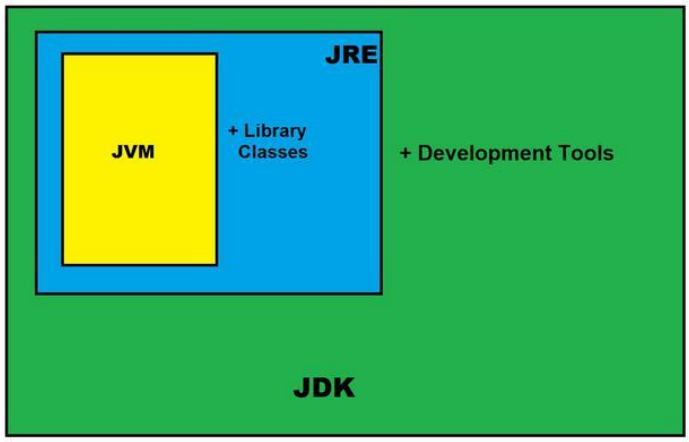
Question 1, what is **java?**

Java is a popular high-level, Java is a popular high-level, object-oriented programming language, which was originally developed by Sun Microsystems and released in 1995. Currently, Java is owned by Oracle, Today Java is being used to develop numerous types of software applications including Desktop Apps, Mobile apps, Web apps, Games, and much more. language, which was originally developed by Sun Microsystems and released in 1995. Currently, Java is owned by Oracle, And Java is being used to develop numerous types of software applications including Desktop Apps, Mobile apps, Web apps.

Question 2, **jdk, jre, jvm?**



JDK (Java Development Kit) is a Kit that provides the environment to develop and execute(run) the Java program. JDK is a kit (or package) that includes two things.

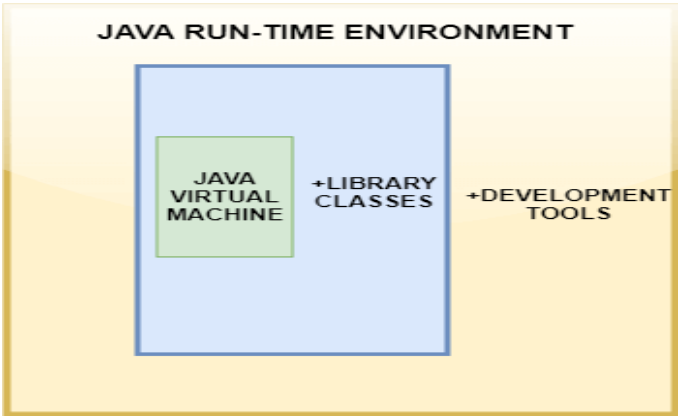
- Development Tools (to provide an environment to develop your java programs)
- JRE (to execute your java program

JRE (Java Runtime Environment) is an installation package that provides an environment to only run (not develop) the java program (or application) onto your machine. JRE is only used by those who only want to run Java programs that are end-users of your system.

JRE = JVM + CLASS LIBRARIES + (AND OTHER COMPONENT THAT ARE REQUIRED TO RUN JAVA APPLICATIONS)

- JRE is the superset of JVM.

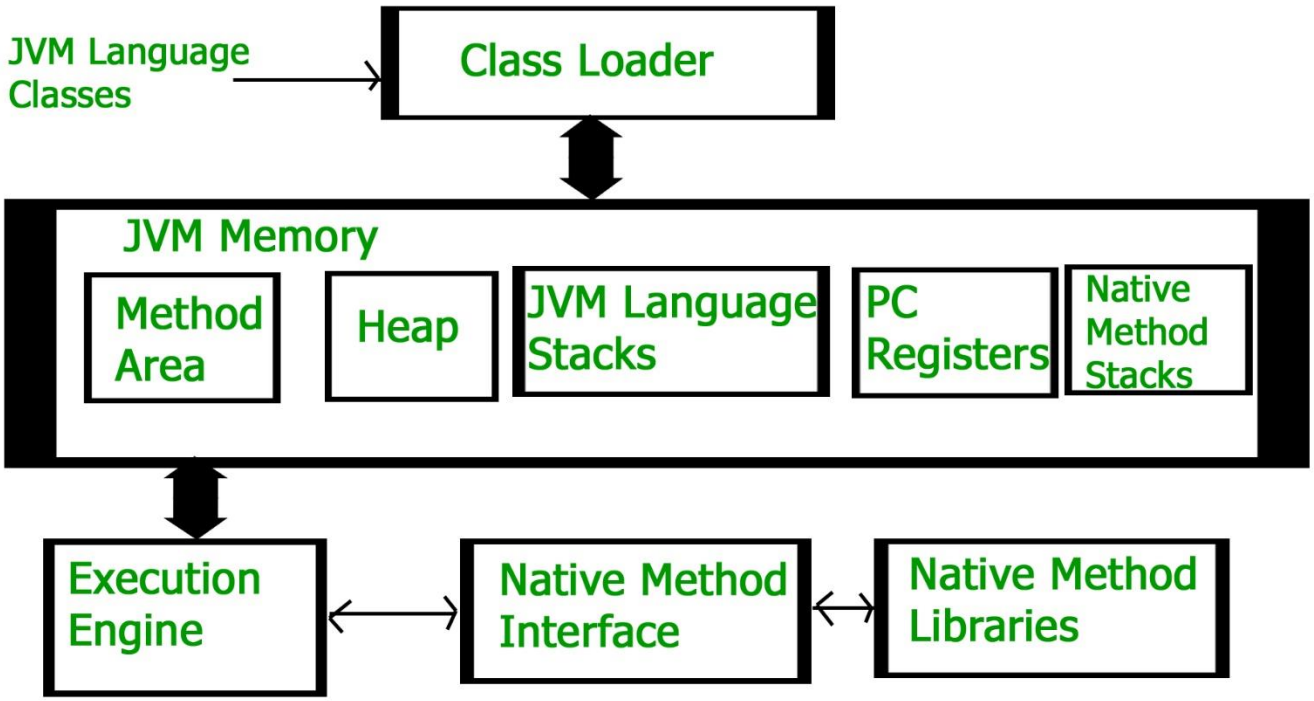
Deep dive into JRE how it works ->



JVM (Java Virtual Machine) is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java program line by line, hence it is also known as an interpreter.

Java program > java byte code > machine code > output

Deep dive into JVM how it works ->



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.
- 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.
- or every loaded ".class" file, only one object of the class is created

Linking:

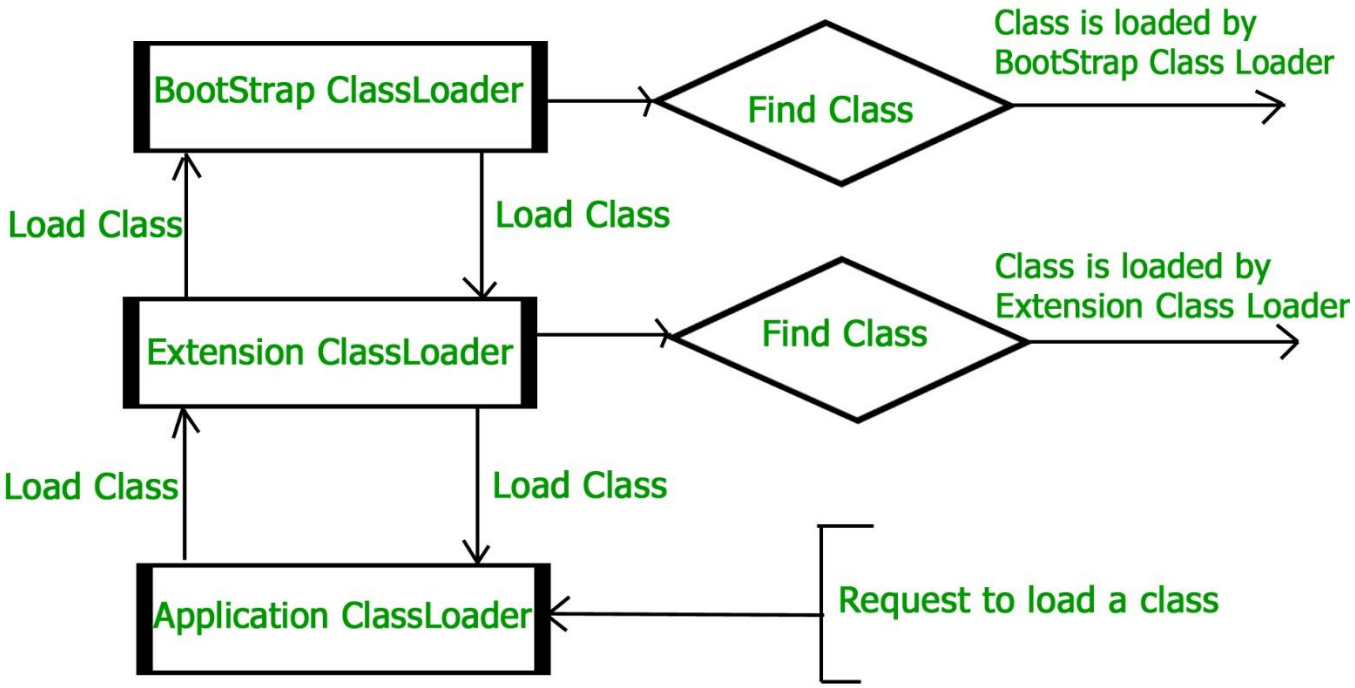
Performs verification, preparation, and (optionally) resolution.

- **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.

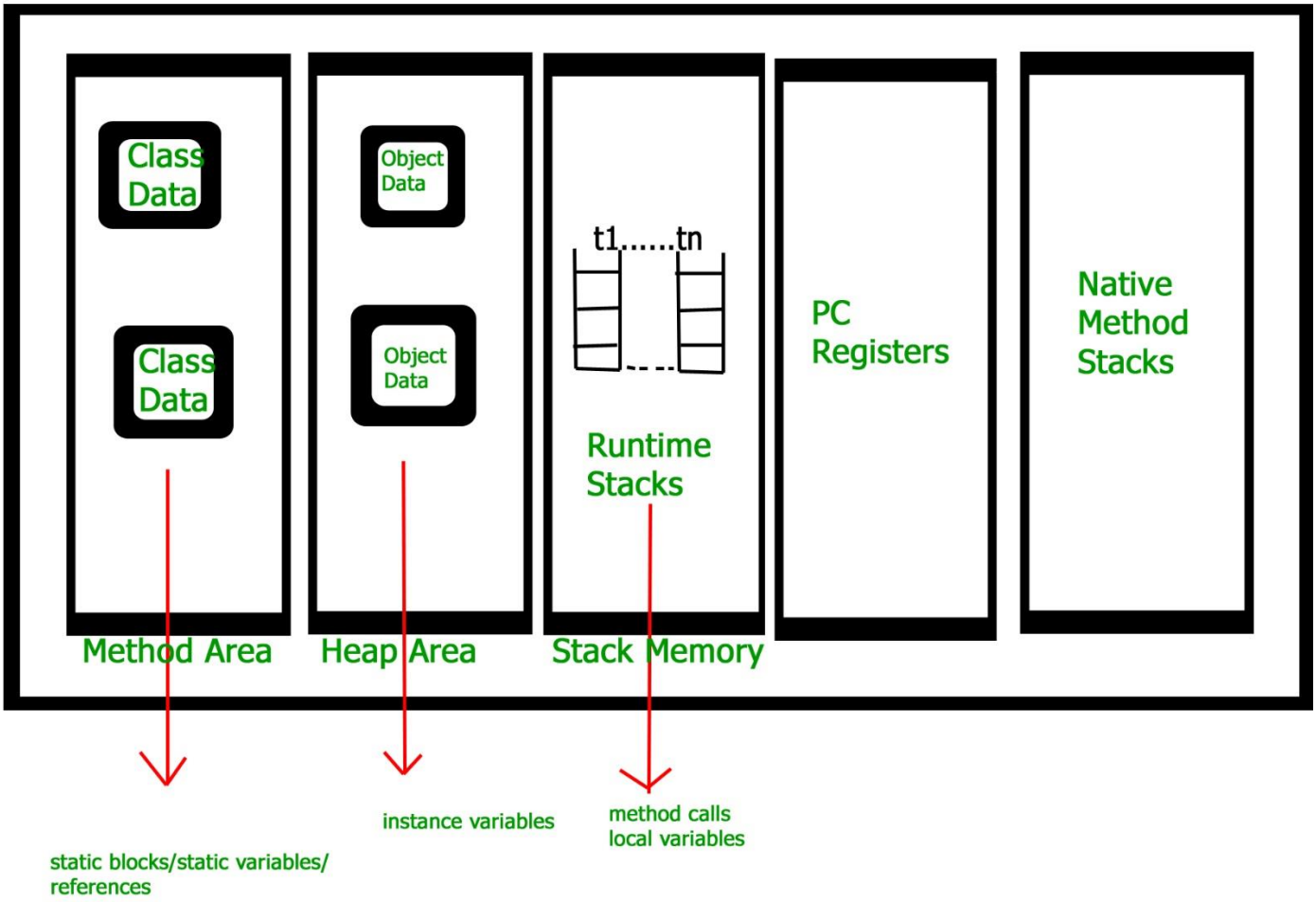
In general, there are three class loaders:



- **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.
- **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.
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.

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.