

# SECURE JAVA CLASS LOADING

LI GONG

*Sun Microsystems*

When Java technology burst onto the Internet scene in 1995, its developers declared the ambitious goal of providing a safe programming environment, especially for Web-based, dynamically composed, and mobile applications.<sup>1,2</sup> OEM vendors and licensees could port the Java platform to their environment, such as browsers and operating systems, and inherit extensive built-in security features. Java's security tools and services enabled independent software vendors to build a wider range of security-sensitive applications—for example, in the enterprise world—with minimal effort.

Java's original security model for these tools and services is known as the sandbox model. This model features a very restricted environment in which to run untrusted code (called applets) obtained from the open network.<sup>3</sup> Essentially, the sandbox model trusts local code to have full access to vital system resources, such as the file system. However, the model does not trust downloaded remote code, so restricts its access to only a small set of limited resources. The Java Development Toolkit, versions 1.0.x, deploy this sandbox model, as do most applications built with JDK, including Java-enabled Web browsers. For more about the sandbox model, see the sidebar "The Mechanisms of Java Sandbox Security."

To extend the sandbox model, Sun Microsystems introduced signed applets with JDK 1.1.x in early 1997. In this model, Java treats a correctly digitally signed applet as trusted local code, if the end system that receives the applet recognizes the signature key as trusted. Developers deliver signed applets, together with their signatures, in the Java Archive format. In this article, I describe the more finely grained, permission-based access control architecture, and its relation to the class loading mechanism, that will be available in the JDK 1.2 release.

The class loading mechanism,  
central to Java, plays a key  
role in JDK 1.2 by enabling  
an improved security policy  
that is permission-based  
and extensible.

## NEW SECURITY MODEL IN JDK 1.2

The forthcoming JDK 1.2 release will introduce a new security architecture that uses a security policy to grant individual access permissions to running code.<sup>4</sup> The policy is based on the code's characteristics; for example, where the code originates, whether it is digitally signed, and, if so, by whom. Code that attempts to access protected resources will invoke security checks that will compare permissions granted with permissions needed for the attempted access. If the former includes the latter, the Java virtual machine permits access; otherwise, it denies access.

The Java runtime system's security behavior is entirely specified by its security policy. As a developer or system administrator, you can set this policy and configure it directly with either a GUI tool or a programming interface. A policy object encapsulates the policy and a security manager object enforces it. This means you can implement different security policies as needed, simply by customizing these objects.

JDK 1.2 builds in support for a commonly used security policy that controls access essentially by listing which code is granted what permissions, as the simple example in Table 1 shows. In abstract terms, this access control policy maps from a set of properties that characterize running Java code to a set of access permissions granted to the concerned code. If users elect not to specify a policy, JDK 1.2 defaults to the sandbox policy.

### Code and Permissions

A piece of code is fully characterized by its origin (its location as specified by a URL) and a set of public

## THE MECHANISMS OF JAVA SANDBOX SECURITY

Java enforces the seemingly simple sandbox security model through a number of sophisticated mechanisms.

- First, Sun designed the Java language to be type-safe and easy to use. Language features such as automatic memory management, garbage collection, and range checking on strings and arrays are examples of how Java helps programmers write safer code.
- Second, compilers and a bytecode verifier ensure that the Java virtual machine executes only legitimate Java code. The bytecode verifier, together with the Java virtual machine, guarantees language type safety at runtime. Moreover, a class loader defines a local name space, which helps to ensure that an untrusted applet cannot interfere with the running of other Java programs.
- Finally, the Java virtual machine mediates access to crucial system resources. A security manager checks this access in advance and restricts actions of untrusted code to the minimum.

keys. The public keys correspond to the set of private keys with which a developer signed the code by means of one or more digital signature algorithms.

A permission is represented with a permission object that is instantiated from a hierarchy of typed and parameterized access permission classes corresponding to all controlled resources. For example, the permission representing file system access is located in the Java I/O package, as `java.io.FilePermission`. All permission classes must implement a method called `implies` where `a.implies(b) == true` means that if, say, a piece of code is granted permission `a`, then it is naturally granted permission `b`. This method is used to compare permissions, and encapsulates resource access semantics locally within a permission class.<sup>4</sup>

Significantly, with semantics encapsulation the actual access control code need not be specialized for a particular type of resource. For example, JDK 1.2 (or any Java application or applet) can use identical access control code for security checks during access to the file system, to networking

**Table 1. A simple example of a security access policy that lists which permissions are granted to a given piece of code.**

Code	Permission
Applets from <code>http://java.sun.com</code>	Read the Java software directory
Applets from <code>http://java.sun.com</code> signed by Sun Microsystems	Write to the Java software directory
Applets from <code>http://mygames.com</code>	No special permission
Applications installed in directory <code>/usr/local/bin</code>	Read and write all files in the <code>/tmp</code> directory

resources, and to system properties. At the same time, semantics encapsulation makes it possible to dynamically add a new type of controlled resource, because programmers can reuse the built-in access control code and need to add only the corresponding permission class.

### How Access Control Works

JDK 1.2 applies the security policy and its associated access controls to both Java applications and applets. The access control process is the same, regardless of whether an applet or an application is involved. You might view an applet, either through a Web browser or the Appletviewer program provided with the JDK. Alternatively, you might run a Java application, possibly from the command line by invoking the program called java. Either way, the following steps occur.

## The security policy and related access controls apply to both Java applications and applets.

1. The Java virtual machine obtains a class file and accepts it if the file passes preliminary bytecode verification.
2. The Java virtual machine determines the class's code source. This step includes signature verification, if the code appears to be signed.
3. The Java virtual machine consults the security policy, and composes the set of permissions to grant to this class. In this step, the policy object will be constructed, if it has not been already.
4. The Java virtual machine loads and defines the class, and marks the class to have been granted the set of permissions.
5. The Java virtual machine instantiates the class into objects, and executes their methods. Runtime type-safety check continues.
6. If at least one method of a class is in the call chain when a security check is invoked, the access control code examines the class's set of granted permissions. It does this to see if there is sufficient permission for the requested access. If yes, the execution continues. If no, a security exception occurs. When a security exception—which is a runtime exception—occurs and is not caught, the Java virtual machine aborts.

7. When the class file and the instantiated objects are no longer in use, they are garbage-collected.

These steps show how the class loading mechanism is integrally connected with security. The class loader locates and fetches the class file, consults the security policy, and defines the class object with the appropriate permissions.

### SECURE CLASS LOADING

Dynamic class loading is an important feature of the Java virtual machine because it enables the Java platform to install software components at run-time.<sup>1</sup> Class loading has several unique characteristics. First, lazy loading means that classes are loaded on demand, on a just-in-time basis. Second, dynamic class loading maintains the type safety of the Java virtual machine by adding link-time checks, which replace certain runtime checks and are performed only once.<sup>5</sup> Moreover, programmers can define their own class loaders that, for example, specify the remote location from which certain classes are loaded, or assign appropriate security attributes to them. Finally, programmers can use class loaders to provide separate name spaces for various software components. For example, a browser can load applets from different Web pages using separate class loaders, thus maintaining a degree of isolation between those applet classes. In fact, these applets can contain classes of the same name—the Java virtual machine treats these classes as distinct types.

### Class Loader Hierarchies

When class loaders load Java software components, the smallest unit is a class. Classes are defined in a machine-independent, binary representation known as the class file format. An individual class representation is called a class file, even though it need not be stored in an actual file. For example, class files can be stored as records or commands in a database.

A class file can contain bytecode as well as symbolic references to fields, methods, and names of other classes. For example, a class named C is declared as follows:

```
class C {
    void f() {
        D d = new D();
        ...
    }
}
```

The class file representing C contains a symbolic reference to class D. The Java virtual machine resolves symbolic references at link time (of class C) to actual class types. To do this, the Java virtual machine must load the class file of D and create the class type.

A class loader L that loads class C is the class's *defining* class loader. The actual class type is fully qualified by both itself and its defining class loader,  $\langle C, L \rangle$ . Therefore, two types in the Java runtime are equal if both the class types are equal and their defining class loaders are identical.

Since there can be multiple instances of class loader objects in one Java virtual machine, how do we determine which class loader to use as the defining loader when loading a class?

In JDK 1.2, the situation is further complicated because JDK introduces multiple class loader classes with distinct properties. Thus another question is, what type of class loader should we use when loading a class?

**Class Hierarchy Root and Tree.** The root of the class loader class hierarchy is an abstract class called `java.lang.ClassLoader`, originally defined in JDK 1.0 and since expanded. Class `java.security.SecureClassLoader`, introduced in JDK 1.2, is a subclass and concrete implementation of the abstract `ClassLoader` class. Class `java.net.URLClassLoader` is a subclass of `SecureClassLoader`.

A utility program called `Appletviewer` uses a private class—`sun.applet.AppletClassLoader`—to load applets. In JDK 1.0, `AppletClassLoader` is a subclass and concrete implementation of `ClassLoader`. In JDK 1.2, however, it is a subclass of `URLClassLoader`. Note that interposing new classes between an existing class and its subclass are binary backward-compatible.<sup>1</sup>

When creating a custom class loader class, users can subclass from any of the above class loader classes, depending on the particular needs of the custom class loader. (Because `AppletClassLoader` is a private class defined in the `sun.*` package, it is not supported and is subject to change, so an application should not subclass from it.)

Each class is loaded by its defining class loader, and each class loader itself is a class and must be loaded by another class loader. This may prompt the obvious chicken-and-egg question: Where does the first class loader come from? It comes from a *primordial* class loader that bootstraps the class loading process. The primordial class loader, generally written in a native language such as C, does

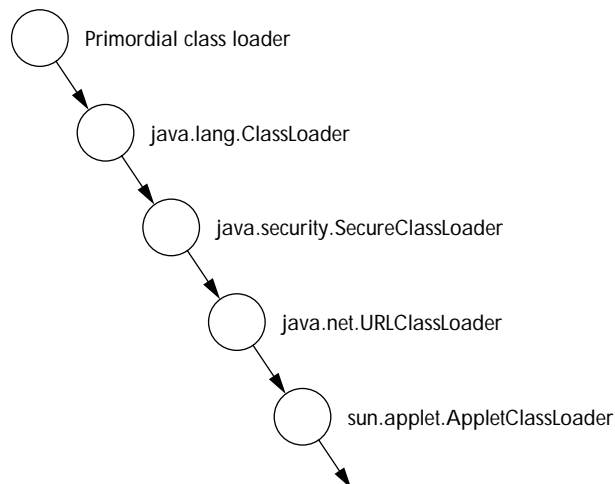


Figure 1. Class loader class hierarchy in JDK 1.2. The arrows indicate subclassing.

not manifest itself in the Java context. The primordial class loader often loads classes from the local file system in a platform-dependent manner.

Some classes, such as those defined in the `java.*` package, are essential for the Java virtual machine and runtime system to function correctly. They are often called system classes. For historical reasons, all system classes have a defining class loader that is a null object. This null class loader, sometimes called the system class loader, is perhaps the only sign that a primordial class loader exists. In fact, it is easier to simply view the null class loader as the primordial class loader. In JDK 1.2, a core set of system classes, called base classes or base system classes, continue to be loaded by the primordial class loader. All other system classes are loaded by instances of the `URLClassLoader`. Figure 1 lists the class loader classes defined in JDK 1.2 and describes their relationship with each other.

Given all classes in one Java runtime environment, we can easily form a class-loading tree to reflect the class-loading relationship. Each class that is not a class loader is a leaf node. Each class's parent node is its defining class loader, with the null class loader being the root class. Such a structure is a tree because there cannot be cycles—a class loader cannot have loaded its own ancestor class loader. Figure 2 depicts such a defining relationship.

**Delegation.** Another relationship between class loader objects is delegation. When the Java virtual machine asks one class loader to load a class, this class loader either loads the class itself or asks another class loader

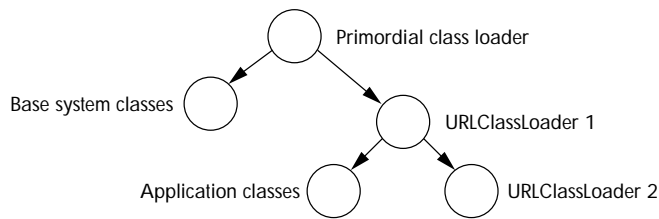


Figure 2. Class-defining relationship in JDK 1.2.

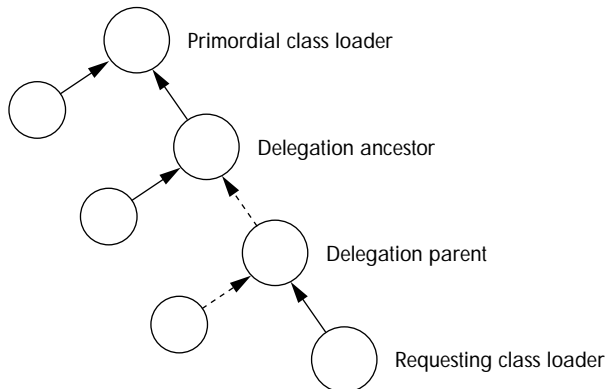


Figure 3. Class-loading delegation relationship in JDK 1.2.

to do so. The first class loader thereby delegates the loading task to the second class loader. The delegation relationship is virtual in the sense that it has nothing to do with which class loader loads or defines which other class loader. Instead, a delegation relationship forms when class loader objects are created, and in the form of a parent-child relationship. Nevertheless, the system class loader is the delegation root ancestor of all class loaders. Applications that create class loaders must be careful to ensure that the delegation relationship does not contain cycles; otherwise, the delegation process may enter into an infinite loop. Figure 3 illustrates the delegation relationship.

### Class Resolution Algorithm

In the default implementation of JDK 1.2, the Java virtual machine searches for classes in the following order:

1. Checks if the class has already been loaded.
2. Delegates, if the current class loader has specified a delegation parent, to the parent to load this class. If there is no parent, delegates to the primordial class loader.
3. Calls a customizable method to find the class elsewhere.

In step 1, the Java virtual machine looks into the class loader's local cache (or its functional equivalent, such as a global cache) to see if a loaded class matches the target class. Step 3 provides a way to customize the mechanism that looks for classes. A custom class loader can thus override this method to specify how to look up a class. For example, an applet class loader can override this method to go back to the applet host, try to locate the class file, and load it over the network. If at any step the loader locates a class, it returns the class to be used by the Java virtual machine.

It is crucial for type safety that the same class loader does not load the same class more than once. If the class is not among those already loaded, the current class loader attempts to delegate the task to the parent class loader. This can occur recursively and ensures the use of the appropriate class loader. For example, when locating a system class, the delegation process continues until it reaches the system class loader.

If the Java virtual machine does not find the class with steps 1–3, a `ClassNotFoundException` occurs. If it finds the class, on the other hand, its type is `<C, L>`, in which `L` is the class loader that actually loaded and defined the class. The eventual class type, therefore, is not directly affected by the variety of intermediate delegation class loaders.

Given the name of any class, which class loader does the Java virtual machine start with in trying to load the class? The rules for it to determine the *requesting* class loader are as follows:

- When loading the first class of an application, use a new instance of the `URLClassLoader`.
- When loading the first class of an applet, use a new instance of the `AppletClassLoader`.
- If an existing class triggers the class-loading request by referring to it or by calling `java.lang.Class.forName` directly, ask the defining class loader for the existing class to load the class.

Rules about the use of `URLClassLoader` and `AppletClassLoader` instances have exceptions and can vary depending on the particular system environment. For example, a Web browser may choose to reuse an existing `AppletClassLoader` to load applet classes from the same Web page.

Because class loaders are so powerful, the Java virtual machine severely restricts who can create instances of them. On the other hand, it is desirable to provide a convenient mechanism for appli-



cations or applets to specify URL locations and load classes from them. JDK 1.2 provides static methods to allow any program to create instances of the `URLClassLoader` class, although not other types of class loaders.

### Class Paths

The class loader classes I have described provide programmable ways to locate and load classes and resources. To simplify installation of software components on a Java-enabled system, there are well-defined and user-specific places to put such components. This allows them to be automatically discovered by the Java runtime system.

JDK 1.0 and 1.1 feature a well-known, built-in systemwide search path called the class path, which is set in a platform-specific way. For example, on Unix systems, the class path can be set via the Shell environment variable `CLASSPATH`. Essentially, all classes or Java Archive files containing classes on the local file system must reside on this path to be discovered. Moreover, this is the same path where all system classes reside. As a result, the Java virtual machine treats all classes from the local file system as system classes and gives them full resource-access privileges. There is therefore no distinction between local classes that really are system code from those classes that are part of some locally installed applications.

This access control situation is clearly not perfect. There are many scenarios where a locally installed application should not be given full system privileges. For example, when trying out a demo program newly received in the mail, it is prudent to run the demo with very limited privileges. Or, when displaying an important document, it is safer to run the display application in read-only mode to prevent possible software bugs from altering document content.

To accommodate such scenarios, the JDK 1.2 security architecture treats locally resident classes like remotely downloaded applet classes—by granting them specific and fine-grained permissions. To achieve this goal, JDK 1.2 distinguishes genuine system classes from all other classes by means of separate class paths. One is the system class path, for storing system classes. The other is the application class path, for storing all other classes. The Java virtual machine still loads classes on the system class path with the primordial class loader or a `URLClassLoader` and trusts them by default. A `URLClassLoader` usually loads classes on the application class path, and the Java virtual machine grants such

classes the appropriate permissions according to the security policy.

### CONCLUSIONS

JDK 1.2 has introduced a powerful and secure class loading mechanism. It not only enforces type safety and name space separation but also has a significant role in the new security architecture that supports fine-grained, permission-based access control. The new class loading mechanism's flexibility—through its delegation scheme and the rich set of class loader classes—gives Java applications and applets greater freedom to customize and specify how, when, and from where classes are loaded. Because the class loading mechanism is central to both the correctness and the security of the Java runtime system, we would like to model and define this mechanism, perhaps in a formal verification system. We can then obtain a formal specification and prove (or disprove) that the mechanism as currently designed is sufficient for security. ■

### REFERENCES

1. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley, Menlo Park, Calif., 1996.
2. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, Menlo Park, Calif., 1997.
3. L. Gong, "Java Security: Present and Near Future," *IEEE Micro*, Vol. 17, No. 3, May/June 1997, pp. 14–19.
4. L. Gong et al., "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2," *Proc. Usenix Symp. Internet Technologies and Systems*, 1997, Usenix Assoc., Berkeley, Calif., pp. 103–112.
5. S. Liang and G. Bracha, "Dynamic Class Loading in the Java Virtual Machine," *Proc. ACM Conf. Object Oriented Programming Systems, Languages, and Applications*, ACM Press, New York, 1998.

**Li Gong** is chief architect, Java security and networking, a Distinguished Engineer, and manager of the security and networking group at Sun Microsystems' Java software division. He is an associate editor of *ACM Transactions on Information and System Security* and is on the editorial board of the *Journal of Computer Security*. He served as program chair of the IEEE Symposium on Security and Privacy, the ACM Conference on Computer and Communications Security, and the IEEE Computer Security Foundations Workshop. He received the BS and MS degrees from Tsinghua University, Beijing, and a PhD degree from the University of Cambridge, England.

Readers may contact Li at Sun Microsystems Inc., 901 San Antonio Rd., Palo Alto, CA 94303; li.gong@sun.com; <http://java.sun.com/people/gong/>.