|  |
| --- |
| 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. | |