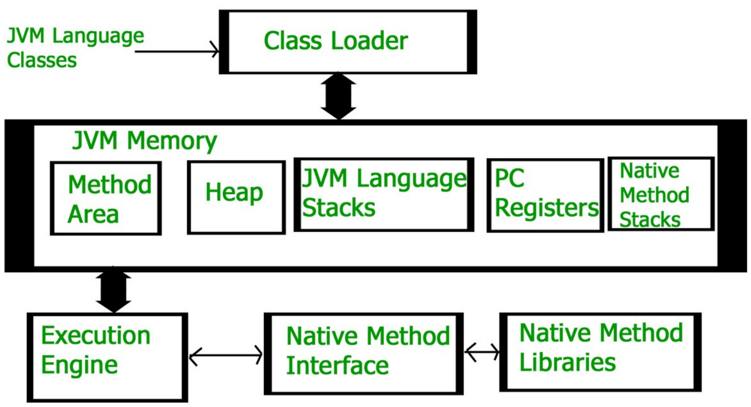
How JVM Works – JVM Architecture ?

- 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, a ***.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.



A. Class Loader Subsystem - It is mainly responsible for three activities -

1. **Loading** : The Class loader reads the **.class** file, generates the corresponding binary data and save it in **method area**.

- For each .class file, 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.
- After loading .class file, JVM creates an object of **type Class** to represent this file in the **heap memory**. 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** class. Example:

```
Student s1 = new Student();        Class c1 = s1.getClass();

String name = c1.getName();        Method m[] = c1.getDeclaredMethods();
```

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

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

- Verification : It ensures the **correctness** of **.class** file i.e. it check whether this file is **properly formatted** and generated by **valid compiler** or not. If verification fails, we get **run-time exception java.lang.VerifyError**.

- Preparation : JVM **allocates** memory for **class 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 **method area** to locate the referenced entity.

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

- Extension class loader : It is **child** of **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 **child** of **extension class loader**. It is responsible to **load classes from application class path**. It internally uses Environment Variable which is mapped to java.class.path. It is also **implemented in Java** by the sun.misc.Launcher$AppClassLoader class.

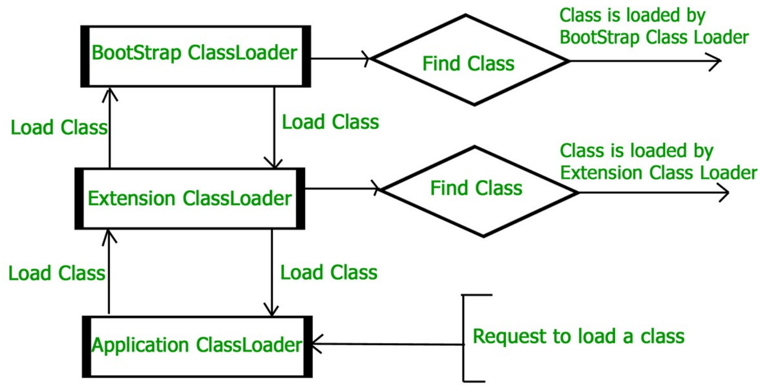
```
// Java code to demonstrate Class Loader subsystem

public class Test {
   public static void main(String[] args) {
      // String class is loaded by bootstrap loader, and bootstrap loader is not Java object, hence null
      System.out.println(String.class.getClassLoader());

      // Test class is loaded by Application loader
      System.out.println(Test.class.getClassLoader());
   }
}
```

Output:
null
sun.misc.Launcher$AppClassLoader@73d16e93

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



<u>B. JVM Memory</u> -

1. <u>Method area</u> :
   - In 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.
   - It is a **shared resource**.
2. <u>Heap area</u> :
   - Information of **all objects** is stored in heap area.
   - There is also **one Heap Area** per JVM.
   - It is also a **shared resource**.
3. <u>Stack area</u> :
   - For **every thread**, JVM create **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.

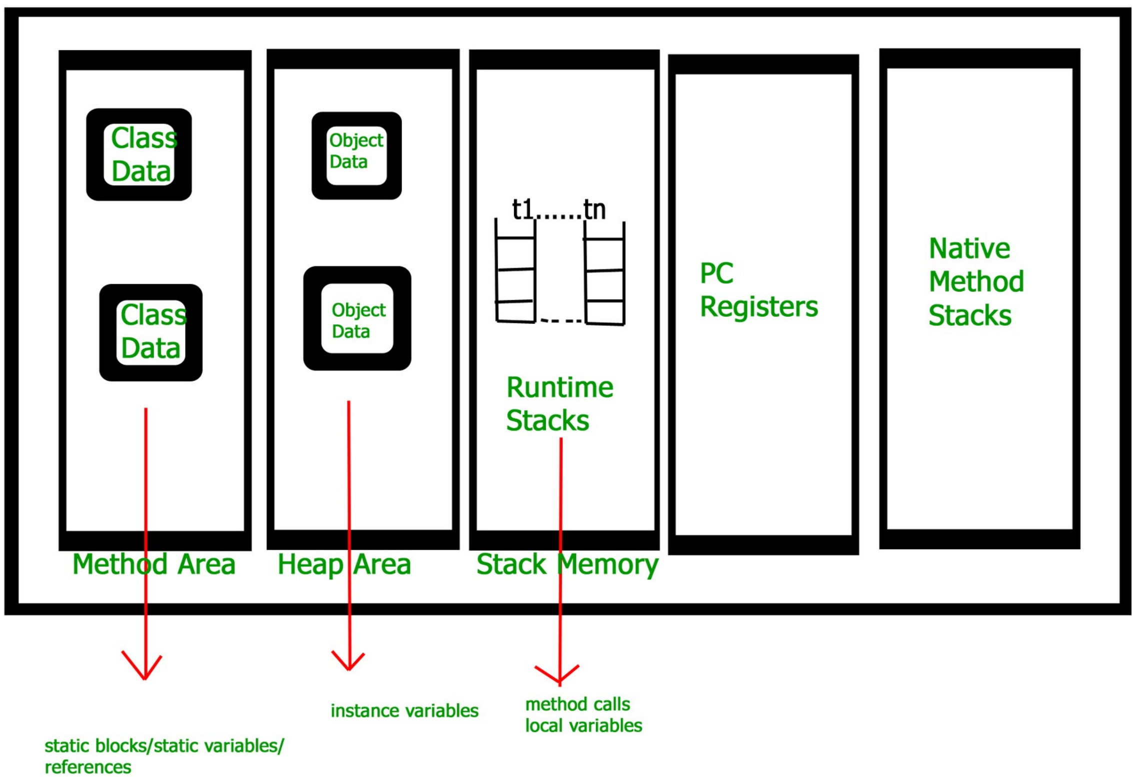
- After a thread terminate, it's run-time stack will be destroyed by JVM.
- It is **not** a shared resource.

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, **separate native stack** is created. It stores native method information.



C. Execution Engine - Execution engine execute the .class (bytecode). It reads the byte-code line by line, use data and information present in various memory area and execute instructions. It can be classified in three parts :-

1. 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.

2. Just-In-Time Compiler(JIT) :

- It is used to **increase efficiency** of interpreter.
- It compiles the entire bytecode and changes it to native code so whenever interpreter see **repeated method calls**, JIT provide direct native code for that part so **re-interpretation** is **not required**,thus efficiency is improved.

3. Garbage Collector :

- It **destroys** un-referenced objects.

D. Java Native Interface (JNI) : It is an **interface** which **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.

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