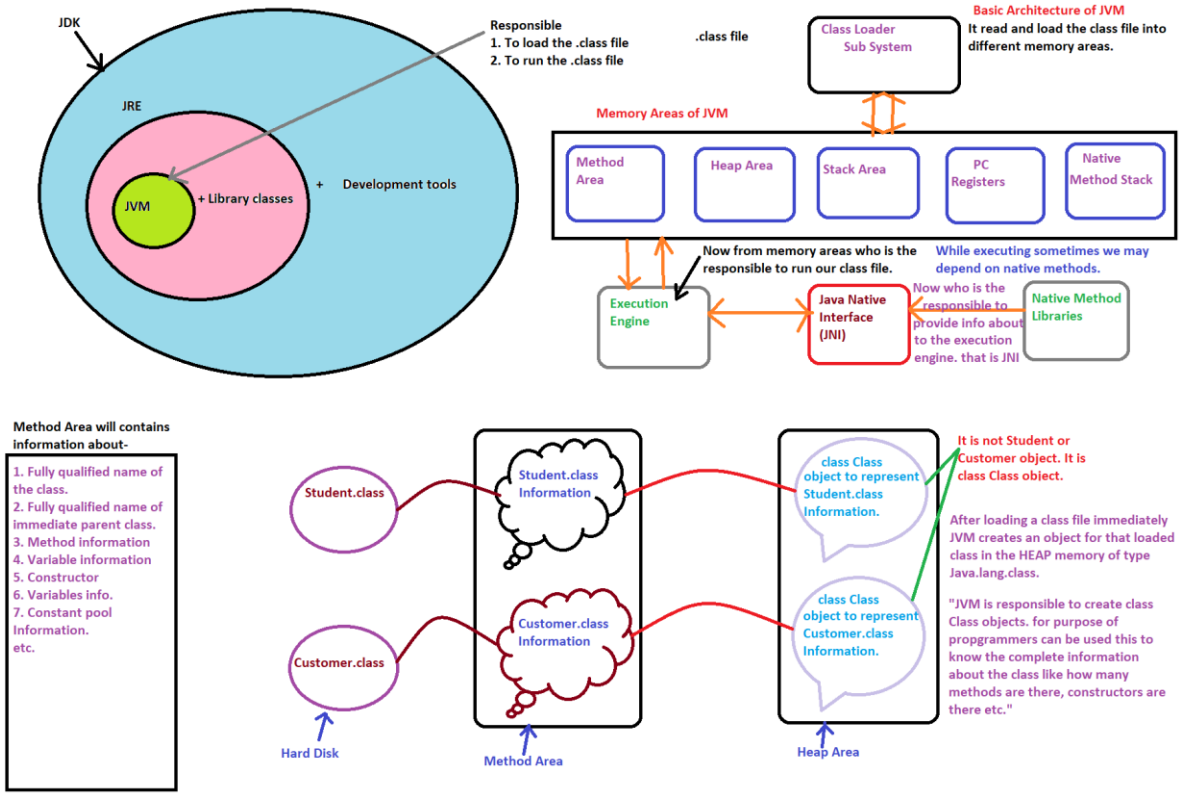
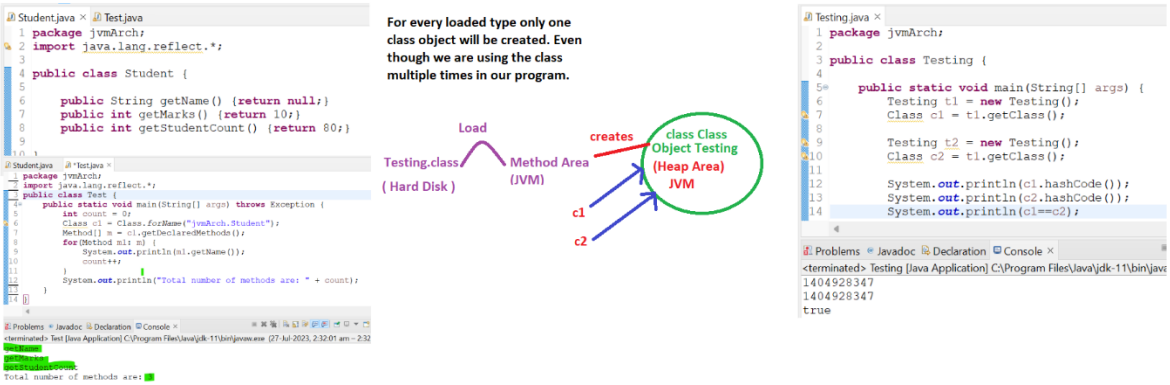
********** **JVM Architecture** **********



And How programmers can see the class information that is shown below:



Loading (Done above)

Linking

Initialization

Now we will talk about the **linking** part:

Byte Code verifier: 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 is generated by valid compiler or not that is 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 run time exception saying "java.lang.verifyerror".

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

In initialization phase original values will be assign to the static variables. And here only default values will be assigned.

Resolution or Resolve: It is the process of replacing syntax names in our program with original memory references from method area.

Class Test

{

Public static void main(String[] args){

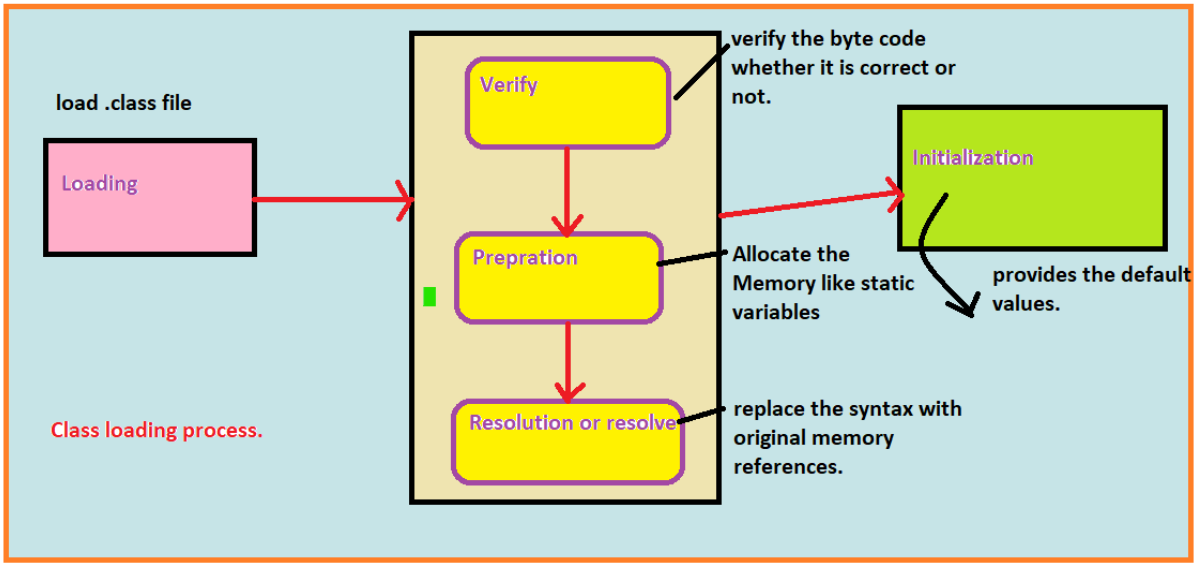
String s = new String("Himanshu");

}

}

For the above class, class loader loads Test.class, Student.class, 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.

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

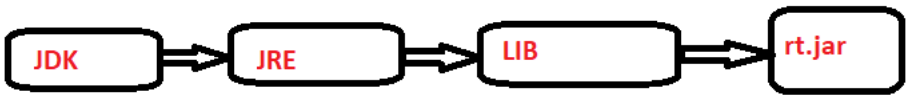


Class loader Sub System

While loading, linking and initialization if any error occurs then we will get run time sception saying "java.lang.linkagerror"

 # Types of Class loaders: It contains 3 types of class loaders

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

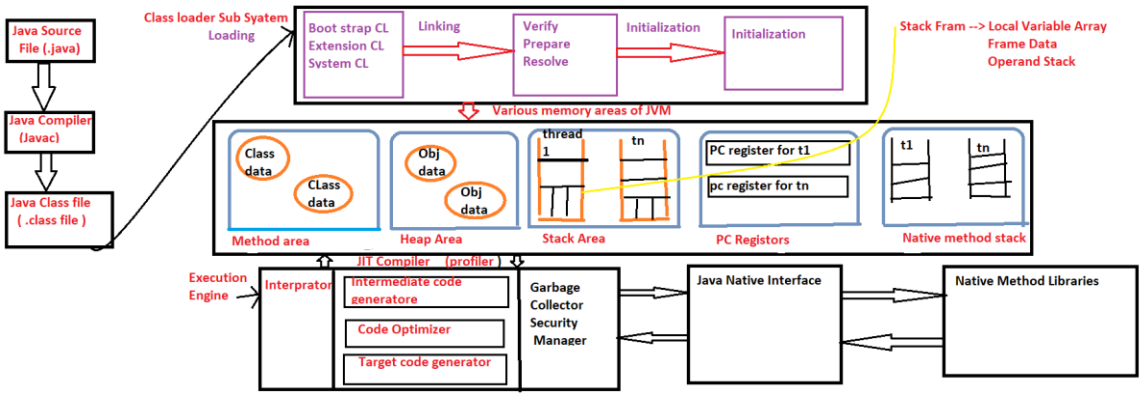


JDK\JRE\LIB\rt.jar

This location is called bootstrap class path. There is bootstrap class loader to 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.

2. **Extension class loader or Platform class loader:** Extension class loader is child class of bootstrap class loader. Extension class loader is responsible to load classes from extension class path. (JDK\JRE\LIB\EXT). Extension class loader is implemented in java and corresponding .class file is "**sun.misc.launcher$extclassloader.class**"
3. **Application class loader or System class loader:** It is the child class of extension class loader. This class loader is responsible to load classes from application class path. It internally uses environment variable class path. It is implemented in java and corresponding .class file is "**sun.misc.launcher$Appclassloader.class"**

JVM Complete Architecture:



Explanation: First I have .java file. Then my java compiler is responsible to compile the code. And responsible to generate .class file. It is byte code. So whenever we are executing, my JVM is going to take .class file as input. So, here first who is responsible to load our .class file ? That is class loader subsystem. And class loader sub system mainly responsible for 3 activities. Loading, Linking and Initialization. So next in Loading how many types of loaders are there. These are 3 types of loaders.

1. **Boot Strap class loader** 2. Extension class loader 3. System or Application class loader.
   And Boot strap CL is responsible to load class from boot strap class path. Boot strap class path means "rt.jar". All core java API classes is going to take care by Boot strap class loader.
2. **Extension class loader**: The classes present inside ext folder like JDK, JRE, lib etc.
3. **System or Application class loader:** It is responsible to load classes from application-level class path.

After this loading process immediately verification process will happen. Byte code verifier is going to verify whether your generated byte code is proper or not. It is generated by valid compiler or not. If verification fails then immediately you will get verify error.

In preparation part for static variables memory will be allocated and default values are assigned. But original values for static variables will be place in initialization part.

And in resolve part all syntax references will replaced with original references from method area.

In **Initialization** original static values will be assigned and static block will be executed. Now after initialization class loading sub system completed successfully.

Now there are 5 memory areas are present in JVM.

Method Area: In method are class level data will be present including static variables.

Heap Area: In heap area object data and corresponding instance variables will be present.

Stack Area: For every JVM there will be one method area and one heap area only but for every thread a separate runtime stack will be there. And all local variables will be created inside Stack memory. For every method call one entry will be stored in the stack. And that entry is consider by default as Stack frame. And each stack frame contains 3 parts. First one is “Local variable array” , so related to that method how many local variables are there and corresponding values will be saved in local variable array.

For every thread one runtime stack is there. Let’s say in JVM 10 threads are executing then 10 stacks are there. But for 10 threads how many heap and method areas are there only one method and heap area. So, heap area and method area are the shared memory for these 10 threads. That is why the data which is stored in method and heap area is not thread safe. Because this data can be accessed by multiple threads. But for every thread there is separate runtime stack is there. So, the data which is stored inside stack memory is always thread safe. Because data can be accessed by only one particular thread.

Next for every thread one PC register will be created to hold address of next executing instruction. And to hold native methods information native method stack by default will be there.

So, this is all about various memory areas. But for programmers’ point of view 3 areas are important i.e., method area, heap area, stack area.

Now our .class file is loaded. Now who is responsible to execute our class file ? that is execution engine.

It contains Interpreter that is responsible to read, interpret and execute java program line by line. If any method repeatedly required multiple times, then instead of interpreting every time just compile once into machine code, then next time directly, we can use that machine code and who is responsible for this. JIT compiler. JIT compiler concept required only for repeatedly required methods not for every method. So, while executing sometimes we required some native method information and it is provided by Java Native Interface.