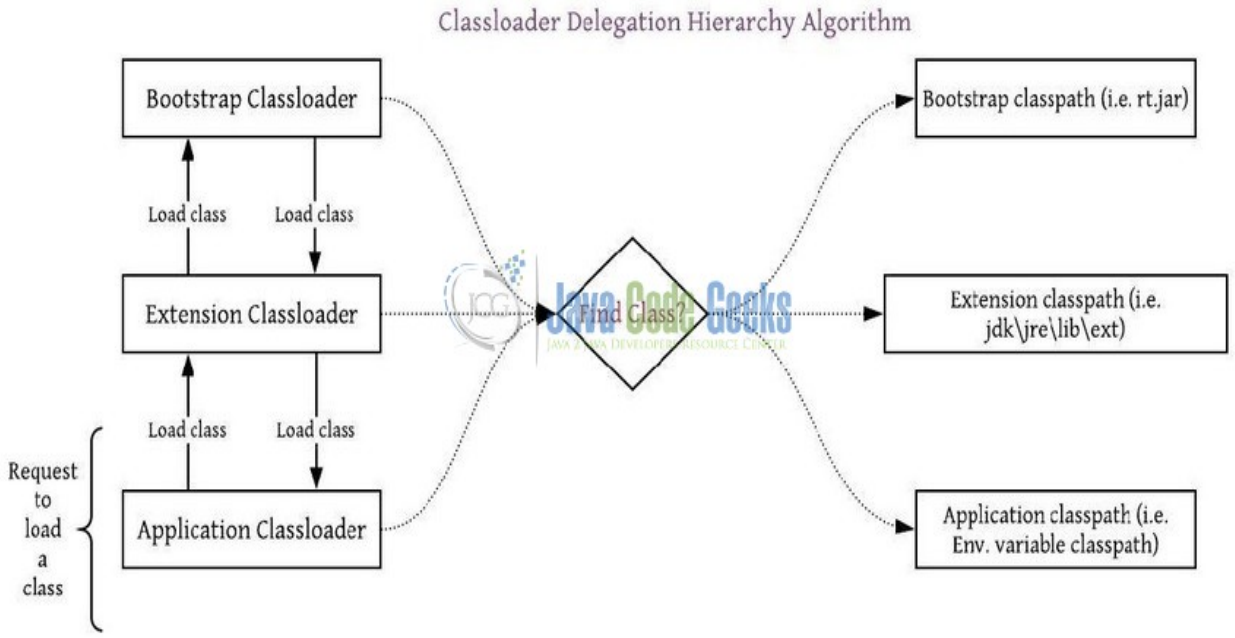
How ClassLoader works in Java

Classloader in Java works in three principles i.e. ***Delegation***, ***Visibility***, and ***Uniqueness***.



Delegation: According to this:

- Whenever the virtual machine come across a class, the JVM will check whether the specified .class file is loaded or not
- If the .class file is already loaded in the method area, then JVM will consider that class. If *not*, JVM requests the classloader subsystem to load that specific class
- The classloader system passes the request to the Application classloader which in turn delegates this request to the Extension classloader. The Extension classloader will again delegate this request to the Primordial classloader
- The primordial classloader will search this class in the bootstrap classpath (i.e. jdk\jre\lib\rt.jar). If found, the corresponding .class file is loaded
- If not, the primordial classloader delegates the request to the extension classloader. This will search the class in the jdk\jre\lib\ext path. If found, the corresponding .class file is loaded
- If not, the extension classloader delegates the request to the application classloader and will search the class in the application's classpath. If found, it is loaded otherwise developers will get a Class-NotFoundException at runtime

Visibility: According to this:

- Application classloader has a visibility to see the classes loaded by the parent classloaders but vice-versa is not true i.e. If a class is loaded by the system classloader and later again trying to explicitly load the same class using the extension classloader will throw a **ClassNotFoundException** at run-time. For instance:

```
// Printing the classloader of this class.
System.out.println("Test.getClass().getClassLoader()?= " + Test.class.getClassLoader());
```

```
// Trying to explicitly load the class again using the extension classloader.
Class.forName("com.jcg.classloading.test.Test", true, Test.class.getClassLoader().getPar-
ent());
```

Uniqueness: According to this:

- Class loaded by the parent classloader should *not* be again loaded by the child classloader

How classes are loaded in Java?

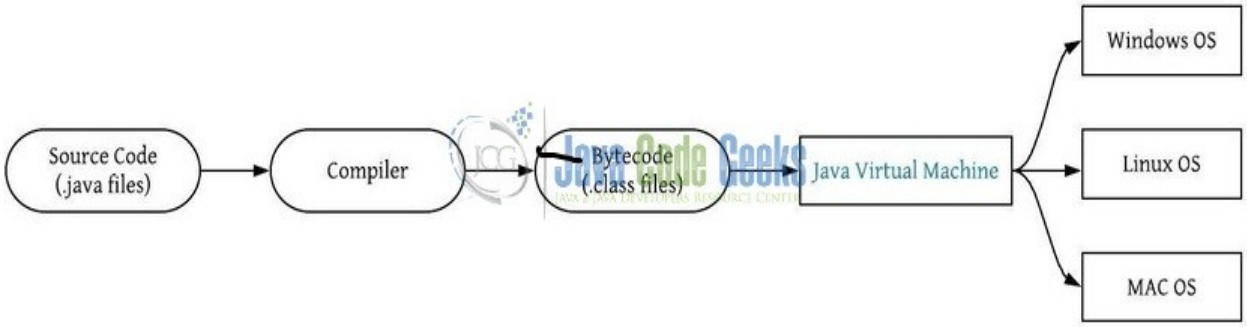
Classloaders are hierarchical. The very first class in an application is specifically loaded with the help of static main() method. All the subsequent classes are either loaded by the Static or the Dynamic class loading techniques.

- **Static class loading**: In this technique classes are statically loaded via the **new operator**
- **Dynamic class loading**: In this technique classes are programmatically loaded by using the **Class.forName()** or the **loadClass()** method. The difference between the two is that the former one initializes the object after loading it while the latter one only loads the class but doesn't initialize the object

1.1 What is Java Virtual Machine (JVM)?

Java Virtual Machine (JVM) is an abstract virtual machine that resides on your computer and provides a runtime environment for the Java bytecode to get executed. JVM is available for many hardware and software platforms but few Java developers know that the **Java Runtime Environment** (JRE) is the implementation of the **Java Virtual Machine** (JVM). JVM analyze the bytecode, interprets it, and execute the same bytecode to display the output.

The basic function of JVM is to execute the compiled .class files (i.e. the bytecode) and generate an output. Do *note*, each operating system has a different JVM, but the generated bytecode output is the same across all operating systems. This means that the bytecode generated on Windows OS can also run on Linux OS and vice-versa, thus making Java as a platform independent language.



What JVM does?

Java Virtual machine performs the following operations:

- Loading of the required .class and jar files
- Assigning references and verification of the code
- Execution of the code
- Provides a runtime environment for the Java bytecode

JVM Internal Architecture
The following diagram shows the key internal components of Java Virtual Machine that conforms to the JVM specification.

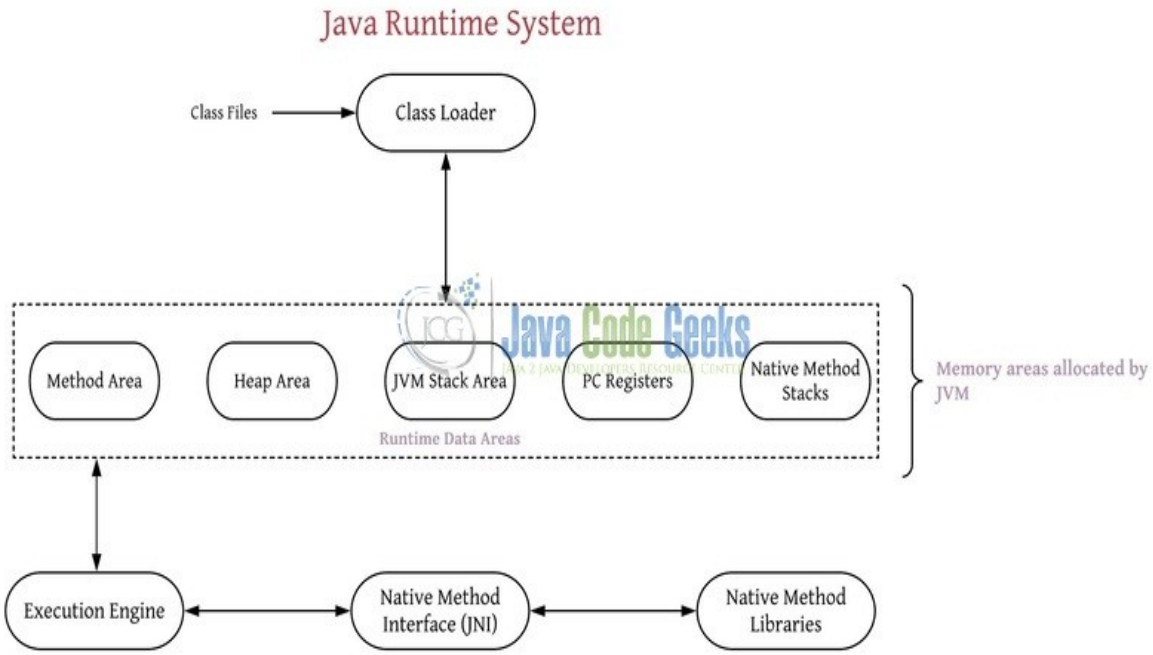


Fig. 2: Java Virtual Machine architecture

The classloader and runtime data areas components that are shown in Fig. 2 are each explained below.

1.2 ClassLoader Subsystem

The classloader subsystem is an essential core of the Java Virtual machine and is used for loading/reading the .class files and saving the bytecode in the JVM method area. This subsystem handles the dynamic class loading functionality and performs three major functions i.e.:

- **Loading**: This component handles the loading of the .class files from the hardware system into the JVM memory and stores the binary data (such as fully qualified class-name, immediate parent class-name, information about methods, variables, constructors etc.) in the method areas. For every loaded .class file, JVM immediately creates an object on the heap memory of type java.lang.class. Do *remember*, even though the developers call a class multiple time, only *one* class object will be created. There are three main types of classloaders:
- **Bootstrap or Primordial ClassLoader**: This classloader is responsible for loading the internal core java classes present in the rt.jar and other classes present in the java.lang.* package. It is by-default available with every JVM and is written in native C/C++ languages. This classloader has no parents and if developers call the String.class.getClassLoader(), it will return null and any code based on that will throw the NullPointerException in Java
- **Extension ClassLoader**: This classloader is the child class of Primordial classloader and is responsible for loading the classes from the extension classpath (i.e. jdk\jre\lib\ext). It is written in Java language and the corresponding .class file is sun.misc.Launcher$ExtClassLoder.class
- **Application or System ClassLoader**: This classloader is the child class of Extension classloader and is responsible for loading the classes from the system classpath. It internally uses the ‘CLASSPATH‘ environment variable and is written in Java language. System classloader in JVM is implemented by sun.misc.Launcher$AppClassLoader.class

- **Linking**: This component performs the linking of a class or an interface. As this component involves the allocation of new data structures, it may throw the OutOfMemoryError and performs the three important activities:
- **Verification**: It is a process of checking the binary representation of a class and validating whether the generated .class file is valid or not. This process is performed by the *Bytecode verifier* and if the generated .class file is not valid, a VerifyError is thrown
- **Preparation**: It is a process of assigning the memory for the class level or interface level static variables and assigns the default values
- **Resolution**: It is a process of changing the symbolic references with the original memory references from the method area

- **Initialization**: This component performs the final phase of the class loading where all the static variables are assigned the original values and the static blocks are executed from the parent to the child class. This process requires careful synchronization as JVM is multithreaded and some threads may try to initialize the same class or interface at the same time.

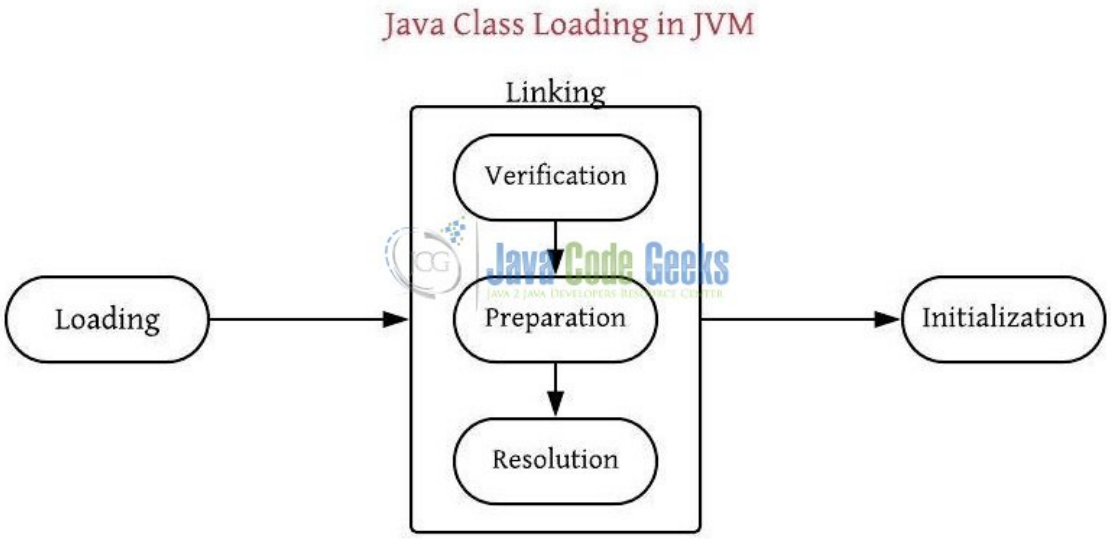


Fig. 3: An overview of ClassLoader Subsystem