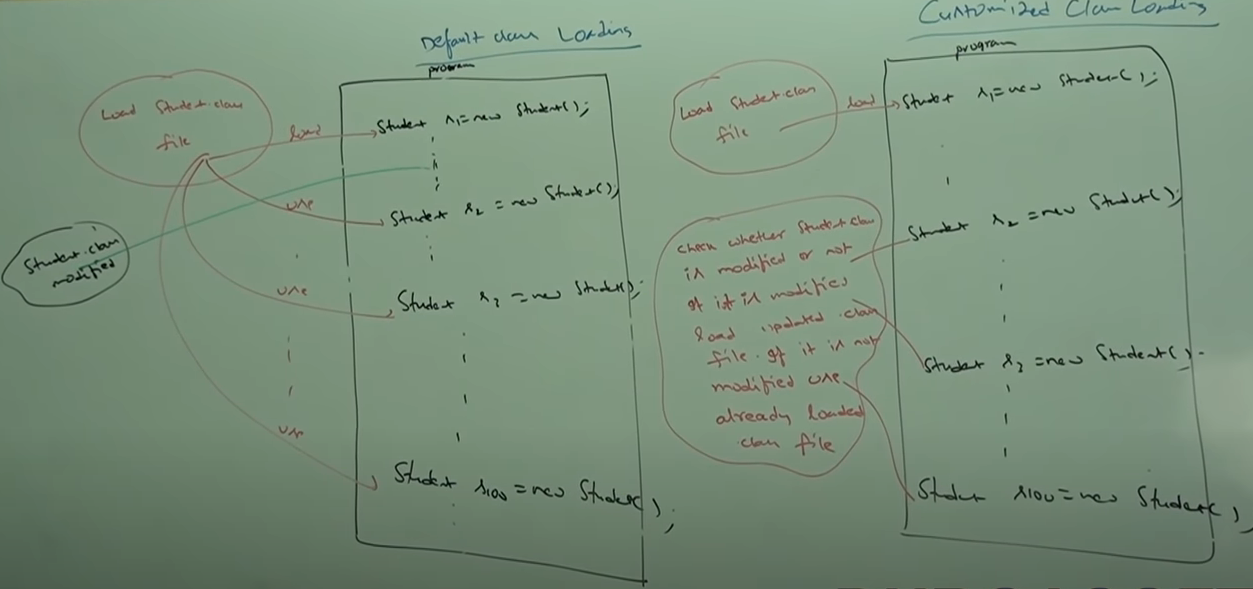


**Note :**

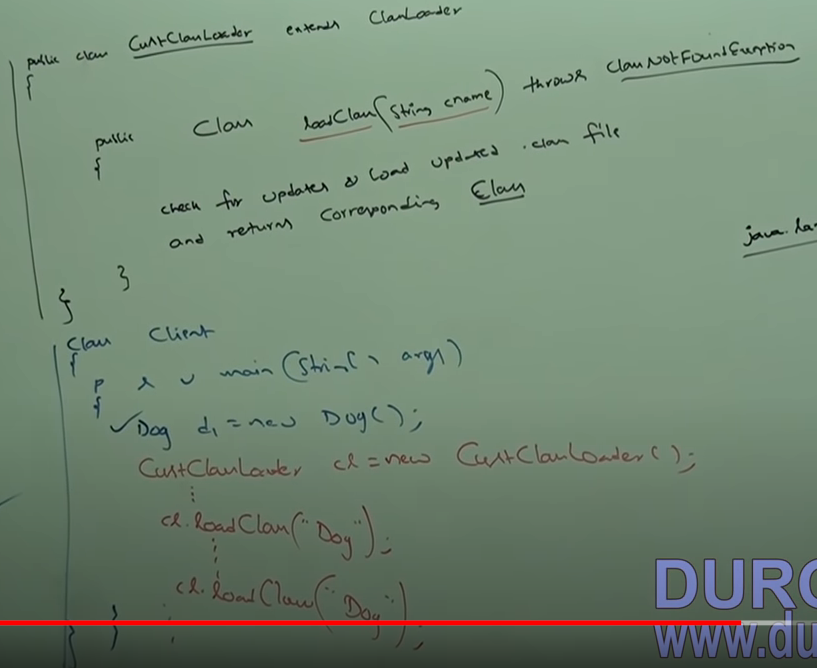
1. Bootstrap class loader is not java object hence we got null in first case but extension and application class loader are java objects hence we are getting corresponding output for the remaining two SPOs.
2. Class loader subsystem will give the highest priority for bootstrap class path and then extension class path followed by application class path.

* **Need of customized class loader :**
* Default class loader will load .class file only once even though we are using multiple time that class in our program.
* After loading .class file if it is modified out side then default class loader wont load updated version of class file (because .class file already available in method area).
* We can resolve this problem by defining our own customized class loader.
* The main advantage of customized class loader is we can control class loading mechanism based on our requirement.

**e.g** we can load .class file separately every time so that updated version available to our program.



* **How to define customized class loader:**
* We can define our own customized class loader by extending java.lang.classloader class



* While developing webserver or application server usually we go for customized class loader to customize class loading mechanism.
* Interview question :

1. What is need/use of class loader class.

* We can use java.lang.classloader class to define our own customized class loaders.
* Every class loader in java should be child class of java.lang.classLoader class either directly or indirectly hence this class acts as base class for all customized class loaders
* **Various memory area in JVM**
* Whenever JVM loads and runs a java program it need memory to store several things like byte code, objects, variable etc
* Total JVM memory organized into following five categories.

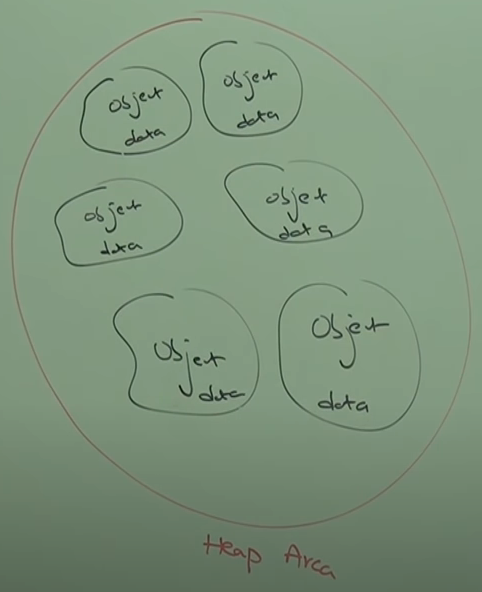
1. Method area
2. Heap area
3. Stack area
4. PC register area
5. Native method stack
6. **Method area :**

* 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 pool of a class will be stored in method area.
* Method area can be accessed by multiple threads simultaneously.



1. **Heap area :**

* For every JVM one heap area is available
* Heap area will be created at the time of JVM startup.
* Object and the corresponding instance variable will be stored in the heap area.
* Every array in java is object only hence arrays also will be stored in the heap area.
* Heap area can be accessed by multiple threads and hence data stored in heap memory is not thread safe.
* Heap area need not be continues.

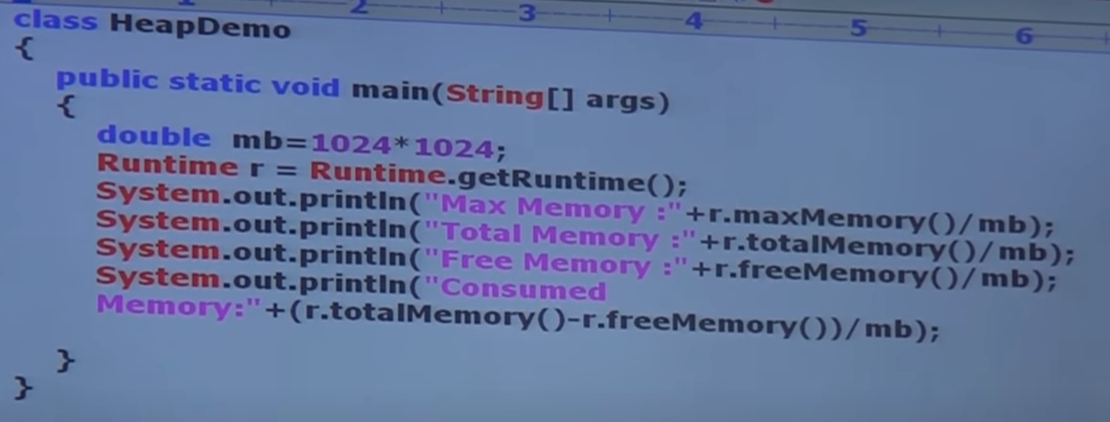


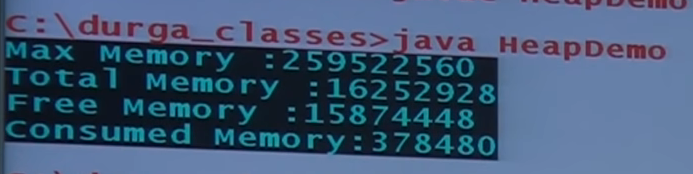
* **Program to display heap memory statistics**
* A java application can communicate with JVM by using Runtime object.
* Runtime class present in java.lang.package and it Is a singleton class.
* we can create runtime object as follows

e.g Runtime r = Runtime.getRuntime();

* once we get runtime object we can call following method.

1. maxMemory : it return number of bytes of max memory allocated to heap
2. totalMemory : it returns number of bytes of total memory allocated to the heap(initial memory).
3. freeMomory : it returns number of bytes of free memory present in heap.





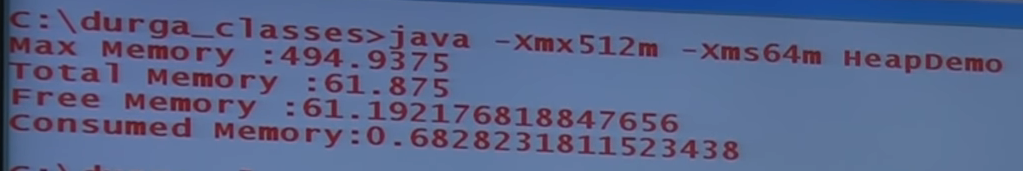
* **how to set maximum and minimum heap sizes :**
* heap memory is finite memory but based on our requirement we can set maximum and minimum heap sizes i.e we can increase or decrease heap size based on our requirement
* we can use following flags with java command.
* To set maximum heap size (max memory)

e.g java -Xmx512m HeapDemo

* This command will set max memory as 512 MB

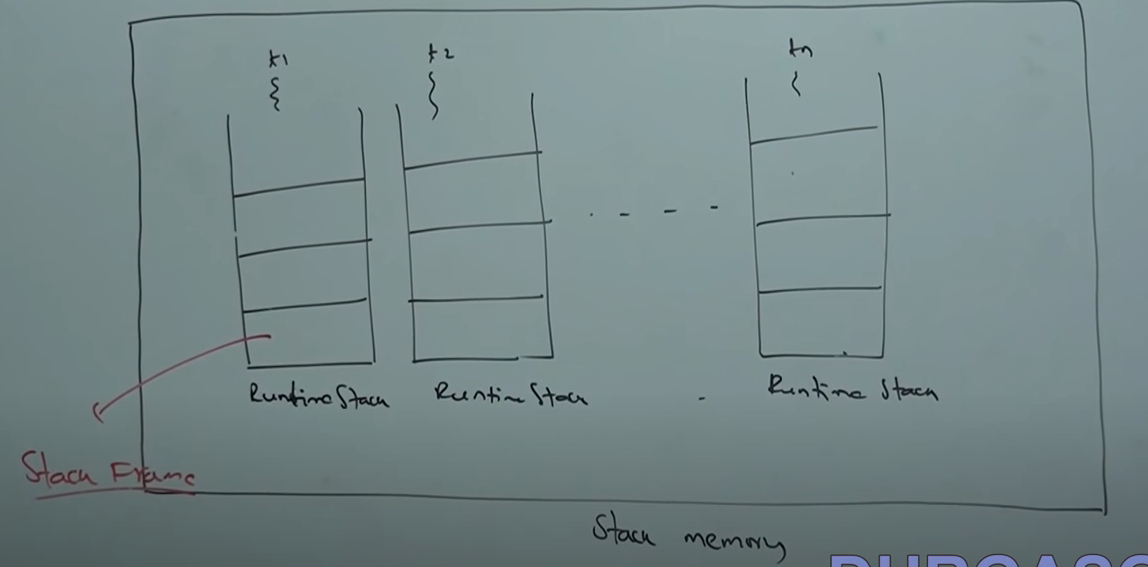
E.g java -Xms64sm HeapDemo

* This command will set min memory as 64 MB



1. **Stack memory :**

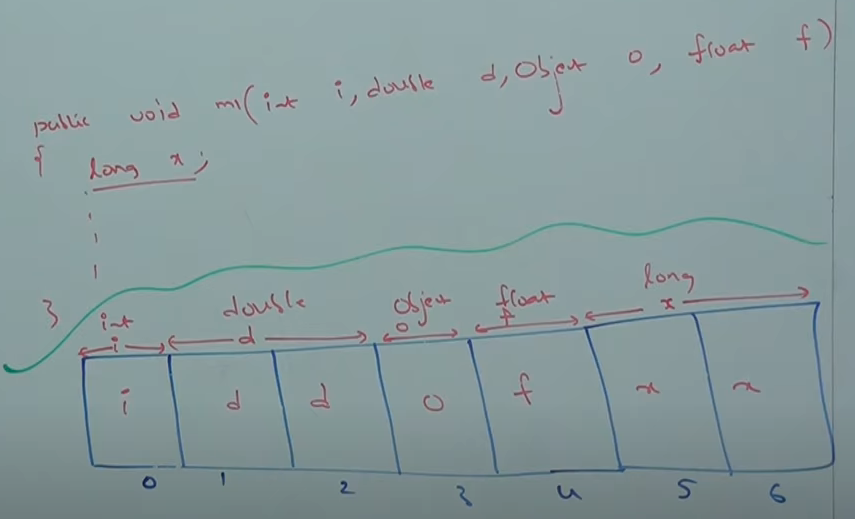
* For every thread JVM will create a separate stack at the time of thread creation.
* Each and every method call performed by that thread will be stored in the stack including local variables also
* After completing a method the corresponding entry will be removed.
* After completing all method calls the stack will become empty and that empty stack will be destroyed by JVM just before terminating thread.
* Each entry in the stack is called stack frame or activation record.
* The data stored in the stack is available only corresponding thread and not available to the remaining threads hence this data is thread safe.



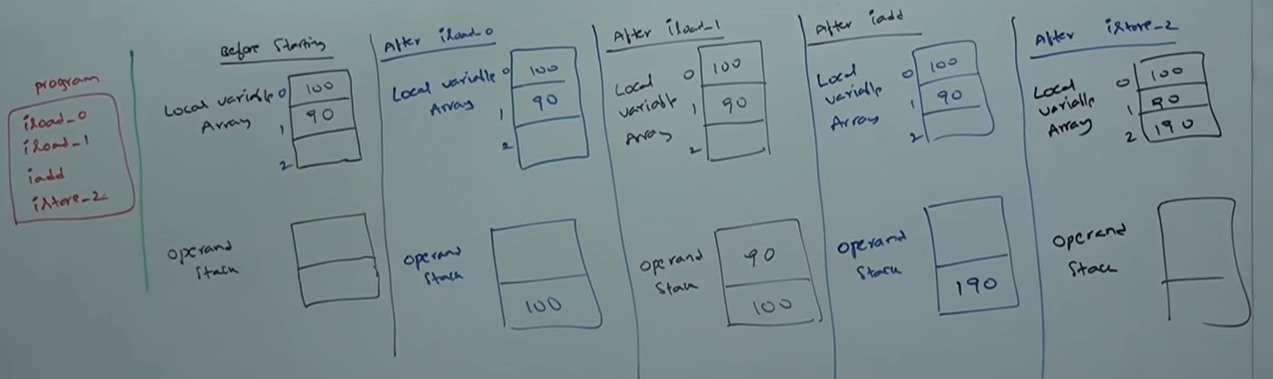
* **Stack frame structure :**
* **Each stack frame contains three parts.**



* **Local variable array :**
* It contains all parameters and local variables of the method.
* Each slot in the array is of 4 bit.
* Values of type int , float and reference occupy one entry in array
* Values of double and long occupy to consecutive entries in the array
* Byte short and char values will be converted to int type before storing and occupying one slot.
* But the way of storing Boolean values is varied from JVM to JVM but most of the JVM follows one slot for Boolean values.



* **Operand stack :**
* JVM uses operand stack as work space.
* Some instructions can push the values to the operand stack and some instractions can pop values from operand stack and some instractions can perform required operations.



* **Frame Data :**
* Frame data contains all symbolic refrances related to that method.
* it also contains a reference to exception table which provides corrosponding catch block information in the case of exceptions.

1. **PC Register(program countor registers) :**

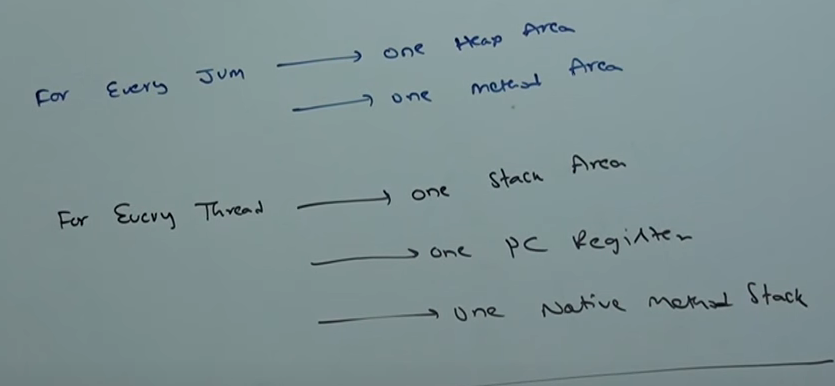
* For every thread a saperate PC register will be created at the time of thread creation.
* PC registers contains the address of current excecuting instraction.
* Once instruction execution complaites automaticaly PC register increment to hold the address of next instruction.

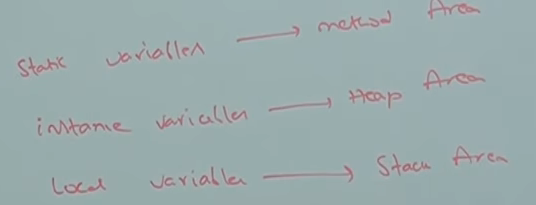
1. **Native Method Stack :**

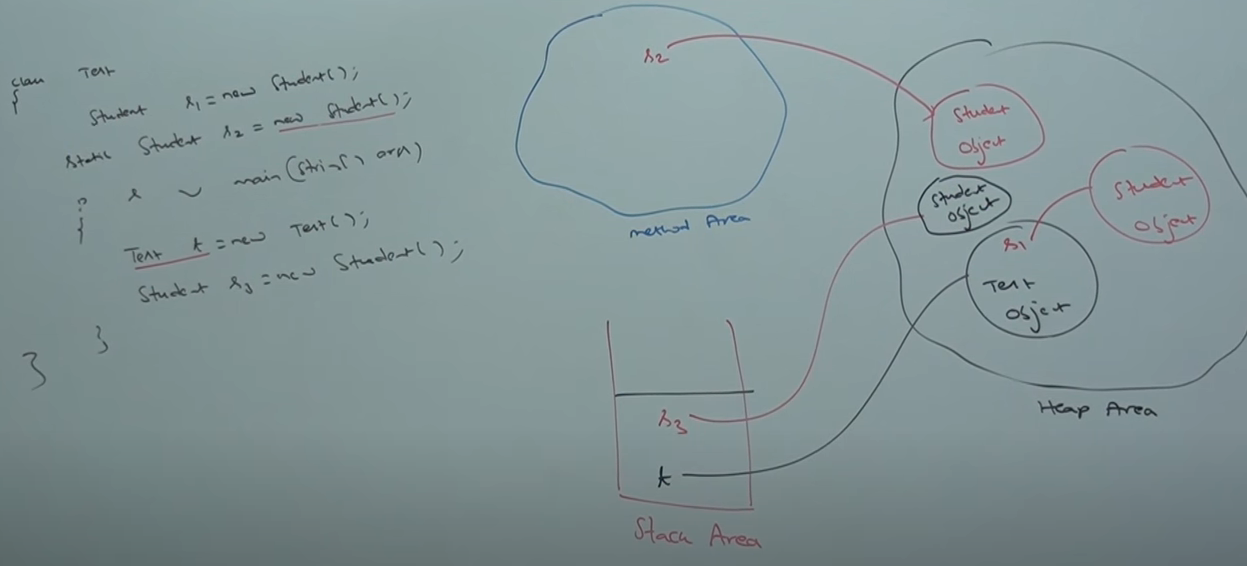
* For every thread JVM will create a saperate native method stack.
* All native method calls invoked by the thread will be stored in the corresponding native method stack.

**Note :**

* Method are, heam area and stack area are considered as important memory areas with respect to programer.
* Method are and heap area are per JVM whereas stack area, PC register and native method stack are per thread.







* **Execution engine :**
* This is the central componant of JVM.
* Excecution engin is responsible to execute java class files.
* Execution engin mainly contain two componants

1. Interpriter
2. JIT compiler
3. **Interpriter :**

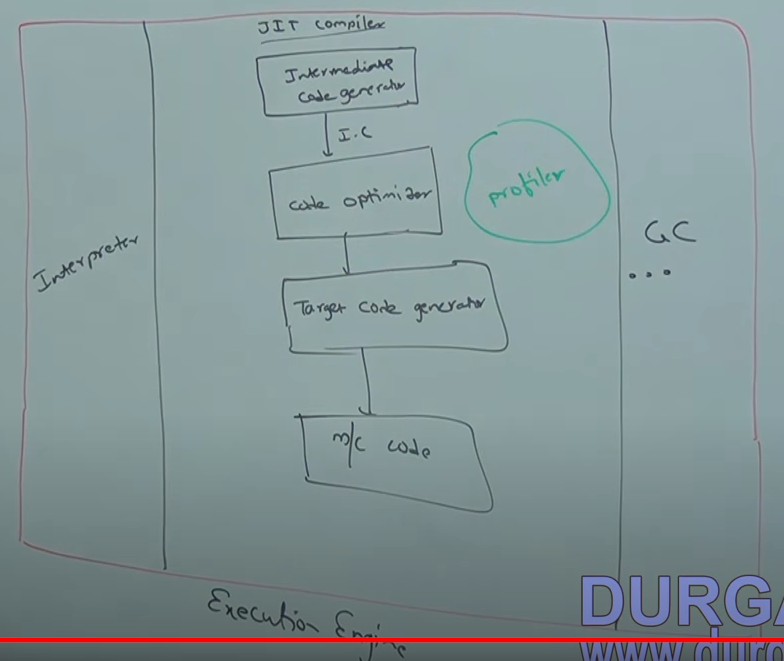
* It is responsible to read byte code and interprite into machine code (native code) and the execute that machine code line by line
* The problem with interpriter is, it interprit every time even same method invoked multiple time which reduces performance of the system
* To overcome this problem sun people introdue JIT compiler in 1.1 verssion.

1. **JIT compiler :**

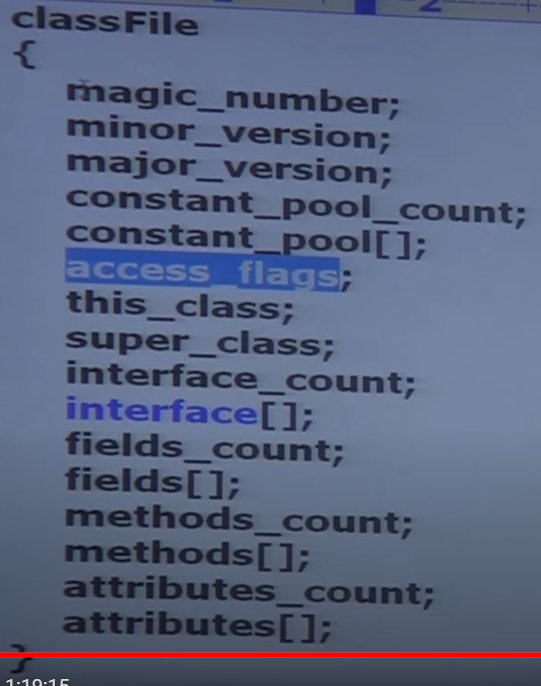
* The main perpose of JIT compiler is to improve performance.
* Internally JIT compiler maintain saperate count for every method.
* Whenever JVM come across any method call first that method will interprited normaly by the interpriter and the JIT compiler increaments the corrsponding count variable.
* This process will be continued for every method.
* Once if any method count reaches threshold value then jit compiler identifies that, method is a repetadely used method also known as **Hot spot.**
* Immidiatly JIT compiler compile that method and generates corresponding native code.
* Next time JVM come across that method call then JVM uses native code directly and executes it instade of interpriting once again so that performance of the system will be improved.
* The threshold count varied from JVM to JVM.
* Some advanced JIT compiler will recompile generated native code if count reaches threshold value secod time so that more optimised machine code will be generated.
* Internally **profiler** which is the part of JIT compiler is responsible to identify hotspot.

**Note :**

* JVM interprits total program atleast once.
* JIT compilation is applicable only for repatedly required methods not for every method.



* **JNI (Java Native Interface) :**
* JNI acts as mediator for java method calls and corresponding native librararies i.e JNI is responsible to provide information about native libraries to the JVM.
* Native method library provides/holds native library information.
* **Class file structure**



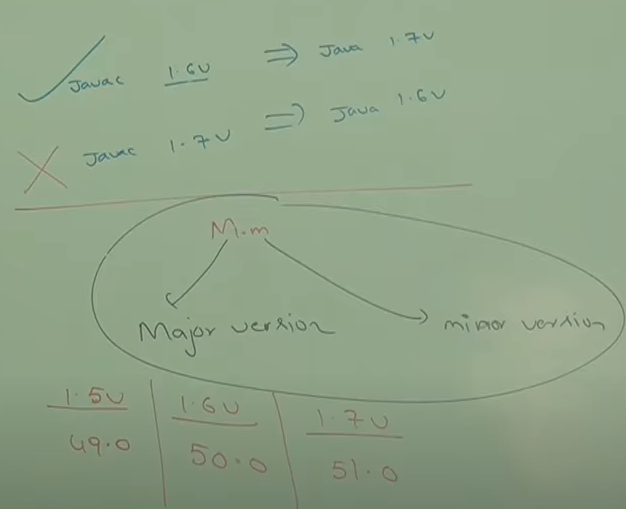
1. **Magic\_Number :**

* The first four bytes of a class file is the magic\_number
* This is predefine value used by JVM to identify .class generated by valid compiler or not
* The value should be 0xCAFEBABE

**Note**: whenever we are executing a java file if JVM unable to find magic number then we will get runtime exception saying java.lang.classFormatExceotion incampatible magic value

1. **Minor and Major Verssion :**

* It represents .class file version JVM will use these versions to identify which version of compiler generates the current .class file.



**Note :**

* Lower version compiler generated class files can be run by higher version JVM but higher version compiler generated .class files cant be run lower version JVMs. If we are trying to run we will get runtime exception saying **UnsupportedClassVersionError**

1. **Constant\_pool\_count :**

* It represent number of constatnt present in constant pool

1. **Constant\_pool\_Arrays :**

* It represent information about constatnt present in constant pool

1. **Access\_flag :**

* It provide information about modifiers which are declaired to the class.

1. **This.class :**

* It represent fully qulified name if the class

1. **Supper class:**

* It represent fully qulified name of immidiate supper class of current class

1. **Interface count :**

* It returns numbe rof interfaces implemented by current class

1. **Interface array :**

* It returns interfaces information implemented by current class.

1. **Fields count :**

* It represents number of fields present in the class

1. **Fields array :**

* It represents fields information present in class.

1. **Method count :**

* It represents number of methods present in current class.

1. **Methods array :**

* It provides information about all methods present in class.

1. **Attributes count :**

* It returns number of attribute present in class

1. **Attributes array :**
2. It provides information about all attribute present in class.