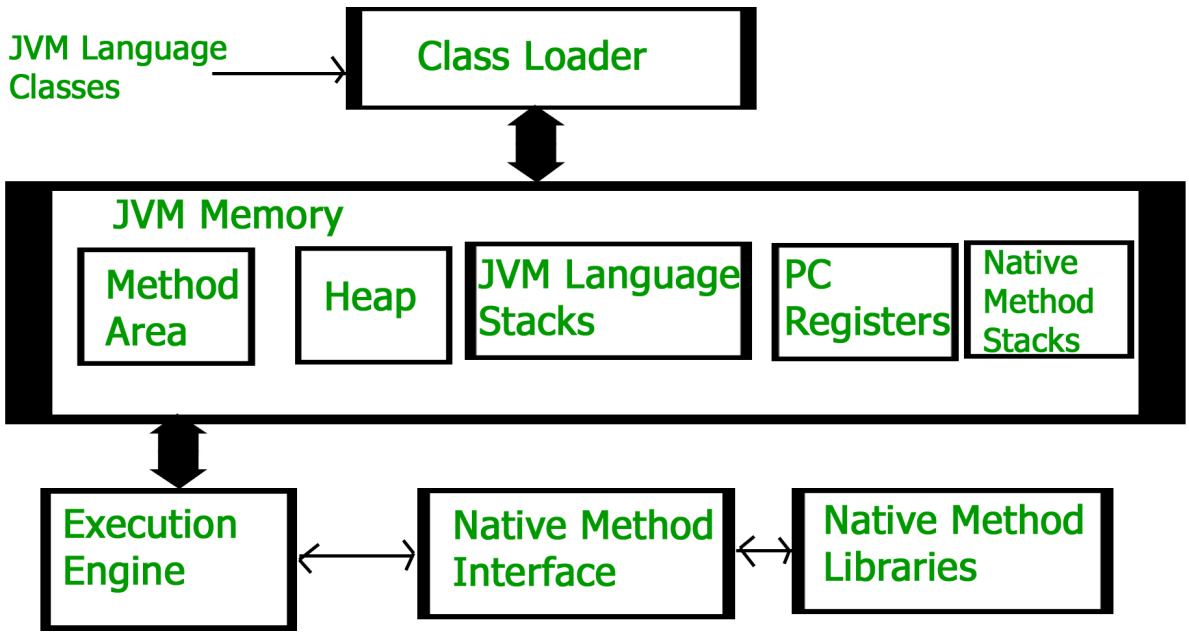
RJ-101 Day 1

How JVM Works – JVM Architecture?

JVM(Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the **main** method present in a java code. JVM is a part of JRE(Java Runtime Environment).

Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java-enabled system without any adjustment. This is all possible because of JVM.

When we compile a *.java* file, *.class* files(contains byte-code) with the same class names present in *.java* file are generated by the Java compiler. This *.class* file goes into various steps when we run it. These steps together describe the whole JVM.



Class Loader Subsystem

It is mainly responsible for three activities.

- Loading

- Linking
- Initialization

Loading: The Class loader reads the "*.class"* file, generate the corresponding binary data and save it in the method area. For each "*.class"* file, JVM stores the following information in the method area.

- The fully qualified name of the loaded class and its immediate parent class.
- Whether the "*.class"* file is related to Class or Interface or Enum.
- Modifier, Variables and Method information etc.

After loading the "*.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. These Class object can be used by the programmer for getting class level information like the name of the class, parent name, methods and variable information etc. To get this object reference we can use *getClass()* method of **Object** class.

```
Programming Example here…
```

Linking: Performs verification, preparation, and (optionally) resolution.

- *Verification*: It ensures the correctness of the *.class* file i.e. it checks whether this file is properly formatted and generated by a valid compiler or not. If verification fails, we get run-time exception *java.lang.VerifyError*. This activity is done by the component ByteCodeVerifier. Once this activity is completed then the class file is ready for compilation.
- *Preparation*: JVM allocates memory for class static variables and initializing the memory to default values.
- *Resolution*: It is the process of replacing symbolic references from the type with direct references. It is done by searching into the method area to locate the referenced entity.

Initialization: In this phase, all static variables are assigned with their values defined in the code and static block(if any). This is executed from top to bottom in a class and from parent to child in the class hierarchy. In general, there are three class loaders :

- *Bootstrap class loader*: Every JVM implementation must have a bootstrap class loader, capable of loading trusted classes. It loads core java API classes present in

the “*JAVA_HOME/jre/lib”* directory. This path is popularly known as the bootstrap path. It is implemented in native languages like C, C++.

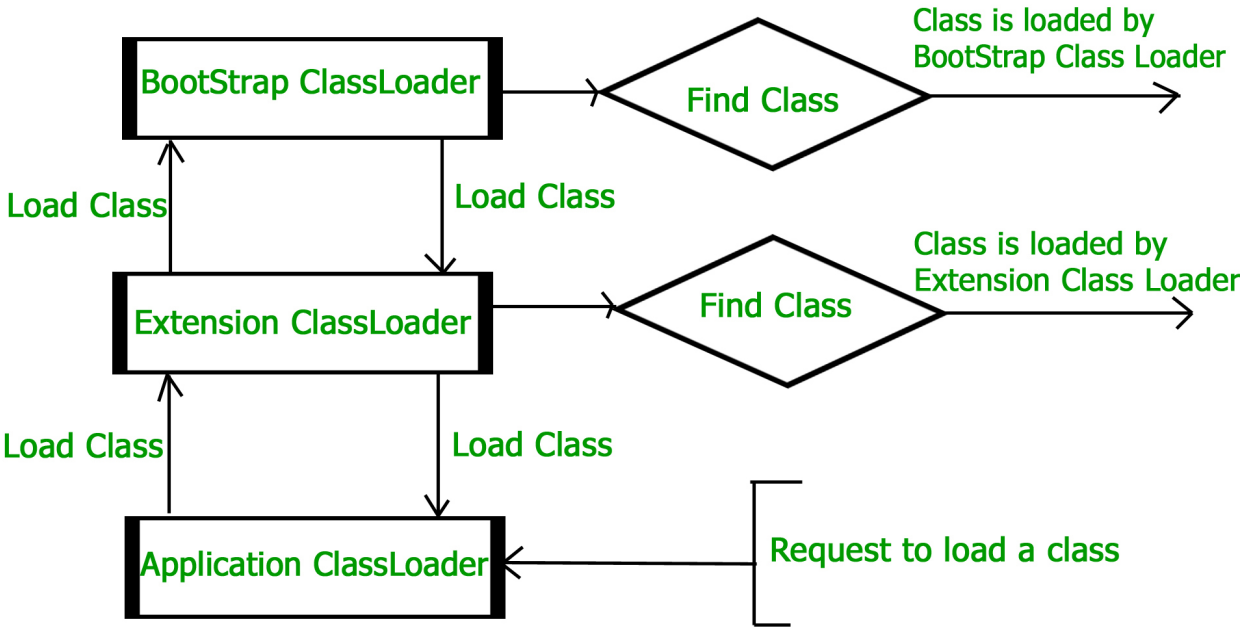
- *Extension class loader*: It is a child of the bootstrap class loader. It loads the classes present in the extensions directories “*JAVA_HOME/jre/lib/ext”*(Extension path) or any other directory specified by the java.ext.dirs system property. It is implemented in java by the *sun.misc.Launcher$ExtClassLoader* class.
- *System/Application class loader*: It is a child of the extension class loader. It is responsible to load classes from the application classpath. It internally uses Environment Variable which mapped to java.class.path. It is also implemented in Java by the *sun.misc.Launcher$AppClassLoader* class.

`Lets see an example here…`

Note:

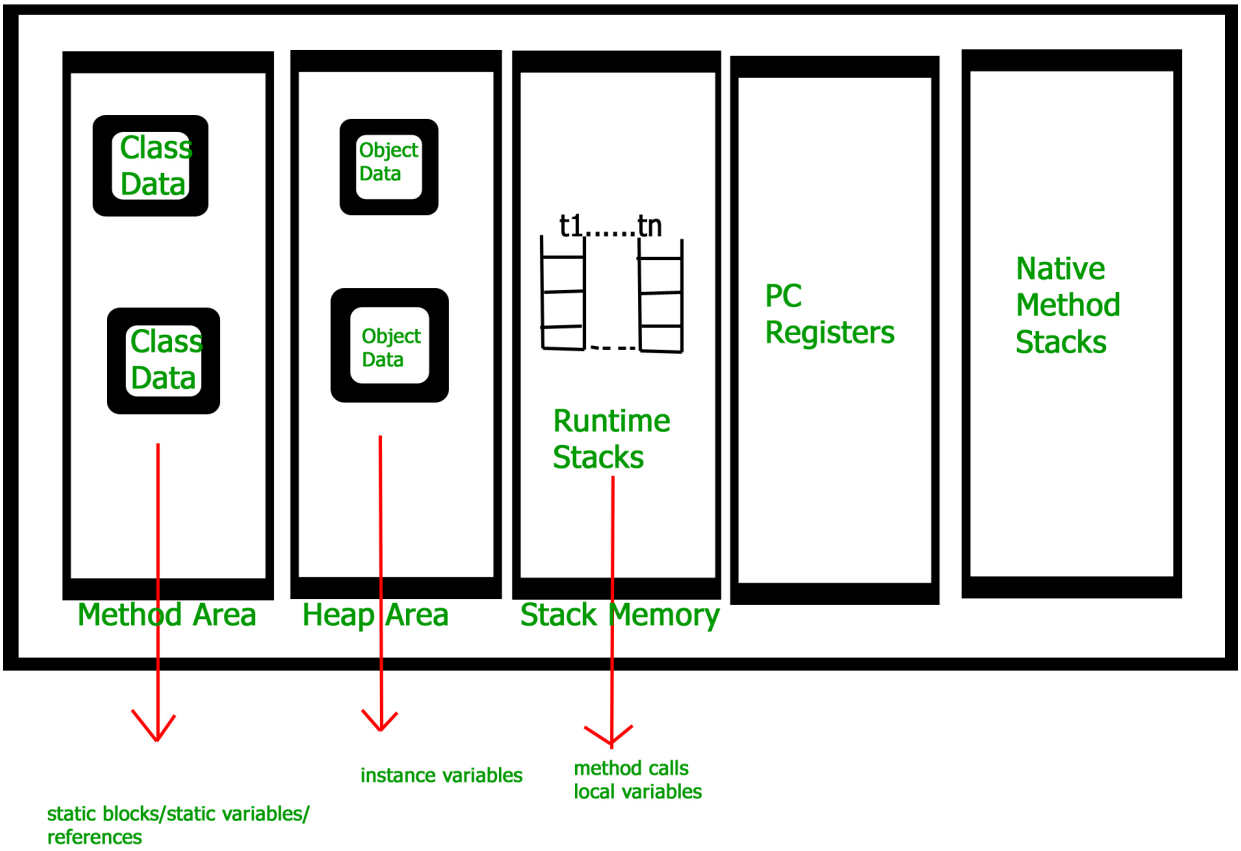
JVM follows the Delegation-Hierarchy principle to load classes. System class loader delegate load request to extension class loader and extension class loader delegate request to the bootstrap class loader. If a class found in the boot-strap path, the class is loaded otherwise request again transfers to the extension class loader and then to the system class loader. At last, if the system class loader fails to load class, then we get run-time exception *java.lang.ClassNotFoundException*

.



JVM Memory

1. **Method area:** In the method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource. From java 8, static variables are now stored in Heap area.
2. **Heap area:** Information of all objects is stored in the heap area. There is also one Heap Area per JVM. It is also a shared resource.
3. **Stack area:** For every thread, JVM creates one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which stores methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminates, its run-time stack will be destroyed by JVM. It is not a shared resource.
4. **PC Registers:** Store address of current execution instruction of a thread. Obviously, each thread has separate PC Registers.
5. **Native method stacks:** For every thread, a separate native stack is created. It stores native method information.



Execution Engine

Execution engine executes the "*.class"* (bytecode). It reads the byte-code line by line, uses data and information present in various memory area and executes instructions. It can be classified into three parts:

- *Interpreter*: It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
- *Just-In-Time Compiler(JIT)* : It is used to increase the efficiency of an interpreter. It compiles the entire bytecode and changes it to native code so whenever the interpreter sees repeated method calls, JIT provides direct native code for that part so re-interpretation is not required, thus efficiency is improved.
- *Garbage Collector*: It destroys un-referenced objects. For more on Garbage Collector, refer **Garbage Collector**.

Java Native Interface (JNI) :

It is an interface that interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

Native Method Libraries :

It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.

Now.. let’s understand the basics of java

→ Know your data types (premitives)

→ Operator and precedence of operators (may get problems in written tests)

→ Understand basic contructs (if-else, switch-case, loops)

→ Also update advanced api after java 8 (traversing array, list using forEach( ) etc)

→ Understand String class and practice tricky questions from string operations

- String reverse (with and without api)
- Character search or character replacement from given String (with and without inbuilt methods)
- String class in immutable. How to work with mutable string and why string is mutable.
- Then how to create your own immutable class

Immutable Objects

Immutable class in java means that once an object is created, we cannot change its content. In Java, all the **wrapper classes** (like Integer, Boolean, Byte, Short) and String class is immutable. We can create our own immutable class as well. Prior to going ahead do go through characteristics of immutability in order to have a good understanding while implementing the same. Following are the requirements:

- The class must be declared as final so that child classes can’t be created.
- Data members in the class must be declared private so that direct access is not allowed.
- Data members in the class must be declared as final so that we can’t change the value of it after object creation.
- A parameterized constructor should initialize all the fields performing a deep copy so that data members can’t be modified with an object reference.
- Deep Copy of objects should be performed in the getter methods to return a copy rather than returning the actual object reference)

> Note: There should be no setters or in simpler terms, there should be no option to change the value of the instance variable.

`Example here …`

Singleton Objects also important..

How to create Singleton object

In object-oriented programming, a singleton class is a class that can have only one object (an instance of the class) at a time. After the first time, if we try to instantiate the Singleton class, the new variable also points to the first instance created. So whatever modifications we do to any variable inside the class through any instance, affects the variable of the single instance created and is visible if we access that variable through any variable of that class type defined.

Remember the key points while defining class as a singleton class that is while designing a singleton class:

1. Make a constructor private.
2. Write a static method that has the return type object of this singleton class. Here, the concept of Lazy initialization is used to write this static method.

Purpose of Singleton Class

The primary purpose of a Singleton class is to restrict the limit of the number of object creation to only one. This often ensures that there is access control to resources, for example, socket or database connection.

The memory space wastage does not occur with the use of the singleton class because it restricts the instance creation. As the object creation will take place only once instead of creating it each time a new request is made.

We can use this single object repeatedly as per the requirements. This is the reason why the multi-threaded and database applications mostly make use of the Singleton pattern in Java for caching, logging, thread pooling, configuration settings, and much more.

For example, there is a license with us, and we have only one database connection or suppose if our JDBC driver does not allow us to do multithreading, then Singleton class comes into the picture and makes sure that at a time, only a single connection or a single thread can access the connection.

How to Design/Create a Singleton Class in Java?

To create a singleton class, we must follow the steps, given below:

1. Ensure that only one instance of the class exists.

2. Provide global access to that instance by

- Declaring all constructors of the class to be private.
- Providing a static method that returns a reference to the instance. The lazy initialization concept is used to write the static methods.
- The instance is stored as a private static variable.

```
Lets create an example of it..!
```