**Overview of JVM**

* Virtual Machine
* Type of Virtual Machine

1. Hardware Virtual Machine
2. Application virtual Machine

* Basic Architecture of Virtual Machine
* Class Loader Subsystem

1. Loading
2. Linking
3. Initialization

* Types of Class Loader

1. Bootstrap class Loader
2. Extension class Loader
3. Application class Loader

* How class Loader Works
* What is the need of Customized class Loader
* Psudedo code for Customized class Loader
* Various Memory Areas of JVM

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

* Program to display heap memory statistics
* How to set Maximum and Minimum heap size?
* Execution Engine

1. Interpreter
2. JIT Compiler

* Java Native Interface(JNI)
* Complete Architecture Diagram of JVM
* Class File Structure.

JVM(Java Virtual Machine)

If is a software simulation of a Machine which can perform operation like a physical machine

There are 2 type of virtual machine 1-Hardware based or System based virtual machine

2-Application based or process based virtual machine.

1. Hardware based or System based virtual machine

It provides several logical system on the same computer with strong isolation from each other. That is on physical machine we are defining multiple logical machine

The main advantage of hardware based virtual machine is hard resources sharing and improve utilization of hardware resources

Eg. KMV(Kernal based virtual machine for Linux systems ,VMWare,Xen ,Cloud computing etc)

2-Application based or Process based Virtual Machine

These virtual machine act as run time engines to run a particular programming language application

* JVM(Java Virtual Machine) acts as Run time engine to run java based application.
* PVM(Perl Virtual Machine) acts as Run time engine to run Perl based application.
* CLR(Common Language Runtime) act as Run time engine to run .Net based application.

JVM(Java Virtual Machine)

JRE is a part of JDK

JVM is part of JRE

JVM

*Important task of JVM is to load .class file and run java .class file.*

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

Basic Architecture Diagram of JVM

**Input .class File Input**

Class Loader SubSystem

***Various Memory Area***

Method Area Heap Area Stack Area PC Register Native Method Area

Java Native Method Library

Class Loader Sub System

Class loader subsystem is responsible for the following 3 Activity

* Loading
* Linking
* Initialization

Loading

Loading mean reading class file 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 quailified name of immediate parent class

3-Method information

4-Variable information

5-Constructor information

6-Modifier Information

7-Constant pool information

Etc.

After loading .class file immediately JVM create an object for that loaded class on the Heap memory of type java.lang.Class

Class class object to represent student .class file

Class class object to represent customer .class file

Hard Disk Method Area(JVM) H Heap Area(JVM)

Student .class Info

Customer.class Info

I

Student.class

It is not student obj

Customer.class

After loading .class file immediately JVM create an object for that loaded class on heap memory of type java.land.Class

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

**Program for class Class.**

**package** com.jvm;

**import** java.lang.reflect.\*;

//import java.lang.reflect.Method;

**class** Student

{

**public** String getName()

{

**return** **null**;

}

**public** **int** getRollNo()

{

**return** 10;

}

}

**public** **class** Test1 {

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

**int** count =0;

Class c=Class.*forName*("com.jvm.Student");

Method[] m=c.getDeclaredMethods();

**for** (Method method : m) {

count++;

System.*out*.println(method.getName());

}

System.*out*.println("Number of Method"+count);

}

}

**For every loaded type only one Class object will be created even though we are using the class Multiple times. Below program for the same.**

**Program**

**package** com.jvm;

**import** java.lang.reflect.\*;

//import java.lang.reflect.Method;

**class** Student

{

**public** String getName()

{

**return** **null**;

}

**public** **int** getRollNo()

{

**return** 10;

}

}

**public** **class** Test1 {

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

Student st1=**new** Student();

Class c1=st1.getClass();

Student st2=**new** Student();

Class c2=st2.getClass();

**int** i=c1.hashCode();

**int** j=c2.hashCode();

System.*out*.println(i==j);

System.*out*.println(c1==c2);

}

}

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

**Linking**

Linking consist of 3 activity 1-Verification 2-Prepare 3-Resolve.

1-Verification

It is a 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 that is whether .class file properly formatted or not.Internaly byte code verifier is responsible for this activity.Bytecode verifier is a part of class loader subsystem.

If verification failed then we will get run time exception saying **java.lang.verify error.**

2-Prepration

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

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

3-Resolution

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

Eg.

Class Test{

Public static void main(String [] args)

{

String s=new String(“Rahul”);

Student s=new Student();

}

}

For the above class loader 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 replace with original memory level references from method area.

**Initilization:**

In this all static are assigned with original values and static block will be executed from parent to child and from top to bottom.

***Class Loading Process***

Verification

Verification

Prepare

Resolve

Resolution

Prepare

Initialization

Loading

Linking

Note While loading linking and initialization if any error accurse then we will get run time exception saying java.lang .liknage error.

**Type of Class Loaders.**

Class loader subsystem contain the following 3 type of class loader

1. Bootstarp class loader or premodial class loader.
2. Extension class Loader
3. Application class Loader or System class Loader

Bootstrap Class Loader

Bootstrap class loader is responsible to load core java API classes that is the classes present in **rt.jar**

**Jdk\jre\lib\rt.jar ->this location is called Bootstarp path that is Bootstrap class loader 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 a child class of bootstrap class loader

Bootstrap class Loder

Extension class Loader

Extension class loader is responsible to load classes from extension class path(jdk\jre\lib\ext)

JDK

JRE

lib

ext

\*.jar

Extension class loader is implemented in java and the corresponding .class file is sun.misc.Launcher$ExtclassLoader.class

Application class Loader or System class class Loader

Application class loader is child class of Extension class Loader

This class loader is responsible to load classes form Application class path .It internally uses environment variable class path.Application class loader is implemented in java and the corresponding .class file name is sun.misc.Launcher$AppclassLoader.class.

BootStrap Class Loader

Extension Class Loader

Application class Loader

**How Class Loader works**

search

Search in (jdk/jre/lib) BootStrap class path

BootStrap Class Loader

delegate delegate

search

Search in Extension class path(jdk/jre/lib/ext

Extension Class Loader

delegate delegae

Search in Application class path(class Path

Application Class Loader

Request re search

Class Loader fallows Delegation hierarchy principal (algorithm) when ever JVM come across a particular class first it will check whether the corresponding .class file is already loaded or not .If it is already loaded in method area then JVM will consider that loaded class .If it not loaded then JVM request class loader subsystem to load that particular class then class loader subsystem handover the request to application class loader .Application class loader delegates the request to Extension class loader which intern delegate the request to BootStrap class loader.Then bootstrap class loader will search in Bootstrap class path if it is available then the corresponding .class will be loaded by bootstrap class loader .If it is not available then bootstrap class loader delegate the request to extension class loader extension class loader will search in extension class path .If it is available then it will loaded otherwise extension class loader delegate the request to application class loader.Application class loader will search in application class path if it is available then it will be loaded otherwise we will get run time exception saying No classDef found error or ClassNotFound Exception.

Example

class Test{

public static void main(String [ ] args)

{

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

System.out.println(Customer.class.getClassLoader());

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

}

}

Assume Customer.class present in both Extension and Application class path and Test.class present in only Application class path.

For String . class BootStrap class loader from BootStrap class path. Output will null as its not implemented in java

For Test.class Application class loader from Application class path.Output: sun.misc.Launcher$AppClassLoader@1db9742

For Customer.class :Extension class loader will from extension class path.

Note :

1-Bootstrap class loader is not java object hence we got null is first case above program.But extension and Application class loaders are java object hence we are getting corresponding out for the remaining sops.(classname@hashCodein hexadecimal form)

2-Class Loader subsystem will give the Highest priority for Bootstrap class path and then Extension class path fallowed by Application class path.

**Need of Customize Class Loader**

Default class Loader will load .class file only once even though we are using multiple time that class in our program .After loading .class file if it is modified outside then default class loader 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 class loader .The main advantage of customized loader is We can control class loading mechanism based on our requirement

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

Default class Loading Diagram

Student s1=new Student()

Student s2=new Student()

Student s3=new Student()

-

-

-

-

-

Student s100=new Student()

load

Use

Use

Use

Customized Class Loading Diagram

Student s1=new Student()

Student s2=new Student()

Student s3=new Student()

-

-

-

-

--

Student s100=new Student()

Load

fLoa

Use

Check whether Student.class is modified or not if it is modifies load upated .class file ,If it is not modified used already loaded .class file

Use

Use

How to Define Customized class Loader

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

public class CustClassLoader extends ClassLoader

{

public Class loadClass(String cname) throws ClassNotFoundException

{

//Code to check for update and load updated .class file oand return corresponding Class class

}

}

Class Client

{

Public static void main(String[] args)

{

Dog d=new Dog()->Use default class loader for first time

CustClassLoader cl=new CustClassLoader();

cl.load(“Dog”);->Loaded by Customized class loader

cl.load(“Dog”);-> Loaded by Customized class loader

}

}

Note :While developing Web Server and Application servers usually we can go for Customized class loader to customize class Loading mechanism.

Q-What is the need or use of Class Loader class

Ans-We can use java.lang.ClassLoader class to define our own customized class loader .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 customized classLoder.

**Various Memory Area inside JVM**

Whenever JVM Load and runs a java program it needs memory to store several things like Byte code ,object variable etc.

Total JVM Memory organized into the following 5 categories

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

**Method Area**

1. For every JVM one method Area will be available.
2. Method area will be created at the time of JVM startup
3. Inside method area class level binary data (.class) including static variable will be stored
4. Constant pool of a class will be stored inside method area.
5. Method area can be access by multiple threads simultaneously

Class level data

Class level data

Class Level data

Class Level data

Class Level data

**Method Area**

**Method Area**

**Heap Area**

1. For every JVM one Heap Area is available
2. Heap area will be created at the time of JVM startup
3. Object and corresponding instance variable will be stored in the Heap Area every array is java is Object only hence arrays also will be stored in the Heap area.
4. Heap area can be accessed by multiple threads and hence the data stored in the Heap memory is not thread safe.
5. Heap Area need not be continuous.

Object

Object

Object

Object

Object

Program to Display Heap memory statistics.

A java application can communicate with JVM by using Runtime object

Runtime class present in java.lang package and it is a Singleton class. We can create runtime object as fallows

Runtime r=Runtime.getRuntime();

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

Max Memory-It returns the number of byte of Max Memory allocated to Heap.

Total Memory-It returns number of Bytes of Total memory allocated to the Heap.(Initial Memory)

Free Memory-It return number of bytes of Free memory present in the Heap.

**public** **class** HeapDemo {

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

Runtime r=Runtime.*getRuntime*();

System.*out*.println(r.maxMemory()/(1024\*1024) +"MB");

System.*out*.println(r.totalMemory()/(1024\*1024) +"MB");

System.*out*.println(r.freeMemory()/(1024\*1024) +"MB");

System.*out*.println("consumed Memory \t"+(r.totalMemory()-r.freeMemory())/(1024\*1024) +"MB");

}

}

How to set Maximum and minimum Heap sizes.

Heap memory is finite memory but based on our requirement we can set maximum and minimum heap sizes that is we can increase or decrease the Heap size based on our requirement .We can use the following flags with java command.

-Xmx to set Maximum Heap size(Max Memory )

Eg java –Xmx512m HeapDemo

This command Maximum Heap size as 512 MB.

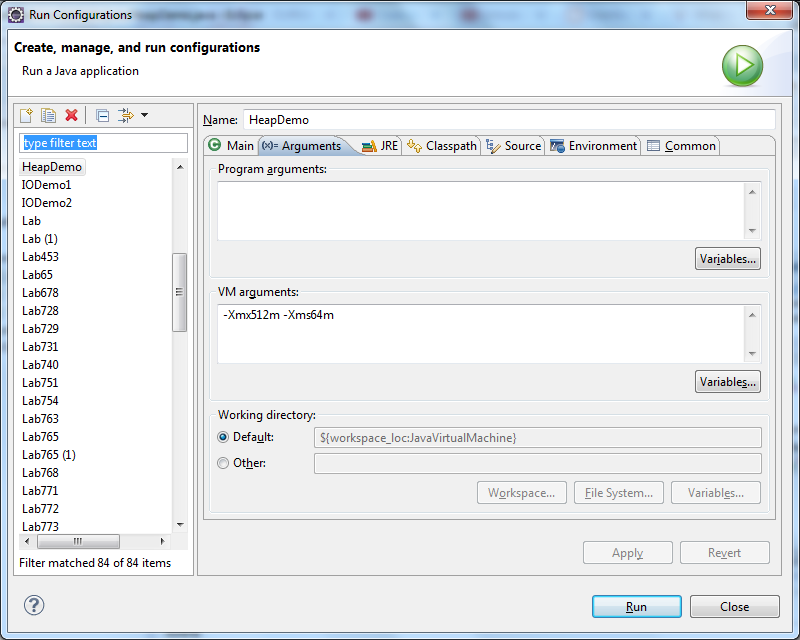
-Xms we can use this commond to set min Heap Size

Ex java –Xms64m HeapDemo to set Minimum Heap size to 64 mb.(total Memory)

We can even set Heap Size for Max and Min Memory as below

-Xmx512m –Xms64m (Here we are setting Max Memory as 512 MB and Min heap memory 64MB)

**Note –In eclipse we have to pass these argument as below Diagram under VM arguements**



**Stack Memory**

For every thread JVM will create a separate stack at the time of Thread creation each and every method performed by that thread will be stored including local variable also .After completing a method the corresponding entry from the stack will be removed after completing all method call the stack will become empty and the empty stack will be destroyed by JVM just before terminating the thread .Each entry in the stack is called **stack frame or Activation Record**

**Stack Memory Diagram**

Thread 1 Thread 2 Thread 3 Thread 4

Method

Methods

Methods

Methods

**RunTime Stack**

The data stored in the stack is available for the corresponding thread and not available to the remaining threads hence this data is Thread safe

**Stack Frame Structure**

Each stack frames contain 3 parts

Frame Data

Operand Data

Local Variable Array

Stack FrameArchitecture

**Local Variable Array:**

It contains all parameter and local variable of the methods. 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 occupy 2 consecutive entry in the array .

byte sort and char value will be promoted to int value and stored but the way storing Boolean value is varied from JVM .But most of the JVM fallow one slot for Boolean value.

**Eg**

**public void m1(int I, double d ,Object o, float f){}**

**4 Byte 4 Byte 4 Byte 4 Byte 4 Byte**

Int I double d double d float f float f

**Operand Stack**

JVM uses Operand stack as work space .Some instructions can push the values to the Operand stack and some instruction can pop values from operand stack and some instruction can perform required operation

Example :Assembly level program

1. Iload 0
2. Iload 1
3. Iadd
4. Istore-2

Before Starting After iload0 After iload 1 After iadd After istore 2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Local Variable Array | LocalVariableArray | Local Variable Array | Local Variable Array | Local Variable Array |
| |  | | --- | | 100 | | 90 | |  | | |  | | --- | | 100 | | 90 | |  | | |  | | --- | | 100 | | 90 | |  | | |  | | --- | | 100 | | 90 | |  | | |  | | --- | | 100 | | 90 | | 190 | |
| Operand Stack | Operand Stack | Operand Stack | Operand Stack | Operand Stack |
| |  | | --- | |  | |  | | |  | | --- | |  | | 100 | | |  | | --- | | 90 | | 100 | | |  | | --- | | 190 | | |  | | --- | |  | |

**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 exception.

**PC Register:**

For every thread a separate PC Register will be created at the time Thread creation .PC register contains the address of current executing instruction once instruction execution complete automatically PC register will be incremented to hold address of next instruction

**Native Method Stack :**

For every thread JVM will create a separate native method stack all native method call invoked by the thread will be stored in the corresponding native method stack.

Note:

1. Method area ,Heap area and stack area are consider as important memory area w.rt to programmer .
2. Method area and Heap area are per JVM where as stack area ,PC register and Native method stack are per thread.
3. **For every**

**JVM->One Heap Area, One Method Area**

**For every Thread->One stack Area,One PC Register , One native method stack.**

1. Static variable will be stored in Method Area , instance variable will be stored in Heap Area, local variable will be stored in stack area.

**Example**

**Class Test**

**{**

**Student s1=new Student();**

**Static student s2=new Student();**

**Public static void main(String [] args)**

**{**

**Test T=new Test();**

**Student s2=new Student();**

**}**

**}**

**Memory Diagram for above code**

**Method Area**

S11

**Heap Area**

S2

T

**Stack Area**

**Execution Engine**

This is the central component of JVM .Execution engine is responsible to execute java class files execution engine mainly contains 2 components

* Interpreter
* JIT Compiler

Interpreter:

It is responsible to read byte code 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 method invoke multiple time 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 count for every method whenever JVM come across any method call first that method will be interpreted normally by the interpreter and JIT compiler increment 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 repeatedly use method (hot spots) 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 advance JIT compiler will re compile generated native code if count reaches threshold value 2nd time so that more optimize machine code will be generated .

Internally Profiler, which is part of JIT compiler is responsible to identify hotspots.

Note:

* JVM interprets total program at least once
* JIT compilation is applicable only for repeatedly required method not for every method.

Execution Engine internal anatomy

**JIT Compiler**

Intermediate Code Generator (interpreted code)

Code Optimizer

Target Code Generator

Machine Code G.C ..

Etc..

Java Native Interface(JNI)

JNI acts as mediator for Java method calls and corresponding native libraries that is JNI is responsible to provide information about native library to the JVM.

Native method library provides holds native libraries information

Execution Engine

Native Libraries