**1-Virtual Machine**

-It is a Software Simulation of a Machine which can Perform Operations Like a Physical Machine.

**2-Types of Virtual Machines**

-There are 2 Types of Virtual Machines

1) Hardware Based OR System Based Virtual Machines (ex-KVM-kernal VM,KEN)

2) Software Based OR Application Based OR Process Based Virtual Machines

**1) Hardware Based OR System Based Virtual Machines**

It Provides Several Logical Systems on the Same Computer with Strong Isolation from Each Other.

Examples:

1) KVM (Kernel Based Virtual Machine) for Linux Systems

2) VMware (Virtual Machine ware)

3) Xen

4) Cloud Computing

The main advantage of Hard-ware based Virtual Machines is for effective utilization of hard-ware resources.

**2) Software Based OR Application Based OR Process Based Virtual Machines**

These ge applications Acts as Runtime Engines to Run a Particular Programming

Examples:

1) JVM Acts as Runtime Engine to Run Java Applications

2) PVM (Parrot VM) Acts as Runtime Engine to Run Scripting Languages Like PERL.

3) CLR (Common Language Runtime) Acts as Runtime Engine to Run Net Based Applications.

**3- Basic diagram of JVM**

**.Class**

**CLASSLOADER SUB SYSTEM**

**Method**

**Stacks**

**Native**

**PC Registers**

**Stack Area**

**Heap Area**

**Method Area**

**VARIOUS MEMORY AREAS OF JVM**

**Native Method Libraries**

**Java Native Interface (JNI)**

**EXECUTION ENGINE**

**4- Class Loader Sub System ->**

- ClassLoader Sub System is Responsible for the following 3 Activities.

1- Loading

2- Linking

• Verification

• Preparation

• Resolution

3- Initialization

**1) Loading:**

• Loading Means Reading Class Files and Store Corresponding Binary Data in Method Area.

• For Each Class File JVM will Store the following Information in Method Area.

1) Fully Qualified Name of the Loaded Class OR Interface OR enum.

2) Fully Qualified Name of its Immediate Parent Class.

3) Whether .class File is related to Class OR Interface OR enum.

4) The Modifiers Information

5) Variables OR Fields Information

6) Methods Information

7) Constant Pool Information and so on.

• After loading class File Immediately JVM will Creates an Object of the Class to Represent Class Level Binary Information on the Heap Memory

class Class object to represent Student.class information

class Class object to represent Student.class information

Customer.class related binary information

Student.class related binary information

Customer.Class

Student.Class

Create

Create

**Heap Area**

**Method Area**

**Hard-Disk**

It is not Student&Customer Object & it is Object of type **java.lang.Class**

Read Store

Read Store

-The Class Object can be used by Programmer to get Class Level Information Like Fully Qualified Name of the Class, Parent Name, Methods and Variables Information Etc.

**Program to print methods and variables information by using Class object:**

import java.lang.reflect.\*;

**Test.java**  
  
public class Test {  
 public static void main(String[] args) throws ClassNotFoundException {  
 Student c = new Student();  
// Class c = Class.forName("Student");  
  
 Field[] f = c.getClass().getFields();  
 for (Field f1:f){  
 System.*out*.println(f1);  
 }  
 Method[] m = c.getClass().getDeclaredMethods();  
 for (Method m1:m){  
 System.*out*.println(m1);  
 }   
 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);  
 }  
}

**Student.java**

public class Student {  
 int x ;  
 private String name;  
 public String m1(){  
 return name;  
 }  
 public void m2(int x){  
 this.x = x;  
 }  
}

In the Above Example by using Student class Class Object, we can get its Methods and Variable Information.

**Note**: For Every loaded .class file Only One Class Object will be Created, even though we are using Class Multiple Times in Our Application.

**2) Linking:**

Linking Consists of 3 Activities

1) Verification

2) Preparation

3) Resolution

i) Verification:

• It is the Process of ensuring that Binary Representation of a Class is Structurally Correct OR Not.

• That is JVM will Check whether class File generated by Valid Compiler OR Not. i.e whether .class File is Properly Formatted OR Not.

• Internally Byte Code Verifier which is Part of ClassLoader Sub System is Responsible for this Activity.

• If Verification Fails then we will get Runtime Exception Saying java.lang.VerifyError.

ii) Preparation:

In this Phase JVM will Allocate Memory for the Class Level Static Variables and Assign Default Values (But Not Original Values).

**Note**: Original Values will be assigned in Initialization Phase.

iii) Resolution:

• It is the Process of Replacing Symbolic References used by the Loaded Type with Original References.

• Symbolic References are Resolved into Direct References by searching through Method Area to Locate the Referenced Entity.

**class Test 1**

**public static void main (String [] args) {**

**String s = new String("Kanha");**

**Student s1 = new Student;**

**}**

**}**

**• Test1.class**

**• String.class**

**• Student.class**

**• Object.class**

* For the Above Class, ClassLoader sub system Loads Test1.class, String.class, Student.class, and Object.class.
* The Names of these Class Names are stored in Constant Pool of Test Class.
* In Resolution Phase these Names are Replaced with Actual References from Method Area.

**3) Initialization:**

In this Phase All Static Variables will be assigned with Original Values and Static Blocks will be executed from from top to bottom and from Parent to Child.

**Linking**

**Initialization**

**Verification**

**Preparation**

**Loading**

**Initialization**

**Resolution**

**Note**: While Loading, Linking and Initialization if any Error Occurs then we will get Runtime Exception Saying java.lang.LinkageError. Of course VerifyError is child class. of LinkageError only

**5- Types of Class Loader**

Every Class Loader Sub System contains the following 3 Class Loaders.

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

**1. Bootstrap Class Loader/ Primordial Class Loader**

-This Class Loader is Responsible for loading Core Java API Classes i.e. the Classes Present in jrt-fs.jar, Means all predefine classes in java.

- jrt-fs.jar is nothing but it the collection of predefine classes like String.class, Integer.class etc....

- This Location is called Bootstrap Class Path i.e. Bootstrap Class Loader is Responsible to load Classes from Bootstrap Class Path. (LOC- C:\Program Files\Java\jdk-18.0.2.1\lib)

-Bootstrap Class Loader is by Default Available with the JVM. It is implemented in Native Languages Like C and C++. (due this when you call the class it shows null)

**2. Extension Class Loader:**

-It is the Child Class of Bootstrap Class Loader. The Class Loader is Responsible to load Classes from Extension Class Path. Means that you are stored in the jar file and that exits in Location - jdk\jre\lib\ext.

-This Class Loader is implemented in Java and the corresponding .class File Name is jdk.internal.loader.ClassLoaders$ExtClassLoader.class.

**NOTE** – Create a jar file jar -cvf <jar file name> <java class file name.class>

**3. Application Class Loader OR System Class Loader:**

-It is the Child Class of Extension Class Loader. This Class Loader is Responsible to loadClasses from Application Class Path. Means it loaded the user define Classes.

- It internally uses Environment Variable Class Path.Application Class Loader is implemented in Java and the corresponding .class File Nameis sun.misc.Launcher$AppClassLoader.class **OR** jdk.internal.loader.ClassLoaders$AppClassLoader.

**Example –**

class Test {

public static void main (String [] args)

System.out.printin(String.class.getClassLoader());

System.out.printin(Student.class.getClassLoader());

System.out.printin(Test.class.getClassLoader());

}

}

- Assume that Student.class present in both extension and application classpaths and Test.class present in only Application classpath.

**For String Class:**

-From Bootstrap Class Path by Bootstrap Class Loader Output is null (b'z BootStrap class Loader is not implemented in java and it is not java object)

**For Student Class:**

-From Extension Class Path by Extension Class Loader

-Output is jdk.internal.loader.ClassLoaders$ExtClassLoader@1234

-For Test Class: From Application Class Path by Application Class Loader

-Output is jdk.internal.loader.ClassLoaders$AppClassLoader@3456

**Note**:

-Bootstrap Class Loader is not Java Object. Hence we are getting null in the 1st Case but Extension Class Loader and Application Class Loader are Java Objects and Hence we are getting proper output.

- ClassName@HexaDecimal String of Hashcode

-Class Loader Subsystem will give Highest Priority for Bootstrap Class Path and then Extension

-Class followed by Application Class Path.

**6- How java class loader work ?**

1. Class Loader follows Delegation Hierarchy Principle.
2. Whenever JVM come across a Particular Class, First it will Check whether the corresponding Class is already loaded or Not.

3. If it is already loaded in Method Area then JVM will use that loaded Class.

1. If it is not already loaded then JVM Requests Class Loader Sub System to load that Particular Class, then Class Loader Sub System Handovers the Request to Application Class Loader.
2. Application Class Loader delegates Request to Extension Class Loader and Extension Class Loader in turn delegates to Boot Strap Class Loader.
3. Boot Strap Class Loader searches in Boot Strap Class Path (JDK/ JRE/Lib). If the specified Class is available then it will be loaded. Otherwise Boot Strap Class Loader delegates the Request to Extension Class Loader.
4. Extension Class Loader will searches in Extension Class Path (UDK/ JRE/Lib/Ext). If the specified Class is available then it will be loaded. Otherwise it delegates the Request to Application Class.
5. Application Class Loader will searches in Extension Class Path. if the specified Class is available then it will be loaded otherwise we will get ERROR : ClassNotFoundException.

**Searches in**

**Searches in**

**Searches in**

**JVM**

**Class Loader Sub System**

**Bootstrap Class Loader**

**Extension Class Loader**

**Application Class Loader**

**Application Class Path (Class Path Variable)**

**Bootstrap Class Path (JDK/JRE/Lib/rt.jar)**

**Extension Class Path (JDK/JRE/Lib/Ext)**

**Delegates**

**Delegates**

**NOTE**: - Class Loader Sub System will give Highest Priority for **Boot Strap** **Class** Path and then **Extension Class** Path followed by the **Application Class** Path.

**7- Need of Customizer class loader ?**

- Sometimes we may not satisfy with default class loaders and with default class loading mechanism.

**Use-Case:**

* Default ClassLoader 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 on fly, because .class File already there in Method Area.
* We can Resolve this Problem by defining Our Own Customized ClassLoader. For Example whenever we are using a class first my class loader will whether updated version is available or not. If it is available then load updated version otherwise use already loaded existing class file, so that Updated Version Available to Our Program.

\*\* The Main Advantage of Customized ClassLoader is we can customize Class loading Mechanism Based on Our Requirement.

**8- Pseudo Code to define Customizer class?**

-We can Define Our Own Customized ClassLoader by extending **java.lang.ClassLoader Class**.

**public class CustomClassLoader extends ClassLoader {  
 public Class loadClass(String cname) throws ClassNotFoundException {  
 /\*  
 Check whether updated version is available or not.  
 If updated version is available then load updated version  
 and returns the corresponding class Class object.  
 otherwise return class Class object of already loaded .class  
 \*/  
 return cname.getClass();  
 }  
}  
  
class CustomClassLoaderTest {  
 public static void main(String[] args) throws ClassNotFoundException {  
 Student s1 = new Student();  
 CustomClassLoader c = new CustomClassLoader();  
 c.loadClass("Student");  
 c.loadClass("Student");  
 }  
}**

**Default class loader loads .class file**

**Customized class Loader checks for updates of updated version available the load updated version otherwise use already loaded .class**

**Q1) What is the Purpose of java.lang.ClassLoader Class?**

**ANS -** We can use ClassLoader class to define our own Customized Class Loaders. Every Customized ClassLoader in Java should extends java.lang.ClassLoader class either Directly OR Indirectly. Hence this class acts as Base Class for developing Our Own Customized ClassLoaders.

**Q1) Where we can use Customized ClassLoader?**

**ANS –** While developing Web Server and Application Server.

**(Module - 2)**

**9- Various Memory area of JVM**

- Whenever JVM load and run a java program its need's memory to stored several things like bytecode, variable etc....

- Totally memory is divided into following 5 types

i- Method Area

ii- Heap area

iii- Stack area

iv- PC register

v- Native method stack

i- Method Area

- For every jvm one method area will be available

- Method area will be create at the time of JVM startup

- Inside method area class level binary data including static variable will be stored.

- Constant polls of a class will be stored inside method area.

- Method area can be access by multiple thread simultaneously.

- So method area data need not be continues

ii- Heap area

- For every JVM one heap area is available

- Heap area will be created at time of JVM Startup

- Object and corresponding instance variable will be stored in the HEAP area

- Every array in java is object only hence Array also stored in the heap area

- It can be access be multiple thread and hence the data stored in the heap memory is a not Thread SAFE.

- Heap area need not be continues.

**10 QNS - WAP to display heap memory statistic?**

public 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);

}

}

Output –

Max Memory 4208984064

Total Memory 264241152

Free Memory 258528656

Consumed Memory 5712496

- 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 run time object as follow

Runtime r = Runtime.getRuntime()

- Once we got runtime object we can call the follow method from that object

1 - Max Memory

- It return the Number of byte of max memory allocated in HEAP

2 - Total Memory

- It return number of byte of total memory allocated to the heap(initially memory)

3- Free Memory

- It return number of byte of free memory present in the heap

iii- 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 variable also.

- After completing a method the corresponding from the stack will be removed after completing all method call the stack will become empty under that empty stack will destroy by the jvm just before terminating thread.

- each entry in the stack is called stack frame are activation record.

- The data stored in the stack is available for only for corresponding thread under not available to the remaining thread hence this data is thread safe.

- Stack Memory are divided into 3 part

1 - Local variable array

2 - Operand Stack (Work Space)

3 - Frame Data (Exception are store)

1 - Local variable array

- It contains all parameter and local variable of the method.

- Each slot in the array is of 4 bytes .

- Values of types int , float and refence occupied one entry in the array.

- values of double and long occupied 2 conjugating entry in the array.

- Byte shot and char values will be converted to int type before storing and occupying one slot.

- But the way of storing Boolean values is valid from JVM to JVM but most of the JVM follow one slot for Boolean values

Program –

12222

Diagram

2 - Operand Stack (Work Space)

- JVM uses Operand Stack (Work Space)

- Some instruction can push the values Operand Stack and some instruction can pop the values from Operand Stack and some instruction can perform required operation.

Diagram

**iLoad - 0 iLoad - 1 iAdd iStore - 2**

**LOCAL VARIABLE ARRAY**

**LOCAL VARIABLE ARRAY**

**LOCAL VARIABLE ARRAY**

**LOCAL VARIABLE ARRAY**

**LOCAL VARIABLE ARRAY**

100

90

190

100

90

100

90

100

90

100

90

1

2

3

1

2

3

1

2

3

1

2

3

1

2

3

**OPERAND STACK**

**OPERAND STACK**

**OPERAND STACK**

**OPERAND STACK**

**OPERAND STACK**

100

**BEFORE STARTING AFTER iLOAD\_0 AFTER iLOAD\_1 AFTER iADD AFTER iSTORE\_2**

190

90

100

3 - Frame Data (Exception are store)

- Frame data contains all symbolic reference related to that method

- It also contains a reference to Exception table which provide corresponding catch block exception in the case on exception.

- It stored also constant poll.

iv- PC register

- PC stand for Program Counter Register.

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

- PC register contains the address of current executing instruction.

- Once Instruction execution complete automatically PC register incremented to hold address of next instruction.

v- Native method stack

- For every thread JVM will create a separated native method stack.

- All native method call invoked by the thread by the thread will be stored in the corresponding native method stack.

**NOTE ----** Method area, Heap area and Stack Area are consider as important memory area with respect to programmer.

- Method area and Heap area are per JVM.

- Whereas Stack area, PC register and Native Method Stack are per Thread.

- Static variable will be stored in Method Area.

- Instance variable will stored in Heap Area.

- Local variable will stored in stack Area.

**Final Example with Diagram –**

Program - ->

**Heap Area**

public class Location {  
 Student s1 = new Student ();  
 static Student *s2* = new Student ();  
  
 public static void main (String [] args) {  
 Location L = new Location ();  
 Student s3 = new Student ();  
 }  
}

**Method Area**

**Stack Area**

Location Object

Student Object

Student Object

Student Object



S1

S

(Module - 3)

**12- Executer Engine ---**

- This is the central component of JVM .

- Execution Engine is responsible for execute the java class file.

- Execution Engine mainly contain 2 component.

i- Interpreter

ii- JIT compiler

i- Interpreter

- It is the responsible to read byte code and Interpreted into machine code (native code) and execute that machine code line by line.

- the problem with Interpreted with it Interpreted every time even same method invoke multiple time which reduce performance of the system.

- To overcome this problem sum people introduce JIT compiler.

ii- JIT compiler

- The main prepose of JIT Compiler is to improve performance internally JIT compiler maintained a separate count for every method.

- Whenever JVM across any method call first that method will be Interpreted by the Interpreted and JIT compiler increment corresponding count variable.

- This process will continue for every method.

- Once if any method counts reaches thresh hold value, then JIT compiler identifies that, the method is a repeatedly (Hot spot) used method and such type of method call Hot spot.

- immediately JIT complier that method and generate the corresponding native code.

- Next time JVM come across that method call then JVM uses native code directly and the execute it's instated of interpreting once again so that performance of the system will improve.

- The thresh hold count varies from JVM to JVM

- Some advanced JIT compiler will re-compile generated native code if count reaches thresh hold value second time so that more optimised machine code will generate.

- Internally "Profiler", Which is the part of JIT compiler is responsible to identify HOT SPOT.

**NOTE ----** JVM interprets total program at least once.

- JIT compiler is applicable only for repeatedly require method not for every method.

**13 - Java native Interface (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 libraries provide or hold native libraries information.