Table of Contents

1 Introduction

1.1 Virtual machines

It's a software simulation of a machine which can perform operations like Physical machines.
There are two kinds of virtual machines –

1. Hardware based or System based virtual machines
2. Software based Or Application based Or Process based virtual machines.

1.1.1 Hardware based Or System based virtual machines

It provides several logical systems on the same machine with strong isolation from each other.

1. KVM (Kernal based virtual machines 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 hardware resources by sharing the between them.

1.1.2 Software based virtual machines

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

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.

1.2 JVM Basic Architecture

JVM is part of JDK, and it is responsible for below 2 activities.

1. Load Java class files (.class files)
2. Run Java class files(.class files)

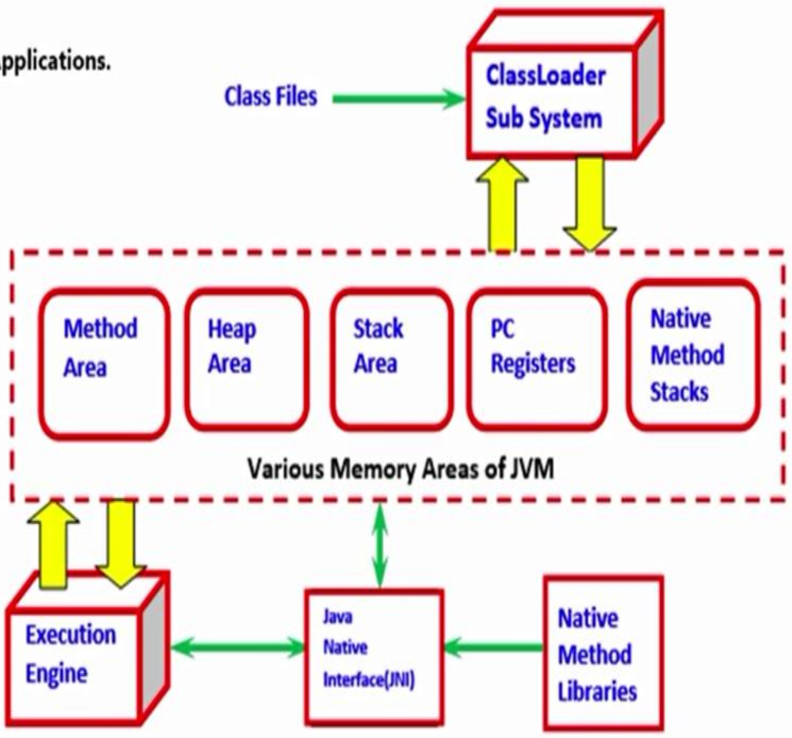
1.2.1 Basic Architecture Diagram of JVM

Below are the different parts of JVM in high level –

1. Class loader Sub system – Take .class as an input and loads it into different memory areas.
2. Method area –
3. Head area
4. Stack area
5. PC registers
6. Native method stacks
7. Execution engine – Is responsible for read .class and run.
8. Native method Libraries –
9. Java Native Interface

JVM

- JVM is the Part of JRE.
- JVM is Responsible to Load and Run Java Applications.



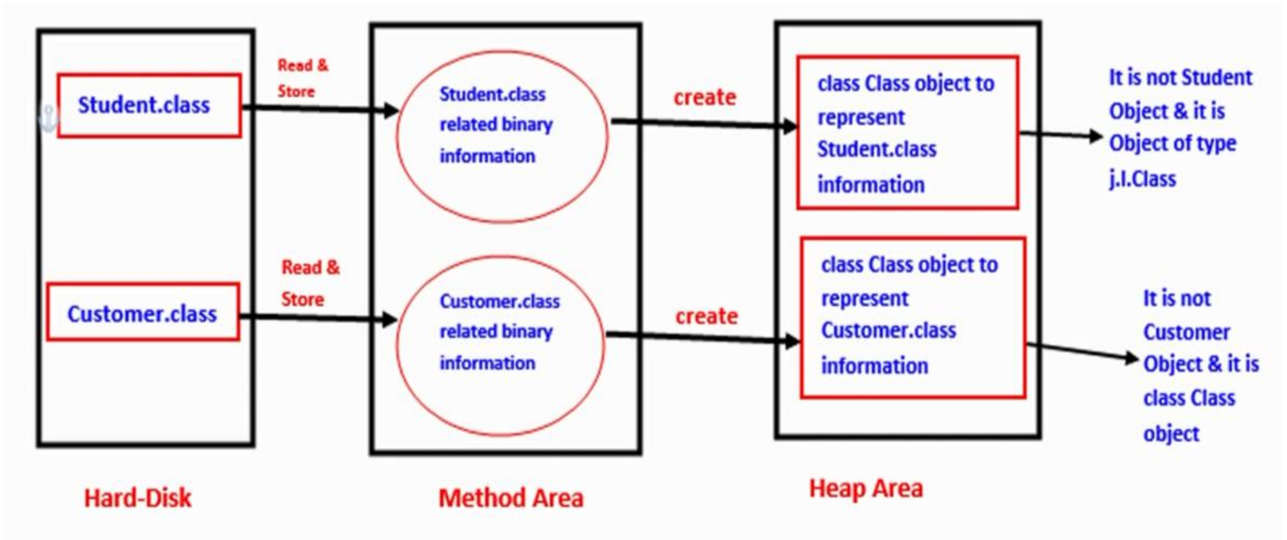
2 JVM Class loader Sub systems

Class loader is subsystem is responsible for following activities

1. Loading
2. Linking
3. Initialization

2.1 Loading

- Loading means reading class files and store corresponding binary data in "method area" may be from "hard disk".
- For each class file JVM will store the following information in "method area" in binary form.
  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. Constructors, Constant pool information and so on.
- After loading .class file immediately JVM will creates an object of the Type class Class (object of the Type "Class" of "Student.class")) to represent Class level Binary information on the "heap memory".
- These class Class objects can be used by programmer to extract class level information, like methods, variables information, constructor information etc.


Student.class
Read & Store
Student.class related binary information
create
class Class object to represent Student.class information
It is not Student Object & it is Object of type j.l.Class
Customer.class
Read & Store
Customer.class related binary information
create
class Class object to represent Customer.class information
It is not Customer Object & it is class Class object
Hard-Disk
Method Area
Heap Area

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.

```
Class Student {
        public String getName(){}
        public String getNum(){}
}
Class Test {
        public static void main (){
                Class c = Class.forName(“com.st.Student”); - This statement Loads Student class
                Method[] methds = c.getDeclaredMethods();
                Int count=0;
                For (Method m: methods) {
                        s.o.p(“Method name ” + m.getName());
                        count++;
                }
                s.o.p(“Number of methods” + count);
        }
}
Class Test {
        public static void main (){
                Student s1 = new Student(); - This statement Loads Student class
                Class c1 = s1.getClass()
                Student s2 = new Student(); - This doesn’t load Student as it already load
                Class c2 = s2.getClass()
                s.o.p(“c1 hashcode” + c1.hashcode());// 12345
                s.o.p(“c2 hashcode” + c2.hashcode()); // 12345
                s.o.p c1.hashcode() == c2.hashcode()); // True :: c1 and c2 refers to same
                Object.
        }
}
```

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

2.2 Linking

Linking phase has 3 activities

1. Verification
2. Preparation
3. Resolution

2.2.1 Verification

- Java is secured because of its .class files are in the form of byte code, and you can’t transfer virus easily by using these .class files as there is some structure for .class file.
- It is the process of ensuring that binary representation of a class structurally.

- That is JVM will check whether .class file generated by valid compiler or not.
- Internally "**Byte code verifier**" is responsible for the activity, which is part of "Class Loader sub systems".
- If verification failed then we will get runtime exception saying "Java.lang.VerifyError".

2.2.2 Preparation

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

Note: Original values will be assigned in Initialization phase and in this only default values will be assigned.

2.2.3 Resolution

- Error like can't find symbol occurs because of failures in this phase.
    - Variable : x unable to find
    - Method : m1 unable to find
    - Class : Test unable to find
- It is the process of replacing symbolic names with original memory references from method area.
- 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 reference entity.

```
Class Test {
        public static void main (){
                String s = new String("test");
                Student s = new Student();
        }
}
```

- For the above class, Class loader subsystems loads below all class level symbols.
    - Test.class, is one symbol
    - String.class, is one symbol
    - Student.class is one symbol
    - Object.class, is one symbol.
- The name of these class names are stored in **constant pool** of Test class.
- In resolution phase these names are replaced with original memory level references from method area.

2.3 Initialization

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

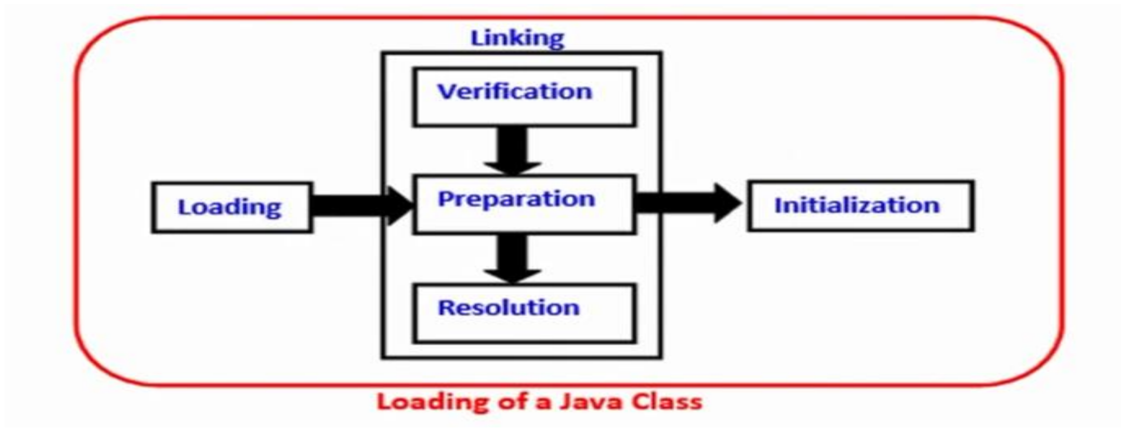


Figure : Classing loading process

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.

3 Types of Class Loaders

Class loaders sub system has following 3 class loaders -

- Bootstrap Class Loaders or Primordial Class Loader
- Extension Class Loaders
- Application Class Loaders Or System class loader

3.1 Bootstrap Class Loaders or Primordial Class Loader

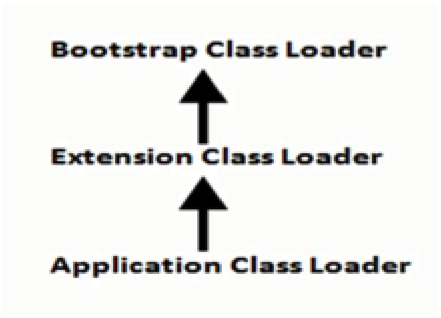
- Bootstrap Class Loaders is responsible to load all core java classes are available in “rt.jar”
- Bootstrap class path is “jdk/jre/lib/rt.jar”, So Bootstrap class loader is responsible for loading the classes from Bootstrap class path.
- Bootstrap class loader is by default available with every JVM.
- It is implemented in native languages like C, C++ and not implemented in JAVA.

3.2 Extension Class Loaders

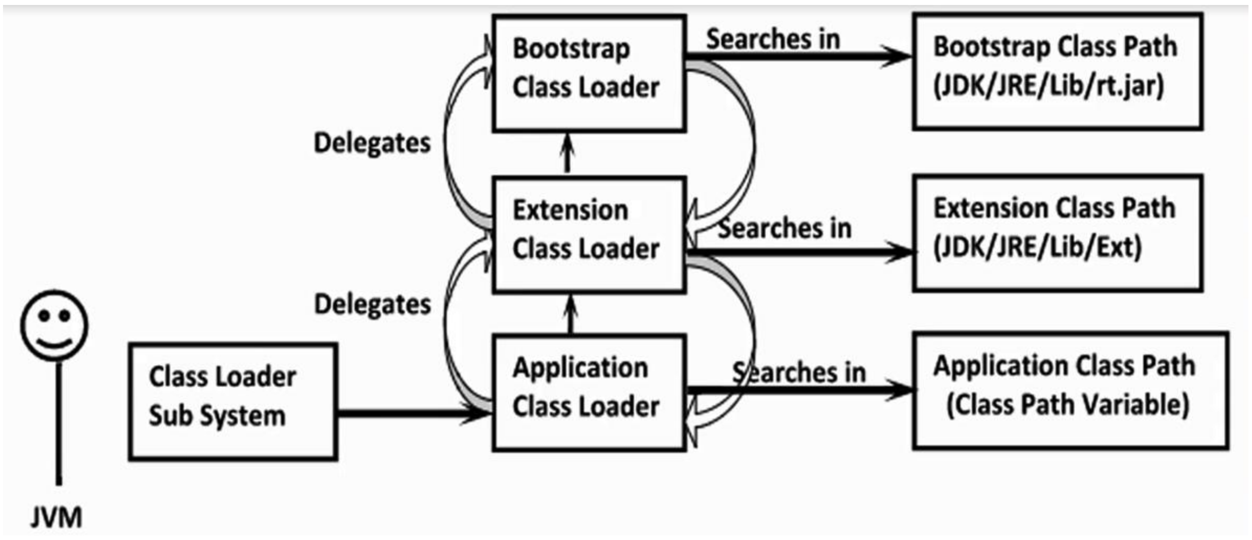
- Extension Class Loaders is child class of “Bootstrap Class Loaders”.
- Extension Class Loaders is responsible to load classes from extension class path.
- Extension class path is- “jdk/jre/lib/ext/*.jar”, so Extension class loader is responsible for loading the classes from Extension class path.
- It is implemented in JAVA, and corresponding .class files is – **“sun.misc.Launcher$ExtClassLoader.class”**

3.3 Application Class Loaders or System class loader

- Application Class Loaders is child class of “Extension Class Loaders”.
- Application class loader is responsible to load the classes from “application class path”, which we used set in environment variables.
- It internally uses environment “ClassPath”.
- It is implemented in JAVA, and corresponding .class files is – **“sun.misc.Launcher$AppClassLoader.class”**



4 How Class Loaders works



Below are the steps Class loader follows to load the classes –

1. Class loader follows delegation hierarchy principle to load the classes.
2. Whenever JVM come across a particular class, First it will check whether the corresponding class is already loaded or Not.
3. If its already loaded in method area then JVM will use that loaded class.
4. 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.
5. Application class loader delegates request to extension class loader and extension class loader in turn delegates to boot strap class loader.
6. Bootstrap class loader searches in bootstrap class path (jdk/jre/lib). If the specified class is available then it will be loaded. Otherwise bootstrap class loader delegates the request to extension class loader.
7. Extension class loader will searches in extension class path (JDK?jre/lib/ext). if the specified class is available then it will loaded. Otherwise it delegates the request to application class loader.
8. Application class loader will search in application class path, if the specified class is already available then it will be loaded otherwise it will throw " ClassNotFoundException or NoClassDefFoundError.

```
Class Test {
public static void main (){
        s.o.p(String.class.getClassLoader()); // Null, as bootstrap classloader is java not a object
        s.o.p(Test.class.getClassLoader()); // sun.misc.Launcher$AppClassLoader
        s.o.p(Customer.class.getClassLoader()); // sun.misc.Launcher$ExtClassLoader
}
}
```

Note: Assume customer.class available in both Extension and Application class path.

For String.class:

Bootstrap class loader from boot strap class path.

Output : **null**

For Test.class:

Application class loader from application class path.

Output : **sun.misc.Launcher$AppClassLoader@1234**

For Customer.class:

Extension class loader from extension class path.

Output : **sun.misc.Launcher$ExtClassLoader@1234**

Note: Bootstrap class loader is not a java object, hence we got a "null" output in first case, but extension and application class loaders are java objects hence we are getting corresponding output for the remaining s.o.p's.

Class loader sub system will get highest priority to bootstrap class path then extension class path followed by application class path.

4.1 Customized Class Loaders

Need of customized class loader –

Sometimes default class loading mechanize in doesn't meet our requirement and need to load the classes on timely basis.

Default class loading :

```
        Student s1 = new Student()
        Student s2 = new Student()
            --
            --
        Student s100 = new Student()
```

Student class loaded only once at the time first use, and next all the times its just use the loaded class.

Meanwhile, If Student class is modified then those changes will not be available to the application as it not be loaded by default class loaders.

In this context we may need to come up with

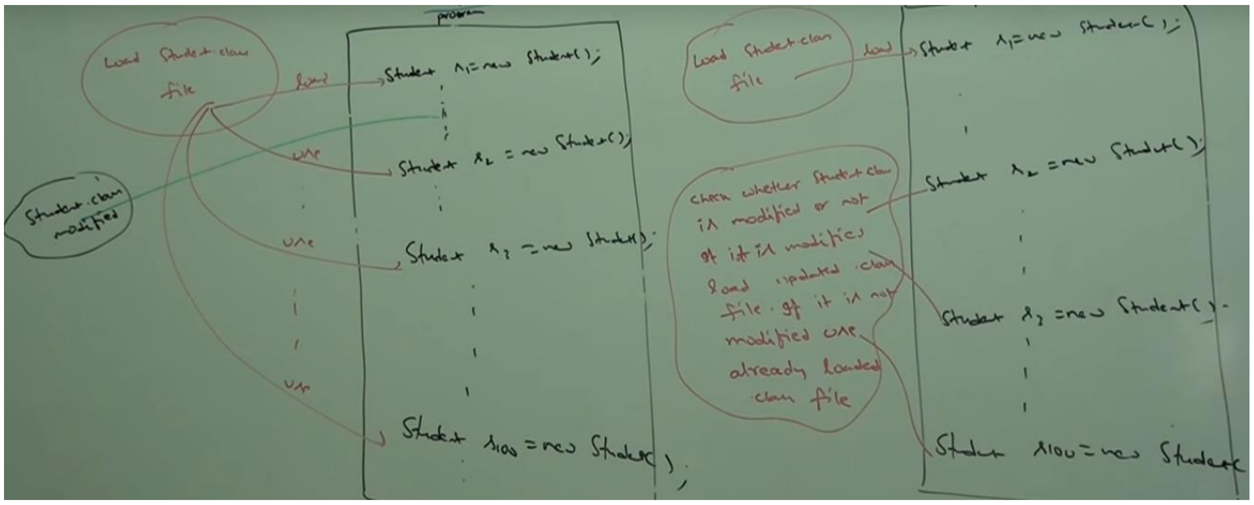
Customized class loading like below :

```
        Student s1 = new Student()
    Check whether student class is modifies, if its modifies then load modified class else use
    already loaded class
        Student s2 = new Student()
            --
            --
        Student s100 = new Student()
```

This approach is used our own JVM's servers, IDE's, or frameworks to pick up latest changes in the system.

Notes:

- 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 own't load updated version of class file (because .class is already available in method area).
- We can resolve this problem by defining out own customize 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 update version available to our program.


program
Load Student class file
load
Student s1 = new Student();
use
Student s2 = new Student();
use
Student s3 = new Student();
use
Student s100 = new Student();
Student.class modified
Load Student class file
load
Student s1 = new Student();
Check whether Student class is modified or not if it is modified load updated class file. If it is not modified use already loaded class file
Student s2 = new Student();
Student s3 = new Student();
Student s100 = new Student

4.2 Pseudo Code for Customized Class Loaders

How to define customized class loader –

- Every class loader is child class of ClassLoader.
- We can define our own customized class loader by extending ClassLoader.

```
Public class CustClassLoader extend ClassLoader {
        Public Class loadClass(String name)  throw ClassNotFoundException {
          Check for updates and load updated .class file and return corresponding .class file.
        }
}
Public Class Client {
        Public static void main(String[] s){
                Dog d =  new Dog(); ==>  Default class loader
                CustClassLoader cl = new CustClassLoader();
                cl.loadClass("Dog");  ==> Loaded by customized class loader.
        }
}
```

Note: while developing webserver and application server we will go with custom class loader to customize

What is the class loader class?

We can use class loader 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.

5 JVM Memory Areas

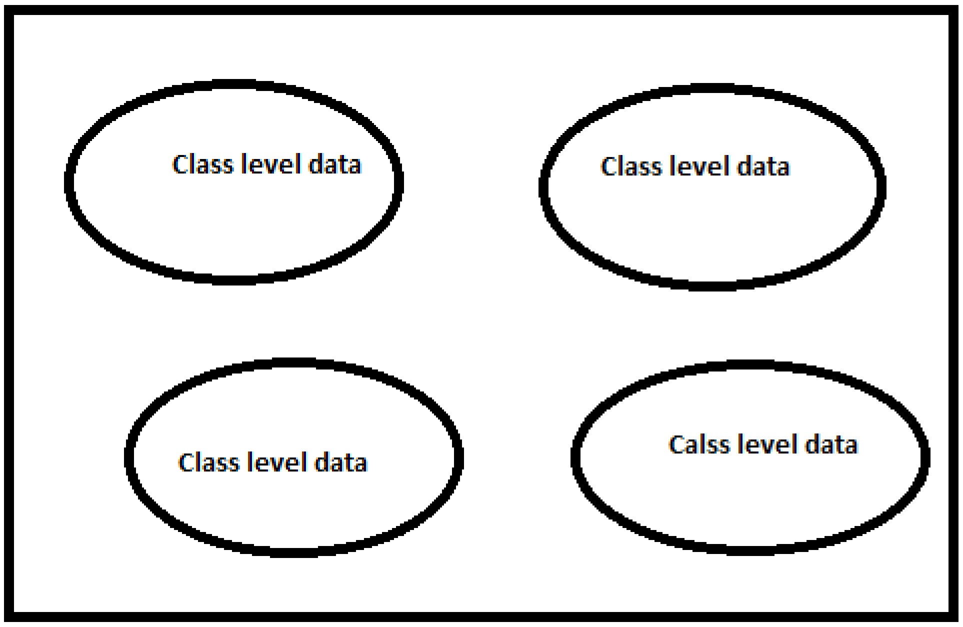
Whenever loads and runs a java program, it needs memory to store several things, like byte code, objects, variable, etc.

Total JVM memory organized into following 5 categories –

1. Method area
2. Head area
3. Stack memory
4. PC registers
5. Native method stacks

5.1 Method Area

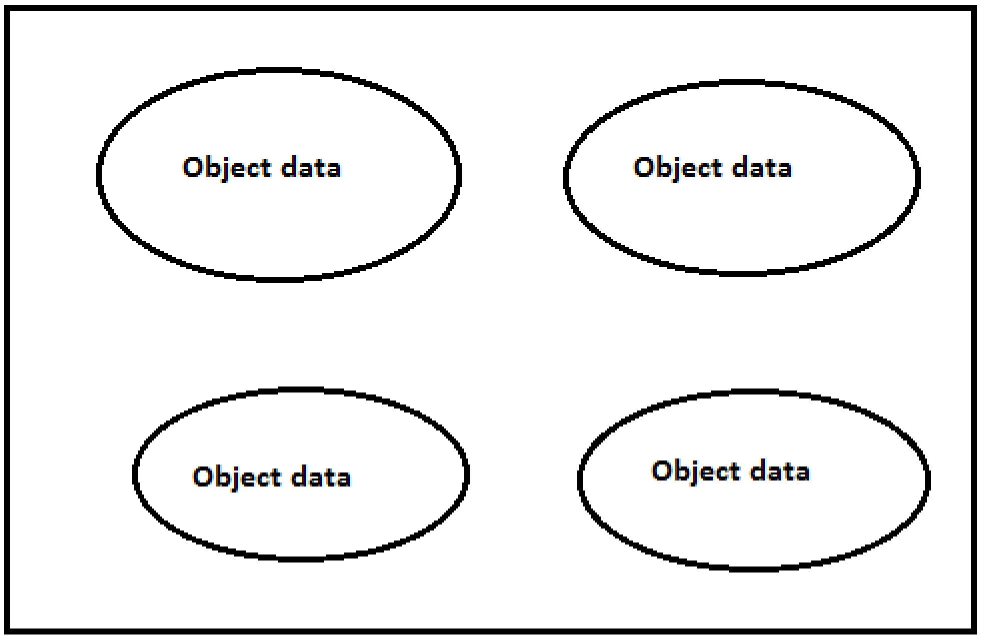
- For every JVM, one method area will be available
- Method area will be created at the time of JVM start up
- Inside method area, class level binary data including static will be stored.
- Constant pools of a class will be stored inside method area
- Method area data is not thread safe.
- Method area need not be continuous memory.



Method area

5.2 Heap Area

- For every JVM, one heap area will be available
- Heap area will be created at the time of JVM start up
- Objects and corresponding instance variables will be stored in the heap area.
- Every array in java is object only, hence arrays alsow ill be stored in heap area.
- Heap are can be accessed by multiple threads, hence it's the data stored in heap memory is not thread safe.
- Heap area is need not be continuous.



Heap Area

Program to display heap memory statistics:

- A Java application can communicate with JVM by using Runtime Object.
- Runtime Class in java.lang package, and it is a singleton class.
- We can create Runtime object, as follows.

  Runtimw r = Runtime.getRuntime()

- Once we got Runime object, we can call the following method on that object -
  r.maxMemory() – It returns number of bytes of max memory allocated to the heap.
  r.totalMemory() – It returns number of bytes of total memory allocated to the heap (initial).
  r.freeMemory() - – It returns number of bytes of free memory available in the heap.

```
Public Class HeapDemo {
        Public static void main(String[] s){
```

```
                    Double mb = 1024  * 1024;
                    Runtimw r = Runtime.getRuntime()
                    s.o.p("Max Memory" + r. maxMemory()/mb)
                    s.o.p("Total Memory" + r.totalMemory())
                    s.o.p("Free Memory" + r.freeMemory())
                    s.o.p("Consumed memory" + r.totalMemory() – r.freeMemory())
          }
    }
```

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 the max heap size (max memory)

Ex - Java **–Xmx**512m HeapDemo

–xms: we can use this command to set the min heap size (minimum/total memory/initial memory)

Java **–Xms**64m HeapDemo

5.3 Stack area

5.4 PC Registry area

5.5 Native method area

5.6 Program to display memory area

5.7 Max memory

5.8 Total memory

5.9 Free memory

5.10 How to set max and min memory sizes.

6 Execution engine

6.1 Interpreter

6.2 JIT Compiler

7 Java Native Interface

8 Complete Architecture of JVM

9 Module - IV

9.1 Class File structure

9.1.1 Magic number

9.1.2 Minor number

9.1.3 Major number