**JVM**

1. **Virtual machine:**

It is a software simulation of a machine which can perform operations like a physical machine. There are two types of virtual machines, hardware based or system based virtual machine and application based or process based virtual machine.

* **Hardware based or system based virtual machine:**

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 advantage of hardware based virtual machines is hardware resources sharing and improves utilization of hardware resources.

Example: KVM (kernel based virtual machine for linux systems), VMWARE gen cloud computing etc.

* **Application based or process based virtual machine:**

These virtual machines act as runtime engines to run a particular programming language applications.

**Example:**

JVM (java virtual machine) acts as run time engine to run java based applications.

PVM (parrot virtual machine) acts as runtime engine to run perl based applications.

CLR (common language runtime) acts as runtime engine to run .net based applications.

1. **JVM:**

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

**Basic architecture diagram of JVM:**

Various memory areas of JVM

.class file

Native method libraries

Java native interface

Execution engines

Class loader sub system

* **Class loader sub system:**

Class loader subsystem is responsible for the following three activities. Loading, linking and 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 method area.

* Fully qualified name of class
* Fully qualified name of immediate parent class
* Methods information
* Variables information
* Constructors information
* Modifiers information
* Constant pool information etc.

After loading .class file immediately, JVM creates an object for that loaded class on the heap memory of type java.lang.class.

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

* **Linking**

Linking consists of three activities. Verify, prepare and resolve.

1. **Verification or 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 file is 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 runtime exception saying “java.lang.VerifyError”

1. **Preparation:**

In this process or 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.

1. **Resolution:**

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

**Example:**

class Test

{

public static void main(String args[])

{

String s=new String(“navya”);

Student s1=new Student();

}

}

For the above class, 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 replaced with original memory level references 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.

Class loader subsystem

Linking

Verify

Prepare

Resolve

Initialization

Loading

**Note:** while loading, linking and initialization, if any error occurs, then we will get runtime exception saying “java.lang.LinkageError”

**Types of class loaders:**

Class loader subsystem contains the following three types of class loaders.

1. **Bootstrap class loader or primordial class loader**

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

Jdk -> jre -> li -> rt.jar. This location is called bootstrap class path. i.e. bootstrap class loader is responsible to load classes from bootstrap class path.

Bootstrap class loader is by default available with every JVM. It is implemented in native languages like C or C++ and not implemented in java.

1. **Extension class loader**

Extension class loader is the child class of bootstrap class loader.

Extension class loader is responsible to load classes from extension class path (jdk -> jre -> lib -> Ext -> \*.jar).

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

1. **Application class loader or system class loader**

Application class loader is the child class of extension class loader. This class loader is responsible to load classes from 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:**

Class loader follows delegation hierarchy principle. Whenever 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 is not loaded then, JVM request class loader subsystem to load that particular class.

Then class loader subsystem handovers the request to application class loader. Application class loader delegates the request to extension class loader which in turn delegates the request to bootstrap class loader.

searches

searches

searches

delegates

delegates

request

request

JVM

Class loader subsystem

Application class loader

Extension class loader

Bootstrap class loader

Application class path (environment variables class path)

Extension class path (jdk\jre\lib\ext)

Bootstrap class path (jdk\jre\lib)

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 delegates the request to extension class loader.

Extension class loader will search in extension class path. If it is available, then it will be loaded. Otherwise, extension class loader delegates 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 runtime exception saying “noClassDefFoundError” or “classNotFoundException”.

**Example:**

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 class path.

For String.class - Bootstrap class loader from bootstrap class path. Output is null

For Test.class – Application class loader from application class path. Output is [sun.misc.launcher$AppClassLoader@1912a56](mailto:sun.misc.launcher$AppClassLoader@1912a56)

For Customer.class – Extension class loader from extension class path. Output is [sun.misc.Launcher$ExtensionClassLoader@1072b90](mailto:sun.misc.Launcher$ExtensionClassLoader@1072b90)

**Note:**

* Bootstrap class loader is not java object. Hence we got null in first case. But extension and application class loaders are java objects. Hence we are getting corresponding output.
* Class loader subsystem will give the highest priority for bootstrap class path and then extension class path followed by application class path.

**Need of customized class loader:**

Default class loaders 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 class loader wont load updated version of class file (because .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 Customized class loading

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

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

Check whether the student class is modified or not. If it is modified load updated .class file. If it is not modified use already existing .class file

Student s3=new Student(); Student s3=new Student();

. .

. .

Student s10=new Student(); Student s10=new Student();

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

public class CustClassLaoder extends ClassLoader

{

public class loadClass(String sname) throws ClassNotFoudException

{

Class for updates and load updates .class file and returns corresponding class

}

}

class Client

{

public static void main(String args[])

{

Dog d1=new Dog(); // loaded by default class loader

CustClassloader cl=new CustCLassLoader();

cl.loadClass(“Dog”);

// loaded by customized class loader

cl.loadClass(“Dog”);

}

}

**Note:** while designing or developing web servers and application servers usually we can go for customized class loaders to customize class loading mechanism.

**ClassLoader class:**

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 acts as base class for all customized class loaders

* **Various memory areas present inside JVM:**

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

Total JVM memory organized into the following 5 categories.

1. **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

Method area

variables will be stored.

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

Method area can be accessed by multiple threads simultaneously.

1. **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 heap memory is not thread safe.

Heap area need not be continuous.

A java application can communicate with JVM by using runtime

Object.

Runtime class is present in java.lang package and it is a

Heap area

singleton class.

We can create runtime object as follows.

Runtime r=Runtime.getRunTime();

Once we got runtime object, we can call the following methods

on that object.

maxMemory() - it returns the number of bytes of max memory allocated to the heap.

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

freeMemory() - it returns number of bytes of free memory present in the heap

**Example:**

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 in bytes:**

Max memory 25922560

Total memory 16252928

Free memory 15874448

Consumed memory 378480

**Output in mb:**

Max memory 247.5

Total memory 15.5

Free memory 15.39

Consumed memory 0.360

**How to set maximum and minimum heap sizes?**

Heap memory is finite memory (fixed). 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 the following flags with a java command.

**-xmx**

To set maximum heap size (max memory)

**Example:** java –xmx 512m classname

This command will set maximum heap size as 512 mb

**-xms**

We can use this command to set minimum heap size.

**Example:** java –xms 64m classname

To set minimum heap size as 64 mb. i.e. total memory.

1. **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 memory

Tn

T2

T1

………..

Runtime stack

Runtime stack

Runtime stack

**Stack frame structure:**

Each stack frame contains three parts.

Local variable army

Operand stack

Frame data

Stack frame

**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 one entry in the array.

Values of double and long 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 JVM’s follow one slot for boolean values.

Public void m1(int I, double d, Object o, float f)

{

Long x;

.

.

.

}

long

float

object

double

int

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| i | d | d | o | f | x | x |

6

5

4

3

0

2

1

**Operand stack:**

JVM uses operand stack as work space.

Some instructions can push the values to the 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 relate to that method. It also contains a reference to exception table which provides corresponding catch block information in the case of exceptions.

**Differences between heap and stack:**

|  |  |
| --- | --- |
| **Heap** | **Stack** |
| All the Objects and its corresponding instance variables and arrays will be stored here. | All local variables will be created in the stack memory. |
| Heap Area is also one per JVM | For every thread, a separate runtime stack will be created. |
| Heap area is not thread safe | Stack is thread safe |

* 1. **Pc Registers**

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

1. **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:**

Method area, heap area and stack area are considered as important memory areas with respect to programmer.

Method area and heap area are per JVM. Whereas stack area and PC registers and native method stack are per thread.

For every JVM – one heap area

* One method area

For every thread – One stack area

* One PC register
* One native method stack

Static variables will be stored in method area. Instance variables will be stored in heap area. 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. Interpreter and 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 invoked 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 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 continued for every method. Once if any method counts reaches threshold value, then JIT compiler identifies that, the method is repeatedly used method (hot spot).

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 second time. So that more optimized machine code will be generated.

Internally profiler, which is the part of JIT compiler is responsible to identify hot spots

**Note:**

JVM interprets total program at least once.

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

**Execution engine**

JIT compiler

interpreter

**Garbage collector**

Intermediate code generator

profiler

Code optimizer

Target code generator

Machine code

* **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:**

Native method library provides or holds native libraries information

The overall working of JVM is as bellow

