

Section 7 -- The Class Loader Architecture



One of the central tenets of Java is making code truly mobile. Every mobile code system requires the ability to load code from outside a system into the system dynamically. In Java, code is loaded (either from the disk or over the network) by a ■ **Class Loader**. Java's class loader architecture is complex, but it is a central security issue, so please bear with us as we explain it.

Recall that all Java objects belong to classes. ■ **Class loaders determine when and how classes can be added to a running Java environment.** ■ **Part of their job is to make sure that important parts of the Java runtime environment are not replaced by impostor code.** The fake Security Manager shown in Figure 2.4 must be disallowed from loading into the Java environment and replacing the real Security Manager. This is known as class spoofing.

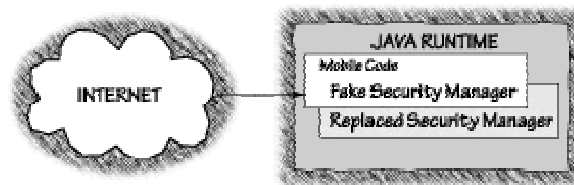


Figure 2.4 Spoofing occurs when someone or something pretends to be something it is not.

In this figure, an external class has arrived from the Internet and declares itself to be the Security Manager (in order to replace the real Security Manager). If external code were allowed to do this, Java's security system would be trivial to break.

Class loaders perform two functions. ■ **First**, when the VM needs to load the byte code for a particular class, it asks a class loader to find the byte code. ■ Each class loader can use its own method for finding requested byte code files: It can load them from the local disk, fetch them across the Net using any protocol, or it can just create the byte code on the spot. This flexibility is not a security problem as long as the class loader is trusted by the party who wrote the code that is being loaded. ■ **Second**, class loaders define the namespaces seen by different classes and how those namespaces relate to each other. Namespaces are a subtle and security-critical issue that we'll have a lot more to say about later. Problems with namespace management have led to a number of serious security holes.

■ It probably would have been better if Java's design had initially separated the two functions of class loaders and provided lots of flexibility in finding byte code but not much flexibility in defining namespaces. In a sense, this is what has come about as successive versions of Java have had increasingly restrictive rules about how namespaces may be managed. ■ Java's class loader architecture was originally meant to be extensible, in the sense that new class loaders could be added to a running system. It became clear early on, however, that malicious class loaders could break Java's type system, and hence breach security. ■ As a result, current Java implementations prohibit untrusted code from making class loaders. This restriction may be relaxed in the future, since there is some possibility that the Java 2 class loader specification is at last safe in the presence of untrusted class loaders.

Varieties of Class Loaders

There are two basic varieties of class loaders: ■ Primordial Class Loaders and ■ Class Loader objects. There is ■ only one Primordial Class Loader, which is an essential part of each Java VM. It cannot be overridden. The Primordial Class Loader is involved ■ in bootstrapping the Java environment. Since most VMs are written in C, it follows that the Primordial Class Loader is typically written in C. ■ This special class loader loads trusted classes, usually from the local disk. Figure 2.5 shows the inheritance hierarchy of Class Loaders available in Java 2.

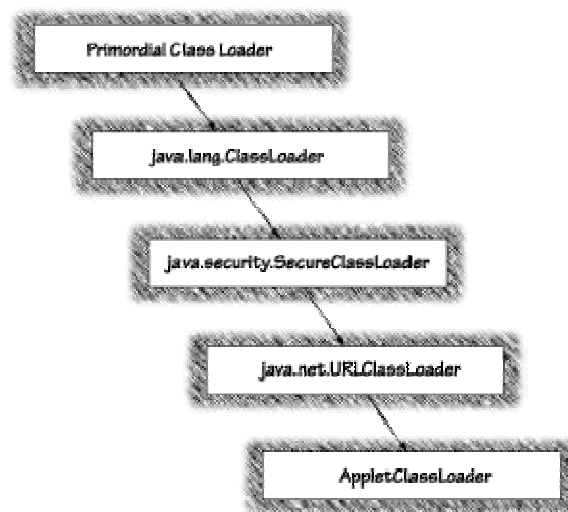


Figure 2.5 Class Loaders provide Java's dynamic loading capability, which allows classes to arrive and depart from the runtime environment.

Java 2 implements a hierarchy of Class Loaders. This figure, after Gong [Gong, 1998], shows the inheritance hierarchy of Class Loaders.

The Primordial Class Loader

The **Primordial Class Loader** uses ■ **the native operating system's file access capabilities to open and read Java class files** from the disk into byte arrays.

This provides Java with the ■ **ability to bootstrap itself and provide essential functions**. The Java **API class files** (stored by default in the classes.zip file) **are usually the first files loaded** by the VM. The **Primordial Class Loader** also typically loads any classes a user has located in the **CLASSPATH**. ■ **Classes loaded by the Primordial Class Loader are not subjected to the Verifier prior to execution.**

Sometimes the Primordial Class Loader is referred to as the "internal" class loader or the "default" class loader. Just to make things overly complicated, some people refer to classes loaded by the Primordial Class Loader as having no class loader at all.

Class Loader Objects

The second basic variety of class loader is made up of ■ **Class Loader objects**. **Class Loader objects load classes that are not needed to bootstrap the VM** into a running Java environment. **The VM treats classes loaded through Class Loader objects as ■ untrusted by default**. **Class Loaders are ■ objects just like any other Java object—they are written in Java, compiled into byte code, and loaded by the VM (with the help of some other class loader).** These Class Loaders give Java its dynamic loading capabilities.

There are three ■ **distinct types of Class Loader objects defined by the JDK itself: Applet Class Loaders, RMI Class Loaders, and Secure Class Loaders**. From the standpoint of a Java user or a system administrator, Applet Class Loaders are the most important variety. Java developers who are interested in ■ rolling their own Class Loaders will likely subclass or otherwise use ■ the RMI Class Loader and Secure Class Loader classes.

Applet Class Loaders are ■ responsible for loading classes into a browser and are defined by the vendor of each Java-enabled browser. Vendors generally implement similar Applet Class Loaders, but they do not have to. Sometimes seemingly subtle differences can have important security ramifications. For example, Netscape now tracks a class not by its name, but by a pointer to actual code, making attacks that leverage Class Loading complications harder to carry out.

Applet Class Loaders help ■ to prevent external code from spoofing important pieces of the Java API. They do this by ■ attempting to load a class using the Primordial Class Loader before fetching a class across the network. If the class is not found by the Primordial Class Loader, the Applet Class Loader typically loads it via ■ HTTP using methods of the URL class. Code is fetched from ■ the CODEBASE specified in the <APPLET> tag. If a fetch across the Web fails, a ClassNotFoundException exception is thrown.

It should be clear why external code must be prevented from spoofing the trusted classes of the Java API. Consider that the essential parts of the Java security model (including the Applet Class Loader class itself) are simply Java classes. If an untrusted class from afar were able to set up shop as a replacement for a trusted class, the entire security model would be toast!

The RMI Class Loader and Secure Class Loader classes were introduced with JDK 1.1 and Java 2, respectively. RMI Class Loaders are very similar to Applet Class Loaders in that they load classes from a remote machine. They also give the Primordial Class Loader a chance to load a class before fetching it across the Net. The main difference is that RMI Class Loaders can only load classes from the URL specified by Java's rmi.server.codebase property. Similar in nature to RMI Class Loaders, Secure Class Loaders allow classes to be loaded only from those directories specified in Java's java.app.class.path property. Secure Class Loaders can only be used by classes found in the java.security package and are extensively used by the Java 2 access control mechanisms.

Roll-Your-Own Class Loaders

Developers are often called upon to write their own class loaders. This is an inherently dangerous undertaking since class loading is an essential part of the Java security model. Homegrown class loaders can cause no end of security trouble. The right approach to take in writing a class loader is to avoid changing the structure of namespaces, and to change only the methods that find the byte code for a not-yet-loaded class. This will allow you to fetch classes in new ways, such as through a firewall or proxy, or from a special local code library, without taking the risk inherent in namespace management. You can do this by overriding only the loadClass methods.

Namespaces

In general, a running Java environment can have many Class Loaders active, each defining its own namespace. Namespaces allow Java classes to see different views of the world depending on where they originate (see Figure 2.6). Simply put, a namespace is a set of unique names of classes loaded by a particular Class Loader and a binding of each name to a specific class object. Though some people say that namespaces are disjoint and do not overlap, this is not true in general. **There is nothing to stop namespaces from overlapping.**

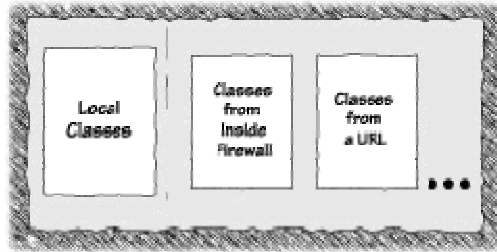


Figure 2.6 Class Loaders have two distinct jobs (which we believe would have been better off separated): (1) fetching and instantiating byte code as classes, and (2) managing name spaces.

This figure shows how Class Loaders typically divide classes into distinct name spaces according to origin. It is especially important to keep local classes distinct from external classes. This figure implies that name spaces do not overlap, which is not entirely accurate.

Most VM implementations have used ■ different class loaders to load code from different origins. This allowed these implementations to assign a single security policy to all code loaded by the ■ same class loader, and to make security decisions ■ based on which class loader loaded the class that is asking to perform a dangerous operation. With the addition of code signing in JDK 1.1, there are now two ■ characteristics for categorization of code: ■ origin (usually represented as a URL) and ■ signer (the identity associated with the private key used to sign the file). Only the Class Loader that loaded a piece of code knows for sure ■ where the code was loaded from.

Applet Class Loaders, which are typically supplied by the browser vendor, load all applets and the classes they reference, usually getting the classes from HTTP servers. When an applet loads across the network, its Applet Class Loader receives the binary data and instantiates it as a new class. Under normal operation, applets are forbidden to install a new Class Loader, so Applet Class Loaders are the only game in town.

A trusted Java application (such as the Java interpreter built in to Netscape Navigator or Internet Explorer) can, however, define its own class loaders. Sun Microsystems provides three template class loader modules as part of the JDK (discussed earlier). If an untrusted applet could somehow install a Class Loader, the applet would be free to define its own namespace. Prior to Java 2, this would allow an attack applet to breach security (see [Chapter 5](#)).

If you are writing an application or built-in extension that defines its own Class Loader, you should be very careful to follow the rules; otherwise, your Class Loader will almost certainly introduce a security hole. It is unfortunate that in order to get the ability to use your own code-finding mechanism, you must also take on responsibility for managing namespaces. One criticism often raised against the Java security model is that because of the presence of objects like application-definable class loaders, the security model is too distributed and lacks central control. Applet Class Loaders install each applet in a separate namespace. This means that each applet sees its own classes and all of the classes in the standard Java library API, but it doesn't see classes belonging to other applets. Hiding applets from each other in this way has two advantages: It allows multiple applets to define classes with the same name without ill effect, so

applet writers don't have to worry about name collisions. It also makes it harder, though not impossible, for applets to team up.

As an example, consider a class called `laptop` with no explicit package name (that is, `laptop` belongs to the default package). Imagine that the `laptop` class is loaded by an Applet Class Loader from `www.rstcorp.com` as you surf the Java Security Web Site. Then you surf over to `java.sun.com` and load a different class named `laptop` (also in the default package). What we have here is two different classes with the same name. How can the VM distinguish between them? The tagging of classes according to which Class Loader loaded them provides the answer. Applets from different CodeBases are loaded by different instances of the browser's Applet Class Loader class. (By the way, distinct namespaces will be created even if the two sites use explicit package names that happen to be the same.) Although the same class is involved in loading the two different classes (i.e., the Applet Class Loader), two different instances of the Applet Class Loader class are involved—one for each CodeBase.

Recall that the default object protection and encapsulation scheme covered earlier in this chapter allows classes that are members of a package to access all other classes in the same package. That means it is important for the VM to keep package membership straight. As a result, Class Loaders have to keep track of packages as well as classes.

When a class is imported from the network, the Applet Class Loader places it into a namespace labeled with information about its origin. Whenever one class tries to reference another, the Applet Class Loader follows a particular order of search. The first place it looks for a class is in the set of classes loaded by the Primordial Class Loader. If the Primordial Class Loader doesn't have a class with the indicated name, the Applet Class Loader widens the search to include the namespace of the class making the reference.

Because the Applet Class Loader searches for built-in classes first, it prevents imported classes from pretending to be built-in classes (something known as "class name spoofing"). This policy prevents such things as applets redefining file I/O classes to gain unrestricted access to the file system. Clearly, the point is to protect fundamental primitives from outside corruption.

Since all applets from a particular source are put in the same namespace, they can reference each other's methods. A source is defined as a particular directory on a particular Web server.

According to the Java specification, every Class Loader must keep an inventory of all the classes it has previously loaded. When a class that has already been loaded is requested again, the class loader must return the already loaded class.

Loading a Class

Class loading proceeds according to the following general algorithm:

- Determine **whether the class has been loaded before**. If so, return the previously loaded class.
- **Consult the Primordial Class Loader** to attempt to load the class **from the CLASSPATH**. This prevents external classes from spoofing trusted Java classes.
- See **whether the Class Loader is allowed to create the class** being loaded. The **Security Manager makes this decision**. If not, throw a security exception.
- **Read the class file into an array of bytes**. The way this happens differs according to particular class loaders. Some class loaders may load classes from a local database. Others may load classes across the network.
- **Construct a Class object** and its methods from the class file.
- **Resolve classes immediately referenced by the class before it is used**. These classes include classes used by static initializers of the class and any classes that the class extends.
- **Check the class file with the Verifier**.

Summary

Each Java class begins as source code. This is then compiled into byte code and distributed to machines anywhere on the Net. A Java-enabled browser automatically downloads a class when it encounters the <APPLET> tag in an HTML document. The Verifier examines the byte code of a class file to ensure that it follows Java's strict safety rules. The Java VM interprets byte code declared safe by the Verifier. The Java specification allows classes to be unloaded when they are no longer needed, but few current Java implementations unload classes.

Java's ability to dynamically load classes into a running Java environment is fraught with security risks. The class-loading mechanisms mitigate these risks by providing separate namespaces set up according to where mobile code originates. This capability ensures that essential Java classes cannot be spoofed (replaced) by external, untrusted code. The Applet Class Loader in particular is a key piece of the Java security model