

Virtual Machine

Type of Virtual Machines

- 1. Hard ware Based VM**
- 2. Application Based VM**

Basic Architecture of JVM

Class Loader SubSystem

- 1. Loading**
- 2. Linking**
- 3. Initialization**

Types of Class Loaders

- 1. Bootstrap Class Loader**
- 2. Extension Class Loader**
- 3. Application Class Loader**

How Class Loader works

What is the need of Customized Class Loader

Pseudo code for Customized Class Loader

Various Memory Areas of JVM

- 1. Method Area**
- 2. Heap Area**

Various Memory Areas of JVM

- 1. Method Area**
- 2. Heap Area**
- 3. Stack Area**
- 4. PC Registers**
- 5. Native Method Stacks**

Program to display heap memory statistics

How to set Maximum and Minimum heap size?

Execution Engine

- 1. Interpreter**
- 2. JIT Compilers**

Java Native Interface(JNI)

Complete Architecture Diagram of JVM

Class File Structure

Virtula Machine:

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

There are two types of vartual machine

1. Hardware based or system based
2. Application based or process based

Hardware based or system based:

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

i.e. on one physical machine we are defining multiple logical machines.

The main advantadge of hardware based virtual machines is hardware resources sharing and improves utilization of hardware resources.

e.g.

KVM(Kernel based VM) for Linux systems

VMWare

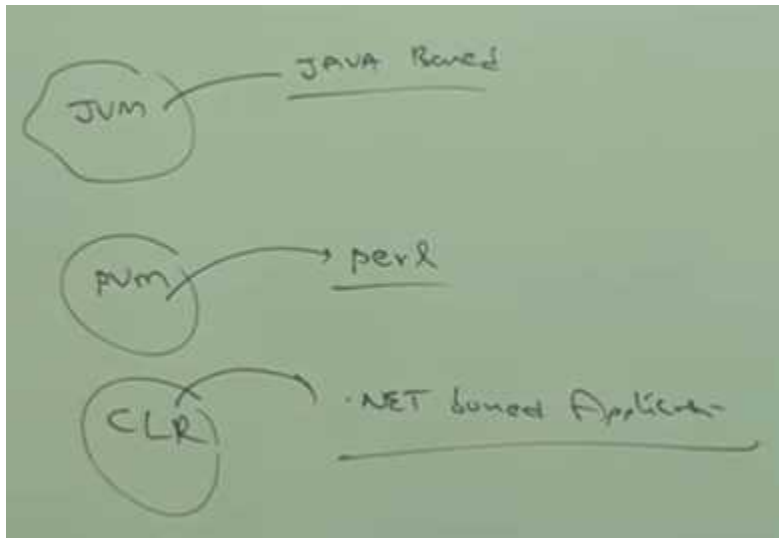
Xen

Could Computing etc..

Application based or process based VM:

These VMs acts as runtime engines to run a particular programming language applications.

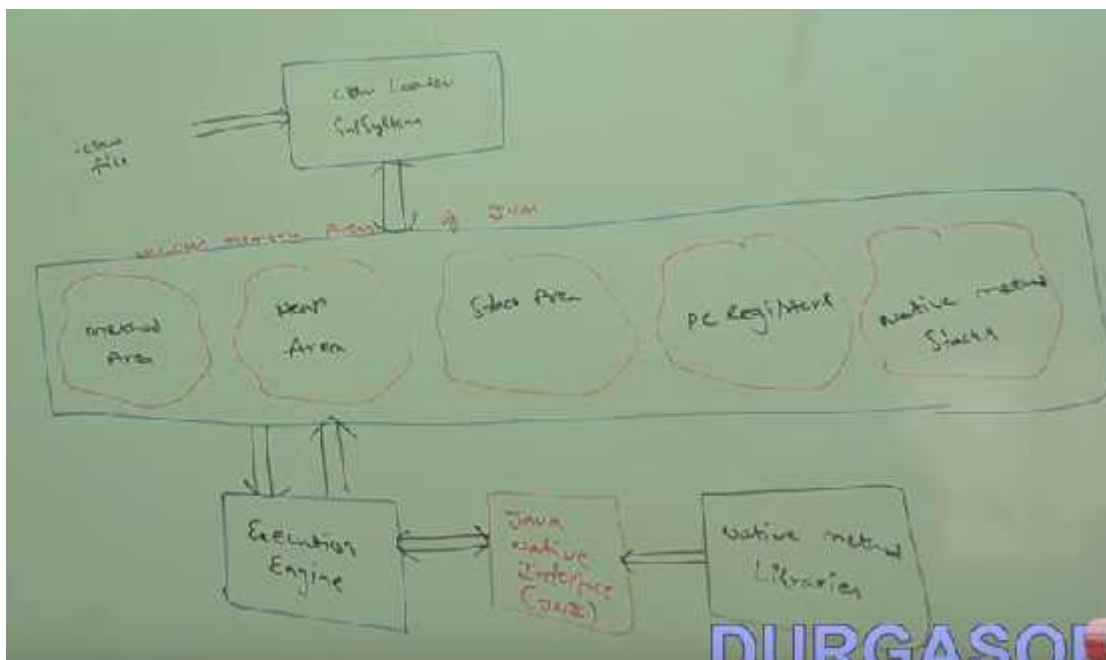
e.g.



JVM:

JVM is the part of JRE and it is responsible to load and run java class files.

Basic Architecture Diagram of JVM:



Class Loader Subsystem:

class loader subsystem is responsible for the following 3 Activities,

1. Loading

2. Linking

3. Initialization

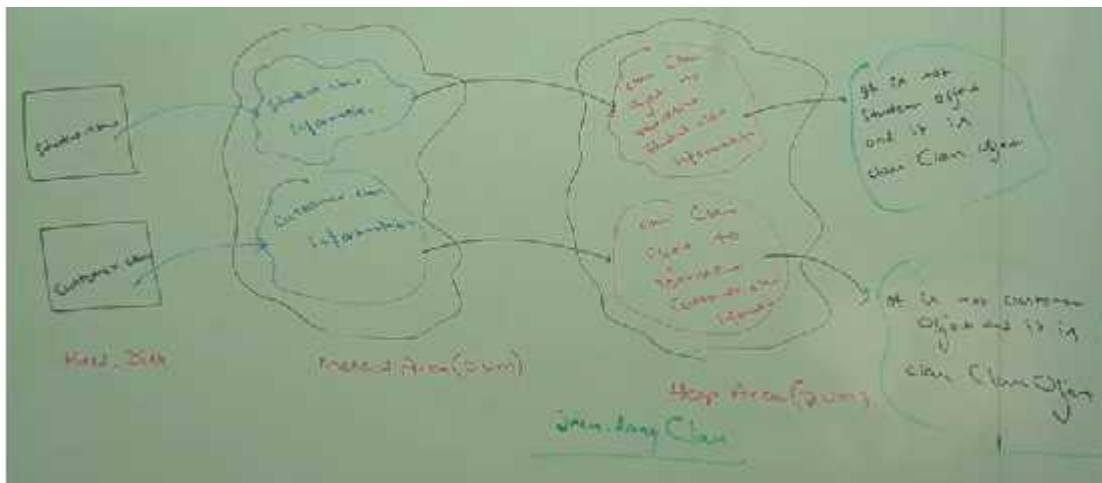
Loading:

Loading means reading .class files and store corresponding binary data in **method area**.

For each class file JVM will store corresponding information in the method area.

1. Fully qualified name of class
2. Fully qualified name of immediate parent class
3. Methods info, variables info, constructors info
4. Modifiers info, constant pool info etc..

After loading .class file immediately JVM creates an object for that loaded class on 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, constructors information etc..

```

class Student
{
    public String name;
    return null;
}

public int getRollNo()
{
    return 10;
}
}

// Main
public static void main(String[] args)
{
    int count = 0;
    Class c = Class.forName("Student");

    Method m = c.getDeclaredMethod("getRollNo");
    m.setAccessible(true);
    Student s = (Student) m.newInstance();
    System.out.println(s.getRollNo());
}
}

```

Note:

For every loaded type only one class object will be created even though we are using the multiple times in our program.

```

// Main
public static void main(String[] args)
{
    Student s1 = new Student();
    Class c1 = s1.getClass();
    Student s2 = new Student();
    Class c2 = s2.getClass();
    System.out.println(c1.hashCode());
    System.out.println(c2.hashCode());
    System.out.println(c1 == c2); // true
}
}

```

Diagram illustrating the JVM's class loading mechanism:

- Two `Student` objects are created, each with its own `Class` object.
- The `Class` objects are linked to a single `Class` object in memory, demonstrating that only one `Class` object is created for a given class.

In the above program even though we are using student class multiple times only one class class got created.

Linking:

Linking consists of three activities

1. Verify
2. Prepare
3. Resolve

Verify:

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

Internally bytecode verifier is responsible for this activity.

ByteCode verifier is the part of classloader subsystem.

If verification fails then we will get runtime exception saying `java.lang.verifyerror`.

Preparation:

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

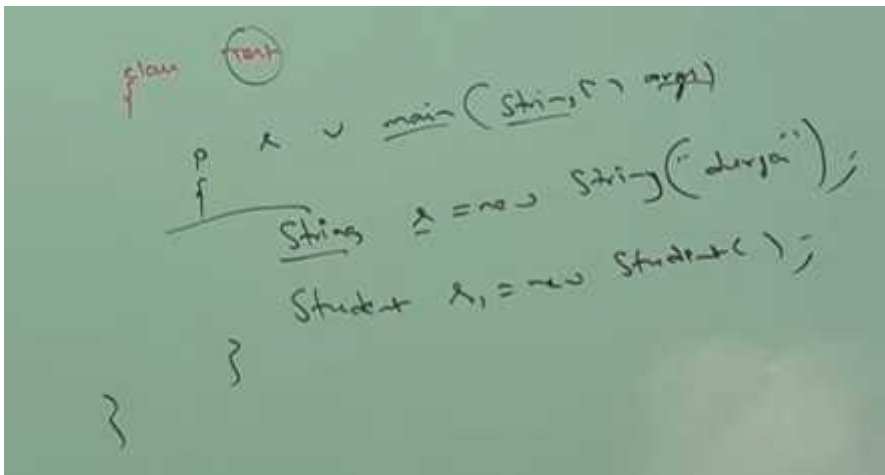
Note:

In initialization phase original values will be assigned to the static variables and here only default values will be assigned.

Resolution:

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

e.g.



```
class Test {  
    public static void main(String[] args) {  
        String s = new String("durga");  
        Student s1 = new Student();  
    }  
}
```

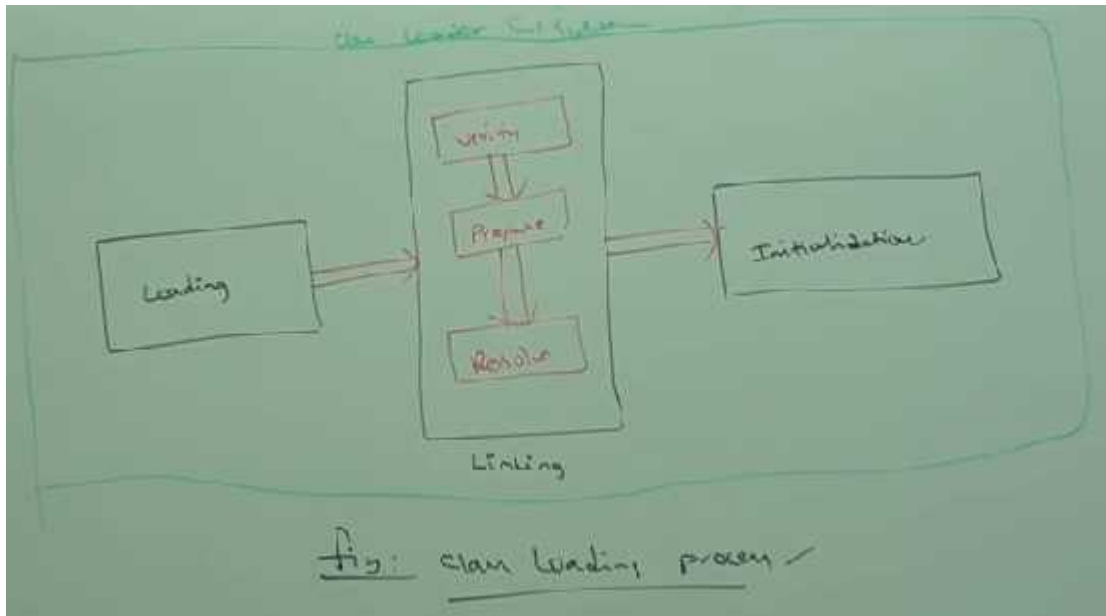
For the above class classloader loads `Test.class`, `String.class`, `Student.class` and `Object.class`

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

In resolution phase these names are replaced with original memory level reference from method area.

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.



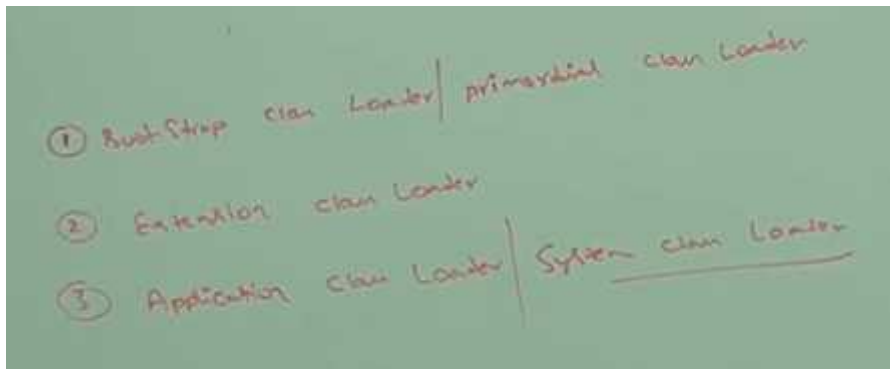
Note:

While loading, linking and initialization if any error occurs then we will get runtime exception saying `java.lang.LinkageError`.



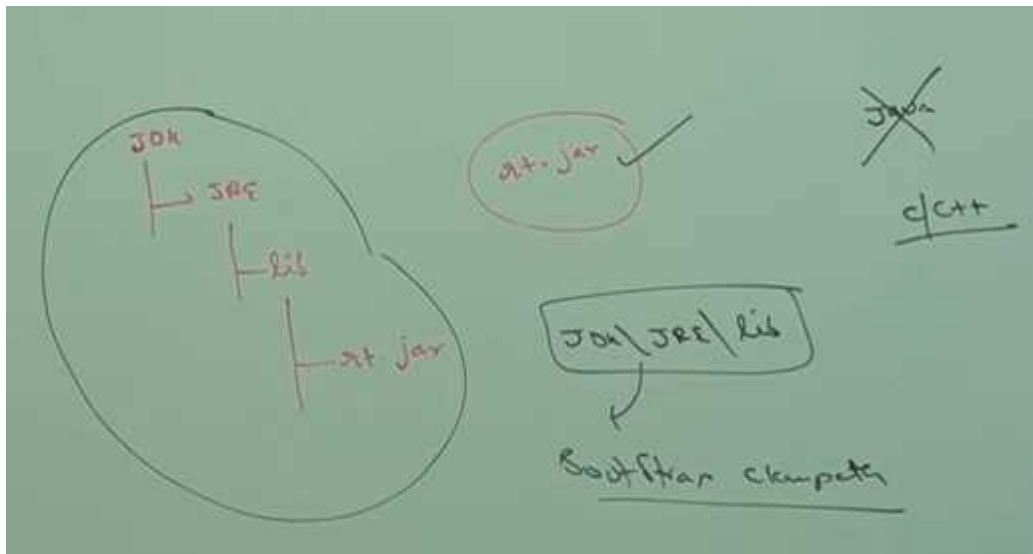
Types of classloaders:

ClassLoader subsystem contains the following three types of classloaders,



Bootstrap classloader:

Bootstrap classloader is responsible to load core java api classes. i.e. the classes present in rt.jar.



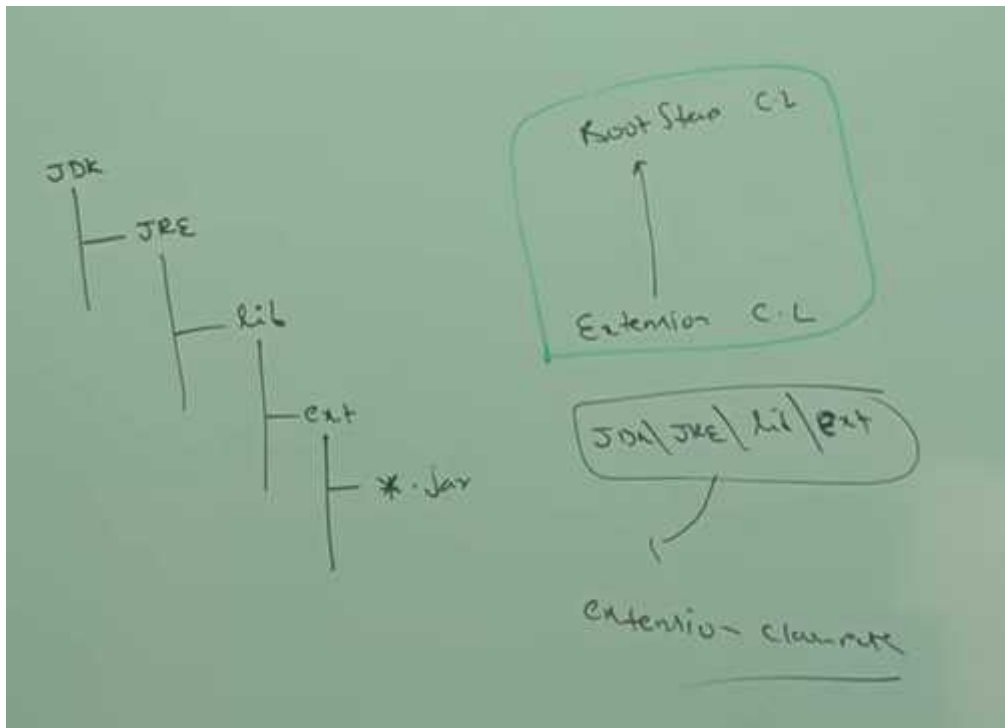
This location is called bootstrap classpath i.e. Bootstrap classloader is responsible to load classes from Bootstrap classpath.

Bootstrap classloader is by default available with every JVM. It is implemented in native languages like C/C++ and not implemented in Java.

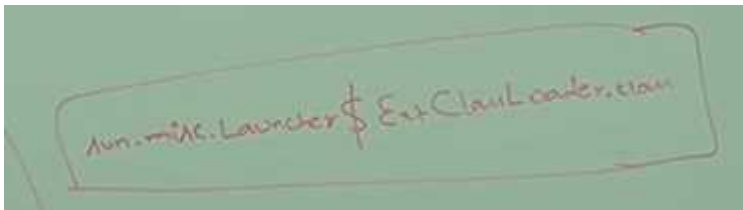
Extension classloader:

Extension classloader is the child class of Bootstrap Classloader.

It is responsible to load classes from extension classpath.



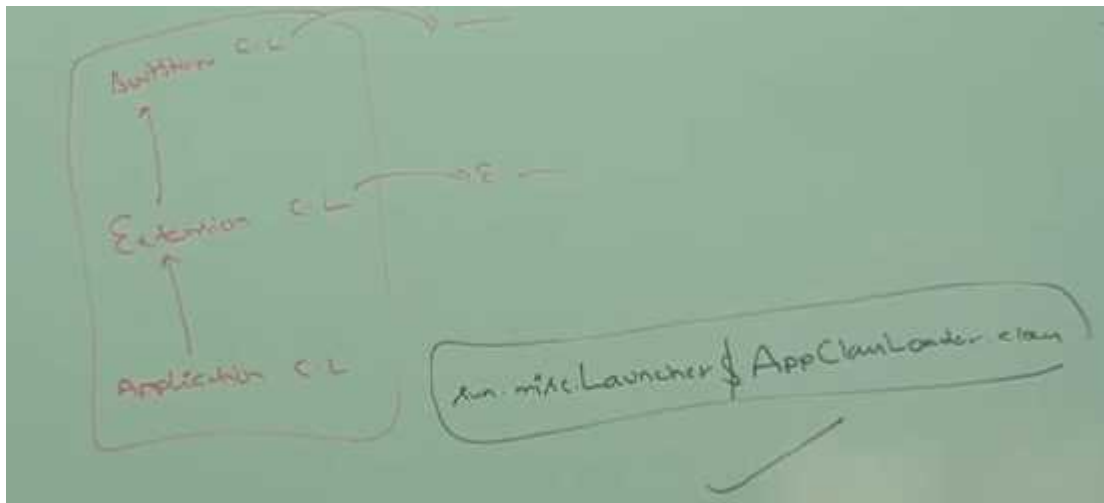
Extension classloader is implemented in java and the corresponding .class file is



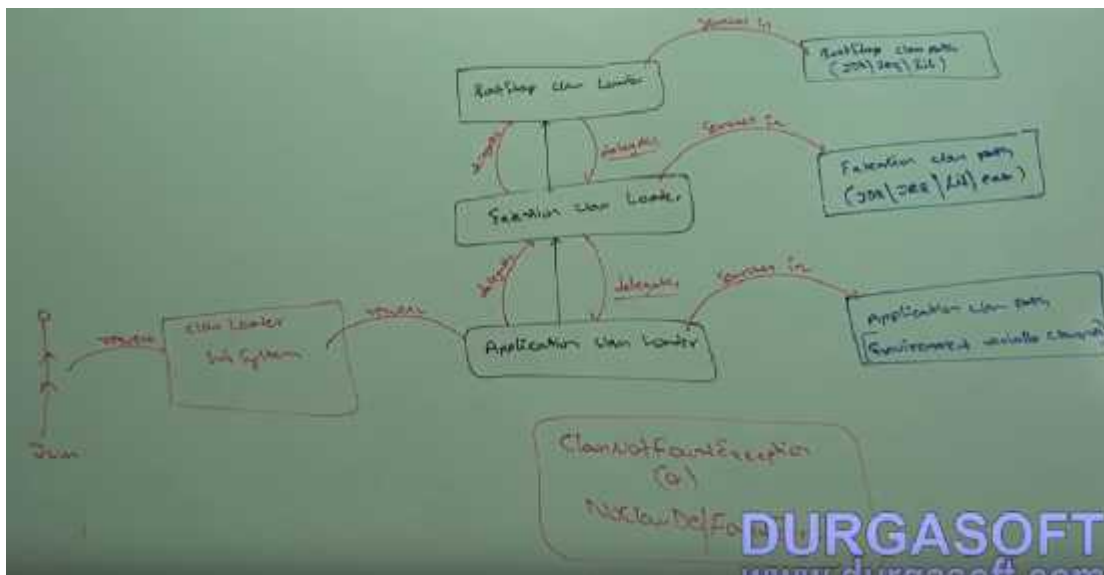
Application ClassLoader:

Application classloader is the child class of Extension classloader. This classloader is responsible to load classes from Application classpath. It internally uses environment variable classpath.

Application classloader is implemented in java and the corresponding .class file name is



How ClassLoader Works?:



Class loader follows delegation hierarchy principle.

Whenever JVM come across a particular class 1st 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 classloader subsystem to load that particular class.

Then classloader subsystem handovers the request to Application classloader.

Application classloader delegates that request to Extension classloader which in turn delegates the request to Bootstrap classloader.

Then Bootstrap classloader will search in bootstrap classpath. If it is available then the

corresponding .class will be loaded by bootstrap classloader.

If it is not available then bootstrap classloader delegates that request to extension classloader.

Extension classloader will search in extension classpath.

If it is available then it will be loaded otherwise extension classloader delegates that request to application classloader.

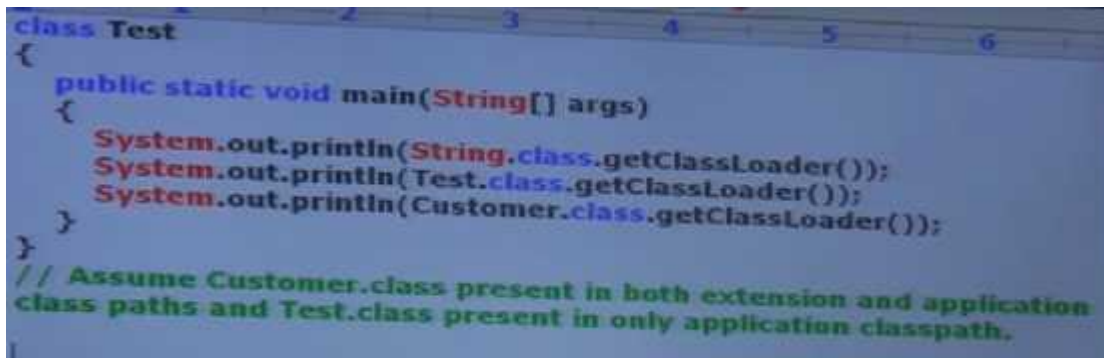
Application classloader will search in application classpath.

If it is available then it will be loaded otherwise we will get Runtime exception saying,

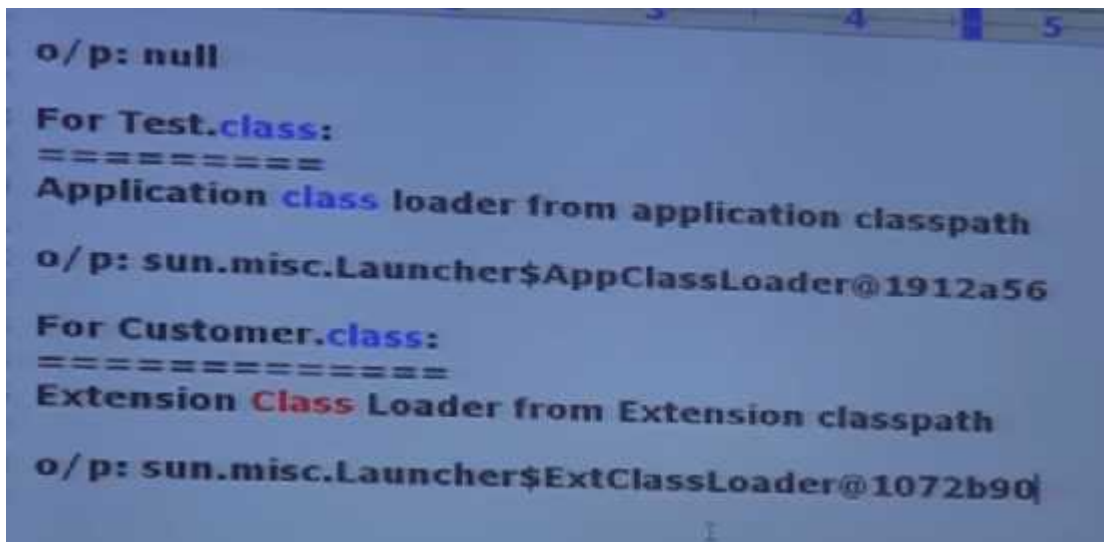
NoClassDefFoundError

OR

ClassNotFoundException



```
class Test
{
    public static void main(String[] args)
    {
        System.out.println(String.class.getClassLoader());
        System.out.println(Test.class.getClassLoader());
        System.out.println(Customer.class.getClassLoader());
    }
}
// Assume Customer.class present in both extension and application
class paths and Test.class present in only application classpath.
```



```
o/p: null
For Test.class:
=====
Application class loader from application classpath
o/p: sun.misc.Launcher$AppClassLoader@1912a56
For Customer.class:
=====
Extension Class Loader from Extension classpath
o/p: sun.misc.Launcher$ExtClassLoader@1072b90
```

Note:

BootStrap ClassLoader is not java object hence we got null in the 1st case.

But Extension and Application classloaders are java objects, hence we are getting corresponding output for the remaining two sops.(className@hashCode in hexadecimal form).

ClassLoader subsystem will give the highest priority for BootStrap classpath and then extension classpath followed by application classpath.

Need of Customized classloader:

default classloaders will load .class file only once even though we are using multiple times, that class in our program.

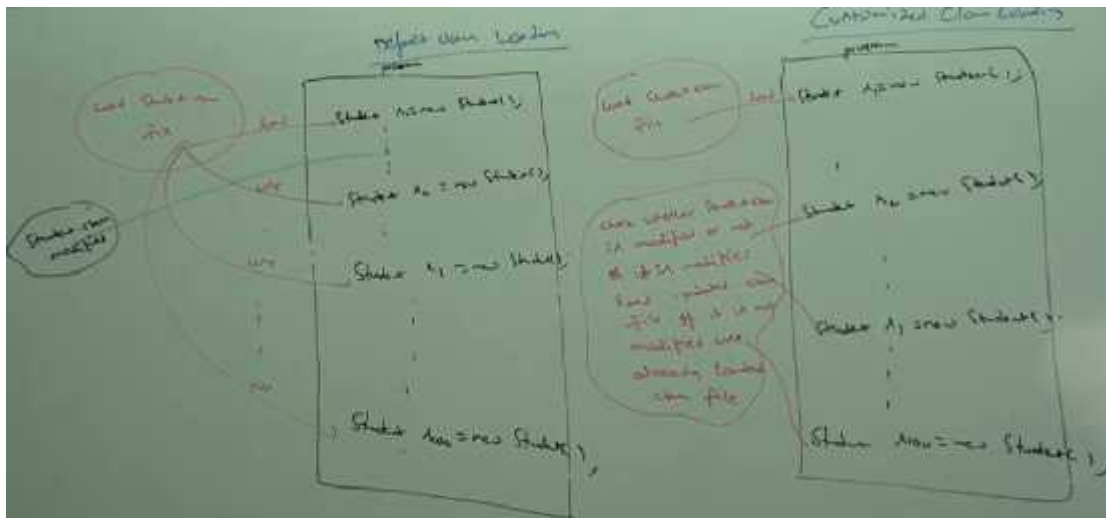
After loading .class file if it is modified outside then default classloader won't load updated version of class file.(because .class file already available in method area)

We can resolve this problem by defining our own customized classloader.

The main advantage of customized classloader is we can control classloading mechanism based on our requirement.

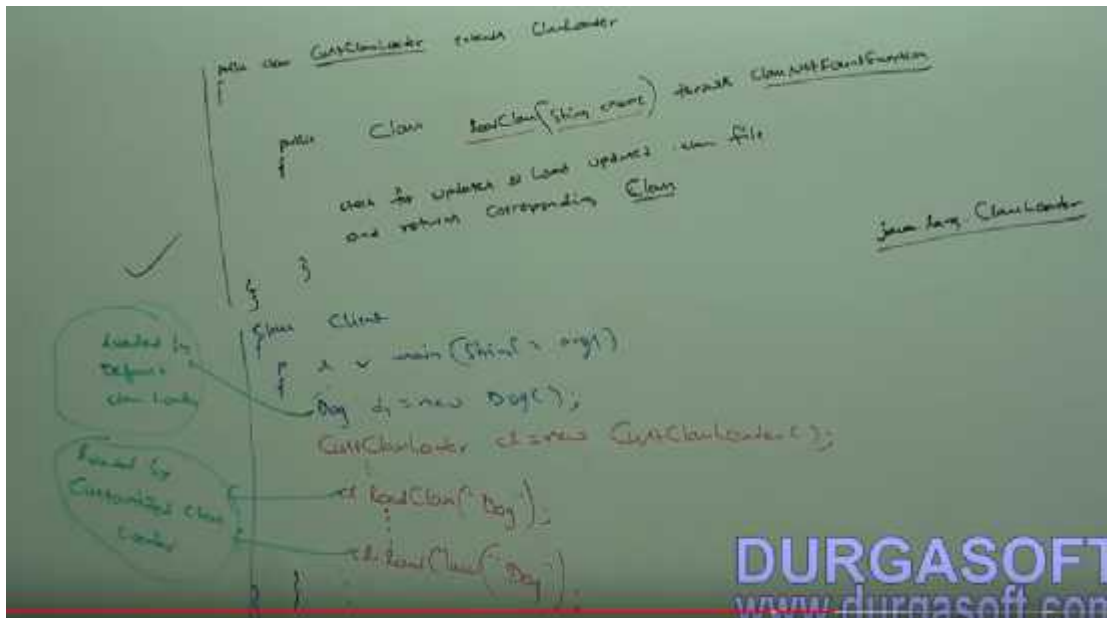
e.g.

We can load .class file separately every time so that updated version available to our program.



How to define customized classloader:

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



Note:

While developing webservers and application servers usually we can go for customized classloaders to customize classloading mechanism.

Q. What is the need of ClassLoader class ?

We can use java.lang.ClassLoader class to define our own customized classloaders.

Every classLoader in java should be child class of java.lang.ClassLoader class either directly and indirectly, hence this class acts as base class for all customized classloaders.

Various Memory Areas Present Inside JVM:

Whenever JVM loads and runs a java program, it needs memory to store several things, like bytecode, objects, variables etc.

Total JVM memory organized into the following 5 categories,

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

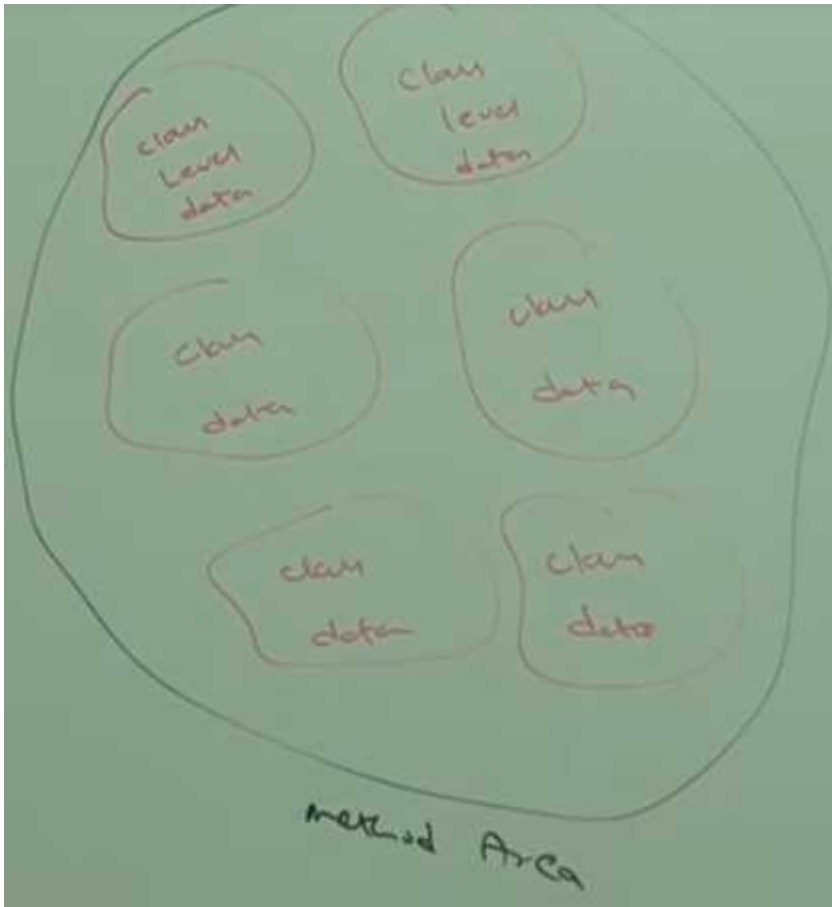
Method Area:

For every JVM one method area will be available.

Method area will be created at the time of JVM start up.

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.

Heap Area:

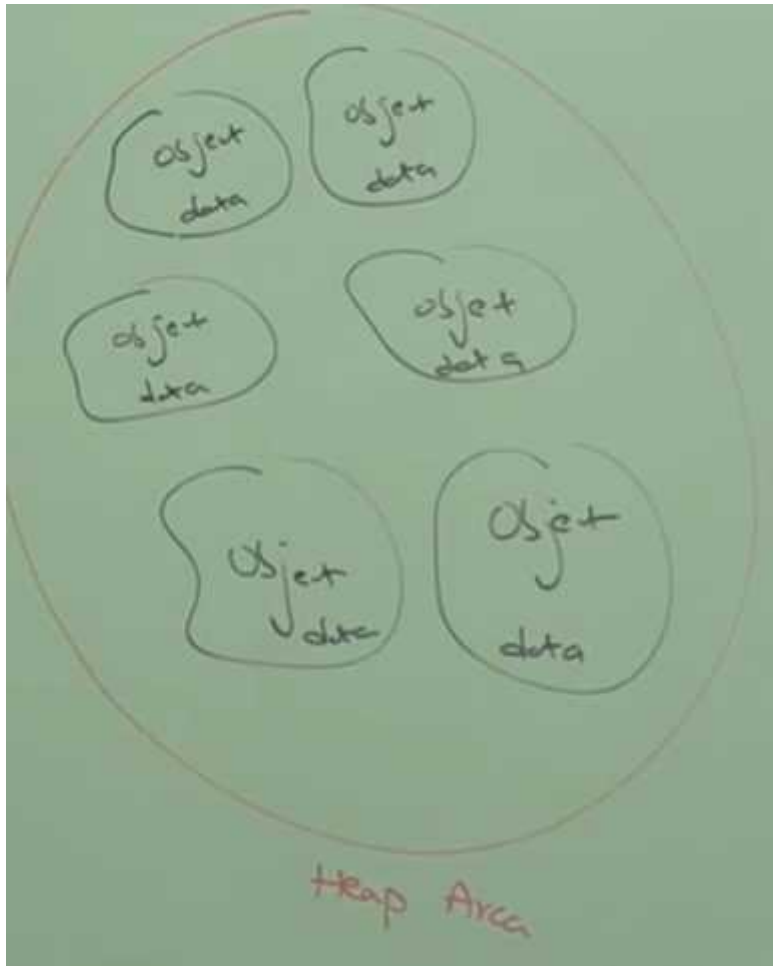
For every JVM one heap area is available.

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.

Heap area can be accessed by multiple threads and hence the data stored in the heap memory is not thread safe.



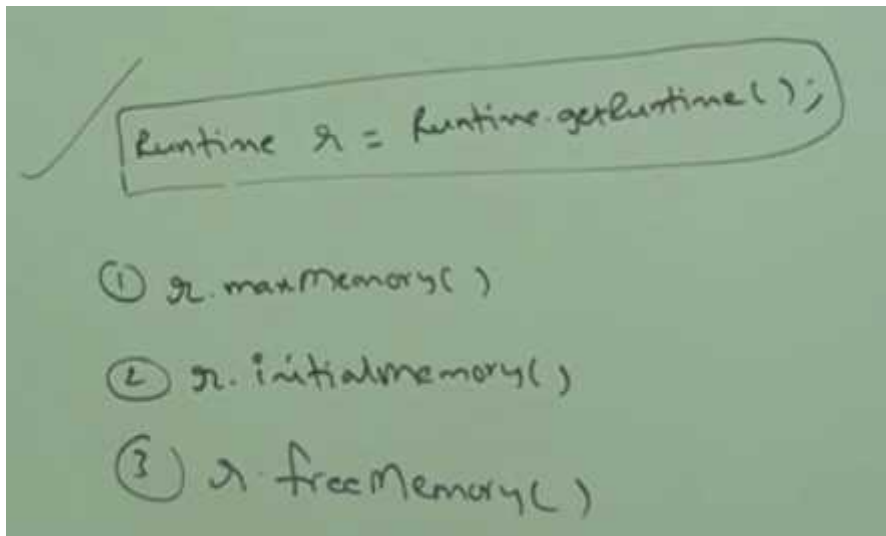
Heap area need not be contineous.

Program to display heap memory statistics:

A java application can communicate with JVM by using Runtime object.

Runtime class present in java.lang pkg and it is a singleton class.

We can create Runtime object as follows,



Once we got Runtime object we can call above methods on that object.

`r.maxMemory()`, number of bytes of memory allocated to heap.

`r.totalMemory()`, it returns number of bytes of total memory allocated to the heap.(initial memory).

`r.freeMemory()`, it returns number of bytes of free memory present in the heap.

```
class HeapDemo
{
    public static void main(String[] args)
    {
        double mb=1024*1024;
        Runtime r = Runtime.getRuntime();
        System.out.println("Max Memory :"+r.maxMemory()/mb);
        System.out.println("Total Memory :"+r.totalMemory()/mb);
        System.out.println("Free Memory :"+r.freeMemory()/mb);
        System.out.println("Consumed Memory:"+(r.totalMemory()-r.freeMemory())/mb);
    }
}
```

```
Max Memory :247.5
Total Memory :15.5
Free Memory :15.139053344726562
Consumed Memory:0.3609466552734375
```

Q. How to set maximum and minimum heap sizes ?


```
C:\durga_classes>java -Xmx512m HeapDemo
Max Memory :494.9375
Total Memory :15.5
Free Memory :15.139053344726562
Consumed Memory:0.3609466552734375
C:\durga_classes>
```

```
C:\durga_classes>java -Xms64m HeapDemo
Max Memory :247.5
Total Memory :61.875
Free Memory :61.192176818847656
Consumed Memory:0.6828231811523438
C:\durga_classes>
```

```
C:\durga_classes>java -Xmx512m -Xms64m HeapDemo
Max Memory :494.9375
Total Memory :61.875
Free Memory :61.192176818847656
Consumed Memory:0.6828231811523438
C:\durga_classes>
```

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 the heap size based on our requirement.

We can use below flags with java command

-Xmx (Maximum heap size, max memory)

This command will set maximum heap size

-Xms (Minimum memory)

This command will set minimum heap size

Stack Memory:

For every thread JVM will create a separate stack at the time of thread creation.

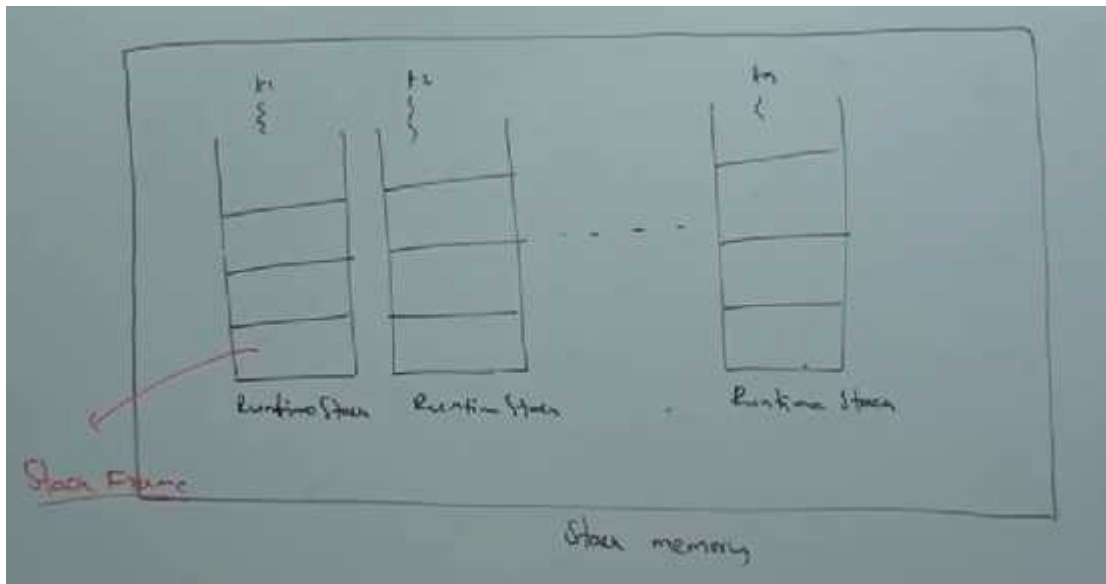
Each and every method call performed by that thread will be stored in the stack including local variables also.

After completing a method the corresponding entry from the stack will be removed. After completing all method calls the stack will become empty and 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.

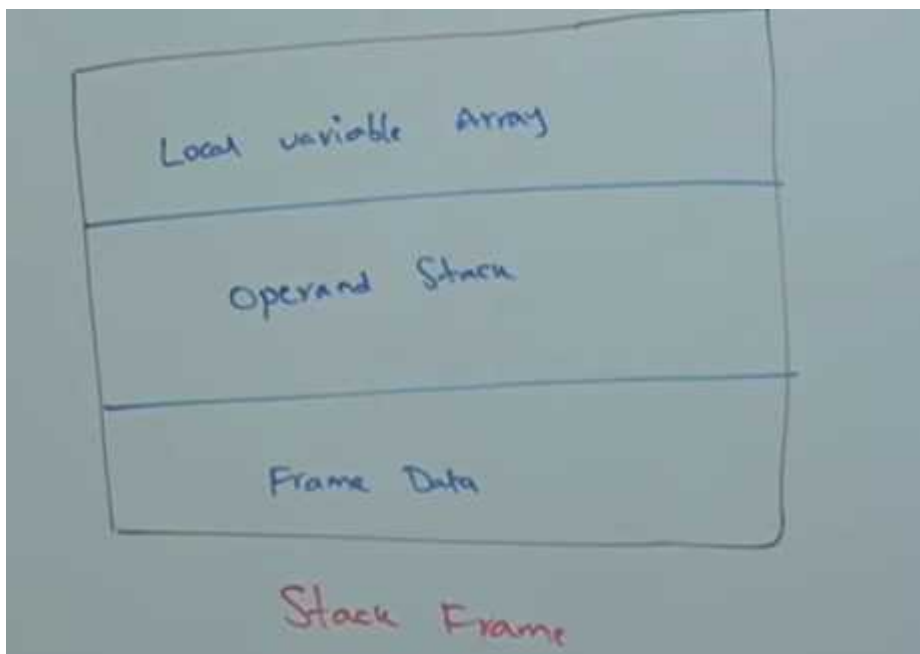
The data stored in the stack is available for the corresponding thread only and not available to

the remaining threads hence this data is thread safe.



Stack Frame Structure:

Each stack frame contains three parts



Local Variable Array:

Contains all parameters and local variables of the method.

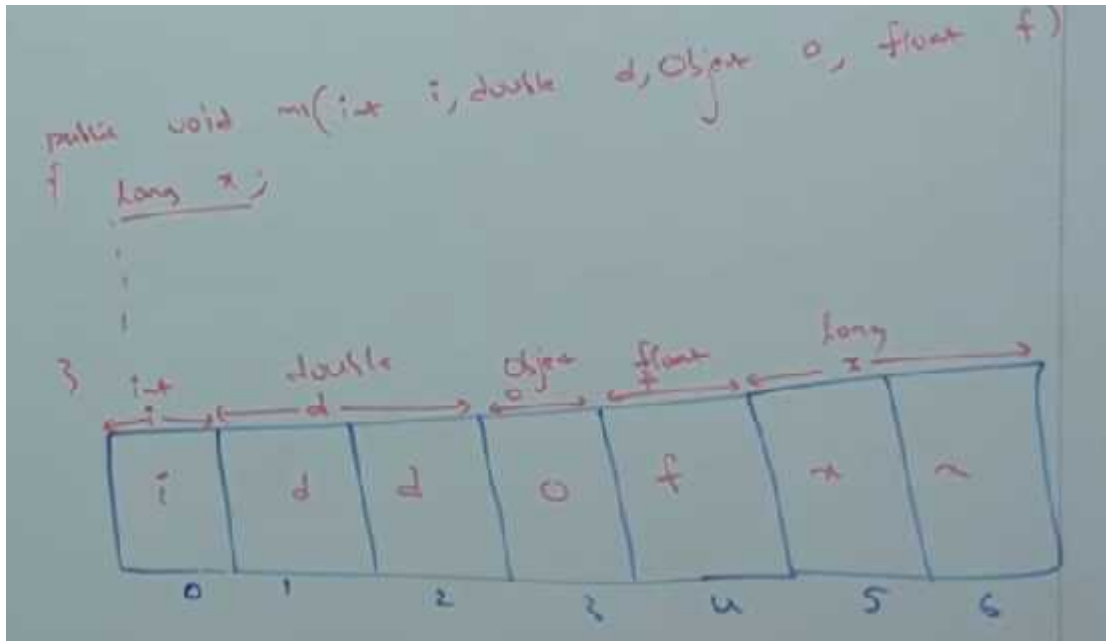
Each slot in the array is of 4 bytes.

Values of type int, float and reference occupy 1 entry in the array.

Values of double and long will occupy two consecutive entries in the 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. But most of the JVMs follow one slot for boolean values.



Operand Stack:

JVM uses operand stack as workspace.

Some instructions can push values to operand stack and some operation can pop values from operand stack and some instructions can perform required operations.



Frame Data:

Frame data contains all symbolic references related to that method.

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

PC Registers:(Program counter Registers)

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

PC registers contains the address of current executing instruction.

Once instruction execution completes automatically PC register will be incremented to hold address of next instruction.

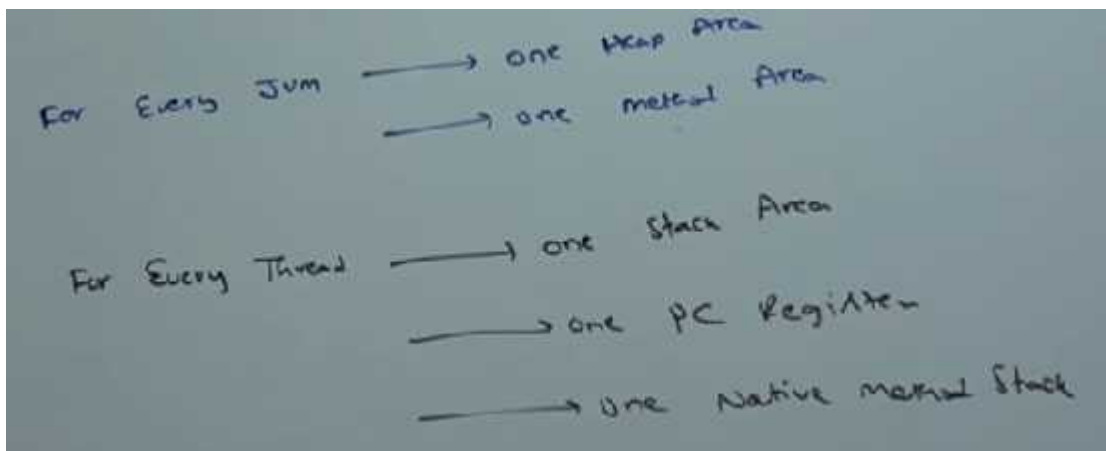
Native Method Stacks:

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.

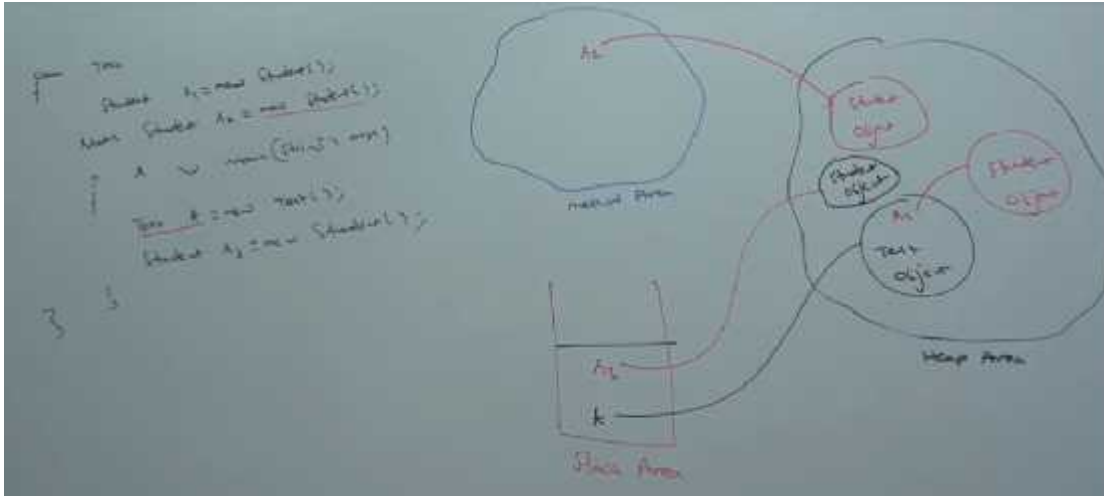
Conclusions:

1. Method area, Heap area and Stack area are considered as important memory areas w.r.t programmer.
2. Method area and heap area are per JVM. where as Stack area, PC Registers and native method stack are per thread i.e.



3. Static variables will be stored in method area, instance variables will be stored in heap area.

4. Local variables will be stored in stack area.



Execution Engine:

This is the central component of JVM.

Execution engine is responsible to execute java class files.

Execution engine mainly contains two components

1. Interpreter
2. JIT Compiler

Interpreter:

It is responsible to read bytecode 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 invoke multiple times which reduces performance of the system.

To overcome this problem Sun people introduced JIT compilers in 1.1 version.

JIT Compiler:

The main purpose of JIT compiler is to improve performance. Internally JIT compiler maintains a separate count for every method.

Whenever JVM comes across any method call 1st 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, that method is a repeatedly used method (hotspot).

Immediately JIT compiler compiles that method and generates the corresponding native code.

Next time JVM come across that method call, then JVM uses native code directly and executes it instead of interpreting once again so that performance of the system will be improved.

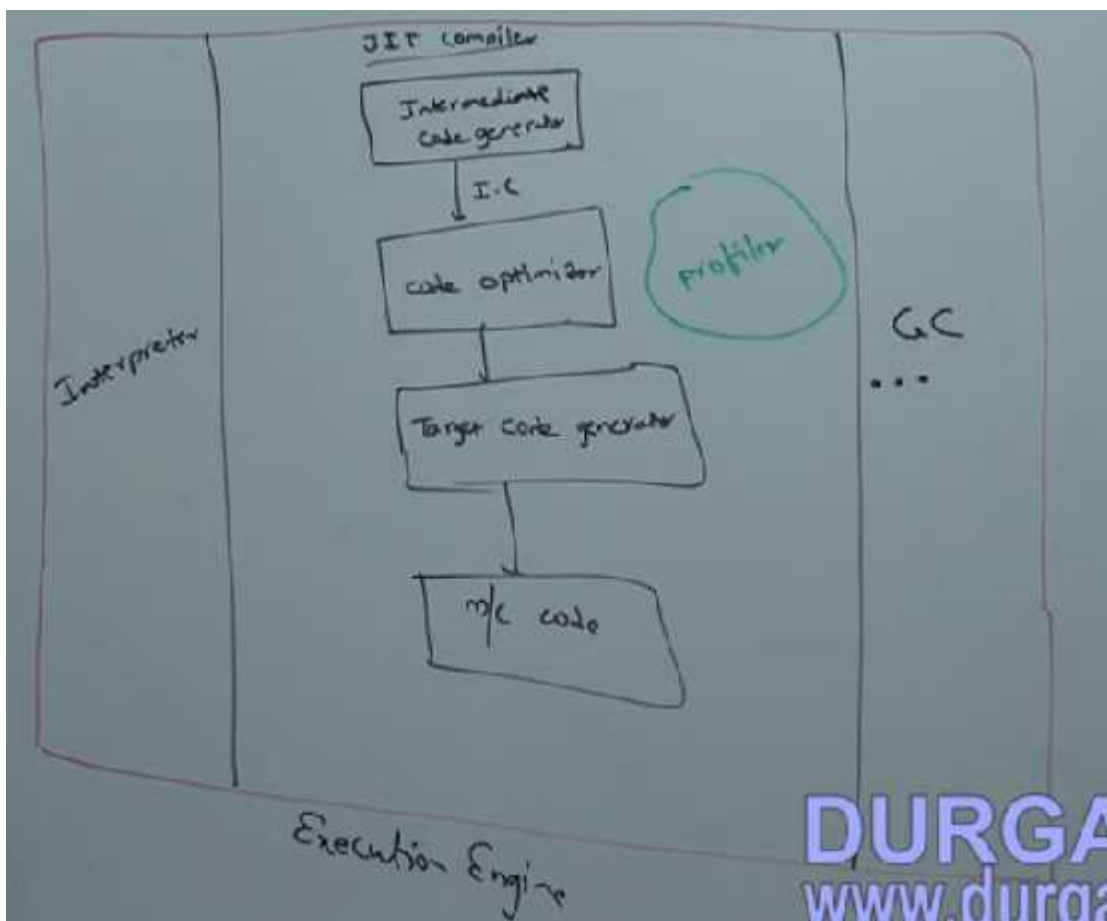
The threshold count varied from JVM to JVM.

Some advanced 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 the part of JIT compiler which is responsible to identify hotspot.

Note:

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

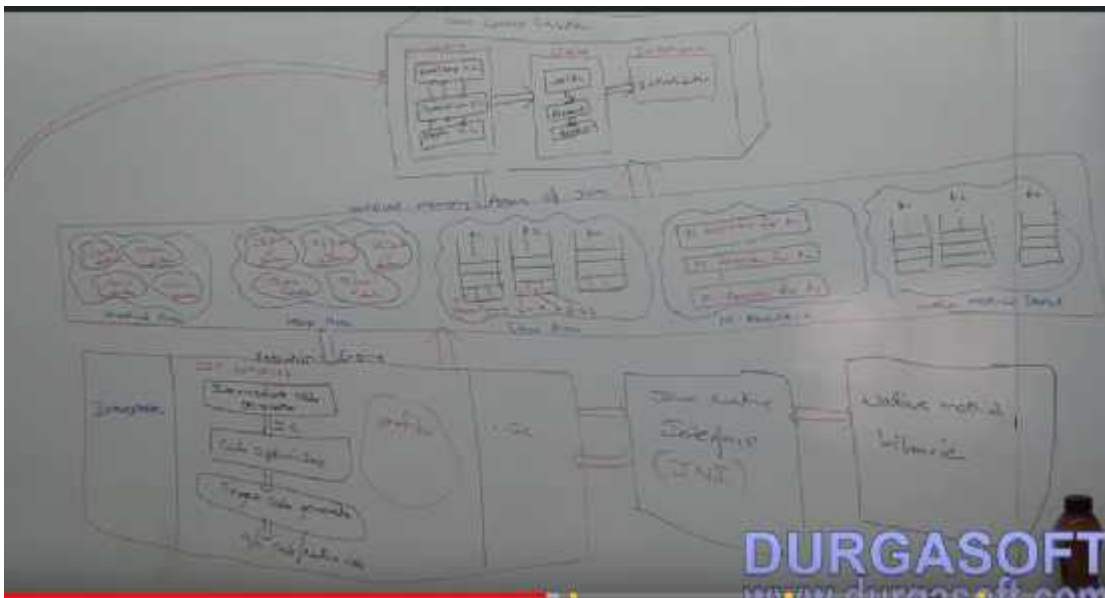


JNI:

JNI acts as mediator for java method calls and 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.

complete jvm architecture:



Class File Structure:

```

classFile
{
    magic_number;
    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_number:

The first 4 bytes of the class file is magic number. This is a pre defined value used by JVM to identify .class file is generated by valid compiler or not.

The value should be 0XCAFEBABE

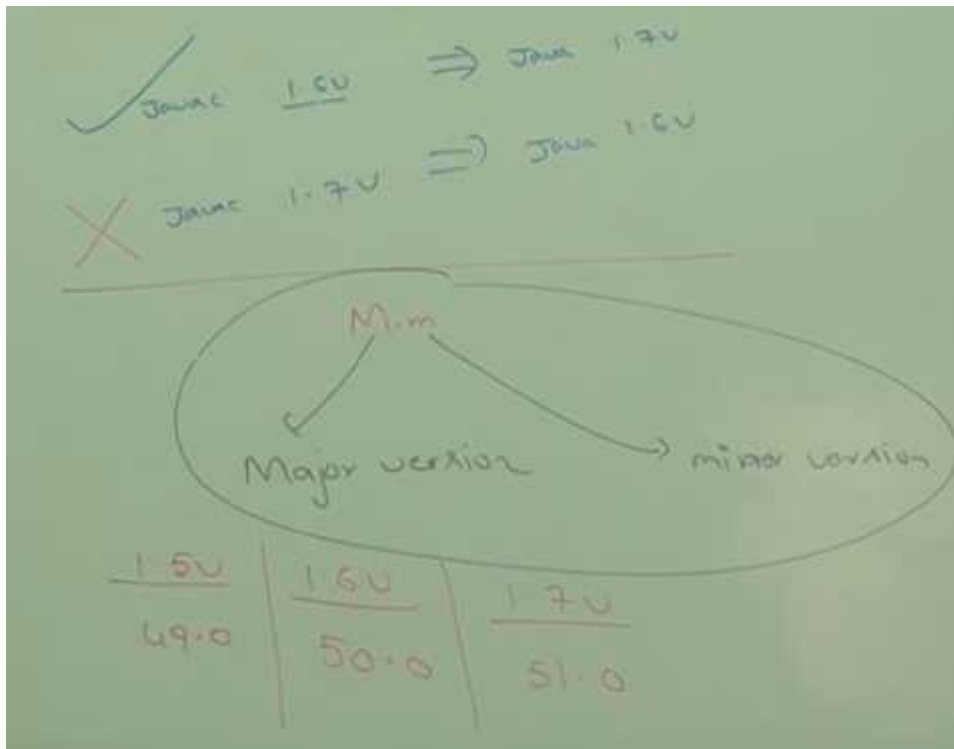
Note:

Whenever we are executing a java class if jvm unable to find valid magic number then we will get runtime exception saying **java.lang.classFormatError: Incompactible magic value.**

minor_version and major_version:

major and minor versions 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 can't be run by lower version JVMs.

if we are trying to run we will get runtime exception saying,

```

java.lang.UnsupportedClassVersionError:
51.0
  loader.defineClass1(Native Method)
  loader.defineClass(ClassLoader)
  
```

Handwritten notes and diagram illustrating Java version compatibility:

- ✓ Java 1.6U \Rightarrow Java 1.7U
- ✗ Java 1.7U \Rightarrow Java 1.6U

Handwritten error message:

RE: UnsupportedClassVersionError

`constant_pool_count:`

It represents number of constants present in constant pool.

`constant_pool[]:`

It represents information about constants present in constant pool.

`access_flags:`

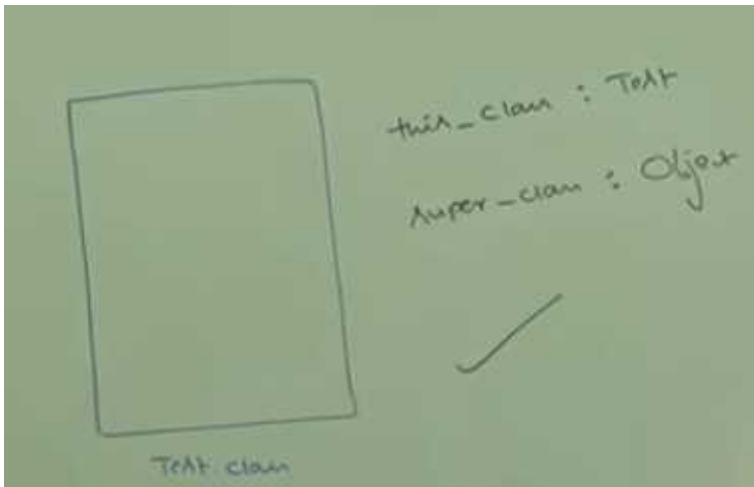
It provides information about modifiers which are declared to the class

`this_class:`

It represents fully qualified name of the class.

`super_class:`

It represents fully qualified name of immediate super class of current class.



`Interface_count:`

it returns number of interfaces implemented by current class.

`interface[]:`

it returns interfaces information implemented by current class.

`fields_count:`

It represents number of field present in the current class(static variables)

`fields[]:`

It represents fields info present in current class.

methods_count:

It represents number of methods preset in current class.

methods[]:

It provides information about all methods present in current class

attributes_count:

it returns number of attributes present in current class

attributes[]:

It provides information about all attributes present in current class.(instance variables)

