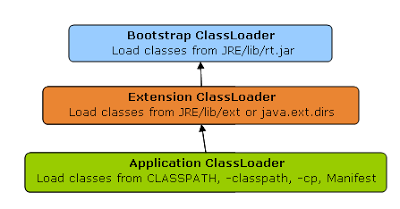
# [How ClassLoader Works in Java](http://javarevisited.blogspot.in/2012/12/how-classloader-works-in-java.html" \o "How ClassLoader Works in Java)

Java class loaders are used to load classes at runtime. ClassLoader in Java works on three principle: delegation, visibility and uniqueness. Delegation principle forward request of class loading to parent class loader and only loads the class, if parent is not able to find or load class. Visibility principle allows child class loader to see all the classes loaded by parent ClassLoader, but parent class loader can not see classes loaded by child. Uniqueness principle allows to load a class exactly once, which is basically achieved by delegation and ensures that child ClassLoader doesn't reload the class already loaded by parent. Correct understanding of class loader is must to resolve issues like [NoClassDefFoundError in Java](http://javarevisited.blogspot.sg/2011/06/noclassdeffounderror-exception-in.html) and [java.lang.ClassNotFoundException](http://javarevisited.blogspot.sg/2011/08/classnotfoundexception-in-java-example.html), which are related to class loading. ClassLoader is also an important topic in advanced Java Interviews, where good knowledge of working of Java ClassLoader and [How classpath works in Java](http://javarevisited.blogspot.ca/2011/01/how-classpath-work-in-java.html)  is expected from Java programmer. I have always seen questions like, **Can one class be loaded by two different ClassLoader in Java** on various [Java Interviews](http://javarevisited.blogspot.sg/2011/04/top-20-core-java-interview-questions.html). In this Java programming tutorial, we will learn what is ClassLoader in Java, How ClassLoader works in Java and some specifics about Java ClassLoader.

**What is ClassLoader in Java**

ClassLoader in Java is a class which is used to load [class files in Java](http://javarevisited.blogspot.ca/2012/05/10-points-about-class-file-in-java.html). Java code is compiled into class file by [javac](http://javarevisited.blogspot.sg/2012/12/javac-is-not-recognized-as-internal-or-external-command.html) compiler and [JVM](http://javarevisited.blogspot.sg/2011/12/jre-jvm-jdk-jit-in-java-programming.html) 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. There are three default class loader used in Java, **Bootstrap** , **Extension** and **System or Application class loader**. 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, if you call String.class.getClassLoader() it will return null and any code based on that may throw [NullPointerException in Java](http://javarevisited.blogspot.com/2012/06/common-cause-of-javalangnullpointerexce.html). Bootstrap class loader is also known as **Primordial ClassLoader** in Java. 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$ExtClassLoader. Third default class loader used by JVM to load Java classes is called System or Application class loader and it is responsible for loading application specific classes from [CLASSPATH](http://javarevisited.blogspot.sg/2011/01/how-classpath-work-in-java.html) environment variable, -classpath or -cp command line option, Class-Path attribute of Manifest file inside JAR. Application class loader is a child of Extension ClassLoader and its implemented by sun.misc.Launcher$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.

[](http://2.bp.blogspot.com/-HCTsr-j_ojw/USTOh1f8JwI/AAAAAAAAAjg/YegPspR5K48/s1600/java_classloader_hierarchy.PNG)

In short here is the location from which Bootstrap, Extension and Application ClassLoader load Class files.

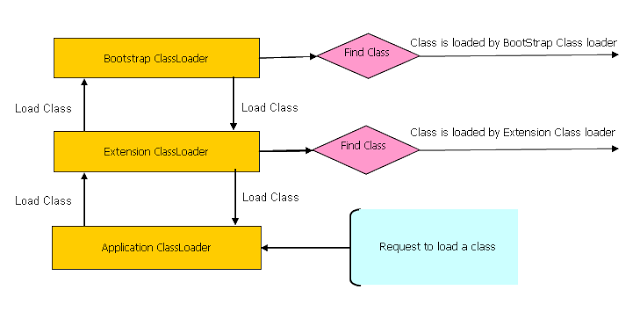
1) Bootstrap ClassLoader - JRE/lib/rt.jar

2) Extension ClassLoader - JRE/lib/ext or any directory denoted by java.ext.dirs

3) Application ClassLoader - CLASSPATH environment variable, -classpath or -cp option, Class-Path attribute of Manifest inside [JAR file](http://javarevisited.blogspot.sg/2012/03/how-to-create-and-execute-jar-file-in.html).

## How ClassLoader works in Java

As I explained earlier Java ClassLoader works in three principles : delegation, visibility and uniqueness. In this section we will see those rules in detail and understand working of Java ClassLoader with example. By the way here is a diagram which explains How ClassLoader load class in Java using delegation.

[](http://1.bp.blogspot.com/-0gOWex7Pb2E/USTOh2K7zpI/AAAAAAAAAjc/_viQADzxrsk/s1600/Java+classloader+working.PNG)

**Delegation principles**

As discussed on [when a class is loaded and initialized in Java](http://javarevisited.blogspot.sg/2012/07/when-class-loading-initialization-java-example.html), 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](http://java67.blogspot.sg/2012/08/what-is-path-and-classpath-in-java-difference.html). Remember Classpath is used to load class files while [PATH](http://javarevisited.blogspot.ca/2011/10/how-to-set-path-for-java-unix-linux-and.html) is used to locate executable like javac or java command.

**Visibility Principle**

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)

**Uniqueness Principle**

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.

## How to load class explicitly in Java

Java provides API to explicitly load a class by Class.forName(classname) and Class.forName(classname, initialized, classloader), remember JDBC code which is used to load JDBC drives we have seen in [Java program to Connect Oracle database](http://javarevisited.blogspot.ca/2012/04/java-program-to-connect-oracle-database.html). As shown in above example you can pass name of ClassLoader which should be used to load that particular class along with binary name of class. Class is loaded by calling loadClass() method of java.lang.ClassLoader class which calls findClass() method to locate bytecodes for corresponding class. In this example Extension ClassLoader uses java.net.URLClassLoader which search for class files and resources in [JAR](http://javarevisited.blogspot.ca/2012/10/5-ways-to-add-multiple-jar-to-classpath-java.html) and directories. any search path which is ended using "/" is considered directory. If findClass() does not found the class than it throws [java.lang.ClassNotFoundException](http://javarevisited.blogspot.de/2012/03/jdbc-javalangclassnotfoundexception.html) and if it finds it calls defineClass() to convert bytecodes into a .class instance which is returned to the caller.

**Where to use ClassLoader in Java**

ClassLoader in Java is a powerful concept and used at many places. One of the *popular example of ClassLoader* is AppletClassLoader which is used to load class by Applet, since Applets are mostly loaded from internet rather than local file system, By using separate ClassLoader you can also loads same class from multiple sources and they will be treated as different class in [JVM](http://javarevisited.blogspot.ca/2011/12/jre-jvm-jdk-jit-in-java-programming.html). J2EE uses multiple class loaders to load class from different location like classes from WAR file will be loaded by Web-app ClassLoader while classes bundled in EJB-JAR is loaded by another class loader. Some web server also supports hot deploy functionality which is implemented using ClassLoader. You can also use ClassLoader to load classes from database or any other persistent store.

That's all about **What is ClassLoader in Java** and **How ClassLoader works in Java**. We have seen delegation, visibility and uniqueness principles which is quite important to debug or troubleshoot any ClassLoader related issues in Java. In summary knowledge of How ClassLoader works in Java is must for any Java developer or architect to design Java application and packaging.

Read more: <http://javarevisited.blogspot.com/2012/12/how-classloader-works-in-java.html#ixzz40t9rWpPQ>

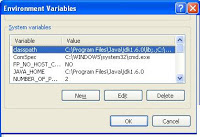
**What is CLASSPATH in Java**

Classpath in Java is the path to directory or list of the directory which is used by ClassLoaders to find and load class in Java program. Classpath can be specified using CLASSPATH environment variable which is case insensitive, -cp or -classpath command line option or Class-Path attribute in manifest.mf file inside JAR file in Java.  In this Java tutorial, we will learn What is Classpath in Java, how Java resolve classpath and how Classpath works in Java alongside How to set the classpath for Java in windows and UNIX environment.  I have experience in finance and insurance domain and Java is heavily used in this domain for writing sophisticated Equity, Fixed income trading applications. Most of these investment banks has written test as part of their [core Java interview questions](http://javarevisited.blogspot.com/2015/10/133-java-interview-questions-answers-from-last-5-years.html)and I always find at least one question related to CLASSPATH in Java on those interviews. Java CLASSPATH is one of the most important concepts in Java,  but,  I must say mostly overlooked. This should be the first thing you should learn while writing Java programs because without the correct understanding of **Classpath in Java**you can't understand how Java locates your class files. Also don't confuse Classpath with [PATH in Java](http://javarevisited.blogspot.sg/2011/10/how-to-set-path-for-java-unix-linux-and.html), which is another environment variable used to find java binaries located in JDK installation directory, also known as [JAVA\_HOME](http://javarevisited.blogspot.com/2012/02/how-to-set-javahome-environment-in.html). 

Unfortunately, books like [Head First Java](http://www.amazon.com/dp/0596009208/?tag=javamysqlanta-20) doesn't teach you much about subtleties of PATH and CLASSPATH. If you really want to test your Java skill, the one book I would suggest reading is [Java Puzzlers](http://www.amazon.com/dp/032133678X/?tag=javamysqlanta-20), whose puzzles and explanation will help more to understand this kind of not so easy concepts.  
  
  
The main [difference between PATH and CLASSPATH](http://java67.blogspot.com/2012/08/what-is-path-and-classpath-in-java-difference.html) is that former is used to locate Java commands while later is used to locate Java class files. So let’s start with basic and then we will see some example and improvisation of Classpath in Java. In fact, CLASSPATH is an environment variable which is used by Java Virtual Machine to locate user defined classes. As I said In this tutorial we will see How to setup classpath for java in windows and Linux, **java -classpath example** in different scenario and use of java -classpath or java -cp. If you have just started learning Java, then I also suggest you follow a good book to learn Java e.g.[Core Java Volume 1 9th Edition by Cay S. Horstmann](http://www.amazon.com/Core-Volume-I-Fundamentals-Edition-Series/dp/0137081898?tag=javamysqlanta-20" \t "_blank). That will help you to grasp most of fundamental concept in Java in a systematic manner. 

**Setting Java Classpath in Windows**

In order to set Classpath for Java in Windows (any version either Windows XP,  Windows 2000 or Windows 7) you need to specify the value of environmentvariable ***CLASSPATH***, the name of this variable is not case sensitive and it doesn’t matter if the name of your *environment variable is Classpath, CLASSPATH or classpath in Java*.  
  
Here is *Step by Step guide for setting Java Classpath in Windows*:

[](http://2.bp.blogspot.com/-FiPjzZMNFJI/TarzPFTdXsI/AAAAAAAAAH8/9Z-_mh8YZvY/s200/setting+classpath+in+windows+for+java.JPG)

Go to Environment variable window in Windows by pressing "Windows + Pause “--> Advanced --> Environment variable " or you can go from right click on my computer than choosing properties and then Advanced and then Environment variable this will open Environment variable window in windows.

Now specify your environment variable CLASSPATH and put the value of your JAVA\_HOME\lib and also include [current directory](http://javarevisited.blogspot.sg/2012/04/how-to-find-current-directory-in-java.html) by including (dot or period sign).

Now *to check the value of Java classpath in windows* type **"echo %CLASSPATH"** in your DOS command prompt and it will show you the value of directory which are included in CLASSPATH.  
  
You can also set classpath in windows by using DOS command like:  
  
set CLASSPATH=%CLASSPATH%;JAVA\_HOME\lib;  
This way you can set the classpath in Windows XP, windows 2000 or Windows 7 and 8, as they all come with command prompt.

**Setting Java Classpath in UNIX or Linux**

To set Classpath for Java In Linux, you can simply export CLASSPATH="your classpath" from either your ***.***bash\_profile or .bashrc script which will run whenever you login into your Linux or Unix Machine. Now to check the value of Java CLASSPATH in Linux type "echo ${CLASSPATH}" this will print the value of [Classpath in command prompt](http://javarevisited.blogspot.sg/2011/11/run-java-program-from-command-prompt.html). By using the export command, you can set the classpath for Java in Unix, Linux, Solaris, IBM AIX or any other UNIX operating system. I hope this example for setting classpath in Java will enable to set classpath by yourself let me know if you face any problem while setting up classpath in Java

**Overriding Classpath in Java**

You can override classpath in Java, defined by environment variable CLASSPATH by providing option "-cp" or "-classpath" while [running Java program](http://javarevisited.blogspot.sg/2011/11/run-java-program-from-command-prompt.html) and this is the best way to have different classpath for different Java application running on same Unix or Windows machine, standard way to define classpath for Java application is creating start-up script for Java program and set classpath there as shown below :  
  
CLASSPATH=/home/tester/classes  
java -cp $CLASSPATH Test  
  
By default, Java CLASSPATH points to current directory denoted by "." and it will look for any class only in the current directory.

**Different example of using Classpath in Java**

In case you have multiple directories defined in CLASSPATH variable, Java will look for a class starting from the first directory and only look the second directory in case it did not find the specified class in the first directory. This is an extremely useful feature of Classpath in java to understand and it’s very useful while [debugging Java application](http://javarevisited.blogspot.sg/2011/07/java-debugging-tutorial-example-tips.html) or  patch release kind of stuff. Let’s see  java -classpath example  
  
  
I have set my classpath environment variable as CLASSPATH=/home/tester/first:/home/tester/second**.**Now I have Test class of different version in both first and second directory. When I give a command "java Test" What will happen? Which Test class would be picked? Since JVM search directory in the order they have listed in CLASSPATH variable it will first go to the "first" directory and if it finds Test.class over there it will not go to /home/tester/second directory. Now if you delete Test.class from /home/tester/first directory it will go to /home/tester/second directory and will pick  Test.class from there.  
  
  
I have used this feature of Java Classpath to test my patch releases, we used to have a folder called "patch" listed as first element in Java CLASSPATH and any point of time we want to put any debug statement or want to test any bug we just modify Java source file, compile it and generate class file and put that inside patch folder instead of [creating JAR file](http://javarevisited.blogspot.sg/2012/03/how-to-create-and-execute-jar-file-in.html) and releasing whole new Java application. This is very handy if you are working in a large project where you don't have development environment setup in Windows and your project only runs on Unix server. This approach is much faster than [remote debugging Java application in Eclipse](http://javarevisited.blogspot.sg/2011/02/how-to-setup-remote-debugging-in.html)  
  
  
It's also worth noting that when you use the  java -jar command line option to run your Java program as an [executable JAR](http://java67.blogspot.sg/2014/04/how-to-make-executable-jar-file-in-Java-Eclipse.html), then the CLASSPATH environment variable will be ignored, and also the -cp and -classpath switches will be ignored. In this case, you can set your Java classpath in the META-INF/MANIFEST.MF file by using the Class-Path attribute. In short Class-pathattribute in manifest file overrides classpath specified by -cp, -classpath or CLASSPATH environment variable.  
  
  
Now a common question if I have my CLASSPATH variable pointing to *current directory****"."*** and I have a class called "Test" inside package "testing" and with below directory structure C:\project\testing\Test.class in my computer.  
  
**What will happen if I run the command "java Test" from directory "C:\project\testing\"? will it run?**  
No, it will not run it will give you:  
[Exception in thread "main" java.lang.NoClassDefFoundError: Test](http://javarevisited.blogspot.sg/2011/06/noclassdeffounderror-exception-in.html)  
Since the name of the class is not Test, instead it’s testing.Test even though your classpath is set to current directory.  
  
Now what will happen if I give command  java testing.Test from C:\project\testing\ it will again not run and give an error?  
  
Exception in thread "main" java.lang.NoClassDefFoundError: testing/Test  
  
Why because now it looking for class called Test which is in package testing, starting from current directory "." but don't find it since there is no directory called "testing after this path "C:\project\testing\".  
  
To run it successfully you need to go back to directory  C:\project and now run   
C:\project>java testing.Test  and It will run successfully because of Classpath issues I prefer to use Eclipse rather than [running Java program from command prompt](http://javarevisited.blogspot.sg/2011/11/run-java-program-from-command-prompt.html). 

**Errors related to Classpath in Java**

If you are working in Java, you must have faced some errors and exception related to the classpath in java, two most common issues related to java classpath is **ClassNotFoundException** and **NoClassDefFoundError**. I have seen that many Java developer tries to solve this error by trial and error; they just don’t look beyond the hood and try to understand what the reason for this java classpath related errors is. They often misunderstood that these two errors are same also.   
  
Here is the reason of these Java classpath errors:   
  
**ClassNotFoundException is an Exception and will be thrown when Java program dynamically tries to load a Java class at Runtime and don’t find the corresponding class file on the classpath. Two keywords here “dynamically” and “runtime”. A classic example of these errors is whey you try to load JDBC driver by using Class.forname(“driver name”) and greeted with**[**java.lang.ClassNotFoundException: com.mysql.jdbc.Driver**](http://javarevisited.blogspot.sg/2012/03/jdbc-javalangclassnotfoundexception.html)**. So this error essentially comes when Java try to load a class using forName() or by loadClass() method of ClassLoader. The key thing to note is that presence of that class on Java classpath is not checked on compile time. So even if those classes are not present on Java classpath your program will compile successfully and only fail when you try to run.**  
  
On the other hand, NoClassDefFoundError is an Error and more critical than ClassNotFoundException which is an exception and recoverable. NoClassDefFoundError comes when a particular class was present in Java Classpath during compile time but not available during run-time. A classic example of this error is using log4j.jar for logging purpose and forgot to include log4j.jar on the classpath in java during run-time. to read more about logging in Java see. Keyword here is,  the class was present at compile time but not available on run-time.  This is normally occurring due to any method invocation on a particular class which is part of library and not available on classpath in Java. This is also asked as common interview questions as   
“[What is difference between NoClassDefFoundError and ClassNotFoundException Exception in Java](http://javarevisited.blogspot.com/2011/07/classnotfoundexception-vs.html)”   or  
“When do you see NoClassDefFoundError and ClassNotFoundException Exception in Java”. By the way NoClassDefFoundError can also comes due to various other reason like static initializer failure or class not visible to Classloaders in J2EE environment. Read 3 ways to resolve NoClassDefFoundError in Java for complete details.

**Summary of CLASSPATH in Java**

1.      **Classpath i**n Java is **an environment variable** used by Java Virtual machine to locate or find  [class files in Java](http://java67.blogspot.sg/2012/08/what-is-class-file-in-java-how-to-create-class.html) during class loading.

2.      You can **override value of Classpath in Java** defined by environment variable CLASSPATH by providing **JVM command line option –cp or –classpath** while running your application.

3.      If two classes with same name exist in Java Classpath then **the class which comes earlier in Classpath** will be picked by Java Virtual Machine.

4.      **By default CLASSPATH in Java points to current directory denoted by "."** and it will look for any class only in current directory.

5.      When you use the **-jar command line  option** to [run your program as an executable JAR](http://javarevisited.blogspot.sg/2012/03/how-to-create-and-execute-jar-file-in.html), then the Java **CLASSPATH environment variable will be ignored**, and also the **-cp and -classpath switches will be ignored** and In this case you can set your java classpath in the *META-INF/MANIFEST.MF file by using the Class-Path attribute*.

6.      **In Unix of Linux Java Classpath** contains names of directory with **colon “:”** separated , On Windows Java Classpath will be  **semi colon “;”** separated while if you defined java classpath in Manifest file those will be **space** separated.

7.       You can check value of classpath in java inside your application by looking at following system property **“java.class.path**”  System.getProperty("java.class.path")  
  
**Class-Path attribute is used to contain classpath inside manifest file**. Also make sure that your manifest file must end with a blank line (carriage return or new line) , here is an example of java classpath in manifest file.  
  
**Main-Class: com.classpathexample.Demo\_Classpath  
Class-Path: lib/tibco.jar lib/log4j.jar**

8.       It’s also important to note that **path specified in manifest file is not absolute** instead **they are relative from application jar’s path**. For example in above if your application jar file is in C:\test directory you must need a lib directory inside test and tibco.jar and log4j.jar inside that.

9.       **ClassNotFoundException**is an Exception and will be thrown when Java program dynamically tries to load a particular Class at Runtime and don’t find that on Java classpath due to result of Class.forName() or loadClass() method invocation.

10. **NoClassDefFoundError**comes when a particular class was present in Java Classpath during compile time but not available during runtime on Classpath in Java.

I hope you find this **Java Classpath tutorial**useful , please let me know if you have any doubt or any question related to **"How to set classpath for java**" and I would be happy to answer :) keep learning. Your suggestions and comments are always welcome.  If you like to read UNIX command tips you may find  [10 tips of using find command in Linux](http://javarevisited.blogspot.com/2011/03/10-find-command-in-unix-examples-basic.html),  [10 tips to increase speed on Unix command](http://javarevisited.blogspot.com/2011/03/unix-command-tutorial-working-fast-in.html)and [10 basic networking Commands in Unix](http://javarevisited.blogspot.com/2010/10/basic-networking-commands-in-linuxunix.html) useful. That's all on What is Classpath in Java, How Classpath works in Java, How to set Classpath in Java on Windows and Linux and how to deal with Classpath issues in Java.  
  
Read more: <http://javarevisited.blogspot.com/2011/01/how-classpath-work-in-java.html#ixzz40sJeEhb6>

# Java Reflection - Dynamic Class Loading and Reloading

* [The ClassLoader](http://tutorials.jenkov.com/java-reflection/dynamic-class-loading-reloading.html#classloader)
* [The ClassLoader Hierarchy](http://tutorials.jenkov.com/java-reflection/dynamic-class-loading-reloading.html#classloaderHierarchy)
* [Class Loading](http://tutorials.jenkov.com/java-reflection/dynamic-class-loading-reloading.html#loading)
* [Dynamic Class Loading](http://tutorials.jenkov.com/java-reflection/dynamic-class-loading-reloading.html#dynamicloading)
* [Dynamic Class Reloading](http://tutorials.jenkov.com/java-reflection/dynamic-class-loading-reloading.html#dynamicreloading)
* [Designing your Code for Class Reloading](http://tutorials.jenkov.com/java-reflection/dynamic-class-loading-reloading.html#designing)
* [ClassLoader Load / Reload Example](http://tutorials.jenkov.com/java-reflection/dynamic-class-loading-reloading.html#example)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2014-05-21 |

It is possible to load and reload classes at runtime in Java, though it is not as straightforward as one might have hoped. This text will explain when and how you can load and reload classes in Java.

You can argue whether Java's dynamic class loading features are really part of Java Reflection, or a part of the core Java platform. Anyways, the article has been put in the Java Reflection trail in lack of a better place to put it.

## The ClassLoader

All classes in a Java application are loaded using some subclass of java.lang.ClassLoader. Loading classes dynamically must therefore also be done using a java.lang.ClassLoader subclass.

When a class is loaded, all classes it references are loaded too. This class loading pattern happens recursively, until all classes needed are loaded. This may not be all classes in the application. Unreferenced classes are not loaded until the time they are referenced.

## The ClassLoader Hierarchy

Class loaders in Java are organized into a hierarchy. When you create a new standard Java ClassLoaderyou must provide it with a parent ClassLoader. If a ClassLoader is asked to load a class, it will ask its parent class loader to load it. If the parent class loader can't find the class, the child class loader then tries to load it itself.

## Class Loading

The steps a given class loader uses when loading classes are:

1. Check if the class was already loaded.
2. If not loaded, ask parent class loader to load the class.
3. If parent class loader cannot load class, attempt to load it in this class loader.

When you implement a class loader that is capable of reloading classes you will need to deviate a bit from this sequence. The classes to reload should not be requested loaded by the parent class loader. More on that later.

## Dynamic Class Loading

Loading a class dynamically is easy. All you need to do is to obtain a ClassLoader and call its loadClass()method. Here is an example:

public class MainClass {

public static void main(String[] args){

ClassLoader classLoader = MainClass.class.getClassLoader();

try {

Class aClass = classLoader.loadClass("com.jenkov.MyClass");

System.out.println("aClass.getName() = " + aClass.getName());

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

## Dynamic Class Reloading

Dynamic class reloading is a bit more challenging. Java's builtin Class loaders always checks if a class is already loaded before loading it. Reloading the class is therefore not possible using Java's builtin class loaders. To reload a class you will have to implement your own ClassLoader subclass.

Even with a custom subclass of ClassLoader you have a challenge. Every loaded class needs to be linked. This is done using the ClassLoader.resolve() method. This method is final, and thus cannot be overridden in your ClassLoader subclass. The resolve() method will not allow any given ClassLoader instance to link the same class twice. Therefore, everytime you want to reload a class you must use a new instance of yourClassLoader subclass. This is not impossible, but necessary to know when designing for class reloading.

## Designing your Code for Class Reloading

As stated earlier you cannot reload a class using a ClassLoader that has already loaded that class once. Therefore you will have to reload the class using a different ClassLoader instance. But this poses som new challenges.

Every class loaded in a Java application is identified by its fully qualified name (package name + class name), and the ClassLoader instance that loaded it. That means, that a class MyObject loaded by class loader A, is not the same class as the MyObject class loaded with class loader B. Look at this code:

MyObject object = (MyObject)

myClassReloadingFactory.newInstance("com.jenkov.MyObject");

Notice how the MyObject class is referenced in the code, as the type of the object variable. This causes theMyObject class to be loaded by the same class loader that loaded the class this code is residing in.

If the myClassReloadingFactory object factory reloads the MyObject class using a different class loader than the class the above code resides in, you cannot cast the instance of the reloaded MyObject class to theMyObject type of the object variable. Since the two MyObject classes were loaded with different class loaders, the are regarded as different classes, even if they have the same fully qualified class name. Trying to cast an object of the one class to a reference of the other will result in a ClassCastException.

It is possible to work around this limitation but you will have to change your code in either of two ways:

1. Use an interface as the variable type, and just reload the implementing class.
2. Use a superclass as the variable type, and just reload a subclass.

Here are two coresponding code examples:

**MyObjectInterface** object = **(MyObjectInterface)**

myClassReloadingFactory.newInstance("com.jenkov.MyObject");

**MyObjectSuperclass** object = **(MyObjectSuperclass)**

myClassReloadingFactory.newInstance("com.jenkov.MyObject");

Either of these two methods will work if the type of the variable, the interface or superclass, is not reloaded when the implementing class or subclass is reloaded.

To make this work you will of course need to implement your class loader to let the interface or superclass be loaded by its parent. When your class loader is asked to load the MyObject class, it will also be asked to load the MyObjectInterface class, or the MyObjectSuperclass class, since these are referenced from within the MyObject class. Your class loader must delegate the loading of those classes to the same class loader that loaded the class containing the interface or superclass typed variables.

## ClassLoader Load / Reload Example

The text above has contained a lot of talk. Let's look at a simple example. Below is an example of a simpleClassLoader subclass. Notice how it delegates class loading to its parent except for the one class it is intended to be able to reload. If the loading of this class is delegated to the parent class loader, it cannot be reloaded later. Remember, a class can only be loaded once by the same ClassLoader instance.

As said earlier, this is just an example that serves to show you the basics of a ClassLoader's behaviour. It is not a production ready template for your own class loaders. Your own class loaders should probably not be limited to a single class, but a collection of classes that you know you will need to reload. In addition, you should probably not hardcode the class paths either.

public class MyClassLoader extends ClassLoader{

public MyClassLoader(ClassLoader parent) {

super(parent);

}

public Class loadClass(String name) throws ClassNotFoundException {

if(!"reflection.MyObject".equals(name))

return super.loadClass(name);

try {

String url = "file:C:/data/projects/tutorials/web/WEB-INF/" +

"classes/reflection/MyObject.class";

URL myUrl = new URL(url);

URLConnection connection = myUrl.openConnection();

InputStream input = connection.getInputStream();

ByteArrayOutputStream buffer = new ByteArrayOutputStream();

int data = input.read();

while(data != -1){

buffer.write(data);

data = input.read();

}

input.close();

byte[] classData = buffer.toByteArray();

return defineClass("reflection.MyObject",

classData, 0, classData.length);

} catch (MalformedURLException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

return null;

}

}

Below is an example use of the MyClassLoader.

public static void main(String[] args) throws

ClassNotFoundException,

IllegalAccessException,

InstantiationException {

ClassLoader parentClassLoader = MyClassLoader.class.getClassLoader();

MyClassLoader classLoader = new MyClassLoader(parentClassLoader);

Class myObjectClass = classLoader.loadClass("reflection.MyObject");

AnInterface2 object1 =

(AnInterface2) myObjectClass.newInstance();

MyObjectSuperClass object2 =

(MyObjectSuperClass) myObjectClass.newInstance();

//create new class loader so classes can be reloaded.

classLoader = new MyClassLoader(parentClassLoader);

myObjectClass = classLoader.loadClass("reflection.MyObject");

object1 = (AnInterface2) myObjectClass.newInstance();

object2 = (MyObjectSuperClass) myObjectClass.newInstance();

}

Here is the reflection.MyObject class that is loaded using the class loader. Notice how it both extends a superclass and implements an interface. This is just for the sake of the example. In your own code you would only have to one of the two - extend or implement.

public class MyObject extends MyObjectSuperClass implements AnInterface2{

//... body of class ... override superclass methods

// or implement interface methods

}