|  |
| --- |
| [ONJava.com -- The Independent Source for Enterprise Java](http://www.onjava.com/) |

|  |  |
| --- | --- |
| [oreilly.com](http://www.oreilly.com/)[Safari Books Online.](http://safari.oreilly.com/)[Conferences.](http://oreilly.com/conferences/) | |
| Top of Form      Bottom of Form |  |
| advertisement | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  | | | http://www.onjava.com/images/trans.gif | **ONJava Topics** [All Articles](http://www.onjava.com/pub/q/all_onjava_articles)[Best Practices](http://www.onjava.com/topics/java/Java_Design)[Enterprise JavaBeans](http://www.onjava.com/topics/java/EJB)[Java and XML](http://www.onjava.com/topics/java/java_xml)[Java Data Objects](http://www.onjava.com/topics/java/jdo)[Java EE (Enterprise)](http://www.onjava.com/topics/java/J2EE)[Java IDE Tools](http://www.onjava.com/topics/java/java_ide)[Java Media](http://www.onjava.com/topics/java/Java_Media)[Java SE (Standard)](http://www.onjava.com/topics/java/J2SE)[Java Security](http://www.onjava.com/topics/java/Java_Security)[Java SysAdmin](http://www.onjava.com/topics/java/javasys)[JDO/JDBC/SQLJ](http://www.onjava.com/topics/java/jdbc_sqlj)[JSP and Servlets](http://www.onjava.com/topics/java/JSP_Servlets)[Open Source Java](http://www.onjava.com/topics/java/open_source_java)[P2P Java](http://www.onjava.com/topics/java/P2P_Java)[Web Services](http://www.onjava.com/topics/java/wsdl_uddi)[Wireless Java](http://www.onjava.com/topics/java/Wireless_Java) | |  | [AddThis Social Bookmark Button](http://www.addthis.com/bookmark.php)  [Print](http://www.onjava.com/lpt/a/5586)[Subscribe to ONJava](http://www.oreillynet.com/pub/feed/7)[Subscribe to Newsletters](http://www.oreillynet.com/cs/nl/home)  [ShareThis](javascript:void(0))  **Internals of Java Class LoadingInternals of Java Class Loading**  by [Binildas Christudas](http://www.onjava.com/pub/au/2121) 01/26/2005  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), 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 the ClassLoader that loaded the class. Thus, if a class named Cl in the package Pg is loaded by an instance kl1 of the class loader KlassLoader, 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 the java.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.pathsystem 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 type java.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/" \t "_blank) (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 thedefineClass() 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" \t "_blank) gives a detailed explanation on the process of [loading](http://java.sun.com/docs/books/jls/second_edition/html/execution.doc.html" \l "44459" \t "_blank), [linking](http://java.sun.com/docs/books/jls/second_edition/html/execution.doc.html" \l "44487" \t "_blank), and the[initialization](http://java.sun.com/docs/books/jls/second_edition/html/execution.doc.html" \l "44557" \t "_blank) 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 andCustomClassLoader2, 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, if MyMainClass asks the custom class loaders to load the Target class,Target will be loaded and Target.class will be defined independently by both CustomClassLoader1 andCustomClassLoader2. This has serious implications in Java. If some static initialization code is put in the Targetclass, 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 ifMyMainClass 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.  http://www.onjava.com/onjava/2005/01/26/graphics/Figure01_MultipleClassLoaders.JPG *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" \t "_blank)."   |  |  | | --- | --- | | Pages: **1**, [2](http://www.onjava.com/pub/a/onjava/2005/01/26/classloading.html?page=2), [3](http://www.onjava.com/pub/a/onjava/2005/01/26/classloading.html?page=3) | [Next Page[arrow](http://www.onjava.com/pub/a/onjava/2005/01/26/classloading.html?page=2)](http://www.onjava.com/pub/a/onjava/2005/01/26/classloading.html?page=2) | |  |  |  | |  | | --- | | **Recommended for You**  **Tagged Articles**  [Be the first to post this article to del.icio.us](http://del.icio.us/post) | |  |  |
|  |  |  |  |  |  |  |  |

|  |
| --- |
|  |
| © 2014, O’Reilly Media, Inc.  (707) 827-7019 (800) 889-8969  All trademarks and registered trademarks appearing on oreilly.com are the property of their respective owners.  **About O'Reilly**   * [Academic Solutions](http://oreilly.com/academic/) * [Jobs](http://oreilly.com/jobs/) * [Contacts](http://oreilly.com/contact.html) * [Corporate Information](http://oreilly.com/about/) * [Press Room](http://press.oreilly.com/index.html) * [Privacy Policy](http://oreilly.com/oreilly/privacy.csp) * [Terms of Service](http://oreilly.com/terms/) * [Writing for O'Reilly](http://oreilly.com/oreilly/author/intro.csp)   **Community**   * [Authors](http://oreilly.com/authors/) * [Community & Featured Users](http://oreilly.com/community/) * [Forums](http://forums.oreilly.com/) * [Membership](https://members.oreilly.com/) * [Newsletters](http://elists.oreilly.com/) * [O'Reilly Answers](http://answers.oreilly.com/) * [RSS Feeds](http://oreilly.com/feeds/) * [User Groups](http://ug.oreilly.com/)   **Partner Sites**   * [makezine.com](http://makezine.com/) * [makerfaire.com](http://makerfaire.com/) * [craftzine.com](http://craftzine.com/) * [igniteshow.com](http://igniteshow.com/) * [PayPal Developer Zone](https://www.x.com/community/ppx/devzone) * [O'Reilly Insights on Forbes.com](http://blogs.forbes.com/oreillymedia/)   **Shop O'Reilly**   * [Customer Service](http://shop.oreilly.com/category/customer-service.do) * [Contact Us](http://shop.oreilly.com/category/customer-service.do) * [Shipping Information](http://shop.oreilly.com/category/customer-service/shipping-information.do) * [Ordering & Payment](http://shop.oreilly.com/category/customer-service/ordering-payment.do) * [The O'Reilly Guarantee](http://shop.oreilly.com/category/customer-service/oreilly-guarantee.do) |

|  |
| --- |
| http://www2.sys-con.com/itsg/virtualcd/java/images/top.jpg |
|

|  |  |  |
| --- | --- | --- |
| http://www2.sys-con.com/itsg/virtualcd/java/images/side.jpg | |  | | --- | | http://www2.sys-con.com/itsg/virtualcd/java/archives/0808/chaudhri/title.jpg  The Java platform was designed to be robust, secure, and extensible in order to support the mobility of code and data. The Java ClassLoader in the Java Virtual Machine (JVM) is a key component in the realization of these goals.  The JVM is responsible for loading and executing code on the Java platform. It uses a ClassLoader to load Java classes into the Java runtime environment. ClassLoaders are architected so that at start-up the JVM doesn't need to know anything about the classes that will be loaded at runtime. Almost all Java-based containers such as EJB or servlet containers implement custom ClassLoaders to support features like hot deployment and runtime platform extensibility. An in-depth understanding of ClassLoaders is important for developers when implementing such Java-based containers.  For enterprise developers who develop components that are deployed on these containers, this knowledge will help you understand how the container works and with debugging problems. This article presents the Java ClassLoader architecture and discusses the implications of ClassLoaders on platform security and extensibility as well as a method to implement user-defined ClassLoaders.  The smallest unit of execution that gets loaded by a ClassLoader is the Java class file. A class file contains the binary representation of a Java class, which has the executable bytecodes and references to other classes used by that class, including references to classes in the Java API. Stated simply, a ClassLoader locates the bytecodes for a Java class that needs to be loaded, reads the bytecodes, and creates an instance of the java.lang.Class class. This makes the class available to the JVM for execution. Initially when a JVM starts up, nothing is loaded into it. The class file of the program being executed is loaded first and then other classes and interfaces are loaded as they get referenced in the bytecode being executed. The JVM thus exhibits lazy loading, i.e., loading classes only when required, so at start-up the JVM doesn't need to know the classes that would get loaded during runtime. Lazy loading plays a key role in providing dynamic extensibility to the Java platform. The Java runtime can be customized in interesting ways by implementing a custom ClassLoader in a program, as I'll discuss later.  **Java 2 ClassLoader Delegation Model** Multiple instances of ClassLoaders exist in the Java 2 runtime, each loading classes from different code repositories. For instance, Java core API classes are loaded by the bootstrap (or primordial) ClassLoader. Application-specific classes are loaded by the system (or application) ClassLoader. In addition, an application can define its own ClassLoaders to load code from custom repositories. Java 2 defines a parent-child relationship between ClassLoaders. Each ClassLoader except the bootstrap ClassLoader has a parent ClassLoader, conceptually forming a treelike structure of ClassLoaders. The bootstrap ClassLoader is the root of this tree and thus doesn't have a parent. The relationship is depicted in Figure 1.  Figure 1  The following is a high-level class-loading algorithm executed by a ClassLoader when a client requests it to load a class: 1.   A check is performed to see if the requested class has already been loaded by the current ClassLoader. If so, the loaded class is returned and the request is completed. The JVM caches all the classes that are loaded by a ClassLoader. A class that has previously been loaded by a ClassLoader is not loaded again. 2.   If the class is not already loaded, the request is delegated to the parent ClassLoader, before the current ClassLoader tries to load it. This delegation can go all the way up to the bootstrap ClassLoader, after which no further delegation is possible. 3.   If a parent fails to return a class because it was unable to load it, the current ClassLoader will then try to search for the requested class. Each ClassLoader has defined locations where it searches for classes to load. For instance, the bootstrap ClassLoader searches in the locations (directories and zip/jar files) specified in the sun.boot.class.path system property. The system ClassLoader searches for classes in the locations specified by the classpath (set as the java.class.path system property) command-line variable passed in when a JVM starts executing. If the class is found, it's loaded into the system and returned, completing the request. 4.   If the class is not found, a java.lang.ClassNotFoundException is thrown.  **Implementing a Java 2 Custom ClassLoader** As mentioned earlier the Java platform allows an application programmer to customize the classloading behavior by implementing a custom ClassLoader. This section shows how.  ClassLoaders (all but the bootstrap ClassLoader, which is implemented in native code in the JVM) are implemented by extending the java.lang.Class Loader class. The following code shows the relevant methods of the Java 2 ClassLoader API:  1. public abstract class ClassLoader extends Object { 2. protected ClassLoader(ClassLoader parent); 3. protected final Class defineClass( 4. String name,byte[] b,int off,int len) 5. throws ClassFormatError 5. protected Class findClass(String 7. className) throws ClassNotFoundException 6. public class loadClass(String className) 7. throws ClassNotFoundException 8.}  Each ClassLoader is assigned a parent when it's created, as per the parent-delegation model. Clients invoke the loadClass method on an instance of a ClassLoader to load a class. This initiates the classloading algorithm as explained earlier. Prior to Java 2, the loadClass method in the java.lang.ClassLoader class was declared abstract, requiring custom ClassLoaders to implement it when extending the java.lang.ClassLoader class. Implementing the loadClass method is rather complicated, so this has been changed in Java 2. With the introduction of the ClassLoader parent delegation model, java.lang.ClassLoader has an implementation of the loadClass method, which is essentially a template method that executes the classloading algorithm. The loadClass method invokes the findClass method (introduced in Java 2) in Step 3 of the classloading algorithm. Custom ClassLoaders should override this method to provide a custom way of locating and loading a Java class. This greatly simplifies the implementation of a custom ClassLoader.[Listing 1](http://www2.sys-con.com/itsg/virtualcd/java/archives/0808/chaudhri/index.html#s1) shows some code from the CustomClassLoader.java class ([the complete source code can be downloaded below](http://www2.sys-con.com/itsg/virtualcd/java/archives/0808/chaudhri/index.html#zip)), which loads classes from a repository specified in the constructor.  The findClass method invokes loadFromCustomRepository that searches for the given class in the repository and, if found, reads and returns the bytecodes for the class. The raw bytecodes for the class are passed into the defineClass method implemented in the java.lang.ClassLoader class, which returns an instance of the java.lang.Class object. This makes a new class available to a running Java program. The defineClass method also ensures that a custom ClassLoader does not redefine core Java API classes by loading them from a custom repository. A SecurityException is thrown if the class name passed to defineClass begins with "java".  It should be noted that at start-up, the JVM doesn't need to know anything about the class represented by the string passed into the loadClass method. A subsequent section shows how a program can use the CustomClassLoader.  **Deviations from the Java 2 Delegation Model** The Java 2 delegation model cannot be followed in all situations. There are cases in which ClassLoaders have to diverge from the Java 2 model. For instance, the servlet specification recommends (section 9.7) that a Web application ClassLoader be implemented so that classes and resources packaged in the Web application archive are loaded in preference to classes and resources residing in container-wide JAR files. To meet this recommendation, a Web application ClassLoader should search for classes and resources in its local repository first, before delegating to a parent ClassLoader, thus deviating from the Java 2 delegation model. This recommendation makes it possible for Web applications to use different versions of classes/resources than those being used by a servlet container. For example, a Web application might be implemented using features available in a newer version of an XML parser than the one being used by a servlet container.  A Web application ClassLoader that meets the recommendation of the servlet specifications can be implemented by overriding the loadClass method of the java.lang.Classloader class. The loadClass method of such a custom ClassLoader may look similar to [Listing 2](http://www2.sys-con.com/itsg/virtualcd/java/archives/0808/chaudhri/index.html#s2).  **Applications of ClassLoaders** ClassLoaders provide some powerful features that can be utilized in Java programs. This section discusses a few ways in which they can be used.  ***Hot Deployment*** Upgrading software in a running application without restarting it is known as hot deployment. For a Java application, hot deployment means upgrading Java classes at runtime. ClassLoaders play an important role in Java-based application servers to achieve hot deployment. This feature is exploited in most Java-based application servers like EJB servers and servlet containers. A ClassLoader cannot reload a class that it has already loaded, but using a new instance of a ClassLoader will reload a class into a running program. The following code from TestCustomLoader.java illustrates how hot deployment may be achieved in a Java application:  1. ClassLoader customLoader = new 2. CustomClassLoader(repository); 3. loadAndInvoke(customLoader,classToLoad); 4. System.out.println("waiting.Hit 5. Enter to continue"); 6. System.in.read(); 7. customLoader = new CustomClassLoader 8. (repository); 9. loadAndInvoke(customLoader,classToLoad);  An instance of the CustomClassLoader is created to load classes from a repository specified as a command-line parameter. loadAndInvoke loads a class, HelloWorld, also specified as a command-line parameter, and invokes a method on its instance, which prints a message on the console. While the program is waiting for user input at line 6, the HelloWorld class can be changed (by changing the message that gets printed on the console) and recompiled. When the program continues execution, a new instance of CustomClassLoader is created at line 7. When loadAndInvoke executes line 9, it loads the updated version of HelloWorld and a new message is printed on the console.  ***Modifying Class Files*** A ClassLoader searches for bytecodes of a class file in the findClass method. After the bytecodes have been located and read into the program, they may be modified before invoking defineClass. For example, extra debugging information may be added to the class file before invoking defineClass. Class file data for some secure applications may be stored encrypted in a repository; the findClass method can decrypt the data before invoking defineClass. A program can generate the bytecodes on the fly instead of retrieving them from a repository. This forms the basis of JSP technology.  **ClassLoaders and Security** Since a ClassLoader is responsible for bringing code into the JVM, it's architected so that the security of the platform may not be compromised. Each ClassLoader defines a separate namespace for the classes loaded by it, so at runtime a class is uniquely identified by its package name and the ClassLoader that loaded it. A class is not visible outside its namespace; at runtime there's a protective shield between classes existing in separate namespaces. The parent delegation model makes it possible for a ClassLoader to request classes loaded by its parent, thus a ClassLoader doesn't need to load all classes required by it.  The various ClassLoaders that exist in a Java runtime have different repositories from which they load code. The idea behind separating repository locations is that different trust levels can be assigned to the repositories. The Java runtime libraries loaded by the bootstrap ClassLoader have the highest level of trust in the JVM. The repositories for user-defined ClassLoaders have lower levels of trust. Furthermore, ClassLoaders can assign each loaded class into a protection domain that defines the permissions assigned to the code as it executes. To define permissions on code based on the system security policy (an instance of java.security.Policy), a custom ClassLoader should extend the java.security.SecureClassLoader class and invoke its defineClass method that takes a java.security.CodeSource object as a parameter. The defineClass method of SecureClassLoader gets the permissions associated with the CodeSource from the system policy and defines a java.security.Protection Domain based on that. A detailed discussion of the security model is beyond the scope of this article. Further details can be obtained from the book *Inside the Java Virtual Machine* by Bill Venners.  **Summary** ClassLoaders offer a powerful mechanism by which the Java platform can be extended in interesting ways at runtime. Custom ClassLoaders can be used to achieve functionality not normally available to a running Java program. Some of these applications have been discussed in this article. ClassLoaders play an important role in some of the technologies offered by current J2EE platforms. For further details about the Java classloading mechanism, read *Inside the Java Virtual Machine*.  **References**   Liang, S., and Barcha, G. "Dynamic Class Loading in the Java Virtual Machine": [www.java.sun.com/people/gbracha/classloaders.ps](http://www.java.sun.com/people/gbracha/classloaders.ps)   Venners, B. (2000). *Inside the Java Virtual Machine*. McGraw-Hill Osborne Media.   Neward, T. (2000). *Server-Based Java Programming*. Manning Publications Company.   Gong, Li. "Secure Java Class Loading" [http://java.sun.com/people/ gong/papers/ieeeic98.pdf](http://java.sun.com/people/gong/papers/ieeeic98.pdf)   The Java Servlet Specification: [http://jcp.org/aboutJava/ communityprocess/first/jsr154/](http://jcp.org/aboutJava/communityprocess/first/jsr154/)  **About The Author** Rohit Chaudhri is a member of the mobile platforms and services lab at Motorola. The team investigates end-to-end application platforms for the wireless Internet. He has an MS in computer science and is a Sun Certified Programmer for the Java 2 Platform. [rohit.chaudhri@motorola.com](mailto:rohit.chaudhri@motorola.com)  http://www2.sys-con.com/itsg/virtualcd/java/images/SourceCode.jpg  **"Understanding the Java Classloading Mechanism"  Vol. 8, Issue 8, p. 16**    **Listing 1**  1. public class CustomClassLoader  2. extends ClassLoader {  3. //search repository  4. private List classRepository;  5. public CustomClassLoader  6. (ClassLoader parent,String searchPath)  7. {S}  8. protected Class findClass(String  9. className) throws  10. ClassNotFoundException {  11. byte[] classBytes =  12. loadFromCustomRepository(className);  13. if(classBytes != null) {  14. return defineClass  15. (className,classBytes,0,classBytes.length);  16. }  17. //else  18. throw new ClassNotFoundException(className);  19. }  20. }  **Listing 2**  1. public Class loadClass(String name)  2. throws ClassNotFoundException  3. {  4. //check if the class is already loaded  5. Class loadedClass = findLoadedClass(name);  6. if (loadedClass == null) {  7. //search for class in local  8. //repository before delegating  9. .......  10. //if class not found delegate to parent  11. loadedClass =  13. this.getClass().getClassLoader().loadClass(name);  13. }  14. return loadedClass;  15. }  [Additional Code for this Article](http://www2.sys-con.com/itsg/virtualcd/java/source/8-8/Chaudhri0808.zip) zip file ~7.62 KB  All Rights Reserved  Copyright ©  2004 SYS-CON Media, Inc.    E-mail: [info@sys-con.com](mailto:info@sys-con.com)  Java and Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. SYS-CON Publications, Inc. is independent of Sun Microsystems, Inc. | |

11-100/21/2