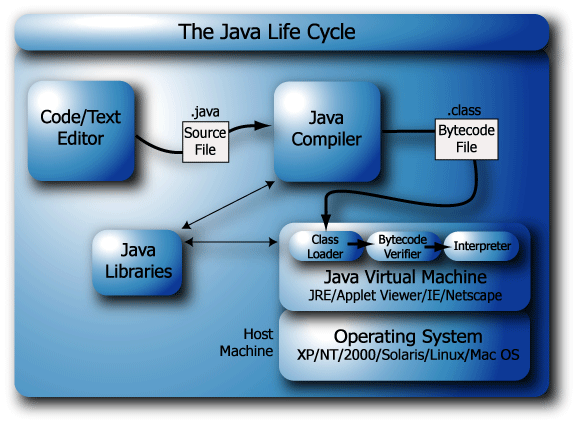
1. **What is JVM?**
2. When you download JRE and install on your machine you got all the code required to create JVM. Java Virtual Machine is get created when you run a java program using java command e.g. java HelloWorld. JVM is responsible for converting byte code into machine specific code and that's why you have different JVM for Windows, Linux or Solaris but one JAR can run on all this operating system. Java Virtual machine is at heart of Java programming language and provide several feature to Java programmer including Memory Management and Garbage Collection, Security and other system level services. Java Virtual Machine can be customized e.g we can specify starting memory or maximum memory of heap size located inside JVM at the time of JVM creation. If we supplied invalid argument to java command it may refuse to create Java Virtual Machine by saying "failed to create Java virtual machine: invalid argument". In short Java Virtual Machine or JVM is the one who provides Platform independence to Java.

JVM, or the Java Virtual Machine, is an interpreter which accepts ‘Bytecode’ and executes it.

Java has been termed as a ‘Platform Independent Language’ as it primarily works on the notion of ‘compile once, run everywhere’. Here’s a sequential step establishing the Platform independence feature in Java:

1. The Java Compiler outputs Non-Executable Codes called ‘Bytecode’.
2. Bytecode is a highly optimized set of computer instruction which could be executed by the Java Virtual Machine (JVM).
3. The translation into Bytecode makes a program easier to be executed across a wide range of platforms, since all we need is a JVM designed for that particular platform.
4. JVMs for various platforms might vary in configuration, those they would all understand the same set of Bytecode, thereby making the Java Program ‘Platform Independent’.



<http://javarevisited.blogspot.com/2011/12/jre-jvm-jdk-jit-in-java-programming.html#ixzz46EawcpTt>

1. **What is Just in Time Compiler (JIT)?**
2. Initially Java has been accused of poor performance because it’s both compiles and interpret instruction. Since compilation or Java file to class file is independent of execution of Java program do not confuse. Here compilation word is used for byte code to machine instruction translation. JIT are advanced part of Java Virtual machine which optimize byte code to machine instruction conversion part by compiling similar byte codes at same time and thus reducing overall execution time.

The JIT compiler is enabled by default, and is activated when a Java method is called. The JIT compiler compiles the bytecodes of that method into native machine code, compiling it "just in time" to run. When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it.

JIT compilation does require processor time and memory usage. When the JVM first starts up, thousands of methods are called. Compiling all of these methods can significantly affect startup time, even if the program eventually achieves very good peak performance.

JIT is part of Java Virtual Machine and also performs several other optimizations such as in-lining function.

http://javarevisited.blogspot.com/2011/12/jre-jvm-jdk-jit-in-java-programming.html#ixzz46Ebc1tKA

1. **What are ClassLoaders?**

ClassLoader in Java is a class which is used to load class files in Java. Java code is compiled into class file by javac compiler and JVM executes Java program, by executing byte codes written in class file. ClassLoader is responsible for loading class files from file system, network or any other source.

The ClassLoader is the part of the JVM.

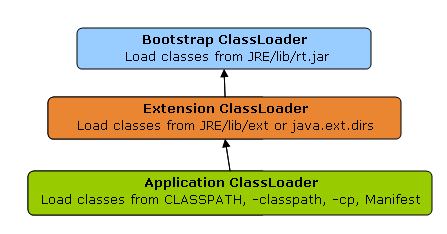
The Java ClassLoader is simply java class. This means that it's easy to create your own ClassLoader.

**ClassLoader Types –**

1. **Bootstrap -** Every class loader has a predefined location, from where they loads class files. Bootstrap ClassLoader is responsible for loading standard JDK class files from rt.jar and it is parent of all class loaders in Java.

Bootstrap class loader don't have any parents. Bootstrap class loader is also known as Primordial ClassLoader in Java.

1. **Extension** - Extension ClassLoader delegates class loading request to its parent, Bootstrap and if unsuccessful, loads class form jre/lib/ext directory or any other directory pointed by java.ext.dirs system property. Extension ClassLoader in JVM is implemented by  sun.misc.Launcher$ExtClassLoader.
2. **Application -** Third default class loader used by JVM to load Java classes is called System or Application class loader and it is responsible for loading application specific classes from [CLASSPATH](http://javarevisited.blogspot.sg/2011/01/how-classpath-work-in-java.html) environment variable, -classpath or -cp command line option, Class-Path attribute of Manifest file inside JAR. Application class loader is a child of Extension ClassLoader and its implemented by sun.misc.Launcher$AppClassLoader class. Also, except Bootstrap class loader, which is implemented in native language mostly in C,  all  Java class loaders are implemented using java.lang.ClassLoader.



Correct understanding of class loader is must to resolve issues like NoClassDefFoundError in Java and java.lang.ClassNotFoundException, which are related to class loading.

**Classloader Principles -**

1. **Delegation** - Delegation principle tells that if class is not loaded by current classloader then it delegates class loading request to its parent classLoader so parent classLoader loads the class this is upto the top most class loader i.e Bootstrap

As discussed on when a class is loaded and initialized in Java, a class is loaded in Java, when its needed. Suppose you have an application specific class called Abc.class, first request of loading this class will come to Application ClassLoader which will delegate to its parent Extension ClassLoader which further delegates to Primordial or Bootstrap class loader. Primordial will look for that class in rt.jar and since that class is not there, request comes to Extension class loader which looks on jre/lib/ext directory and tries to locate this class there, if class is found there than Extension class loader will load that class and Application class loader will never load that class but if its not loaded by extension class-loader than Application class loader loads it from Classpath in Java. Remember Classpath is used to load class files while PATH is used to locate executable like javac or java command.

1. **Visibility** - Visibility principle allows child class loader to see all the classes loaded by parent ClassLoader, but parent class loader cannot see classes loaded by child.

According to visibility principle, Child ClassLoader can see class loaded by Parent ClassLoader but vice-versa is not true. Which mean if class Abc is loaded by Application class loader than trying to load class ABC explicitly using extension ClassLoader will throw either [java.lang.ClassNotFoundException](http://javarevisited.blogspot.ca/2011/08/classnotfoundexception-in-java-example.html). as shown in below Example

**package** test;  
  
**import** java.util.logging.Level;  
**import** java.util.logging.Logger;  
  
/\*\*  
 \* Java program to demonstrate How ClassLoader works in Java,

 \* in particular about visibility principle of ClassLoader.

 \*  
 \* @author Javin Paul  
 \*/  
  
**public** **class** ClassLoaderTest {  
    
    **public** **static** **void** main(**String** args[]) {  
        **try** {            
            *//printing ClassLoader of this class*  
            **System**.out.println("ClassLoaderTest.getClass().getClassLoader() : "  
                                 + ClassLoaderTest.**class**.getClassLoader());  
  
            
            *//trying to explicitly load this class again using Extension class loader*  
            **Class**.forName("test.ClassLoaderTest", **true**   
                            ,  ClassLoaderTest.**class**.getClassLoader().getParent());  
        } **catch** (**ClassNotFoundException** ex) {  
            **Logger**.getLogger(ClassLoaderTest.**class**.getName()).log(**Level**.SEVERE, **null**, ex);  
        }  
    }  
  
}  
  
**Output:**  
ClassLoaderTest.getClass().getClassLoader() : sun.misc.Launcher$AppClassLoader@601bb1  
16/08/2012 2:43:48 AM test.ClassLoaderTest main  
SEVERE: **null**  
java.lang.**ClassNotFoundException**: test.ClassLoaderTest  
        at java.net.**URLClassLoader**$1.run(**URLClassLoader**.java:202)  
        at java.security.**AccessController**.doPrivileged(Native **Method**)  
        at java.net.**URLClassLoader**.findClass(**URLClassLoader**.java:190)  
        at sun.misc.Launcher$ExtClassLoader.findClass(Launcher.java:229)  
        at java.lang.**ClassLoader**.loadClass(**ClassLoader**.java:306)  
        at java.lang.**ClassLoader**.loadClass(**ClassLoader**.java:247)  
        at java.lang.**Class**.forName0(Native **Method**)  
        at java.lang.**Class**.forName(**Class**.java:247)  
        at test.ClassLoaderTest.main(ClassLoaderTest.java:29)

1. **Uniqueness** -Uniqueness principle allows loading a class exactly once, which is basically achieved by delegation and ensures that child ClassLoader doesn't reload the class already loaded by parent.

According to this principle a class loaded by Parent should not be loaded by Child ClassLoader again. Though its completely possible to write class loader which violates Delegation and Uniqueness principles and loads class by itself, its not something which is beneficial. You should follow all class loader principle while writing your own ClassLoader.

### Methods in ClassLoader -

#### loadClass

- loadClass is load the class disk into memory.but class is not initilized likely class.forName()

class MainClassLoader {  
public static void main(String[] args){  
  
ClassLoader mainLoader = MainClassLoader.class.getClassLoader();  
try {  
Class ourCLass = classLoader.loadClass("com.javawora.MainClassLoader");  
System.out.println("ourCLass.getName() = " + ourCLass.getName());  
} catch (ClassNotFoundException e) {  
e.printStackTrace();  
}  
  
}

#### defineClass -

This method mainly used for converting the raw array bytes to class object. it parses the bytecode format into a run-time data structure, checks for validity,verification etc.this method is final so we can not override it.

#### findSystemClass -

The findSystemClass method is used to loads the class files from the local filesystem.i.e bootstrap classloader after the loading the class it call defineClass method to convert raw byte array to class object   
1.classLoader requires particuler class to load  
2.check class exist or not if it is there loaded into memory.  
3.if it is not there calls findSystemClass to load from the local file system.

#### resolveClass -

if we want to load the partially then this method is used

#### findLoadedClass -

findLoadedClass works as a cache.when any classloader ask to load the class then child class loader calls findLoadedClass() so it gives all loaded class by parent classloader if it is already loaded then then it can not load the class otherwise it loads the class.This method is called first when any request come to load the class.

Now let’s get our hands dirty with some real code. Consider the following example: class A instantiates class B.

|  |
| --- |
| **public** **class** A {  **public** **void** doSomething() {  B b = **new** B();  b.doSomethingElse();  }  } |

The statement B b = new B() is semantically equivalent to

B b = A.class.getClassLoader().loadClass(“B”).newInstance()

As we see, every object in Java is associated with its class (A.class) and every class is associated with classloader (A.class.getClassLoader()) that was used to load the class.

When we instantiate a ClassLoader, we can specify a parent classloader as a constructor argument. If the parent classloader isn’t specified explicitly, the virtual machine’s system classloader will be assigned as a default parent.

<http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/lang/ClassLoader.java#ClassLoader.findLoadedClass%28java.lang.String%29>

<http://www.javawora.com/classloader>

<http://javarevisited.blogspot.in/2012/12/how-classloader-works-in-java.html>

<http://zeroturnaround.com/rebellabs/rebel-labs-tutorial-do-you-really-get-classloaders/>