JVM Architecture

Date
Page

1. Virtual machine
2. Type of virtual machine
   - Hardware Based VM
   - Application Based VM
3. Basic Architecture of JVM
4. Class Loader subsystem
   - Loading
   - Linking
   - Initialization
5. Types of classloaders
   - Bootstrap class Loader
   - Extension class Loader
   - Application class Loader
6. How classloader works
7. What is the need of customized class loader
8. pseudo code for customized class loader
9. Various Memory Areas of JVM
   - Method Area
   - Heap Area
   - Stack Area
   - PC Registers
   - Native Method stacks
10. program to display heap memory stats.
11. How to set min & max heap size.

(12) Execution Engine
- Interpreter
- JIT compiler

(13) Java Native Interface (JNI)

(14) Complete Architecture Diagram of JVM

(15) Class file structure

(*) Virtual Machine

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

→ There are two types of VM
(1) Hardware Based / System Based
(2) Application Based / process Based

→ **Hardware Based VM :-**

- It provides several logical systems on the same computer with the strong isolation from each other ie on one physical machine, we are defining multiple logical machines

- The main advantage of H/w Based VM is Hardware Resource sharing & improve utilization of H/w resources.

ex. KVM [kernel based VM] for linux systems

- VMware, Xen, Cloud computing etc.

→ **Application Based VM :-**

- This VM acts as Runtime engines to run a particular programming language applications

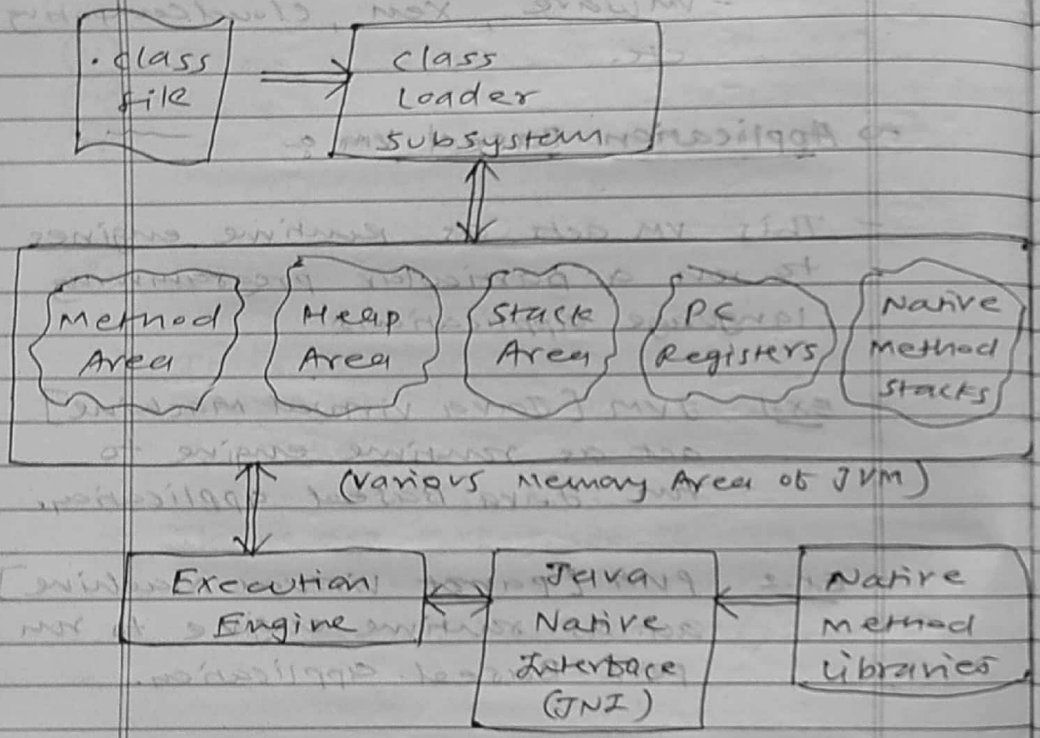
- Ex.1 JVM [Java virtual Machine] act as runtime engine to run Java Based applications.

Ex.2 PVM [parrot virtual Machine] act as runtime engine to run perl Based application.

Ex.3 CLR [common language Runtime] act as runtime engine to run .NET Based application.

- JVM is a part of JRE & it is responsible to load & 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 & store corresponding Binary data into ~~memory~~ Method Area.
- For each .class file, JVM will store corresponding info^{n} in Method Area.

1) Fully Qualified name of class.
2) Fully Qualified name of immediate parent class.
3) Methods Info^{n}
4) Variables Info^{n}
5) constructor Info^{n}
6) Modifiers Info^{n}
7) Constant Pool Info^{n}

:
etc.

- After loading .class file, immediately JVM will creates an Object for that loaded class on the Heap Memory Area of type java.lang.Class.

* The class Class object can be used by programmer to get class level info" like methods, variable info" etc.

Ex:

```
import java.lang.reflect.*;
class Student
{
  public String getName()
  {
    return null;
  }
  public int getMarks()
  {
  }
}
class Test
{
  public static void main(String[] args)
  throws Exception
  {
    int count = 0;
    Class c = Class.forName
              ("Student");
    Method[] m =
      c.getDeclaredMethods();
```

```
for(Method m:m)
{
    count++;
    S.O.P(m.getName());
}
S.O.P("Number of methods:"+
                count);
}
```

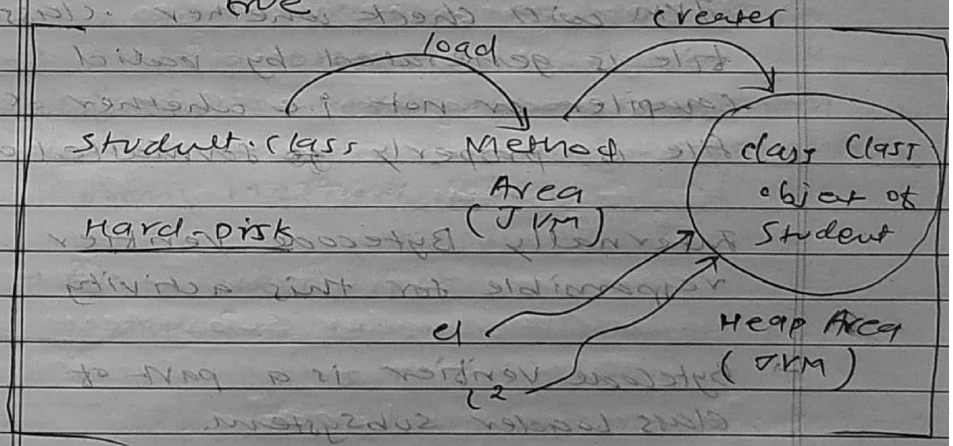
O/P : getName
getMarks
Number of methods : 2

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

Ex:

```
class Test
{
    P S V M(Sting
{
    Student S1 = new Student();
    Class c1 = s1.getClass();
    Student s2 = new Student();
    Class c2 = s2.getClass();
    S.O.P(c1.hashCode());
    S.O.P(c2.hashCode());
    S.O.P(c1==c2);
}
}
```

O/P : -26117675
-26117675
true



- In the above program, even though we are using Student class multiple times, only one Class class object got created.

Linking

- Linking consist of three activities
  1) verify (Bytecode Verifier)
  2) prepare
  3) Resolve

- **Verify :-**

→ It is a process of ensuring that binary representation of a class is structurally correct or not i.e JVM will check whether .class file is 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 a part of Class Loader subsystem.

→ If verification fails then we will get Runtime Exception saying java.lang.VerifyError.

- **Prepare :-**

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

Note :- In Initialization phase, original values will be assigned to the static variable & here only default values will be assigned

- **Resolution :-**

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

→ Ex.

```
class Test
{
    p s v main(S—)
    {
        String s = new String("durga");
        Student st = new Student();
    }
}
```

→ For the above class, classLoader loads Test.class, String.class, Student.class & Object.class

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

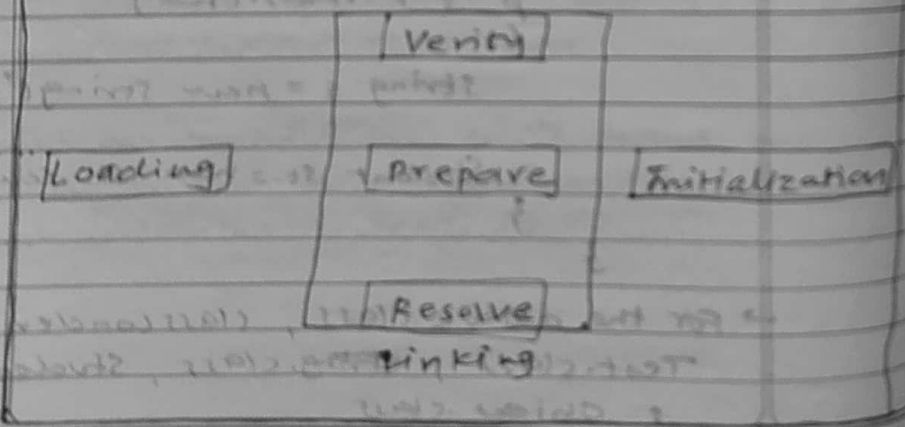
→ In this phase, these names are replaced with original memory level references from method area.

Test → 0A9E1D

→ Initialization :

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

Class Loader SubSystem



Class Loading Process.

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

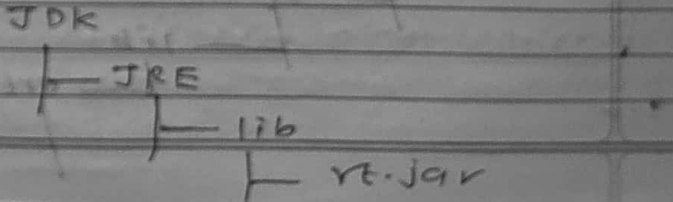
✱ Types of Class Loaders

→ Class Loader subsystem contains the following 3 types of class loaders :

1) BootStrap Class Loader / Premordial Class Loader
2) Extension Class Loader
3) Application Class Loader / System Class Loader

→ Bootstrap Class Loader :

→ Bootstrap class loader is responsible to load 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 to load classes from "Bootstrap classpath."

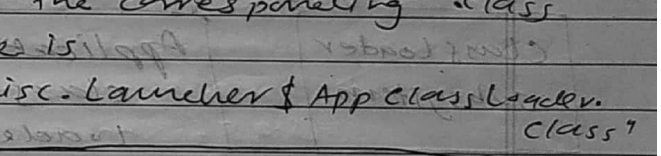
→ Bootstrap class loader is by default available with every JVM.

→ It is implemented in native languages like C/C++ & not Implemented in Java.

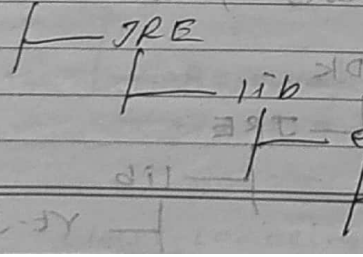
→ Extension class loader :-

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

→ Extension class loader is responsible to load classes from "Extension classpath" i.e



JDK / JRE / lib / ext



→ Extension class loader is implemented in java & the corresponding .class file is "sun.misc.Launcher$ExtClassLoader.class".

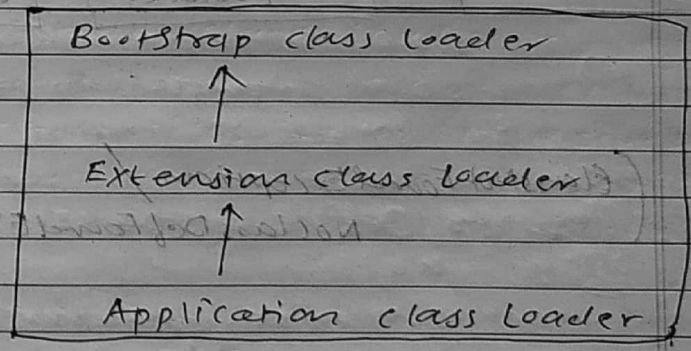
→ Application class loader :-

→ Application class loader is a child class of Extension class loader.

→ This class loader is responsible to load classes from "Application classpath".

→ It internally uses "environment variable classpath."

→ Application class loader is implemented in java & the corresponding .class file name is "sun.misc.Launcher$AppClassLoader.class"



Extension CP. If it is available then it will be loaded by Extension CL, otherwise Extension CL delegates the request to Application CL.

→ Application CL will search in Application CP, If it is available then it will be loaded by Application CL otherwise we will get R.E. NoClassDefFoundError / ClassNotFoundException.

→ Ex:-

```
class Test
{
    public static void main(String[] sm)
    {
        S.O.P(String.class.getClassLoader());
        S.O.P(Test.class.getClassLoader());
        S.O.P(customer.class.getClassLoader());
    }
}
```

O/P :- null
sun.misc.Launcher$AppClassLoader@123406
sun.misc.Launcher$ExtClassLoader@52186

Note :-
We got null first up because BootstrapClassLoader is not written in java,

Assume → We have put customer.class in ext folder also, so Extension CL will get priority.

Need of Customized Class Loader

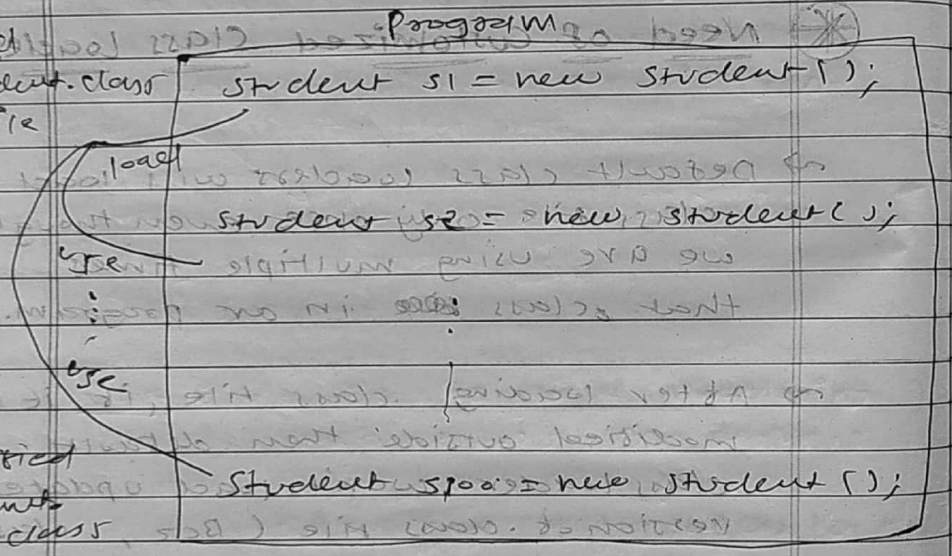
→ Default class loaders will load .class file only once even though we are using multiple times that class in an program.

→ After loading .class file, if it is modified outside then default class loaders won't load updated version of .class file (Bcz, .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.

For example, we can load .class file separately every time, so that updated version available to our program.

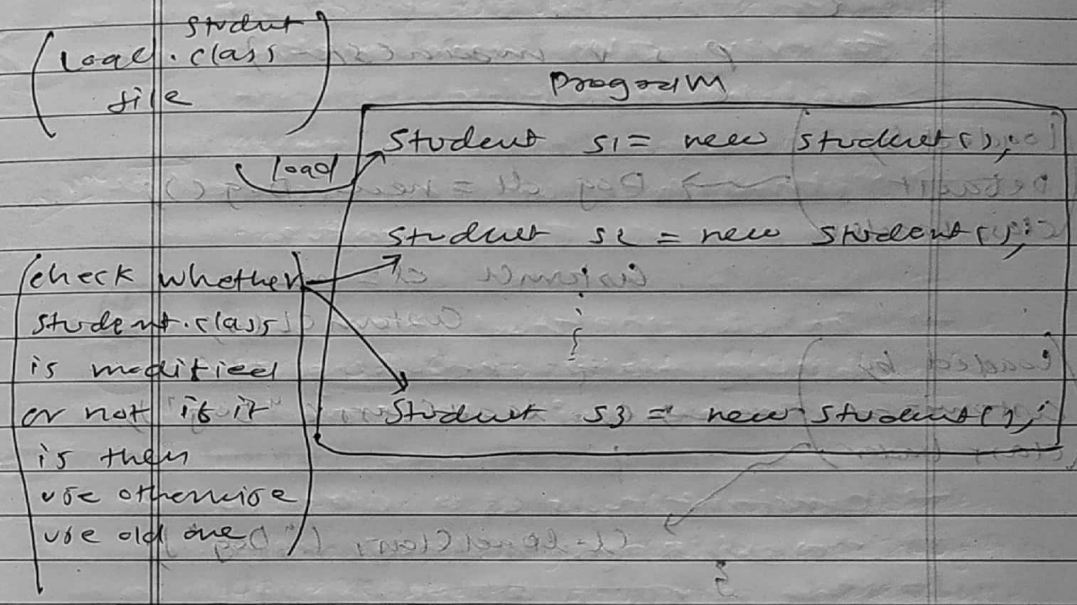
Default class loading



(*) How to define customized class loader

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

Customized class loading



Ex: public class CustomCL extends ClassLoader
{
@Override
public Class loadClass(String name)
throws ClassNotFoundException
{
// check for updates & load updated .class file & returns corresponding class
}
}

```
class Client
{
    p s v main(SA)
    {
        Dog d1 = new Dog();

        CustomCL cl = new CustomCL();
        cl.loadClass("Dog");

        cl.loadClass("Dog");
    }
}
```

(loaded by Default class loader) → `Dog d1 = new Dog();`

(loaded by customized class loader) → `cl.loadClass("Dog");`

Note: While designing / developing web servers / App servers usually we can go for customized class loader to customize class loading mechanism.

Q. What is the need of ClassLoader class?

Ans: 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 act as base class for all customized class loaders.

✱ Various Memory Areas Present Inside JVM

⇝ Whenever JVM loads & Runs a Java program it needs memory to store several things like Bytecode, Objects, variables etc.

⇝ Total JVM memory organized into following five categories:-

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

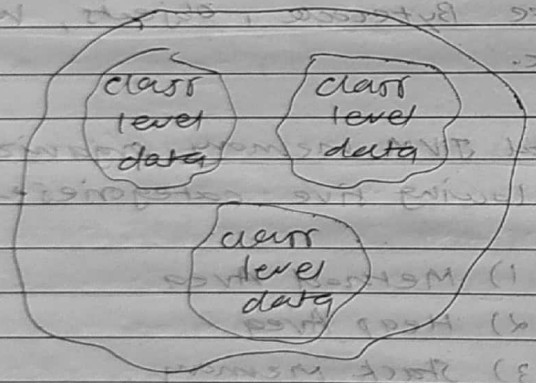
⇒ Method Area :

→ For every JVM, one method area will be available.

→ Method Area will be created at the time of JVM startup.

→ Inside Method Area, class level binary data including static variables will be stored.

→ Constant pools of a class will be stored inside method area.



Method Area

→ Method Area can be accessed by multiple threads simultaneously

→ Method Area need not to be continuous.

⇒ Heap Area :

→ For every JVM, one Heap area is available.

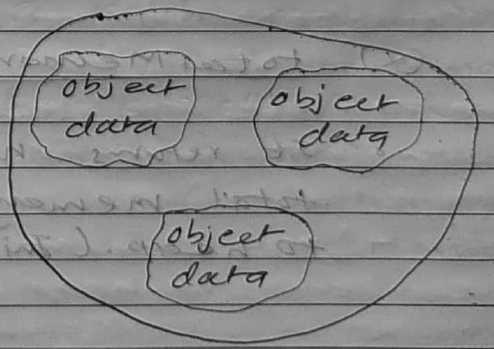
→ Heap Area will be created at the time of JVM startup.

→ Objects & corresponding instance variable will be stored in the Heap Area

→ Every Array in Java is object only, hence Arrays also store in the Heap Area.

→ Heap Area can be accessed by multiple threads & hence the data stored in the heap memory is not thread-safe.

→ Heap Area need not be continuous.



Heap Area

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

→ Runtime class present in java.lang package & it is singleton class.

→ we can create Runtime object as follows:

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

→ Once we got Runtime object, we can call following methods on that object.

(1) maxMemory()

- It returns the no. of bytes of max memory allocated to heap

(2) totalMemory()

- It returns no. of bytes of total memory allocated to heap. (Initial Memory)

(3) freeMemory()

- It returns no. of bytes of free memory present in the heap

Ex.

```
class HeapDemo
{
    public static void main(String)
    {
        double mb = 1024 * 1024;

        Runtime r = Runtime.getRuntime();

        S.O.P("Max Mem: " +
            r.maxMemory()/mb);

        S.O.P("Total Mem: " +
            r.totalMemory()/mb);

        S.O.P("Free Mem: " +
            r.freeMemory()/mb);

        S.O.P("Consumed mem: " +
            (r.totalMemory() -
            r.freeMemory())/mb);
    }
}
```

O/P :- (If in bytes)

Max Mem : 2595225 60
Total Mem : 16252928
Free Mem : 15874448
Consumed Mem : 378480

(Here in MB)

Max mem : 247.5
Total Mem : 15.5
Free Mem : 15.1390
Consumed Mem : 0.36 09

→ How to set Maximum & Minimum Heap sizes ?

- Heap Memory is finite memory, but based on our requirement, we can set maximum & minimum heap sizes i.e we can increase & decrease the heap size based on our requirement.
- we can use the following flags with java command

(1) -Xmx :- To set maximum heap size (Max Memory)

- (java -Xmx512m HeapDemo ↵)
- This command will set max mem heap size as 512 MB.

(2) -Xms :- To set minimum heap size

- (java -Xms64m HeapDemo ↵)
- This command will set minimum heap size to 64 MB (i.e total mem.)

- (java -Xmx512m -Xms64m HeapDemo)

O/P :-

Max Mem : 494.9375
Total Mem : 61.875
Free Mem : 61.1921768
Consumed Mem : 0.682823

Stack Memory Area :-

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

→ Each & every method call performed by that thread will be stored in the

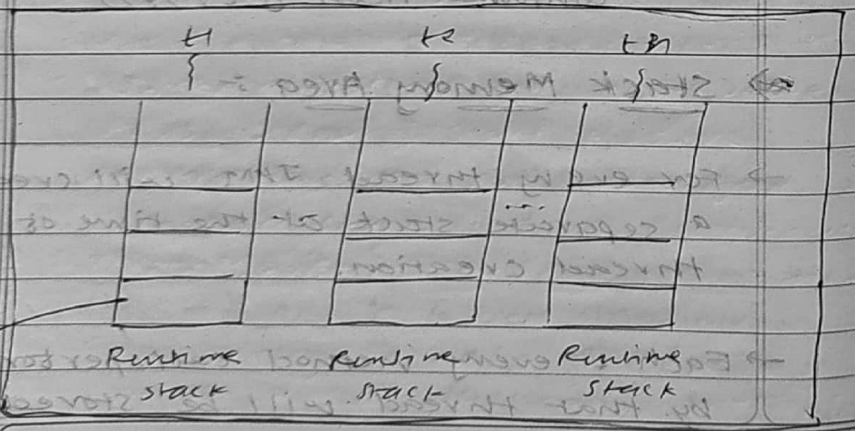
stack, including local variables also.

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

→ After completing all method calls the stack will became empty & that empty stack will be destroyed by JVM just before terminating the thread.

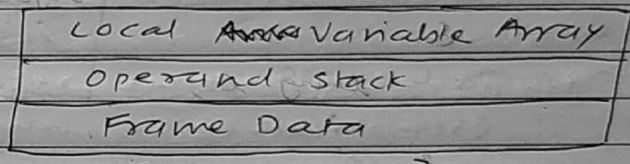
→ Each entry in the stack is called "Stack Frame / Activation Record".

→ The data stored in the stack is available only for the corresponding thread & not available to the remaining thread, hence this data is thread-safe



→ Stack Frame structure :-

→ Each stack Frame contains 3 parts.



(Stack Frame)

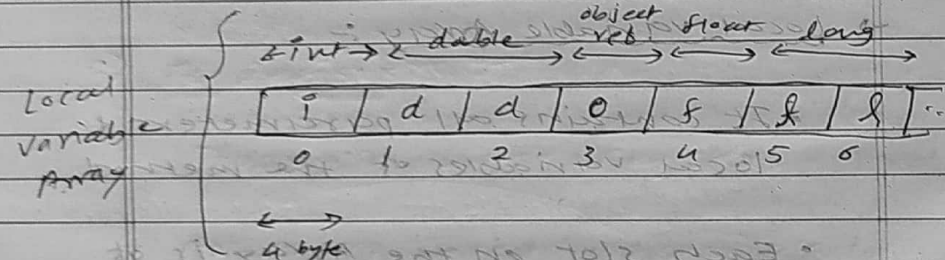
→ Local variable Array :-

- It contains all parameters & local variables of the method
- Each slot in the Array is of 4-bytes.
- Values of type - int, float & reference occupy 1 entry (1 slot in the Array.
- Values of type - double & long occupy 2 entry / 2 slots (consecutive)
- Values of type - byte, short & char will be converted to int type before storing & occupy 1 slot. but, the way of storing boolean values is varied from JVM to JVM

but most of the JVM follow 1 slot.

- ex.

```
public void my(int i, double d,
               object o, float f)
{
    long l;
    :
}
```



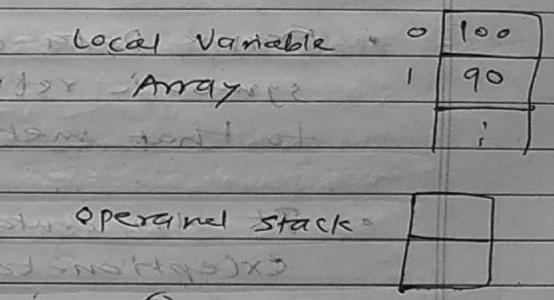
→ Operand stack :-

- JVM uses operand stack as workspace.
- some instructions can push values to the operand stack & some instruction can pop values from operand stack & some instructions can perform required operations.

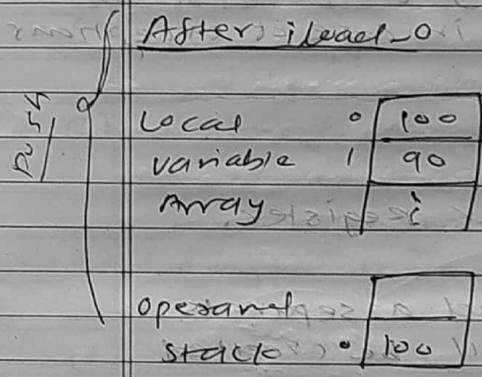
Program

```
iload_0
iload_1
iadd
istore_2
```

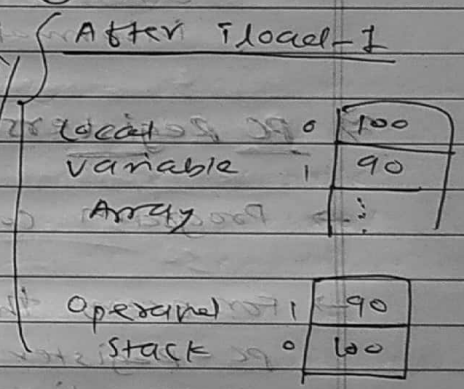
① Before starting



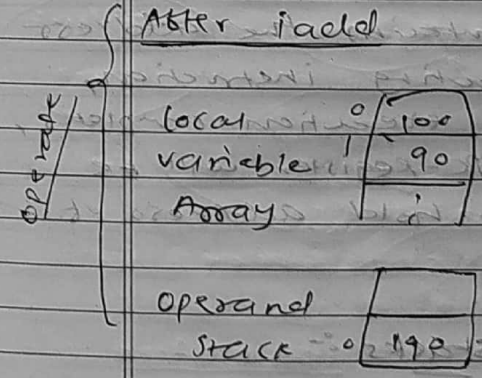
② After iload_0



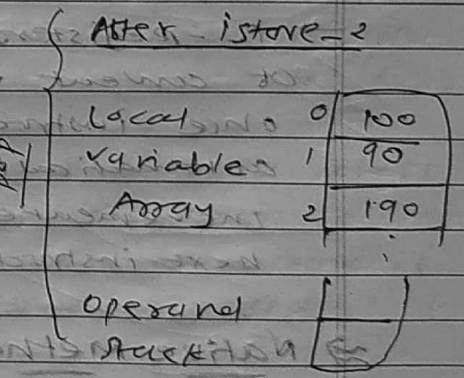
③ After iload_1



④ After iadd



⑤ After istore_2



→ **Frame Data :-**

- Frame Data containing all symbolic references related to that method.
- It also contains reference to exception-table, which provides corresponding catch block information in case of exceptions.

⇒ **PC Registers :-**

→ Program counter register.

→ For every thread a separate PC-Register will be created at the time of thread creation.

→ PC-Registers containing 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 :-**

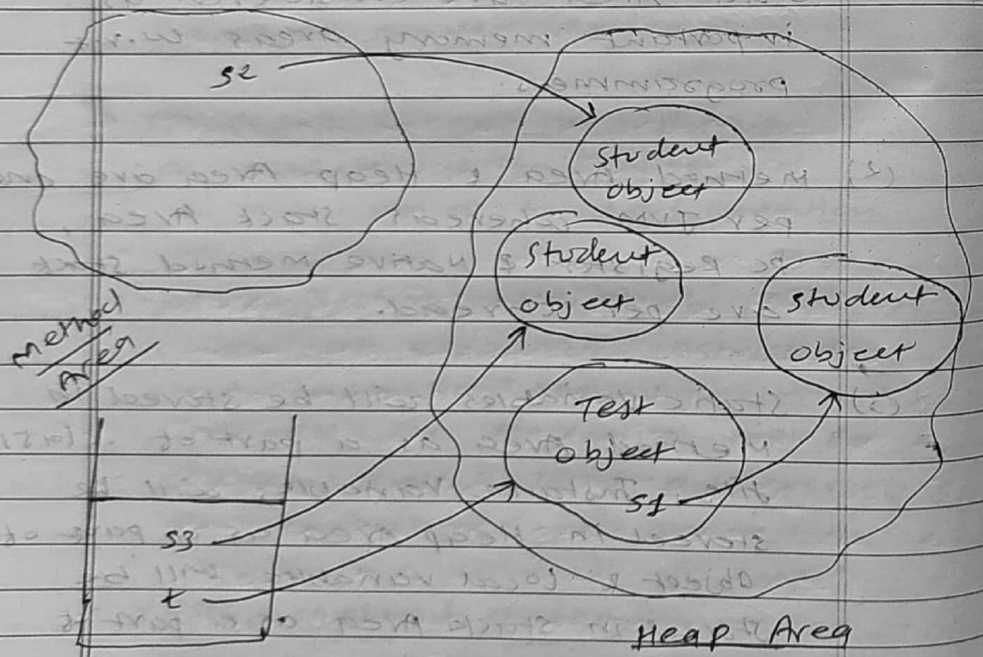
(1) Method Area, Heap Area & stack Area are considered as important memory areas w.r.t programmers.

(2) Method Area & Heap Area are one per JVM whereas stack Area, PC Register & Native method stack are per thread.

(3) Static variables will be stored in Method Area as a part of .class file, Instance variables will be stored in Heap Area as a part of Object & Local variables will be stored in stack Area as a part of Local variable Array.

Ex.

```
class Test
{
    Student S1 = new Student();
    static Student S2 = new Student();
    public static void main (String )
    {
        Student S3 = new Student();
        Test t = new Test();
    }
}
```



Runtime stack for main-Thread

Execution Engine

⇒ This is a central component of JVM

⇒ Execution Engine is responsible to execute java class file.

⇒ Execution Engine mainly contains 2 components :

1) Interpreter
2) JIT compiler

⇒ Interpreter :-

→ It is responsible to read byte-code & interpret into machine (native) code & execute that m/c 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 introduce JIT compiler in 1.1 version.

⇒ JIT Compiler :-

→ The main purpose of JIT compiler is to improve performance.

→ Internally JIT compiler maintains a separate counter for every method.

→ whenever JVM came across any method call, first that method will be interpreted normally by the interpreter & JIT compiler increments corresponding count variable

→ This process will be continued for every method.

→ Once if any method count threshold value then JIT compiler identifies that, that method is repeatedly used method (Hot-spot).

→ Immediately JIT compiler compiles that method & generates native code.

→ Next time JVM came across that method call then JVM uses that native code directly & executes it instead of interpreting once again so that performance will improved.

→ The threshold count varies from JVM to JVM.

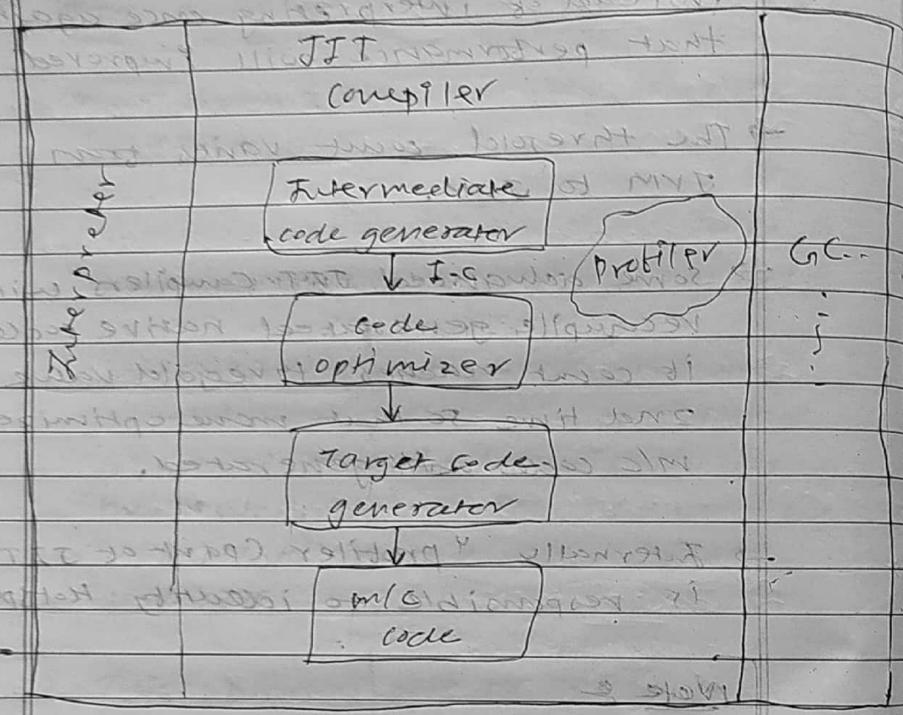
→ Some advanced JIT compilers will recompile generated native code if count reaches threshold value 2nd time so that more optimized m/c code will generated.

→ Internally 'profiler (part of JIT)' is responsible to identify Hotspots.

Note :-

→ JVM interprets total program atleast once.

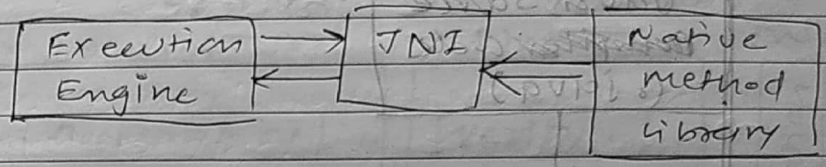
→ JIT compilation is applicable only for repeatedly required method, not for every methods.



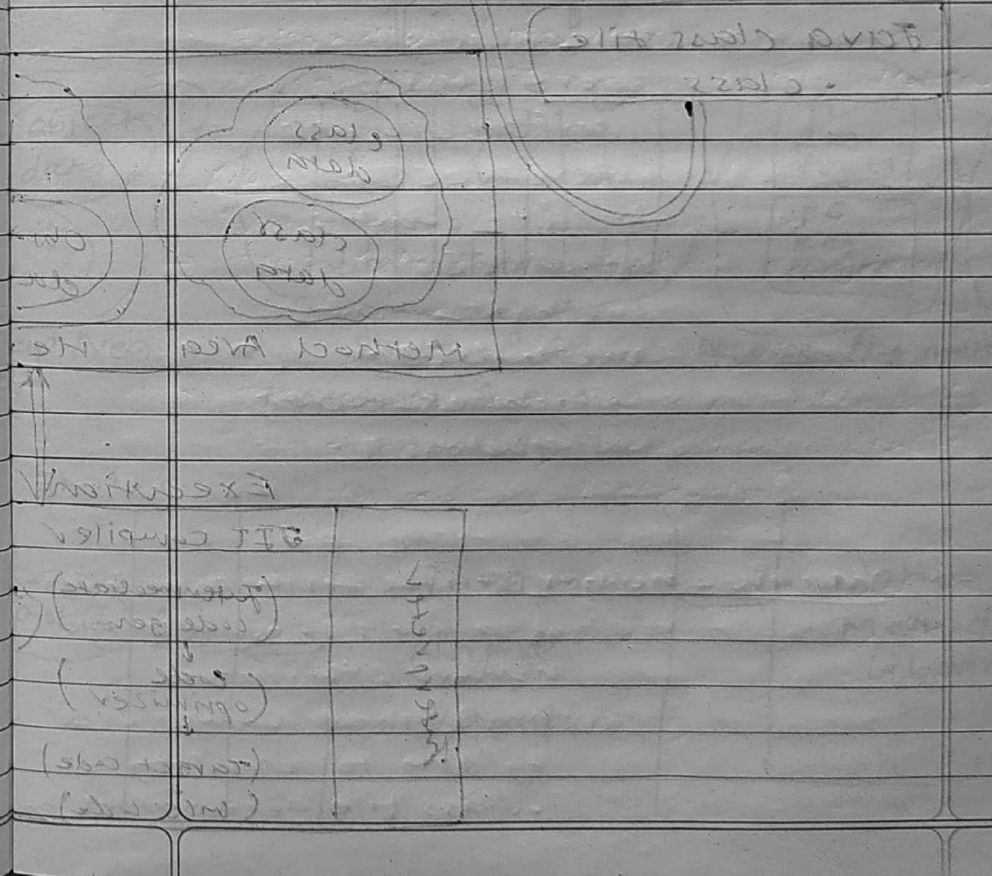
Execution Engine

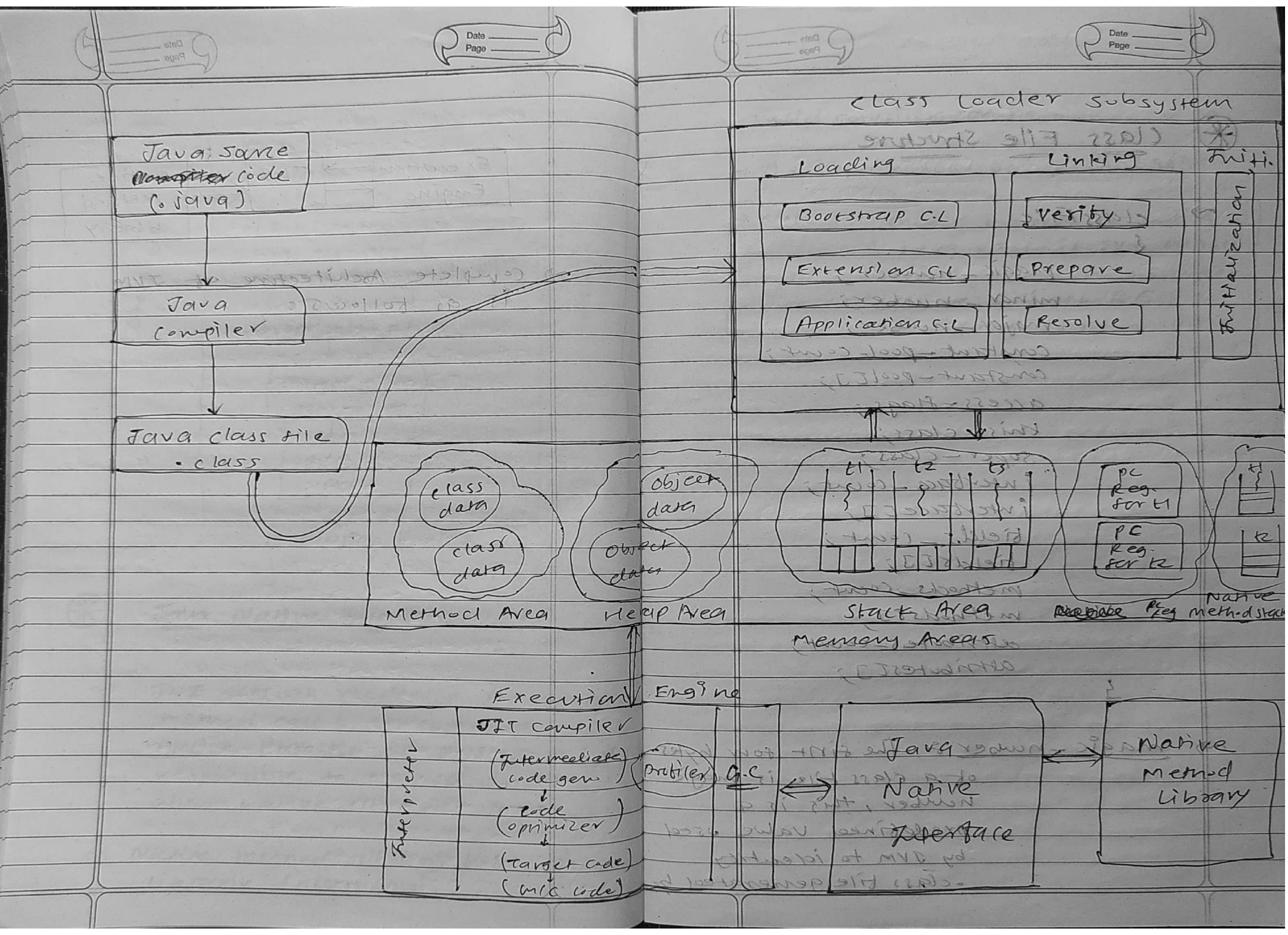
Java Native Interface (JNI)

- JNI acts as mediator for Java method call & corresponding native libraries i.e JNI is responsible to provide info about native libraries to the JVM.
- Native method library holds native library information.



- Complete Architecture of JVM is as follows:




Java source code (.java)
Java compiler
Java class file .class
class loader subsystem
Loading
Linking
Init.
Bootstrap C.L
Extension C.L
Application C.L
Verify
Prepare
Resolve
Initialization
class data
class data
object data
object data
Method Area
Heap Area
t1
t2
t3
PC Reg. for t1
PC Reg. for t2
Native method stack
Stack Area
Memory Areas
Execution Engine
Interpreter
JIT compiler
(Intermediate code gen.)
(code optimizer)
(Target code)
(m/c code)
Profiler
G.C
Java Native Interface
Native Method Library

(*) Class File Structure

→ classFile
{
magic_number;
minor_number;
major_number;
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 four bytes of a class file is magic number, this is a predefined value used by JVM to identify .class file generated by valid compiler or not.

→ The value should be OXCAFEBABE

→ Note :- whenever we are executing a java class it JVM unable to find valid magic number then we will get R.E ClassFormatError