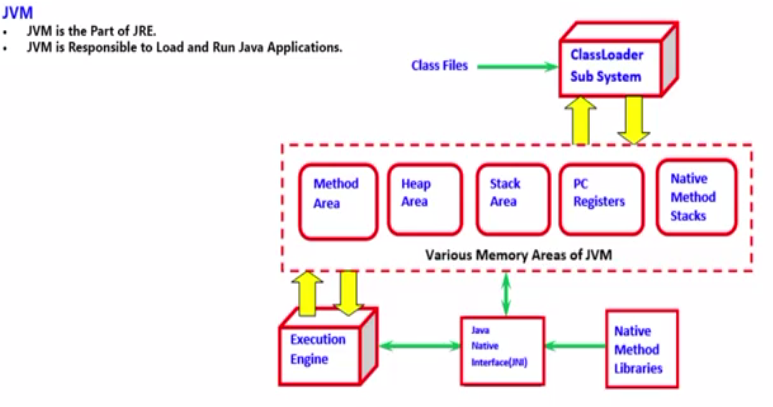
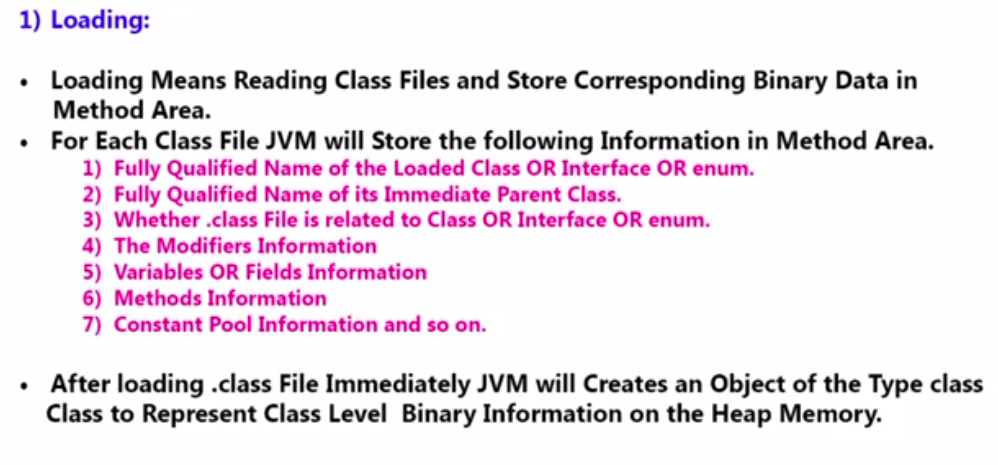
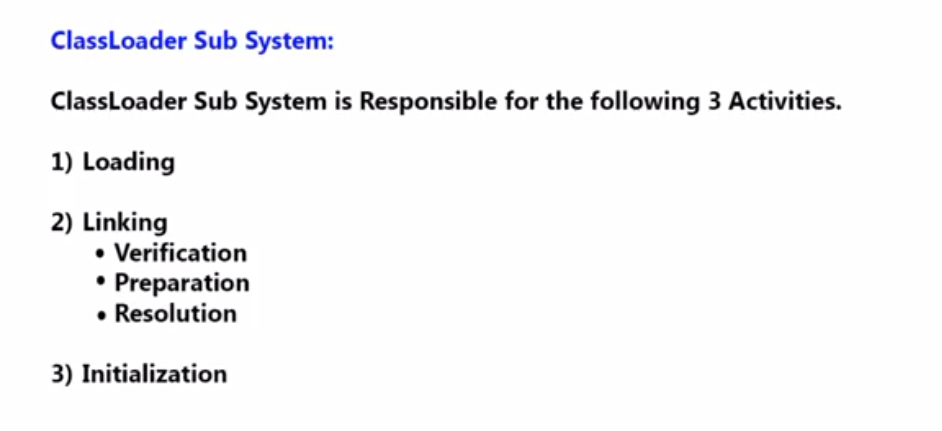


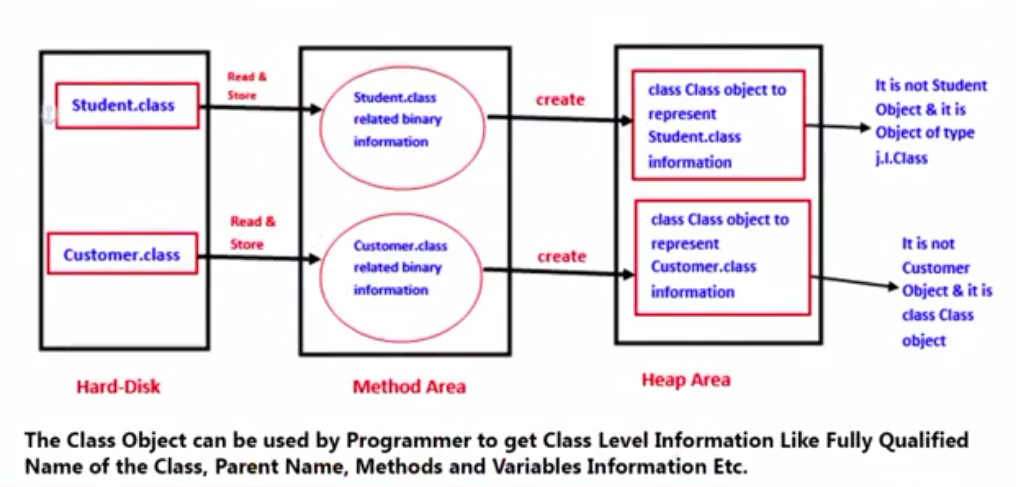
****

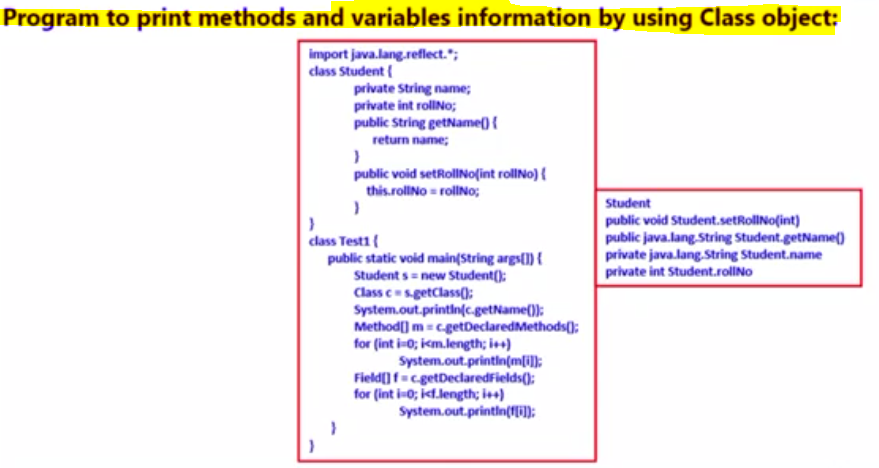
**3) Basic Architecture Diagram**

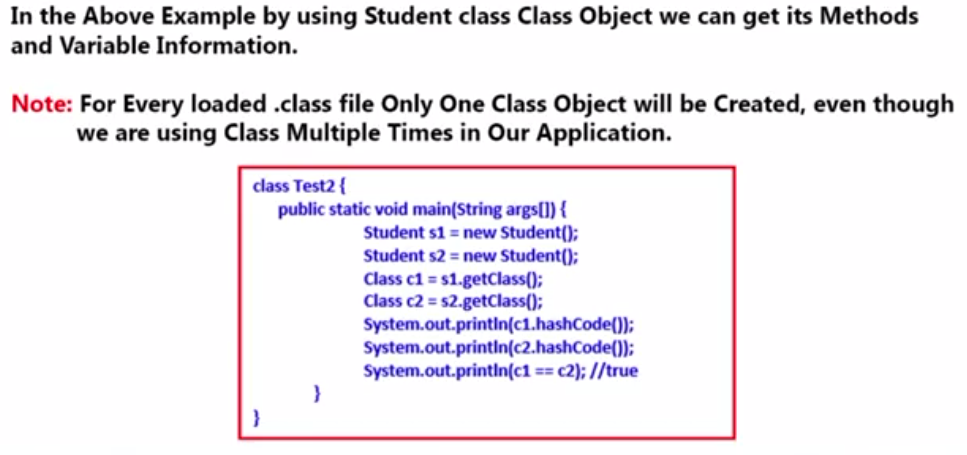


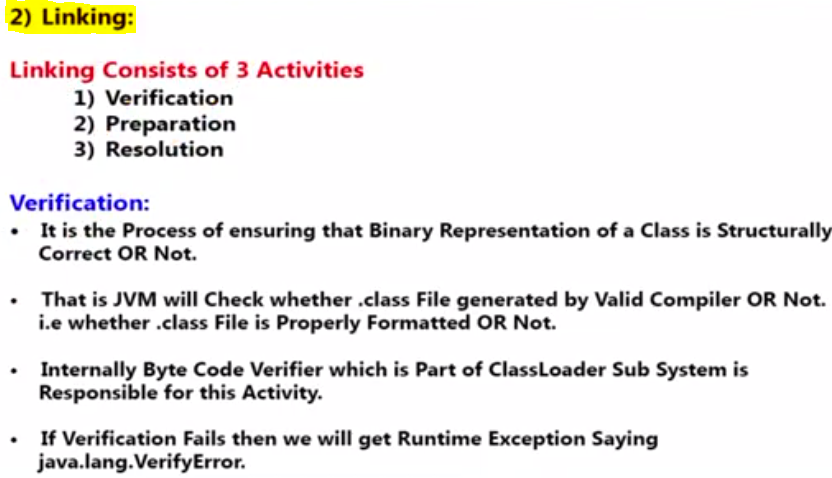
**4. Class Loader Sub System**

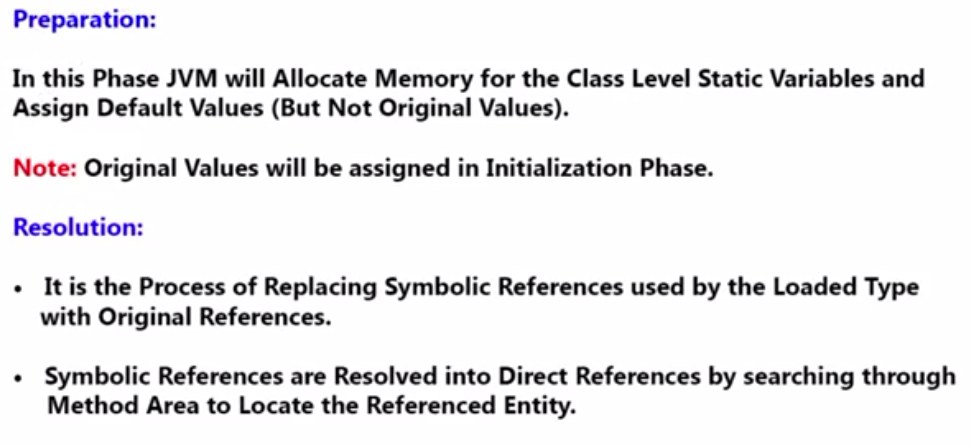


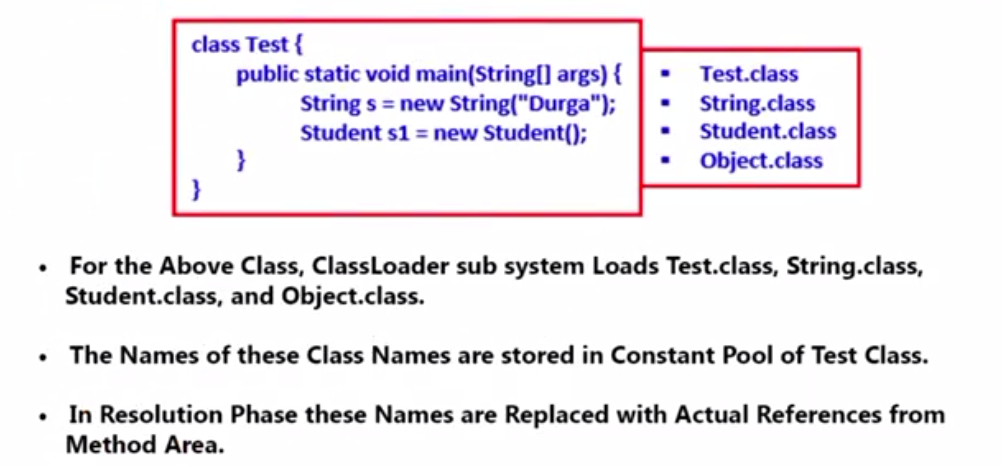


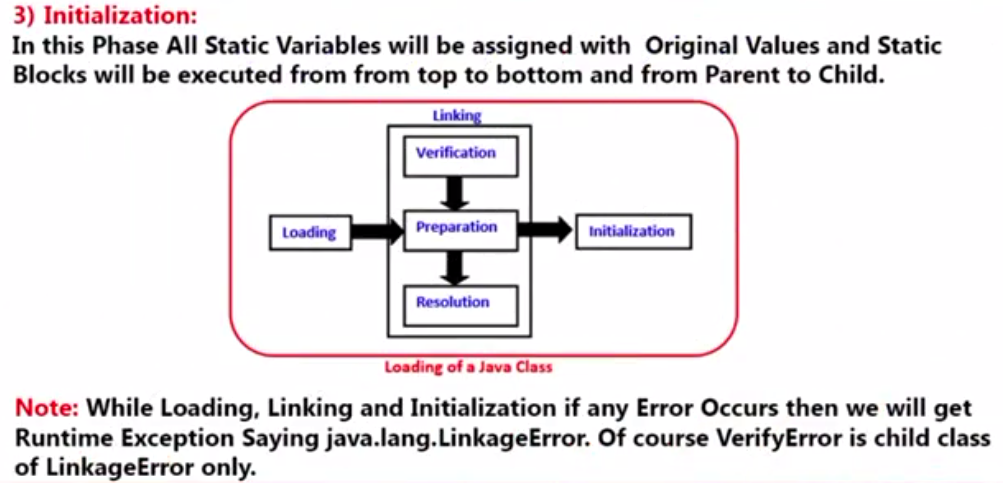












**Note:** For every loaded .class file, only **one** object of Class is created.

**4) CLASS LOADER SUBSYSTEM**

Class loader sub system contains the three parts.

1. Class Loader
2. Linking
3. Initialization

**5) Class Loader Sub system or types of Class Loader**

Class Loader sub system contains following three types of class loaders

* 1. Boot Strap class Loader or Primordial class Loader
  2. Extension class loader
  3. Application class loader or System class loader.

**1.1) Boot Strap class Loader or Primordial class Loader**

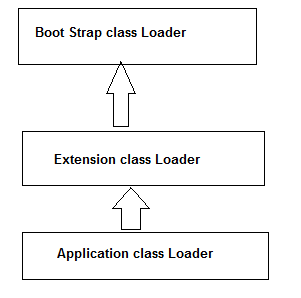
* Boot Strap class loader is responsible to load core java API classes.
* The class present in rt.jar
* In rt.jar all the core java API classes are available in it.
* JDK🡪JRE🡪LIB🡪rt.jar
* Above location is known as boot strap class path.
* Boot Strap class loader is responsible to load classes from boot strap class path.
* Boot Strap class loader is by default available with every JVM.
* It is implementing in native language like C or C++ not implemented in java.

**1.2) Extension class loader**

* Extension class is the child class of Boot strap 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 dot class file is **sun.misc.Launcher$ExtClassLoader.class.**

**1.3) 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 the environment variable class path.
* Application class loader is implemented in java and the corresponding dot class file name is **sun.misc.Launcher$AppClassLoader.class.**



**2. Linking**

Linking consists of three activities

* 1. Verify or Verification
  2. Prepare or Preparation
  3. Resolve or Resolution

**2.1) Verify or Verification**

* Inside the JVM there is a ByteCode verifier
* It is the process of insuring that binary representation of a class is structurally correct or not.
* JVM will check whether the dot class file is generated by valid compiler or not that is whether dot 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 the verification fails then we will get the run time exception saying **java.lang.VerifyError**.
* This is the one of the reasons why java is secure.

**2.2) Prepare or Preparation**

* In this process or phase JVM will allocate memory for class level static variables and assign the default values to it.
* **Note:** In initialization phase original values will be assigned to the static variables and here only default values will be assigned.

**2.3) Resolve or Resolution**

* Cannot find symbol.
* It is the process of replacing the symbolic names in our program with the original memory references from the method area.

**public** **class** Test {

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

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

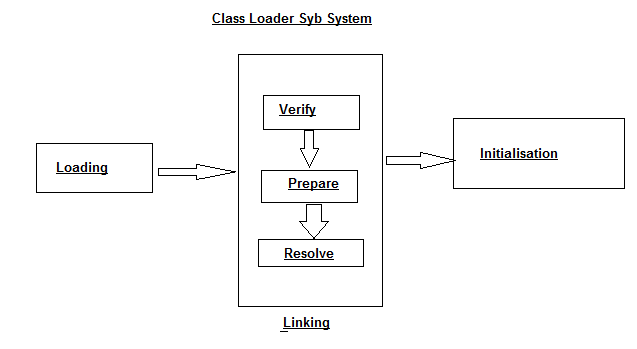
Student s1 = **new** Student();}

}

* **How many dot classes will be loaded for above program?**
* Test.class
* Object.class
* String.class
* Stduent.class
* All the above dot class is stored in the method area.
* For the above classes, class loader load the Test.class, Object.class, String.class and Student.class the names of these classes are stored in the constant pool of Test class. In the Resolution phase these names are replaced with original memory level references from the method.

**3) Initialization**

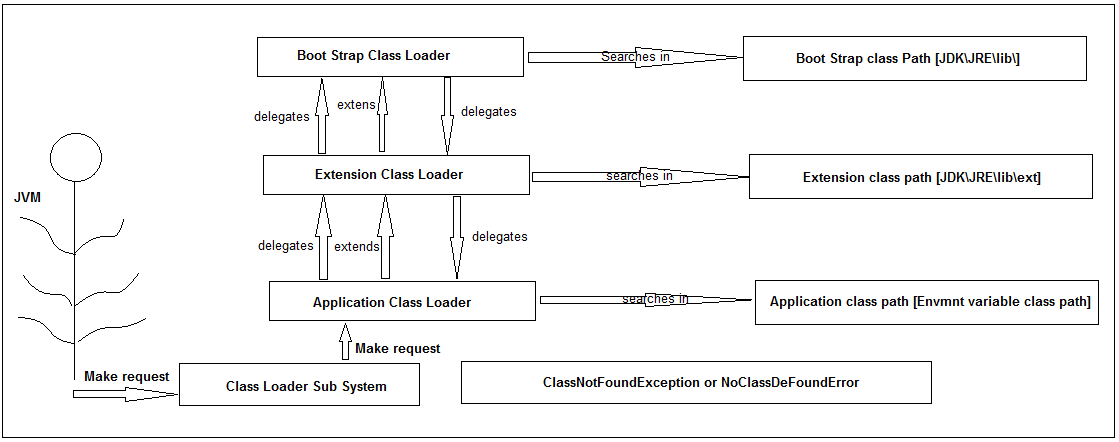
* In this phase all static variable are assigned with the original values and static block will be executed from parent to child and from top to bottom.



**Note:** 🡪 While Loading, linking and Initialization if any error occurs then we will get the Runtime Exception saying Java.lang.LinkageError.

* VerifyError is child class of LinkageError only.

**6) How java class loader or class loader sub system or class loader works?**

****

* Class loader follows the delegation hierarchy principle or algorithm.
* Whenever JVM come across for a particular class – first it will check whether the corresponding dot class file is already loaded or not if it is already loaded in the method area then JVM will consider it is the loaded class if it is not loaded then JVM request class loader sub system to load the particular class.
* Then class loader sub system handovers the request to Application class loader application class loader delegates the request to extension class loader which in turn delegates the request to boot strap class loader.
* Boot strap class loader will search the class in the boot strap class path if it is available then the corresponding class will be loaded by the boot strap class loader. If it is not available then boot strap class loader delegates the request to Extension class loader.
* Extension class loader class 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 the application class path if it is available then it will be loaded otherwise we will get run time exception saying **NoClassDefFoundError or ClassNotFoudException**.
* Class loader sub system will give the highest priority for Boot Strap class path and then Extension class path followed by Application class path.

**public** **class** Customer{ }

Javac Customer.java -- > compile it

Jar –cvf cust.jar Customer.class🡪Enter [jar creation]

**Place the cust.jar file in the jdk🡪jre🡪lib🡪ext directory**.

[E:\InstalledSofteware\Jdk\jre\lib\ext]

**public** **class** Test {

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

//which class loader is responsible to load the String class

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

//Test class is present in the current working directory

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

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

}

}

//sun.misc.Launcher$AppClassLoader@1234

//sun.misc.Launcher$ExtClassLoader@7654

**Test.java - > save it**

**Javac Test.java 🡪 compiles it**

**Java Test 🡪 runs it**

**Output**

**Null – Boot strap class is not the java object.**

[**sun.misc.Launcher$AppClassLoader@6d06d69**](mailto:sun.misc.Launcher$AppClassLoader@6d06d69)**5**

[**sun.misc.Launcher$ExtClassLoader@33909752**](mailto:sun.misc.Launcher$ExtClassLoader@33909752)

* Assume Customer. **Class** present in the both the extension and application **class** path and Test. **class** present in only application class path.
* **For String. class**

🡪 Boot Strap Class Loader from Boot Strap class path

-- O/p: null - we have got.

🡪 **For Test. class**

-- Application Class Loader from the application class path.

-- O/p: sun.misc.Launcher$AppClassLoader@6d06d69c

🡪 **For Customer. class**

-- Extension Class Loader from the Extension class path.

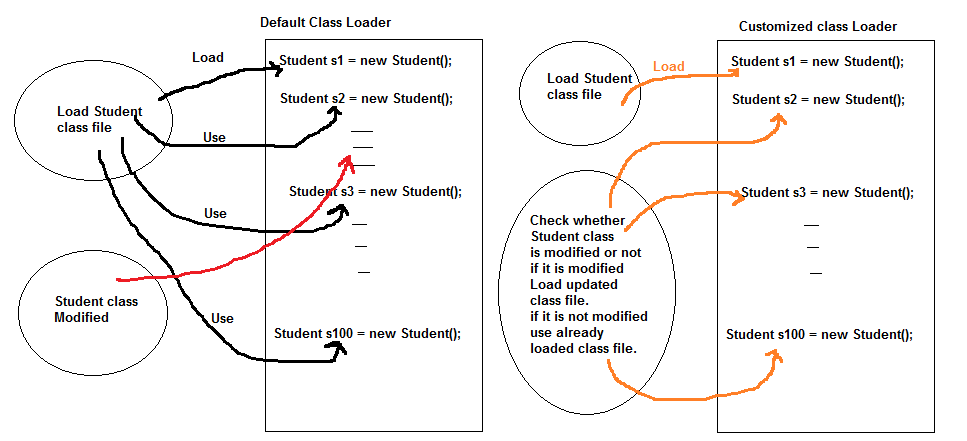
-- O/p: sun.misc.Launcher$ExtClassLoader@33909752

**Note:**

* Boot Strap is not the java object hence we got null in the first case but Extension and Application class loader are java object hence we are getting the corresponding output for the remaining two sop.
* **className@hashCode\_in\_hexadecimalform**
* Class loader sub system will give the highest priority for Boot Strap class path and then Extension class path followed by Application class path.

**7) Need of customizer class loader**

* Default class loader will load dot class file only once even though we are using multiple times that class in our program.
* After loading dot class file if it is modified outside then default class loader will not load the updated version of class file [Because that dot class file is already available in method area].
* We can resolve this problem by defining our own customizer class loader.
* The main advantage of customizer class loader is we can control the class loading mechanism based on our requirement.
* For example we can load dot class file separately every time so that updated version of class is available to our program.

🡪Need of customizer class loader if we are not satisfy with default class loader then we go for customizer class loader.

**8) Pseudo code to define customized Class Loader or How to define the customizer class loader?**

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

**public** **class** CustClassLoader **extends** ClassLoader {

**public** Class loadClass(String cname) **throws** ClassNotFoundException{

/\*

check for updates and load dot class file and

return the corresponding class

\*/

**return** **null**;

}

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

{

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

CustClassLoader classLoader = **new** CustClassLoader();

classLoader.loadClass("Dog");

//Dog will be loaded by customized class loader

--

--

classLoader.loadClass("Dog");

}

}

**class** Dog{}

* While developing web server and application server usually we can go for customizer class loader to customize the class loading mechanism.

**What is the need or use of ClassLoader class?**

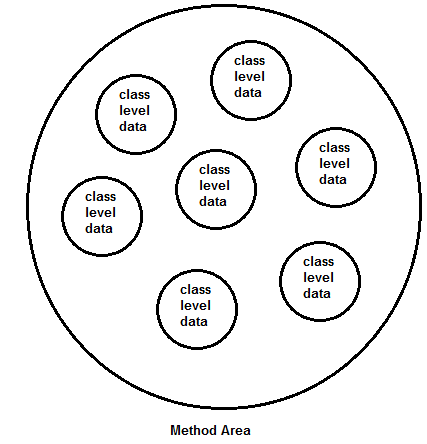
* We can use java.lang.ClassLoader class to define our own customizer class loaders.
* Every class loader in java should be child class of java.lang.ClassLoader class either directly or indirectly hence this class access base class for all the customizer class loaders.

**9) Various memory area present inside JVM**

* Whenever JVM load and run a java program it need memory to store several things like byte code, object, variable etc.
* Total JVM memory is organized into the following file categories.

1. Method Area
2. Heap Area
3. Stack Area
4. PC Registers
5. Native method stacks.
6. **Method Area**

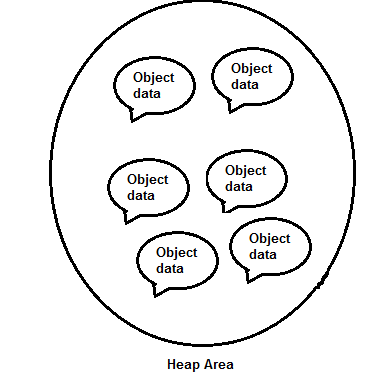
* For Every JVM one method area will be available.
* Method area will be created at the time of JVM startup.
* Inside method area class level binary data including static variable will be stored.
* Constant pools of a class will be stored in the method area.

****

* Method area can be accessed by multiple threads simultaneously.
* Its memory allocation is not continuous.
* It is not thread safe.
* JVM stores following information in method area.
* Fully qualified name of the loaded class and its immediate parent class.
* Whether *.class* file is related to Class or Interface or Enum
* Modifier, Variables and Method information etc.

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 variable will be stored in the heap area.
* Every array in java is object only hence arrays also will be stored in heap area.
* Heap area can be accessed by multiple threads and hence the data stored in the heap memory is not thread safe.
* Heap area need not be continuous.



**11) Importance of Runtime Class or program to display heap memory statistic.**

* A java application can communicate with the JVM by using runtime object.
* Runtime class present in the java.lang package and it is a single tone class.
* We can create run time object as follows.
* Runtime r = Runtime.getRuntime();
* Once we got the runtime object we can call the following methods on that object.
* **maxMemory()** – it returns no of bytes of max memory allocated to the heap
* **totalMemory()** – it return number of bytes of total memory allocated to the heap(initial memory).
* **freeMemory()** – 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("Max Memory : "+r.maxMemory());

System.***out***.println("Initial memory : "+r.totalMemory());

System.***out***.println("Free Memory : "+r.freeMemory());

System.***out***.println("Consumed Memory : "+(r.totalMemory()-r.freeMemory()));

}

}

Output

**Max Memory : 1404043264**

**Initial memory : 96468992**

**Free Memory : 94959016**

**Consumed Memory : 1509976**

/\*

1KB = 1024 bytes

1MB = 1024\*1024 bytes

\*/

In Mb

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

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

Runtime r = Runtime.*getRuntime*();

**long** mb = 1024\*1024;

System.***out***.println("Max Memory : "+r.maxMemory()/mb);

System.***out***.println("Initial memory : "+r.totalMemory()/mb);

System.***out***.println("Free Memory : "+r.freeMemory()/mb);

System.***out***.println("Consumed Memory : "+(r.totalMemory()-r.freeMemory())/mb);

}

}

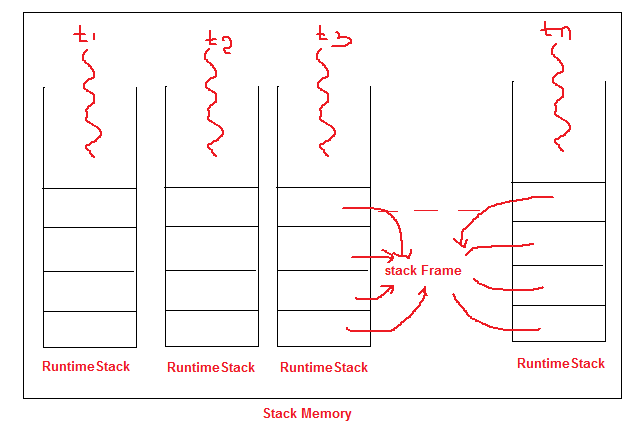
After loading *.class* file, JVM creates an object of type Class to represent this file in the heap memory. Please note that this object is of type Class predefined in *java.lang* package. This Class object can be used by the programmer for getting class level information like name of class, parent name, methods and variable information etc. To get this object reference we can use *getClass()* method of [Object](https://www.geeksforgeeks.org/object-class-in-java/) class.

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

* Heap memory is finite memory but based on our requirement we can set the maximum and minimum heap sizes that is we can increase and decrease the heap size based on our requirement.
* We can use the following flags with java command
* **-Xmx** - to set the maximum heap size [maxmemory()]
* Using command command prompt
* **Java –Xmx512m HeapDemo**
* Above command will set the maximum heap size as 512 MB
* **-Xms** – we can use this command to set the minimum heap size.
* **Java –Xms64m HeapDemo**
* To set the minimum heap size as 64MB (totalMemory()).
* **Java –Xmx512m -Xms64m HeapDemo**

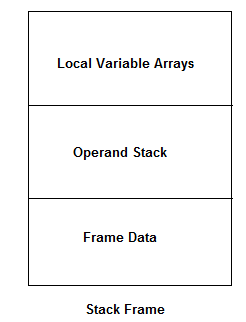
1. **Stack Area**

* For every thread JVM will creates a separate stack at the time of thread creation.
* Each and every method call performed by that thread will be stored in the corresponding stack including local variable also.
* After completing a method the corresponding entry from the stack will be removed.
* After completing all the methods calls the stack will become empty and that empty stack will be destroyed by 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 thread hence this data has thread safe.



**Stack Frame structure**

Each stack frame contains three parts



**Local Variable Arrays**

* It contains all the parameter and local variable of the method.
* Each slot in the arrays is of 4 bytes.
* Values of type int, float and reference occupy one entry in the arrays.
* Values of double and long occupy two consecutive entries in the array.
* Byte, short and char values will be converted into int type before storing and occupy one one slot.
* But the way of storing the Boolean values varied from JVM to JVM but most of the JVM followed one slot for Boolean values.

**public** **static** **void** m1(**int** i, **double** d, Object o, **float** f) {

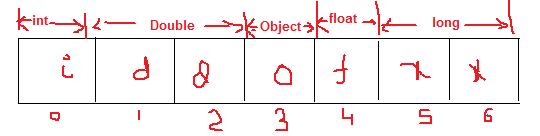
**long** x;

--

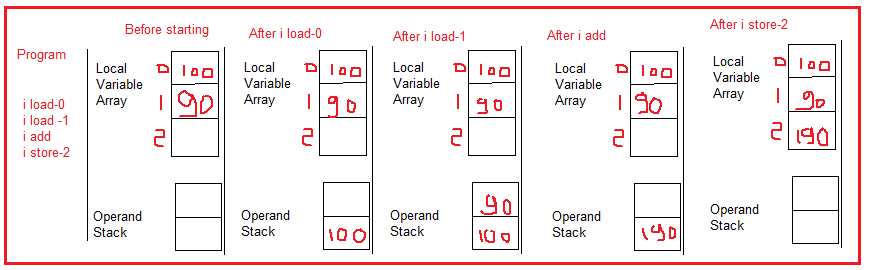
--

--

}

**Operand Stack**

* JVM uses operand stack as work space.
* Some instruction can push the value to the operand stack and some instruction can pop values from the operand stack and some instruction can perform required operations.



**Frame Data**

* Frame data contains all the symbolic reference related to that method.
* It also contains a reference to exception table which provide the corresponding catch block information in the cash of exception.

1. **PC Registers [Program counter Registers]**

* PC 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 completes automatically pc register will be incremented to hold the address of next instruction.
* Internally it is used by the JVM.

1. **Native method stacks.**

* For every thread JVM will creates a separate native method stack.
* All native method call invoked by the thread will be stored in the corresponding native method stack.

**Native method stacks: For** every thread, separate native stack is created. It stores native method information.

**Conclusion or Note:**

* 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 where as stack area, pc registered 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 variable will be stored in the method area.
* Instance variable will be stored in heap area.
* Local variable will be stored in stack area.

**public** **class** Test {

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

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

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

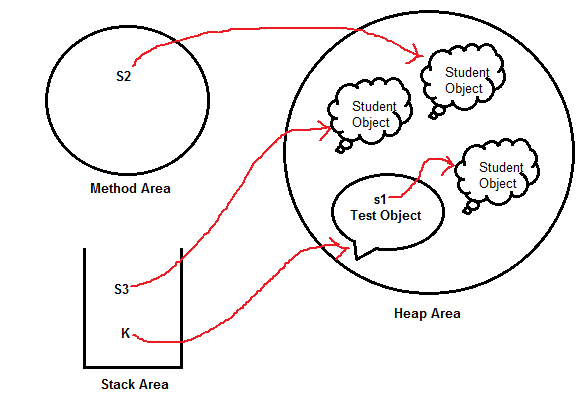
Test k = **new** Test();

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

}

}

**class** Student {}



**13) Execution Engine**

* This is the central component of JVM.
* Execution engine is responsible to execute the java class files.
* Execution engine mainly contains two components.
* Interpreter and JIT Compiler

**Interpreter**

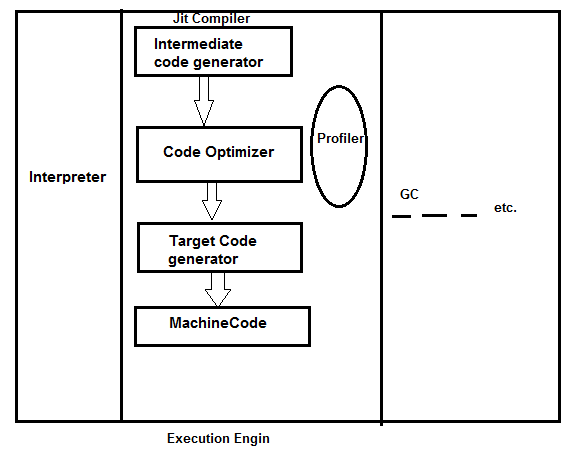
* Interpreter is responsible to read the byte code and interpret (convert) into the machine code (native code) and execute that machine code line by line.
* The problem with interpreter is it interprets every time even same method invoked multiple times which reduces the performance of the system.
* To overcome this problem sun people introduces JIT compiler in jdk 1.1 versions.

**JIT Compiler**

* The main purpose of JIT compiler is to increase performance – internally JIT compiler maintains a separate count for every method call.
* Whenever JVM come across any method call first that method will be interpreted normally by the interpreter and the JIT compiler will increase the corresponding count variable.
* This process will be continued for every method, once if any method count reaches to threshold value then the JIT compiler identify that, that method is repeatedly used method (Hot spot method).
* Immediately JIT compiler compile that method and generates the corresponding native code or machine code and next time JVM come across that method call then JVM uses the native code directly and execute it instead of interpret once again so that performance of the system will improve.
* The threshold count very to JVM to JVM.
* Some advanced JIT compiler will re-compile generated native code if count reaches threshold values second time so that more optimized machine code will be generated.
* Internally Profiler – which is the part of JIT compiler, is responsible to identify the hot spot.

**Note:**

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

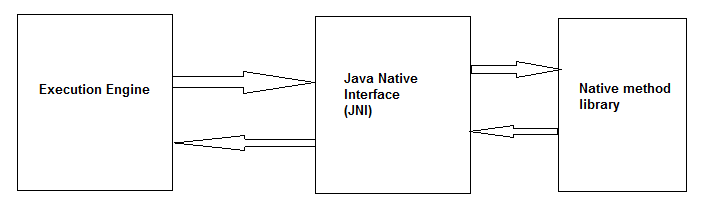


**14) Java Native Interface (JNI)**

JNI access mediator for java method calls and corresponding native libraries.

That is JNI is responsible to provide the information about the native libraries to the JVM.

Native method library provides or holds native libraries information.



**15) Class File Structure**

**class** Class {

magic\_number;

minor\_version;

major\_version;

constant\_pool\_count;

constant\_pool[];

access\_flags;

this\_class;

super\_class;

interface\_count;

**interface**[];

fields\_count;

fields[];

methods\_count;

methods[];

attributes\_count;

attributes[];

}

**Magic Number**

* The first four bytes of the class file is a magic number, this is a predefined values used by JVM to identify dot class file is generated by valid compiler or not.
* Value should be **0XCAFEBABE.**
* **Note:**
* Whenever we are executing the java file if the JVM unable to find the valid magic number then we will get run time exception saying java.langa.ClassFormatError: incompatible magic value.
* Example: Remove the first line from the any of the dot class file and then run it.
* JVM will throw runtime exception saying java.lnag.ClassFormatError.

**Minor and major version**

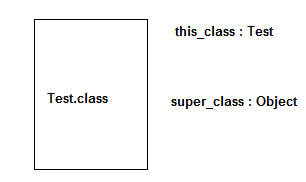
* To check the current version of JVM use bellow command.
* javac –version. [**javac 1.8.0\_151**]
* Major and minor version represents dot class file version.
* JVM will use this version to identify which version of compiler generates the current dot class file.



* **versions – 49.0** - if java file compiled with jdk 1.5 version then major versions should be 49.0 it is present in the corresponding dot class file.
* **Versions - 50.0-** if java file compiled with jdk 1.6 version then major versions should be 50.0 it is present in the corresponding dot class file.
* **Versions – 51.0-** if java file compiled with jdk 1.7 version then major versions should be 51.0 it is present in the corresponding dot class file.

**Note**

* Lower version compiler generated dot class file can be run by the higher version JVM
* But higher version compiler generated dot class file cannot be run by the lower version JVM If we are trying to run we will get run time exception saying java.lang.UnsupportedClassVersionError.
* Javac 1.6 V ------ java 1.7 – valid
* Java 1.7 V -------- java 1.6 – invalid
* **constant\_pool\_count**
* It represents number of constant present in constant pool.
* **constant\_pool[]**
* It represents information about constant present in the constant pool.
* **access\_flags**
* It provides information about the modifiers which are declared to the class.
* **this.class**
* It represents fully qualified name of the class.
* **super\_class**
* It represents fully qualified name of the immediate super class of current class.



* **interface\_count** - It return number of interfaces implemented by current class.
* **interface[]** - It returns interfaces information implemented by current class.
* **fields\_count** - It represents number of fields present in the current class – fields or nothing but static variables.
* **fields[]** - It represents field’s information presents in the current class.
* **methods\_count** - It represents number of methods presents in the current class.
* **methods[]** - It provides information about all methods present in the current class.
* **attributes\_count** - It returns number of attributes (instance variable) presents in the current class.
* **attributes[]** - It provides information about all attributes present in the current class.
* **javap –verbose Test.class**
* It gives the dot class file information at the certain level.