## Internals of Java Class Loading

<http://www.onjava.com/pub/a/onjava/2005/01/26/classloading.html?page=1>

Class loading is one of the most powerful mechanisms provided by the Java language specification. Even though the internals of class loading falls under the "advanced topics" heading, all Java programmers should know how the mechanism works and what can be done with it to suit their needs. This can save time that would otherwise have been spent debugging[ClassNotFoundException](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ClassNotFoundException.html), [ClassCastException](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ClassCastException.html" \t "_blank), etc.

This article starts from the basics, such as the difference between code and data, and how they are related to form an instance or object. Then it looks into the mechanism of loading code into the JVM with the help of class loaders, and the main type of class loaders available in Java. The article then looks into the internals of class loaders, where we cover using the basic algorithm (or *probing*), followed by class loaders before it loads a class. The next section of the article uses code examples to demonstrate the necessity for developers to extend and develop their own class loaders. This is followed by explanation on writing your own class loaders and how to use them to make a generic task-execution engine that can be used to load the code supplied by any remote client, define it in the JVM, and instantiate and then execute it. The article concludes with references to J2EE-specific components where custom class loading schemas becomes the norm.

**Class and Data**

A *class* represents the code to be executed, whereas *data* represents the state associated with that code. State can change; code generally does not. When we associate a particular state to a class, we have an instance of that class. So different instances of the same class can have different state, but all refer to the same code. In Java, a class will usually have its code contained in a .class file, though there are exceptions. Nevertheless, in the Java runtime, each and every class will have its code also available in the form of a first-class Java object, which is an instance of [java.lang.Class](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Class.html" \t "_blank). Whenever we compile any Java file, the compiler will embed a public, static, final field named class, of the type java.lang.Class, in the emitted byte code. Since this field is public, we can access it using dotted notation, like this:

java.lang.Class klass = Myclass.class;

Once a class is loaded into a JVM, the same class (I repeat, the **same** class) will not be loaded again. This leads to the question of what is meant by "the same class." Similar to the condition that an object has a specific state, an identity, and that an object is always associated with its code (class), a class loaded into a JVM also has a specific identity, which we'll look at now.

In Java, a class is identified by its fully qualified class name. The fully qualified class name consists of the package name and the class name. But a class is uniquely identified in a JVM using its fully qualified class name along with the instance of theClassLoader that loaded the class. Thus, if a class named Cl in the package Pg is loaded by an instance kl1 of the class loaderKlassLoader, the class instance of C1, i.e. *C1.class* is keyed in the JVM as (Cl, Pg, kl1). This means that the two class loader instances (Cl, Pg, kl1) and (Cl, Pg, kl2) are not one and the same, and classes loaded by them are also completely different and not type-compatible to each other. How many class loader instances do we have in a JVM? The next section explains this.

**Class Loaders**

In a JVM, each and every class is loaded by some instance of a [java.lang.ClassLoader](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ClassLoader.html" \t "_blank). The ClassLoader class is located in thejava.lang package and developers are free to subclass it to add their own functionality to class loading.

Whenever a new JVM is started by typing java MyMainClass, the "bootstrap class loader" is responsible for loading key Java classes like java.lang.Object and other runtime code into memory first. The runtime classes are packaged inside of the*JRE\lib\rt.jar* file. We cannot find the details of the bootstrap class loader in the Java documentation, since this is a native implementation. For the same reason, the behavior of the bootstrap class loader will also differ across JVMs.

In a related note, we will get null if we try to get the class loader of a core Java runtime class, like this:

log(java.lang.String.class.getClassLoader());

Next comes the Java extension class loader. We can store extension libraries, those that provide features that go beyond the core Java runtime code, in the path given by the java.ext.dirs property. The ExtClassLoader is responsible for loading all .jar files kept in the java.ext.dirs path. A developer can add his or her own application .jar files or whatever libraries he or she might need to add to the classpath to this extension directory so that they will be loaded by the extension class loader.

The third and most important class loader from the developer perspective is the AppClassLoader. The application class loader is responsible for loading all of the classes kept in the path corresponding to the java.class.path system property.

"[Understanding Extension Class Loading](http://java.sun.com/docs/books/tutorial/ext/basics/load.html)" in Sun's Java tutorial explains more on the above three class loader paths. Listed below are a few other class loaders in the JDK:

* java.net.URLClassLoader
* java.security.SecureClassLoader
* java.rmi.server.RMIClassLoader
* sun.applet.AppletClassLoader

java.lang.Thread, contains the method public ClassLoader getContextClassLoader(), which returns the context class loader for a particular thread. The context class loader is provided by the creator of the thread for use by code running in this thread when loading classes and resources. If it is not set, the default is the class loader context of the parent thread. The context class loader of the primordial thread is typically set to the class loader used to load the application.

**How Class Loaders Work**

All class loaders except the bootstrap class loader have a parent class loader. Moreover, all class loaders are of the typejava.lang.ClassLoader. The above two statements are different, and very important for the correct working of any class loaders written by developers. The most important aspect is to correctly set the parent class loader. The parent class loader for any class loader is the class loader instance that loaded that class loader. (Remember, a class loader is itself a class!)

A class is requested out of a class loader using the loadClass() method. The internal working of this method can be seen from the source code for java.lang.ClassLoader, given below:

protected synchronized Class<?> loadClass

(String name, boolean resolve)

throws ClassNotFoundException{

// First check if the class is already loaded

Class c = findLoadedClass(name);

if (c == null) {

try {

if (parent != null) {

c = parent.loadClass(name, false);

} else {

c = findBootstrapClass0(name);

}

} catch (ClassNotFoundException e) {

// If still not found, then invoke

// findClass to find the class.

c = findClass(name);

}

}

if (resolve) {

resolveClass(c);

}

return c;

}

To set the parent class loader, we have two ways to do so in the ClassLoader constructor:

public class MyClassLoader extends ClassLoader{

public MyClassLoader(){

super(MyClassLoader.class.getClassLoader());

}

}

or

public class MyClassLoader extends ClassLoader{

public MyClassLoader(){

super(getClass().getClassLoader());

}

}

The first method is preferred because calling the method getClass() from within the constructor should be discouraged, since the object initialization will be complete only at the exit of the constructor code. Thus, if the parent class loader is correctly set, whenever a class is requested out of a ClassLoader instance, if it cannot find the class, it should ask the parent first. If the parent cannot find it (which again means that its parent also cannot find the class, and so on), and if the findBootstrapClass0() method also fails, the findClass() method is invoked. The default implementation of findClass() will throw ClassNotFoundException and developers are expected to implement this method when they subclass java.lang.ClassLoader to make custom class loaders. The default implementation of findClass() is shown below.

protected Class<?> findClass(String name)

throws ClassNotFoundException {

throw new ClassNotFoundException(name);

}

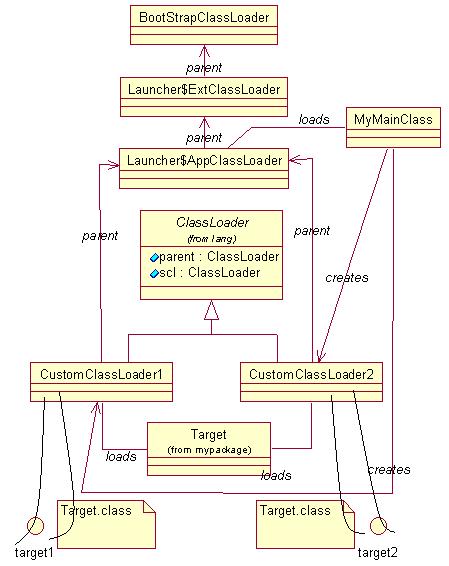
Inside of the findClass() method, the class loader needs to fetch the byte codes from some arbitrary source. The source can be the file system, a network URL, a database, another application that can spit out byte codes on the fly, or any similar source that is capable of generating byte code compliant with the Java byte code specification. You could even use [BCEL](http://jakarta.apache.org/bcel/) (Byte Code Engineering Library), which provides convenient methods to create classes from scratch at runtime. BCEL is being used successfully in several projects such as compilers, optimizers, obsfuscators, code generators, and analysis tools. Once the byte code is retrieved, the method should call the defineClass() method, and the runtime is very particular about which ClassLoader instance calls this method. Thus, if two ClassLoader instances define byte codes from the same or different sources, the defined classes are different.

The [Java language specification](http://java.sun.com/docs/books/jls/second_edition/html/jTOC.doc.html) gives a detailed explanation on the process of [loading](http://java.sun.com/docs/books/jls/second_edition/html/execution.doc.html#44459), [linking](http://java.sun.com/docs/books/jls/second_edition/html/execution.doc.html#44487), and the [initialization](http://java.sun.com/docs/books/jls/second_edition/html/execution.doc.html#44557) of classes and interfaces in the Java Execution Engine.

Figure 1 shows an application with a main class called MyMainClass. As explained earlier, MyMainClass.class will be loaded by the AppClassLoader. MyMainClass creates instances of two class loaders, CustomClassLoader1 and CustomClassLoader2, which are capable of finding the byte codes of a fourth class called Target from some source (say, from a network path). This means the class definition of the Target class is not in the application class path or extension class path. In such a scenario, ifMyMainClass asks the custom class loaders to load the Target class, Target will be loaded and Target.class will be defined independently by both CustomClassLoader1 and CustomClassLoader2. This has serious implications in Java. If some static initialization code is put in the Target class, and if we want this code to be executed one and only once in a JVM, in our current setup the code will be executed twice in the JVM: once each when the class is loaded separately by both CustomClassLoaders. If theTarget class is instantiated in both the CustomClassLoaders to have the instances target1 and target2 as shown in Figure 1, then target1 and target2 are not type-compatible. In other words, the JVM cannot execute the code:

Target target3 = (Target) target2;

The above code will throw a ClassCastException. This is because the JVM sees these two as separate, distinct class types, since they are defined by different ClassLoader instances. The above explanation holds true even if MyMainClass doesn't use two separate class loader classes like CustomClassLoader1 and CustomClassLoader2, and instead uses two separate instances of a single CustomClassLoader class. This is demonstrated later in the article with code examples.

  
*Figure 1. Multiple ClassLoaders loading the same Target class in the same JVM*

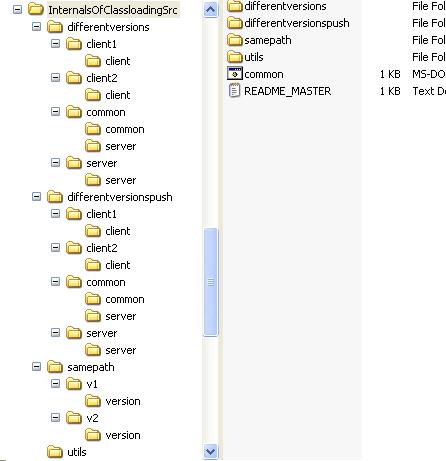
A more detailed explanation on the process of class loading, defining, and linking is in Andreas Schaefer's article "[Inside Class Loaders](http://www.onjava.com/pub/a/onjava/2003/11/12/classloader.html)."

**Why Do We Need our Own Class Loaders?**

One of the reasons for a developer to write his or her own class loader is to control the JVM's class loading behavior. A class in Java is identified using its package name and class name. For classes that implement java.io.Serializable, theserialVersionUID plays a major role in versioning the class. This stream-unique identifier is a 64-bit hash of the class name, interface class names, methods, and fields. Other than these, there are no other straightforward mechanisms for versioning a class. Technically speaking, if the above aspects match, the classes are of "same version."

But let us think of a scenario where we need to develop a generic Execution Engine, capable of executing any tasks implementing a particular interface. When the tasks are submitted to the engine, first the engine needs to load the code for the task. Suppose different clients submit different tasks (i.e., different code) to the engine, and by chance, all of these tasks have the same class name and package name. The question is whether the engine will load the different client versions of the task differently for different client invocation contexts so that the clients will get the output they expect. The phenomenon is demonstrated in the sample code download, located in the [References](http://www.onjava.com/pub/a/onjava/2005/01/26/classloading.html?page=2#references) section below. Two directories, samepath and differentversions, contain separate examples to demonstrate the concept.

Figure 2 shows how the examples are arranged in three separate subfolders, called samepath, differentversions, anddifferentversionspush:

  
Figure 2. Example folder structure arrangement

In samepath, we have version.Version classes kept in two subdirectories, v1 and v2. Both classes have the same name and same package. The only difference between the two classes is in the following lines:

public void fx(){

log("this = " + this + "; Version.fx(1).");

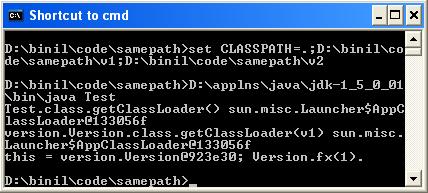
}

inside of v1, we have Version.fx(1) in the log statement, whereas in v2, we have Version.fx(2). Put both these slightly different versions of the classes in the same classpath, and run the Test class:

set CLASSPATH=.;%CURRENT\_ROOT%\v1;%CURRENT\_ROOT%\v2

%JAVA\_HOME%\bin\java Test

This will give the console output shown in Figure 3. We can see that code corresponding to Version.fx(1) is loaded, since the class loader found that version of the code first in the classpath.

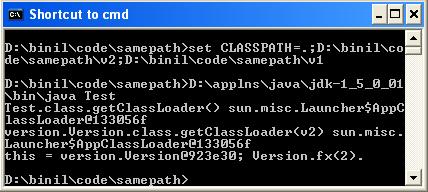
  
Figure 3. samepath test with version 1 first in the classpath

Repeat the run, with a slight change in the order of path elements in class path.

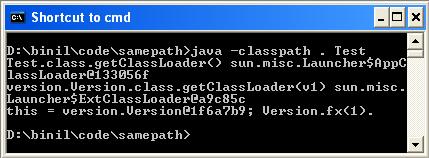
set CLASSPATH=.;%CURRENT\_ROOT%\v2;%CURRENT\_ROOT%\v1

%JAVA\_HOME%\bin\java Test

The console output is now changed to that shown in Figure 4. Here, the code corresponding to Version.fx(2) is loaded, since the class loader found that version of the code first in the classpath.

  
Figure 4. samepath test with version 2 first in the classpath

From the above example it is obvious that the class loader will try to load the class using the path element that is found first. Also, if we delete the version.Version classes from v1 and v2, make a .jar (myextension.jar) out of version.Version, put it in the path corresponding to java.ext.dirs, and repeat the test, we see that version.Version is no longer loaded byAppClassLoader but by the extension class loader, as shown in Figure 5.

  
Figure 5. AppClassLoader and ExtClassLoader

Going forward with the examples, the folder differentversions contains an RMI execution engine. Clients can supply any tasks that implement common.TaskIntf to the execution engine. The subfolders client1 and client2 contain slightly different versions of the class client.TaskImpl. The difference between the two classes is in the following lines:

static{

log("client.TaskImpl.class.getClassLoader

(v1) : " + TaskImpl.class.getClassLoader());

}

public void execute(){

log("this = " + this + "; execute(1)");

}

Instead of the getClassLoader(v1) and execute(1) log statements in execute() inside of client1, client2 hasgetClassLoader(v2) and execute(2) log statements. Moreover, in the script to start the Execution Engine RMI server, we have arbitrarily put the task implementation class of client2 first in the classpath.

CLASSPATH=%CURRENT\_ROOT%\common;%CURRENT\_ROOT%\server;

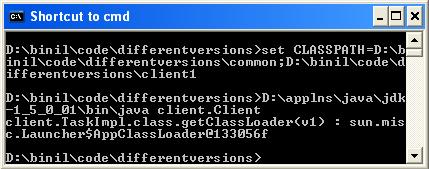
%CURRENT\_ROOT%\client2;%CURRENT\_ROOT%\client1

%JAVA\_HOME%\bin\java server.Server

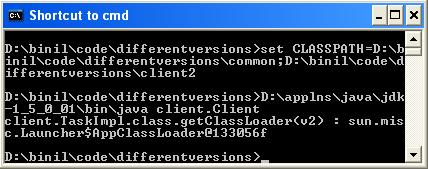
The screenshots in Figures 6, 7, and 8 show what is happening under the hood. Here, in the client VMs, separateclient.TaskImpl classes are loaded, instantiated, and sent to the Execution Engine Server VM for execution. From the server console, it is apparent that client.TaskImpl code is loaded only once in the server VM. This single "version" of the code is used to regenerate many client.TaskImpl instances in the server VM, and execute the task.

  
Figure 6. Execution Engine Server console

Figure 6 shows the Execution Engine Server console, which is loading and executing code on behalf of two separate client requests, as shown in Figures 7 and Figure 8. The point to note here is that the code is loaded only once (as is evident from the log statement inside of the static initialization block), but the method is executed twice for each client invocation context.

  
Figure 7. Execution Engine Client 1 console

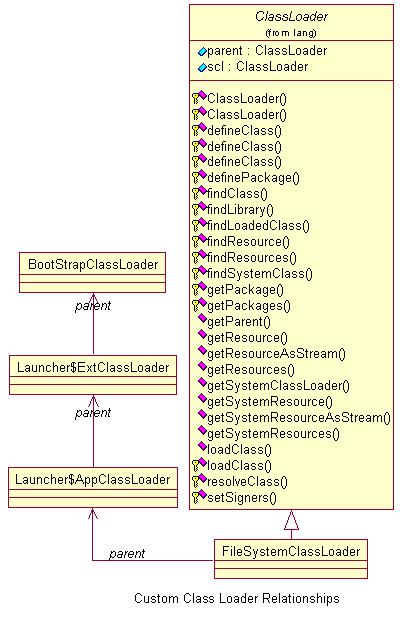
In Figure 7, the code for the TaskImpl class containing the log statement client.TaskImpl.class.getClassLoader(v1) is loaded by the client VM, and supplied to the Execution Engine Server. The client VM in Figure 8 loads different code for the TaskImplclass containing the log statement client.TaskImpl.class.getClassLoader(v2), and supplies it to the Server VM.

  
Figure 8. Execution Engine Client 2 console

Here, in the client VMs, separate client.TaskImpl classes are loaded, instantiated, and sent to the Execution Engine Server VM for execution. A second look at the server console in Figure 6 reveals that the client.TaskImpl code is loaded only once in the server VM. This single "version" of the code is used to regenerate the client.TaskImpl instances in the server VM, and execute the task. Client 1 should be unhappy since instead of his "version" of the client.TaskImpl(v1), it is some other code that is executed in the server against Client 1's invocation! How do we tackle such scenarios? The answer is to implement custom class loaders.

**Custom Class Loaders**

The solution to fine-control class loading is to implement custom class loaders. Any custom class loader should havejava.lang.ClassLoader as its direct or distant super class. Moreover, in the constructor, we need to set the parent class loader, too. Then, we have to override the findClass() method. The *differentversionspush* folder contains a custom class loader calledFileSystemClassLoader. Its structure is shown in Figure 9:

  
*Figure 9. Custom class loader relationship*

Below are the main methods implemented in common.FileSystemClassLoader:

public byte[] findClassBytes(String className){

try{

String pathName = currentRoot +

File.separatorChar + className.

replace('.', File.separatorChar)

+ ".class";

FileInputStream inFile = new

FileInputStream(pathName);

byte[] classBytes = new

byte[inFile.available()];

inFile.read(classBytes);

return classBytes;

}

catch (java.io.IOException ioEx){

return null;

}

}

public Class findClass(String name)throws

ClassNotFoundException{

byte[] classBytes = findClassBytes(name);

if (classBytes==null){

throw new ClassNotFoundException();

}

else{

return defineClass(name, classBytes,

0, classBytes.length);

}

}

public Class findClass(String name, byte[]

classBytes)throws ClassNotFoundException{

if (classBytes==null){

throw new ClassNotFoundException(

"(classBytes==null)");

}

else{

return defineClass(name, classBytes,

0, classBytes.length);

}

}

public void execute(String codeName,

byte[] code){

Class klass = null;

try{

klass = findClass(codeName, code);

TaskIntf task = (TaskIntf)

klass.newInstance();

task.execute();

}

catch(Exception exception){

exception.printStackTrace();

}

}

This class is used by the client to convert the client.TaskImpl(v1) to a byte[]. This byte[] is then send to the RMI Server Execution Engine. In the server, the same class is used for defining the class back from the code in the form of byte[]. The client-side code is shown below:

public class Client{

public static void main (String[] args){

try{

byte[] code = getClassDefinition

("client.TaskImpl");

serverIntf.execute("client.TaskImpl",

code);

}

catch(RemoteException remoteException){

remoteException.printStackTrace();

}

}

private static byte[] getClassDefinition

(String codeName){

String userDir = System.getProperties().

getProperty("BytePath");

FileSystemClassLoader fscl1 = null;

try{

fscl1 = new FileSystemClassLoader

(userDir);

}

catch(FileNotFoundException

fileNotFoundException){

fileNotFoundException.printStackTrace();

}

return fscl1.findClassBytes(codeName);

}

}

Inside of the execution engine, the code received from the client is given to the custom class loader. The custom class loader will define the class back from the byte[], instantiate the class, and execute. The notable point here is that, for each client request, we use separate instances of the FileSystemClassLoader class to define the client-supplied client.TaskImpl. Moreover, theclient.TaskImpl is not available in the class path of the server. This means that when we call findClass() on theFileSystemClassLoader, the findClass() method calls defineClass() internally, and the client.TaskImpl class gets defined by that particular instance of the class loader. So when a new instance of the FileSystemClassLoader is used, the class is defined from the byte[] all over again. Thus, for each client invocation, class client.TaskImpl is defined again and again and we are able to execute "different versions" of the client.TaskImpl code inside of the same Execution Engine JVM.

public void execute(String codeName, byte[] code)throws RemoteException{

FileSystemClassLoader fileSystemClassLoader = null;

try{

fileSystemClassLoader = new FileSystemClassLoader();

fileSystemClassLoader.execute(codeName, code);

}

catch(Exception exception){

throw new RemoteException(exception.getMessage());

}

}

Examples are in the *differentversionspush* folder. The server and client side consoles are shown in Figures 10, 11, and 12:

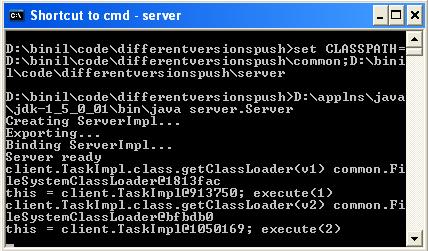
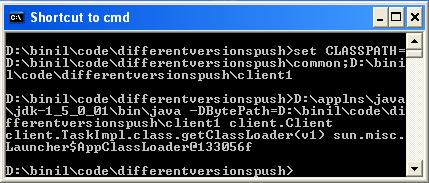
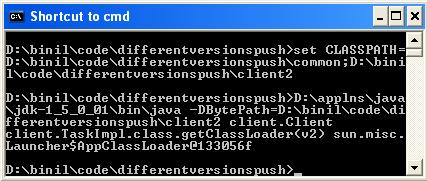
  
*Figure 10. Custom class loader execution engine*

Figure 10 shows the custom class loader Execution Engine VM console. We can see the client.TaskImpl code is loaded more than once. In fact, for each client execution context, the class is newly loaded and instantiated.

  
*Figure 11. Custom class loader engine, Client 1*

In Figure 11, the code for the TaskImpl class containing the log statement client.TaskImpl.class.getClassLoader(v1) is loaded by the client VM, and pushed to the Execution Engine Server VM. The client VM in Figure 12 loads a different code for theTaskImpl class containing the log statement client.TaskImpl.class.getClassLoader(v2), and pushes to the Server VM.

  
*Figure 12. Custom class loader engine, Client 2*

This code example shows how we can leverage separate instances of class loaders to have side-by-side execution of "different versions" of code in the same VM.

**Class Loaders In J2EE**

The class loaders in some J2EE servers tend to drop and reload classes at different intervals. This will occur in some implementations and may not on others. Similarly, a web server may decide to remove a previously loaded servlet instance, perhaps because it is explicitly asked to do so by the server administrator, or because the servlet has been idle for a long time. When a request is first made for a JSP (assuming it hasn't been precompiled), the JSP engine will translate the JSP into its page implementation class, which takes the form of a standard Java servlet. Once the page's implementation servlet has been created, it will be compiled into a class file by the JSP engine and will be ready for use. Each time a container receives a request, it first checks to see if the JSP file has changed since it was last translated. If it has, it's retranslated so that the response is always generated by the most up-to-date implementation of the JSP file. Enterprise application deployment units in the form of .ear, .war, .rar, etc. will also needs to be loaded and reloaded at will or as per configured policies. For all of these scenarios, loading, unloading and reloading is possible only if we have control over the application server's JVM's class-loading policy. This is attained by an extended class loader, which can execute the code defined in its boundary. Brett Peterson has given an explanation of class loading schemas in a J2EE application server context in his article "[Understanding J2EE Application Server Class Loading Architectures](http://www.theserverside.com/articles/article.tss?l=ClassLoading)" at [TheServerSide.com](http://www.theserverside.com/).

**Summary**

The article talked about how classes loaded into a Java virtual machine are uniquely identified and what limitations exist when we try to load different byte codes for classes with the same names and packages. Since there is no explicit class versioning mechanism, if we want to load classes at our own will, we have to use custom class loaders with extended capabilities. Many J2EE application servers have a "hot deployment" capability, where we can reload an application with a new version of class definition, without bringing the server VM down. Such application servers make use of custom class loaders. Even if we don't use an application server, we can create and use custom class loaders to finely control class loading mechanisms in our Java applications. Ted Neward's book [Server-Based Java Programming](http://www.manning.com/neward3) throws light onto the ins and outs of Java class loading, and it teaches those concepts of Java that underlie the J2EE APIs and the best ways to use them.

**References**

* [Sample code](http://www.onjava.com/onjava/2005/01/26/examples/InternalsOfClassloadingSrc.zip) for this article
* [JDK 1.5 API Docs](http://java.sun.com/j2se/1.5.0/docs/api/)
* The [Java language specification](http://java.sun.com/docs/books/jls/second_edition/html/jTOC.doc.html)
* "[Understanding Extension Class Loading](http://java.sun.com/docs/books/tutorial/ext/basics/load.html)" in the Java tutorial
* "[Inside Class Loaders](http://www.onjava.com/pub/a/onjava/2003/11/12/classloader.html)" from [ONJava](http://www.onjava.com/)
* "[Inside Class Loaders: Debugging](http://www.onjava.com/pub/a/onjava/2004/06/30/classloader2.html)" from [ONJava](http://www.onjava.com/)
* "[What version is your Java code?](http://www.javaworld.com/javaqa/2003-05/02-qa-0523-version.html)" from [JavaWorld](http://www.javaworld.com/)
* "[Understanding J2EE Application Server Class Loading Architectures](http://www.theserverside.com/articles/article.tss?l=ClassLoading)" from [TheServerSide](http://www.theserverside.com/)
* [Byte Code Engineering Library](http://jakarta.apache.org/bcel/)
* [Server-Based Java Programming](http://www.manning.com/neward3) by Ted Neward