

Java Virtual Machine (JVM)

This chapter includes the following topics.

- Virtual Machine and Types of Virtual Machine
- Class Loader Subsystem
- Types of Class Loaders
- How Class Loader Works?
- What is the need of Customized Class Loader
- Pseudo Code for Customized Class Loader
- Various Memory Areas of JVM
- Program to display Heap Memory Statistics
- How to set Maximum and Minimum Heap Size?
- Execution Engine
- Java Native Interface (JNI)
- Complete Architecture Diagram of JVM
- Class File Structure

Virtual Machine

- It is a software simulation of a Machine which can perform operation just like a physical machine.
- There are two types of Virtual Machines:
 1. Hardware Based | System Based Virtual Machine
 2. Application Based | Process Based Virtual Machine

Hardware Based or System Based Virtual Machine

- It provides several logical systems on the same computer with the strong isolation from each other. i.e., On one physical machine we are defining multiple logical machines.
- The main advantage of Hardware based virtual machine is Hardware resource sharing and improves utilisation of hardware resources.
- Example – KVM (Kernal Based Virtual Machine) for Linux, VMware for Cloud Computing etc.

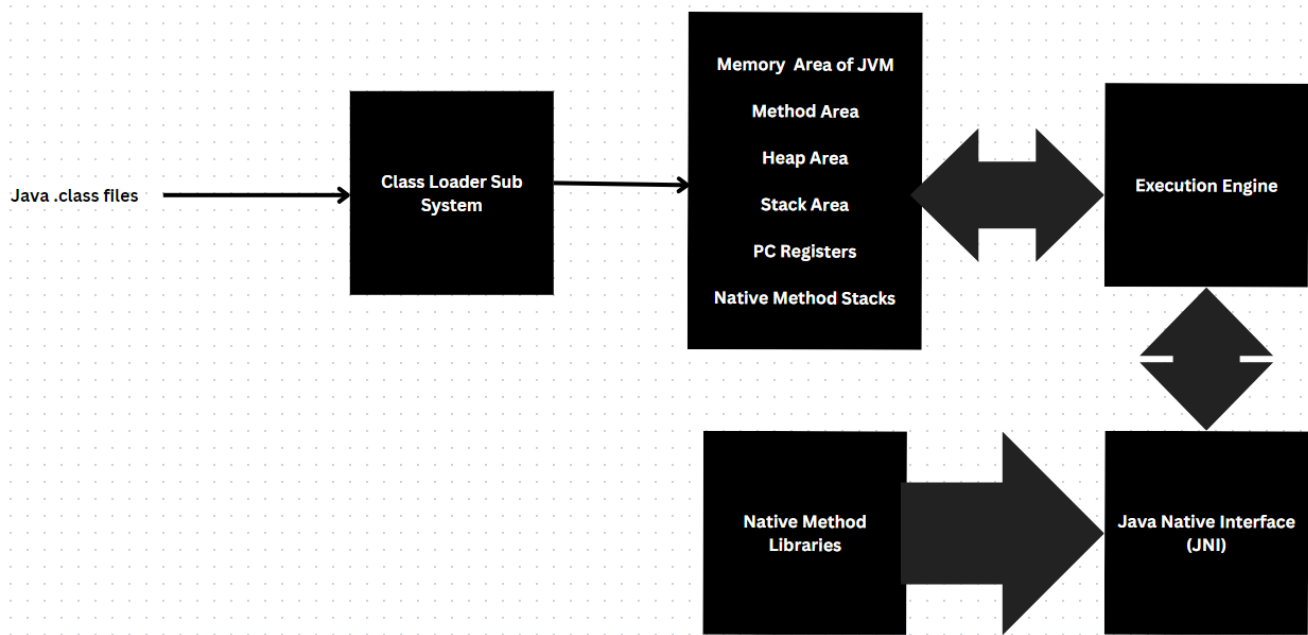
Application Based or Process Based Virtual Machine

- These virtual machines act as run time engines to run a particular programming language application.
- Example – JVM (Java Virtual Machine) acts as Run time engine to run Java applications. PVM (Parrot Virtual Machine) acts as Run time engine to run applications written in scripting language like Perl, Python etc. CLR (Common Language Runtime) acts as Run time engine to run .Net based applications.

JVM

- JVM (Java Virtual Machine) acts as Run time engine to run Java applications.
- JVM is a part of JRE and JRE is a part of JDK.
- JVM is responsible to load and run .class files of Java.

Basic Architecture Diagram of JVM



Since main objective of JVM is to load and run java .class files, hence, the input which comes to JVM is .class files of Java.

The very first component of JVM is Class Loader Sub System which is responsible to load java .class files into various memory area which includes (Heap, Stack, PC Registers, Method Area etc.). Execution engine is responsible to take the components from Memory area and execute and produce the results back to the memory area.

If require through JNI various Native Method libraries can be used.

Class Loader Sub System

- Class Loader Sub System is responsible for following three activities:
 1. Loading
 2. Linking
 3. Initialization

Loading

- Loading means reading class file present in Secondary memory (Hard Disk) and store corresponding binary data in First memory area of JVM which is Method Area.
- For each class file, JVM will store following information in the Method area:
 1. Fully Qualified Class Name
 2. Fully Qualified Parent Class Name
 3. Method Information
 4. Variable Information
 5. Constructor Information
 6. Modifiers Information
 7. Constant Pools, etc.
- After loading .class file in the Method Area, immediately JVM Creates an Object for the Loaded Class in the Heap Memory, which is the next memory area after Method Area. The created Object will be of type `java.lang.Class`. i.e., Class Object.

- The class Class Object can be used by programmer to get class level information, like method information, variable information, constructor information etc.

Main.java	Run	Output
<pre> 1- import java.lang.reflect.*; 2 class Student 3- { 4 Student() 5- { 6 System.out.println("Constructor"); 7 } 8 public String getName() 9- { 10 return null; 11 } 12 public int getRollNumber() 13- { 14 return 10; 15 } 16 } 17- class LoadDemo { 18- public static void main(String[] args) throws Exception { 19 int count = 0; 20 Class c = Class.forName("Student"); 21 Method[] m = c.getDeclaredMethods(); 22- for(Method m1:m){ 23 count++; 24 System.out.println(m1.getName()); 25 } 26 System.out.println("Total Methods : " + count); 27 } 28 } </pre>	Run	<pre> java -cp /tmp/HwpeeBjG8x LoadDemo getRollNumber getName Total Methods : 2 </pre>

Main.java	Run	Output
<pre> 1- import java.lang.reflect.*; 2 class Student 3- { 4 Student() 5- { 6 System.out.println("Constructor"); 7 } 8 public String getName() 9- { 10 return null; 11 } 12 public int getRollNumber() 13- { 14 return 10; 15 } 16 } 17- class LoadDemo { 18- public static void main(String[] args) throws Exception { 19 Student s1 = new Student(); 20 Class c1 = s1.getClass(); 21 Student s2 = new Student(); 22 Class c2 = s2.getClass(); 23 System.out.println("C1 hashCode : " + c1.hashCode()); 24 System.out.println("C2 hashCode : " + c2.hashCode()); 25 System.out.println(c1 == c2); 26 } 27 } </pre>	Run	<pre> java -cp /tmp/HwpeeBjG8x LoadDemo Constructor Constructor C1 hashCode : 932583850 C2 hashCode : 932583850 true </pre>

For every loaded type, only one Class object will be created Even though we are using class multiple times in our program. In above program, even though we have used Student class multiple times only one Class Object has been 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 the class generated by valid compiler or not i.e., whether .class file is properly formatted or not.

Internally Byte Code Verifier is responsible for this activity. Byte Code Verifier is the part of Class Loader Sub System.

If verification fails then we will get a run time exception saying java.lang.VerifyError

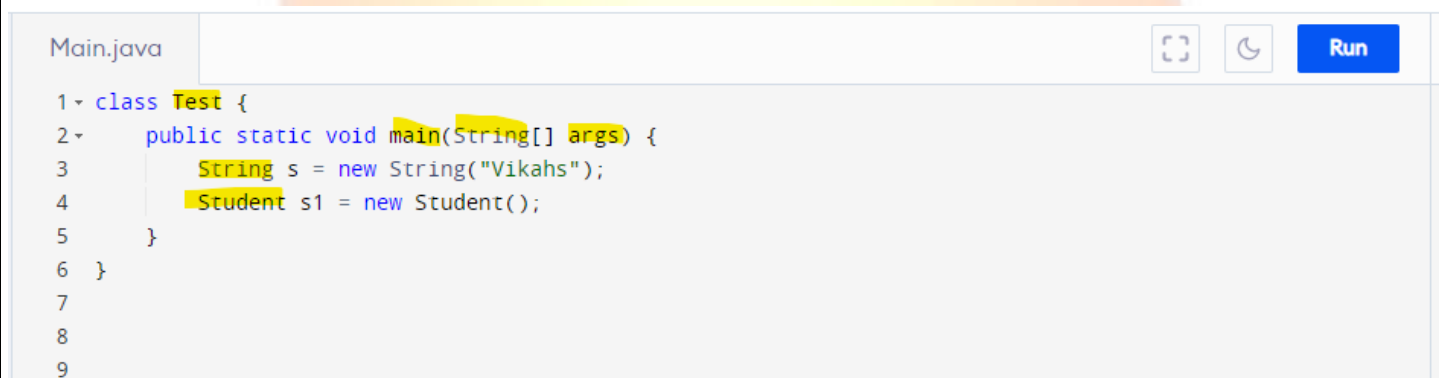
Prepare

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.

Resolve

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

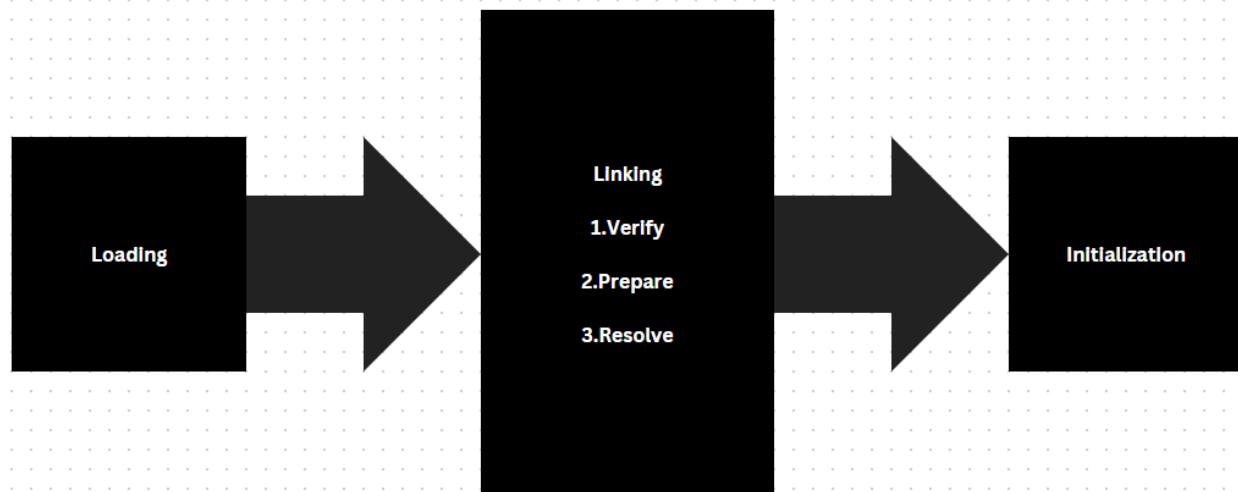


```
Main.java
1 class Test {
2     public static void main(String[] args) {
3         String s = new String("Vikahs");
4         Student s1 = new Student();
5     }
6 }
7
8
9
```

In above program, Class Loader Loads Test.class, Object.class, String.class and Student.class in method area. The names of these classes are stored in Constant Pool of test class. In resolve phase, these names gets replaced with Original memory level reference from method area.

Initialization

- In this process all static variables gets initialized with Original Values.
- Static Blocks gets executed from parent to child and from top to bottom.



During loading, linking and initialization, if any error occurs then we will get run time exception saying `java.lang.LinkageError` which is a parent class of `java.lang.VerifyError`.

Types of Class Loaders

There are three types of class loaders:

1. Bootstrap Class Loader or Primordial Class Loader
2. Extension Class Loader
3. Application Class Loader or System Class Loader

Bootstrap Class Loader or Primordial Class Loader

- Bootstrap Class Loader is responsible to load core java api classes i.e., classes present in `rt.jar` which is present in bootstrap class path which is `jdk/jre/lib/rt.jar` i.e., Bootstrap class loader is responsible for loading all the classes present in Bootstrap class path.
- It is by default available with every JVM.
- It is implemented in native languages like C/C++ and not implemented in Java.

Extension Class Loader

- Extension Class Loader is responsible for loading all the classes present in Extension class path which is `jdk/jre/lib/ext/*.jar`
- Extension Class Loader is the child of Bootstrap Class Loader.
- Extension class Loader is implemented in Java and its corresponding Java class is `sun.misc.Launcher$ExtClassLoader.class`.

Application or System Class Loader

- Application or System Class Loader is the child class of Extension Class Loader which is a child of Bootstrap Class Loader.
- Application or System Class Loader is responsible for loading all the classes present in Application class path which is our environment variable class path.
- Application class Loader is implemented in Java and its corresponding Java class is `sun.misc.Launcher$AppClassLoader.class`.

How the Class Loader Works?

- Class Loader Follows Delegation Hierarchy Principle.
- Whenever JVM will come across any .class file, it will send a request to Class Loader Sub System. If Class Loader sub system found that class is already loaded then it will use the already loaded class if it is not loaded, it will forward the request to Application Class Loader.
- Application Class loader will first send the request to Extension Class Loader and Extension Class Loader will send it to Bootstrap Class Loader.
- Bootstrap Class Loader will first look for the class in Bootstrap Class Path, if found it will load the class and if class is not found, it will send the request to Extension Class Loader.
- Extension Class Loader will then look for the class in Extension Class Path, if found it will load the class and if class is not found, it will send the request to Application Class Loader.
- Application Class Loader will first look for the class in Application configured Class Path, if found it will load the class and if class is not found, it will throw run time exception saying `ClassNotFoundException` or **NoClassDefFoundError**.
- So, here highest priority is with Bootstrap Class Loader, then with Extension Class Loader and then with Application Class Loader.

Need of Customized Class Loader

- Default Class Loaders will loads .class files only once even though we use that class Multiple times in our program.
- After Loading .class files if it is modified outside then default class loader won't load an updated version of class file because .class file is 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.

Suppose I have a Program in which I am using Student class 100 times. So in default class loading Student class will be loaded only once whenever it encounter Student class for the very first time and then it will use the same class for the remaining 99 encounters of Student class and doesn't care if .class file got modified or not.

If We are using Customized Class Loader here, it will Load Student class whenever it encounter Student class for the very first time and then from next encounter onwards, it will check if .class file is modified or not. If it is modified, it will use latest .class file or else will use the same .class file.

Pseudo Code for Customized Class Loader

```
Main.java ⌵ ⌵ Run  
1 class CustomizedClassLoader extends ClassLoader  
2 {  
3     public Class loadClass(String className) throws ClassNotFoundException  
4     {  
5         /*  
6         Check for Updates and Loads Updated .class Files and  
7         returns Corressponding Class  
8         */  
9     }  
10 }
```


- Every Customized Class Loader which we create is a Child of ClassLoader class present in java.lang package.
- We have to Override loadClass() method for our custom Logic which will return Class type Object and Throws ClassNotFoundException.

```
class Client
{
    public static void main(String[] args)
    {
        Dog d1 = new Dog(); //Loaded by Default class Loader
        CustomizedClassLoader c1 = new CustomizedClassLoader();
        c1.loadClass("Dog"); //Loaded by Customized Class Loader
        .....
        c1.loadClass("Dog");
    }
}
```

Various Memory Areas of JVM

Whenever JVM loads and runs a Java Program it needs memory to store several information like Bytecode, Variables, Methods etc.

Total JVM Memory is Organized into following 5 Categories:

1. Method Area
2. Heap Area
3. Stack Area
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 in a Method Area.
- Method Area can be accessed by Multiple threads Simultaneously.

Heap Area

- For Every JVM, one Heap area will be 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 Simultaneously and hence data stored in the Heap Memory is not Thread-Safe.
- Heap Area need not be continuous.

Program to display Heap Memory Statistics

A Java application can communicate with JVM by using Runtime class Object. Runtime class is present in java.lang package and it is a singleton class. We can create Runtime Object as Follows:

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

Once we gets Runtime Object we can call the following methods on that object:

1. `maxMemory()` : It returns number of bytes of max memory allocated to the Heap.
2. `totalMemory()` : It returns number of bytes of total memory allocated to the Heap.
3. `freeMemory()` : It returns number of bytes of free memory present in the Heap.

Main.java	Run	Output
<pre>1- class HeapStatistics { 2- public static void main(String[] args) { 3 Runtime r = Runtime.getRuntime(); 4 System.out.println("Maximum Memory Allocated to Heap : " + r.maxMemory()); 5 System.out.println("Total Memory Allocated to Heap : " + r.totalMemory()); 6 System.out.println("Free Memory Present in Heap : " + r.freeMemory()); 7 System.out.println("Consumed Memory Present in Heap : " + (r.totalMemory() - r.freeMemory())); 8 } 9 }</pre>		<pre>java -cp /tmp/gWFTCNlKBS HeapStatistics Maximum Memory Allocated to Heap : 243269632 Total Memory Allocated to Heap : 16252928 Free Memory Present in Heap : 14894944 Consumed Memory Present in Heap : 1357984</pre>

Main.java	Run	Output
<pre>1- class HeapStatistics { 2- public static void main(String[] args) { 3 long mb = 1024 * 1024; 4 Runtime r = Runtime.getRuntime(); 5 System.out.println("Maximum Memory Allocated to Heap : " + r.maxMemory()/mb); 6 System.out.println("Total Memory Allocated to Heap : " + r.totalMemory()/mb); 7 System.out.println("Free Memory Present in Heap : " + r.freeMemory()/mb); 8 System.out.println("Consumed Memory Present in Heap : " + (r.totalMemory() - r.freeMemory())/mb); 9 } 10 } 11</pre>		<pre>java -cp /tmp/gWFTCNlKBS HeapStatistics Maximum Memory Allocated to Heap : 232 Total Memory Allocated to Heap : 15 Free Memory Present in Heap : 14 Consumed Memory Present in Heap : 1</pre>

How to set Maximum and Minimum Heap Size?

While running Java Program we can pass an option to set Maximum and Minimum Heap Size.

Java -Xmx512m <ClassName> - Sets Maximum Heap Size to 512 MB

Java -Xms64m <ClassName> - Sets Minimum Heap Size to 64 MB

We can use both together as:

Java -Xmx512m -Xms54m <ClassName>

Stack Area

- 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 only for the corresponding Thread and not available to the remaining Threads. Hence, this data is a Thread Safe.

Stack Frame or Activation Record Structure

Each Stack Frame or Activation Record consists of Three Parts:

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

Local Variable Array

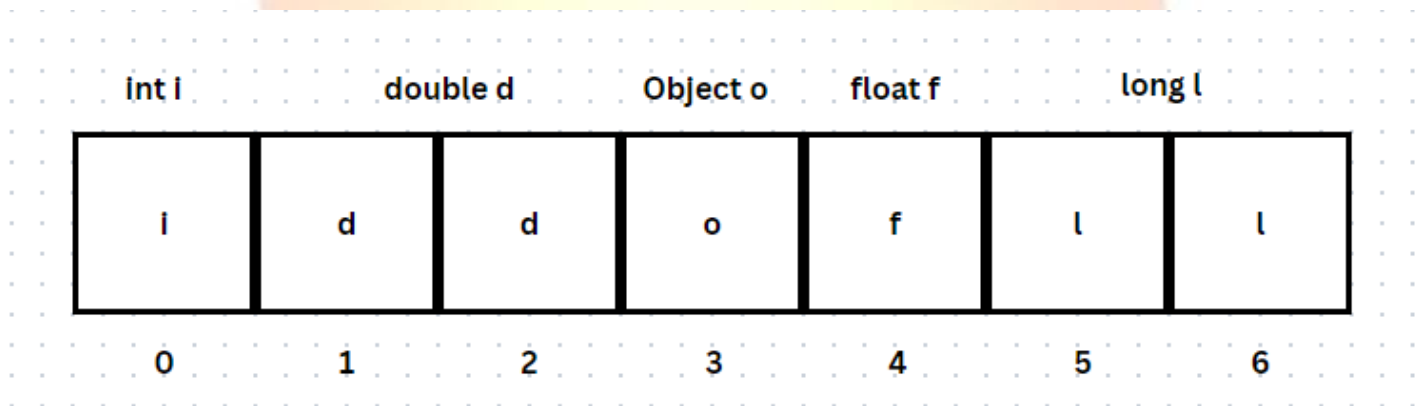
It contains all parameters and local variables of the method. Each slot in the array is of 4 bytes. Values of Type int, float and reference occupy 1 slot of Local Variable Array. Variables of types long and double will occupy 2 slots of Local Variable Array. Variables of types byte, short and char whose size is less than 4 bytes will be promoted to int type of 4 bytes and occupy 1 slot each. Boolean value we can't predict as it will vary from JVM to JVM.

Example:

Consider a method definition as below:

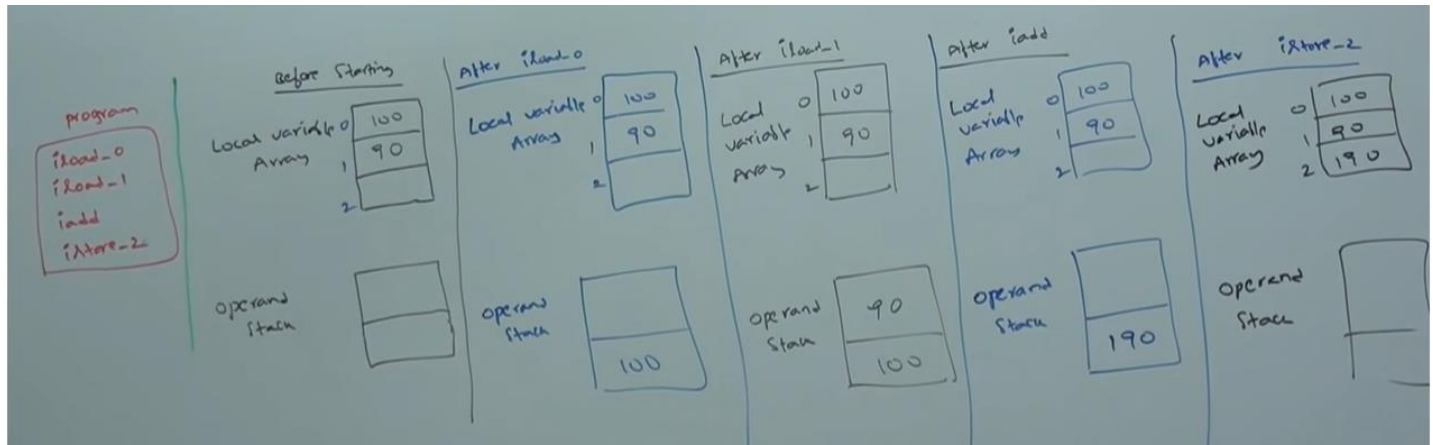
```
Main.java
1 public void m1(int i, double d, Object o, float f)
2 {
3     long x;
4     ;
5     ;
6 }
```

Local Variable Array for this method will be:



Operand Stack

JVM uses Operand Stack as Workspace. Some Instructions can push the values to Operand Stack and some instructions 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 reference to exception table which provides corresponding catch block information for the exceptions.

PC Registers

- PC Registers stands for Program Counter Register.
- PC Registers holds the address of current executing instruction and when the instruction got completed PC registers will be incremented to hold the next instruction.
- For every thread a separate PC register will be created at the time of thread creation.

Native Method Stacks

- For every thread JVM will create separate Native Method Stacks.
- All native method calls invoked by the thread will be stored in the corresponding native method stacks.

Execution Engine

- This is the central component of JVM.
- Execution Engine is mainly responsible for executing our Java class files.
- Execution Engine is mainly consisting of two parts:
 1. Interpreter
 2. JIT Compiler

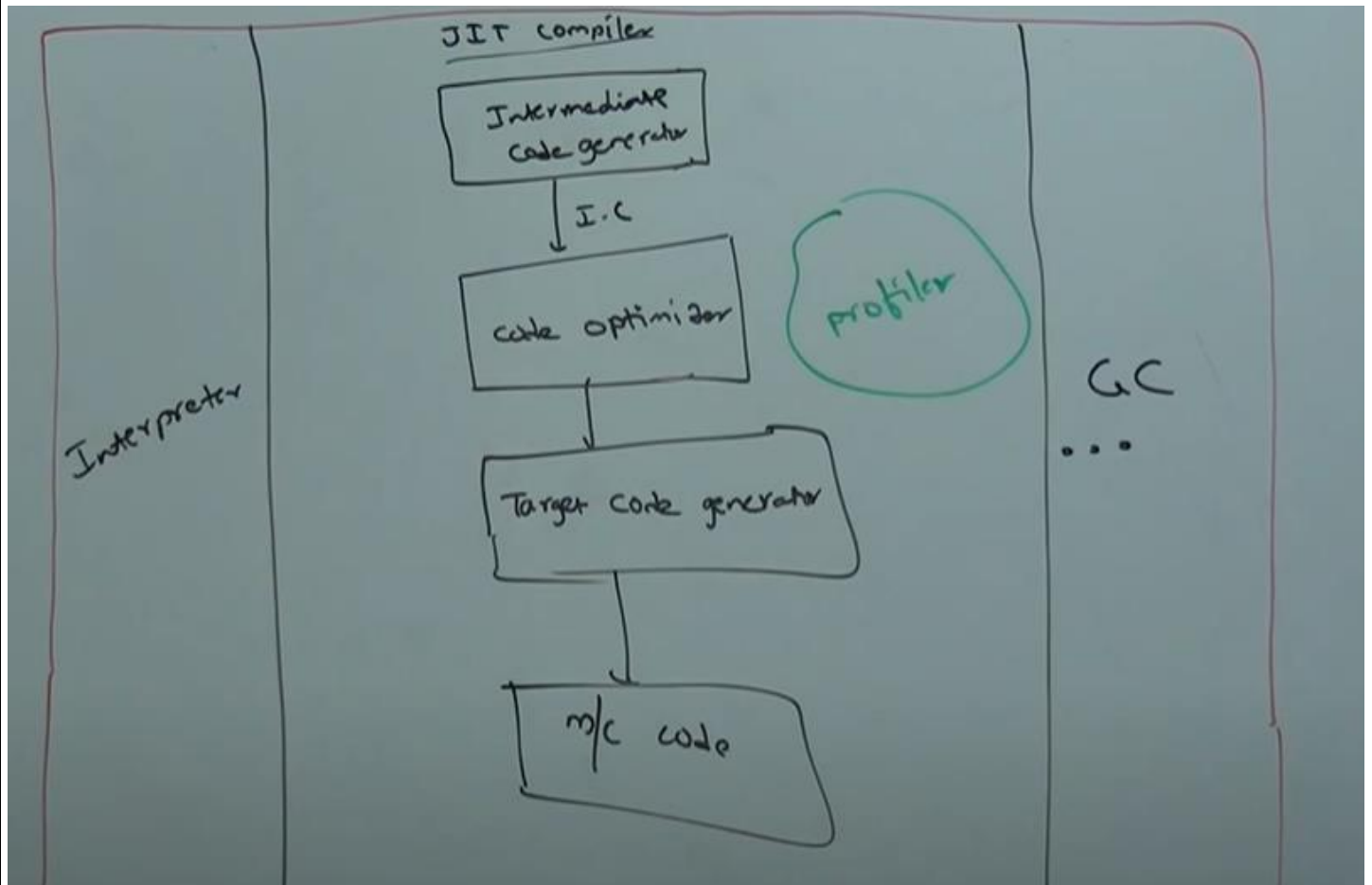
Interpreter

- Interpreter is responsible for reading byte code and interpret it into machine code also known as native code and execute that machine code.
- Executions will happen line by line.
- But there is a problem with interpreter. If there is a method say m1() containing 100 lines of code and if we call m1() 10 times then each time m1() is called, interpreter is going to interpret and execute its 100 lines, which will degrade the performance of the system.
- To Overcome this problem JIT Compiler is introduced In 1.1 version.

JIT Compiler

- The main purpose of JIT Compiler is to improve performance of the system.
- Internally, JIT Compiler maintains a separate count for every method. Whenever JVM come across any method call, first that method will be interpreted normally by the Interpreter and JIT compiler increments the corresponding count variable.
- This process will be continue for every method.

- Once if any method count reaches threshold value then JIT compiler identifies that, that method is a repeatedly used method (Hot Spot). Immediately JIT Compiler compiles that method and generates the corresponding native Code.
- Next time whenever JVM comes 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.
- Threshold count vary from JVM to JVM.
- Some advanced JIT compilers will recompile generated native code if count reaches threshold value second time so that more optimized machine code will be generated.
- Internal Profiles, which is the part of JIT Compiler, is responsible for identifying Hot Spots.



Execution Engine

Java Native Interface (JNI)

- JNI Acts as a mediator for Java Method calls and corresponding native libraries. i.e., JNI is responsible for providing information from the Native Method Libraries to the JVM.
- Native Method Libraries hold Native Libraries Information.