



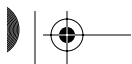
Chapter 5

Customizing Class Loading

Chapter 2 described using the class loaders installed by the Java launcher and also using your own instances of `URLClassLoader`. These techniques solve many of the class loader problems you are likely to face. However, there are times when it would be nice to use a custom class loader. For instance, you may want to distribute classes with a protocol other than http, load classes from an object database, or extract classes from a version control system. More interestingly, you may want to modify the semantics of class loading. You might insert additional information, such as instrumentation for debugging, profiling, or auditing, or you might want to process custom metadata that you have added to the binary class format. In these situations, you will want to customize class loading.

This chapter will present two different techniques for customizing class loading. The first, and most obvious, is writing your own subclass of `ClassLoader`. Your own class loader implementation is free to choose any strategy it wants for mapping class names to class bytes, but you must be careful about security. Class loaders tell the security architecture where code came from, and the security manager uses this information to grant permissions. So before I show you how to write a custom class loader, I will take a brief detour through the security architecture. You will see that in most cases, you should subclass `SecureClassLoader`, not `ClassLoader`.

The second option for customizing class loading splits the security and resource-resolution tasks into separate classes, leveraging the core API. A standard `URLClassLoader` instance manages the details of Java security, and a



Java class called a *protocol handler* resolves resources by parsing a customized URL protocol string, like

```
objectdb://server/MyClass.class
```

instead of

```
http://server/MyClass.class
```

This separation of concerns provides two benefits over simply extending `SecureClassLoader`. First, the protocol handler can be implemented in relatively untrusted code since the `URLClassLoader` handles security. This is useful because though you might trust a piece of unknown code to go and find other classes that it needs, you certainly would not trust it to tell you what permissions those classes should have! Second, the protocol handler can request any kind of resource, not just binary classes.

§5.4 explains how to create and use class loaders in a secured environment. Most application code in a secured environment does not have permission to create a class loader because of the sensitive role that class loaders play in assigning permissions to classes as they are loaded. However, you will often want the ability to request a specific class loader from application code. There are several ways to work around this issue. Usually application code does not need to *create* a class loader; all it needs is *access* to a class loader pointed at the correct classes. Instead of attempting to instantiate a class loader directly, you call back into more trusted code, requesting a class loader that uses a resource resolution strategy defined by you. You can write code to do this yourself, or you can use `URLClassLoader`, which includes a factory method specifically designed to handle this problem.

§5.5 shows you how to modify the class bytes after you have compiled a class. You can use this technique to insert or remove debugging information, performance instrumentation, or optimization hints. In combination with a custom class loader, you can modify classes as they are loaded, or you can simply change the binary classes offline in the file system or whatever other repository you use. To illustrate the power of this technique, I create a new custom class attribute that tracks version dependencies between packages and then implement

a `URLClassLoader` subclass that always finds the correct version of dependent classes.

Writing a custom class loader is an interesting exercise, but you should not reinvent the wheel. The core API already handles the most common class loading scenarios. Before you write a custom loader, you should study the source for `SecureClassLoader` and `URLClassLoader` to ascertain that you cannot accomplish your purpose simply by leveraging these classes.

5.1 Java 2 Security

Brace yourself for some massive simplification;¹ it's time to talk about Java security. In Java 2 security, classes are assigned *permissions* based on their *code source*. A permission is simply a description of some secured operation that you might want to perform. Permissions are defined as subclasses of `java.security.Permission`, and have optional targets and actions. For example, a class might have the `FilePermission` permission with target `<<ALL FILES>>` and action `delete`. This means exactly what you think it means—that the class can delete any or all files.

A code source contains the URL a class came from, plus any certificates used to sign the code. These two data items are stored in an instance of `java.security.CodeSource`. An instance of `java.security.Policy` manages the mapping between code sources and permissions.

The reference implementation of `Policy` is file-based. If you wanted to give your code permission to delete all files, you would create a policy file like the one in Listing 5–1. Both the `signedBy` and `codeBase` attributes are optional. If you omit `signedBy` or `codeBase` then the permissions are granted regardless of signer or location, respectively.

Listing 5–1 A Simple Policy File

```
//file your.policy
grant signedBy "you" codeBase "file:/yourclassdir" {
    permission java.io.FilePermission "<<ALL FILES>>", "delete";
};
```

1. See [Gon99] for the full story.

```
//command line to use your.policy
>java -Djava.security.manager \
    -Djava.security.policy=your.policy MainClass
```

A class's permissions come into play at runtime when a security manager is installed and the class attempts to perform an operation that is protected by the security manager. Consider the `SelfDestruct` class, shown in Listing 5–2, which attempts to destroy its own class file on the local file system. If you run this class from the folder where the file is located, it will delete itself and be unavailable for future runs. However, if you turn on Java security with the `-Djava.security.manager` flag but do not specify a policy file, your code will run with a minimal set of permissions² and you will see the exception trace shown in Listing 5–3. The security manager rejects the call to `File.delete` because the `SelfDestruct` class does not have the necessary permission. The standard security manager will grant permission only if *all* the classes on the call stack have the requisite permission.³

Listing 5–2 The SelfDestruct Class

```
import java.io.*;
public class SelfDestruct {
    public static void main(String[] args) {
        try {
            File f = new File("SelfDestruct.class");
            System.out.println("deleted file? " + f.delete());
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Listing 5–3 Blocked by the Security Manager

```
$ java -Djava.security.manager SelfDestruct
java.security.AccessControlException: access denied
    (java.io.FilePermission SelfDestruct.class delete)
```

2. The default behavior of the security manager can be modified by editing the `java.security` and `java.policy` files in your `$(JAVA_HOME)/jre/lib/security` directory.

3. The default behavior of checking the entire call stack can be modified with privileged scopes.

```

at java.security.AccessControlContext.checkPermission(...)
at java.security.AccessController.checkPermission(...)
at java.lang.SecurityManager.checkPermission(...)
at java.lang.SecurityManager.checkDelete(...)
at java.io.File.delete(...)
at SelfDestruct.main(SelfDestruct.java:6)

```

In the call stack shown in Listing 5–3, the `SelfDestruct` class is the only problem because all the other classes on the call stack were loaded by the bootstrap loader and are exempt from permission checks. In order to give `SelfDestruct` the requisite permission, you could reference a policy file similar to the one in Listing 5–1. You would accomplish this by removing the `signedBy` field and setting the `codeBase` value to a file URL where the `SelfDestruct.class` is located.

The permission granted by the policy file does not have to exactly match the permission listed in the `AccessControlException`. The policy file in Listing 5–1 grants `FilePermission, <<ALL FILES>>, delete`, but the security manager in Listing 5–3 is looking for `FilePermission, SelfDestruct.class, delete`. This difference causes no confusion because the permission architecture supports *implication*. Each permission subclass defines its own notion of implication, so for example, the `FilePermission` class knows that `<<ALL FILES>>` implies any particular file name. The built-in permission classes support implication without any special effort by the developer.

That's Java 2 security in brief. When you deploy an application, you determine the permissions that code will need, and you associate those permissions with code locations and signers in the policy file. The default policy implementation then extracts these permissions when the class is loaded. When the security manager wants to check an operation, it verifies that every class on the call stack has the necessary permission, directly or by implication. The relationships between the players are shown in Figure 5–1. There is only one thing missing: What does all of this have to do with class loaders?

5.1.1 The Role of Class Loaders

Class loaders are the bridge between the policy and the security manager. A class loader takes a code source and permissions pair from the policy and binds

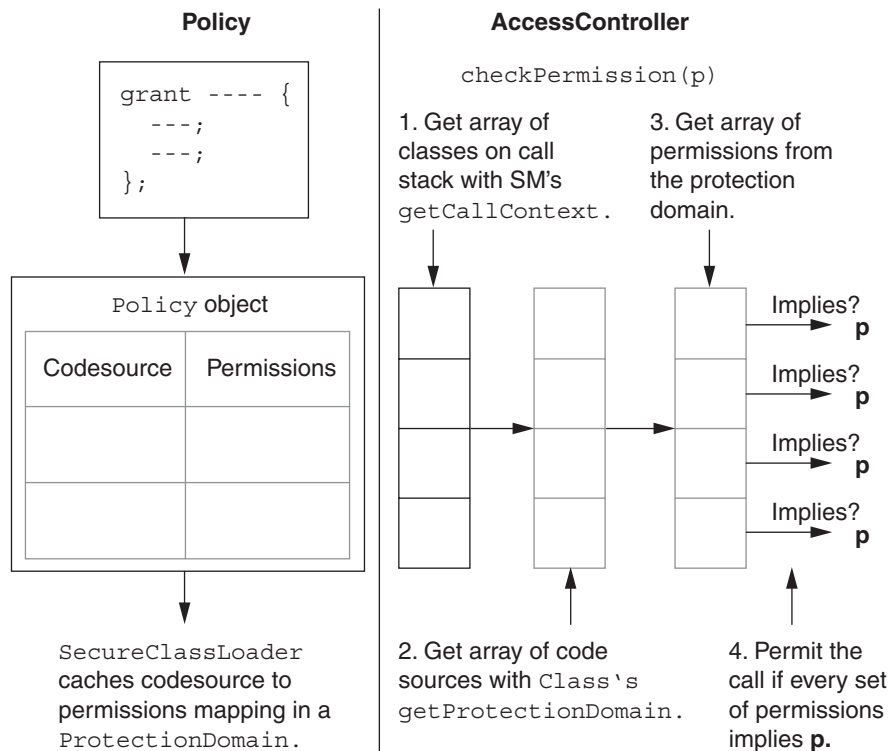


Figure 5–1 The Java 2 security architecture

them into an instance of `java.security.ProtectionDomain`. The class loader then hands the protection domain to the virtual machine via `ClassLoader`'s native method `defineClass`. When the security manager wants to verify a call stack, it extracts all the protection domains for classes on the stack via a native method of `java.security.AccessControlContext`. The Java 2 security architecture depends on honest class loaders telling the truth when they define classes.

The key role that class loaders play in security has two consequences for application developers. First, you should rarely grant the `RuntimePermission createClassLoader`. Code with this permission can create a class loader that lies about protection domains in an attempt to compromise security. Second, because you cannot allow untrusted code to create class loaders, you may need to provide some other mechanism to help untrusted code find the classes and

resources it needs. §5.2 presents the basics of custom class loaders, and §5.4 shows how to make class loading services available to untrusted code without compromising security.

5.2 Custom Class Loaders

The rules for writing class loaders have changed over time, but the principle has remained basically the same. You implement a subclass of `ClassLoader` that knows, given a class name, how to create or find a byte array that is the class bytes for that class.

5.2.1 Pre-Java 2 Custom Class Loaders

Prior to SDK version 1.2, the relevant methods were `loadClass`, `defineClass`, and `resolveClass`, as shown in Listing 5–4. To implement a custom loader, you override the abstract method `loadClass`. Using the `name` argument, you find or create an array of bytes that has the correct binary class format, and then you pass these bytes to the `defineClass` method. `ClassLoader`'s `defineClass` implementation then calls to native code inside the virtual machine that loads the class. If the `resolve` flag is set, your subclass should also call `resolveClass` after `defineClass` completes. `ClassLoader`'s `resolveClass` makes the class ready for use by verifying, resolving, and initializing the class.⁴

Listing 5–4 Pre-Java 2 Custom Class Loader APIs

```
package java.lang;
public class ClassLoader {
    protected abstract Class loadClass(String name,
                                      boolean resolve);
    protected final Class defineClass(byte[] data, int offset,
                                      int length);
    protected final Class defineClass(String name, byte[] data,
                                      int offset, int length);
    protected final void resolveClass(Class c);
    //other methods omitted for clarity
}
```

4. This process is described in detail in [Ven99].

5.2.2 Class Loading since SDK 1.2

The pre-1.2 method of implementing class loaders worked, but it suffered from two important flaws. First, there was no mechanism for passing security information from the `Policy` implementation into the virtual machine. Second, there was no guarantee that custom class loaders would follow the rules for class loaders laid out in Chapter 2.⁵ An overridden `loadClass` method could cheat by either refusing to delegate to its parent loader, or by reloading classes that had already been loaded.

The 1.2 Java SDK introduced several modifications to the `ClassLoader` class to address these problems. The 1.2 `ClassLoader` includes a new overloaded form of `defineClass` that passes security information to the VM via a `ProtectionDomain` argument. 1.2 also provides a concrete implementation of `loadClass` that enforces the basic rules of class loading. Developers should no longer override `loadClass`; instead, they should override a new method, `findClass`. The `findClass` method is called only after `loadClass` fails to find the class from a parent class loader or the current loader's cache. The changes to the 1.2 `ClassLoader` API are summarized in Listing 5–5.

Listing 5–5 SDK 1.2 Enhancements to `ClassLoader`

```
package java.lang;
public class ClassLoader {
    //only new/changed methods shown here
    protected final Class defineClass(String name, byte[] b,
                                     int off, int len, ProtectionDomain pd);

    //override this instead of loadClass
    protected Class findClass(String name)
        throws ClassNotFoundException
    {
        throw new ClassNotFoundException(name);
    }
}
```

5. A more historically accurate account would acknowledge that both the `Policy` and the class loader rules did not exist in SDK 1.1 either, so there was no way that `ClassLoader` *could* work with them. In my account, I am following the tradition that winners get to rewrite history so that the outcome appears obvious and inevitable.


```

//do not override this method
protected synchronized Class loadClass(String name,
                                         boolean resolve) throws ClassNotFoundException
{
    // First, check if the class has already been 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 call findClass in order
            // to find the class.
            c = findClass(name);
        }
    }
    if (resolve) {
        resolveClass(c);
    }
    return c;
}

```

The 1.2 API represents a big improvement over previous incarnations of `ClassLoader`. The `loadClass` method pulls all the common code into the core API where it belongs, so all you have to worry about is finding or creating the class bytes. Notice, however, that `loadClass` is not marked `final`. You could still override `loadClass` and implement an old-style class loader. This loophole was left in deliberately to preserve binary compatibility with existing code. In a world without legacy software, `loadClass` would be `final`, and you should treat it as such when writing new programs.

The various signatures of `defineClass` are also a concession to legacy code. You should always use the version that takes a `ProtectionDomain` argument so that code loaded by your class loaders can participate fully in the Java 2 security model. An easy way to accomplish this is to always extend the 1.2 class `java.security.SecureClassLoader`. `SecureClassLoader` provides yet

another `defineClass` method, as shown in Listing 5–6. This version of `defineClass` takes a `CodeSource` instance, and the `getProtectionDomain` method manages a cache of `ProtectionDomains` for efficiency. If you are going to write a custom class loader, your best bet is to subclass `SecureClassLoader` and override `findClass` to call the `CodeSource`-aware version of `defineClass`.

Listing 5–6 `SecureClassLoader`'s `defineClass` Implementation

```
protected final Class defineClass(String name, byte[] b,
                                int off, int len, CodeSource cs)
{
    if (cs == null)
        return defineClass(name, b, off, len);
    else
        return defineClass(name, b, off, len,
                           getProtectionDomain(cs));
}
```

5.2.3 A Transforming Class Loader

As an example of a custom class loader, consider a `TransformingClassLoader`. This loader is very similar to a `URLClassLoader`, except that it may perform arbitrary transformations on the bytes of a class before handing them off to the virtual machine. These transformations could modify the class bytes to add logging, profiling, debugging, or other services. Since the transforming loader needs to handle URLs anyway, it will extend `URLClassLoader`, and thereby implicitly extend `SecureClassLoader`.

The transforming loader and related classes are shown in Listing 5–7. The `ResourceTransformer` class defines the various transformations that might be made. For now, the interesting method is `transformClassBytes`, which modifies an array of class bytes after the `URLClassLoader` retrieves the bytes but before they are passed to `defineClass`. The `TransformingClassLoader` repeats all the constructors of `URLClassLoader` but with a `ResourceTransformer` parameter added. The transformer implementation will get a chance to modify the class bytes before they are used to define a class.

Listing 5-7 The TransformingClassLoader and Related Classes

```
//class com.develop.xload.ResourceTransformer
package com.develop.xload;
import java.io.IOException;
import java.net.URL;
import java.util.Enumeration;

public interface ResourceTransformer {
    public byte transformClassBytes(byte[] inout,
                                    int start, int len);
    public URL transformResourceURL(URL resource);
    public Enumeration transformResources(Enumeration resrcs);
}

//class com.develop.xload.TransformingClassLoader
package com.develop.xload;

import java.io.*;
import java.lang.reflect.*;
import java.net.*;
import java.security.*;
import java.util.jar.*;
import java.util.jar.Attributes.*;
import sun.misc.*;

public class TransformingClassLoader extends URLClassLoader {
    private final ResourceTransformer xr;

    public TransformingClassLoader(URL[] urls,
                                   ResourceTransformer xr) {
        super(urls);
        this.xr = xr;
    }

    public TransformingClassLoader(URL[] urls,
                                   ClassLoader parent,
                                   ResourceTransformer xr) {
        super(urls, parent);
        this.xr = xr;
    }

    public TransformingClassLoader(
        URL[] urls, ClassLoader parent,
```

```

        URLStreamHandlerFactory fact,
        ResourceTransformer xr)
    {
        super(urls, parent, fact);
        this.xr = xr;
    }

    private URL getURLBase(URL url) {
        URL[] urls = getURLs();
        int length = urls.length;
        String stringForm = url.toExternalForm();
        for (int n=0; n<length; n++) {
            if (stringForm.startsWith(urls[n].toExternalForm())) {
                return urls[n];
            }
        }
        return null;
    }

    protected Class findClass(final String name)
        throws ClassNotFoundException
    {
        String className = name.replace('.', '/') + ".class";
        URL url = super.getResource(className);
        if (url == null) {
            return null;
        }
        URL urlBase = getURLBase(url);
        if (urlBase == null) {
            throw new Error("url has no base");
        }
        InputStream is = null;
        try {
            is = url.openStream();
            if (is == null) { return null; }
            ByteArrayOutputStream baos =
                new ByteArrayOutputStream();
            for (int ch=0; -1 != (ch=is.read()); )
                baos.write(ch);
            byte[] classbytes = baos.toByteArray();
            xr.transformClassBytes(classbytes, 0,
                classbytes.length);
            return defineClass(name, classbytes, 0,
                classbytes.length, new CodeSource(urlBase, null));
        }
    }

```

```

    }
    catch (IOException ioe) {
        return null;
    }
}
}

```

The meat of the example is the `findClass` implementation. First, `findClass` takes the string class name passed into it and turns it into a relative URL string by replacing occurrences of `'.'` with `'/'` and then appending `'.class'`. Then, the superclass's `getResource` method locates the URL for the class. The method then opens a stream to the URL and reads it into an array of bytes. Before these bytes are passed to `defineClass`, the call to `xr.transform` converts the bytes, using whatever algorithm the transformer implements.

The helper method `getURLBase` returns the base URL; for instance,

```
http://server/MyClass.class
```

would have a base URL of

```
http://server/
```

The base URL is used to construct a `CodeSource`, which is then passed to the security-aware version of `defineClass`. The `TransformingClassLoader` has most of the abilities of a `URLClassLoader`,⁶ and it adds the ability to plug in arbitrary transformations as code is loaded.

As a simple transformation example, consider a `ClassNotter` that decrypts a binary class that was encrypted by NOTting every bit in the binary class format. Such encryption is not very secure, but it is easy to implement for a quick example. The `ClassNotter` is shown in Listing 5–8. `ClassNotter` extends `NoOpResourceTransformer`, which is an adapter class that provides empty implementations of `ResourceTransformer` methods. This allows the `ClassNotter` to implement only the method(s) of interest, in this

6. Notice that this implementation passes `null` as the second argument to the `CodeSource` constructor. This loses any signer information, so this implementation supports location-based security only—not digital certificates. The design of `URLClassLoader` does not encourage inheritance-based reuse because it does not make the certificate information easily available to derived classes. It is tucked away in private members of `URLClassLoader` and requires some classes in the `sun.misc` package. Since you should not use `sun.misc` code, a certificate-aware version of `TransformerClassLoader` would be nontrivial.

case `transformClassBytes`. The `ClassNotter` implementation inverts every bit in the class byte array.

The test harness `test.TestTransformingLoader` creates a `TransformingLoader` that uses the `ClassNotter` to transform bytes after they are loaded from the file system and before they are handed off to the virtual machine. If you encrypted your classes as part of deployment, then anyone trying to use a standard `URLClassLoader` would be unable to interpret the class and would see a `ClassFormatError` (bad magic number).

Of course, this encryption is very simple, and would be defeated by any but the most casual adversary.⁷ The point here is that the `TransformingLoader` enables any transformation you can imagine. You focus on the transformation process, and let the built-in capabilities of `URLClassLoader` take care of correctly setting your `ProtectionDomain`.

Listing 5–8 `ClassNotter`, a Very Simple `ResourceTransformer`

```
package com.develop.xload;
import java.io.IOException;
import java.net.URL;
import java.util.Enumeration;

public class NoOpResourceTransformer
    implements ResourceTransformer {
    public byte[] transformClassBytes(byte[] inout,
                                       int start,
                                       int length) {
        return inout;
    }
    public URL transformResourceURL(URL resource) {
        return resource;
    }
    public Enumeration transformResources(
        Enumeration resources) {
        return resources;
    }
}
```

7. In fact, even far more complex “unbreakable” encryption schemes are easily defeated unless you have physical control over every box in which the decryption will occur. Otherwise, an adversary can simply use debugging tools to grab the class bytes *after* they are decrypted without troubling to attack the encryption head-on.

```
package test;
import com.develop.xload.NoOpResourceTransformer;

public class ClassNotter extends NoOpResourceTransformer {
    public byte[] transformClassBytes(byte[] inout,
                                      int start,
                                      int len) {
        int end = start+len;
        for (int n=start; n<end; n++) {
            inout[n] = (byte)~inout[n];
        }
        return inout;
    }
}

package test;
import com.develop.xload.*;
import java.io.*;
import java.net.*;

public class TestTransformingLoader {

    public static void main(String [] args) {
        try {
            if (args.length != 4) {
                System.out.println("usage: test.TestTransformingLoader " +
                                   " url1 url2 cls1 cls2");
                System.exit(-1);
            }
            URL u1 = new URL(args[0]);
            URL u2 = new URL(args[1]);
            URLClassLoader cl = new TransformingClassLoader(
                new URL[]{u1,u2}, new ClassNotter());
            System.out.println(cl);
            URL[] urls = cl.getURLs();
            Class cls1 = Class.forName(args[2], true, cl);
            System.out.println(cls1);
            Class cls2 = Class.forName(args[3], true, cl);
            System.out.println(cls2);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



5.3 Protocol Handlers

In most cases, writing your own custom class loader is overkill, thanks to yet another feature of the ubiquitous `URLClassLoader`—pluggable URL protocols called protocol handlers. If you look at the code for the `java.net.URL` class, you will see that it doesn't actually do very much. `URL` includes code to parse a URL string into its component parts: protocol, host, port, and file. Everything else is delegated to helper classes called protocol handlers, or synonymously, stream handlers. Protocol handlers can be used with any `URL`, so you can use them for other purposes besides just class loading.

The Java 2 platform comes with several protocol handlers built in, including handlers for the all-important `http` and `file` protocols. In addition to these, you can install your own protocol handlers to do any sort of resource lookup that you want. Integration with Java security is automatic. The policy file already understands URL syntax, so you can combine your custom handler with a standard `URLClassLoader` and policy file.

To demonstrate protocol handlers in action, I will rebuild the simple encryption scheme from the previous example, this time using a protocol handler. The first step is to define the custom URL protocol stream to use. The syntax for using the sample protocol `not` is shown in Listing 5–9. Notice that the second example syntax does not include the hostname. This follows the convention for URL syntax, where the host name defaults to `localhost`, if it is omitted. This behavior is built into the `URL` class parsing logic, and it will be available for free to the `not` stream handler.

Listing 5–9 The 'not' Custom Protocol

```
'not' URL syntax:
not://host:port/file/
example xform for NOTting every byte:
not://localhost/d/halloway/src/
not:/halloway/src
```



5.3.1 Implementing a Handler

To create a protocol handler, you must create at least two classes: a `java.net.URLStreamHandler` subclass that understands your protocol syntax, and a `java.net.URLConnection` subclass that the handler can return to clients. Annotated listings for these base classes appear in Listing 5–10.

Listing 5–10 `URLStreamHandler` and `URLConnection`

```
package java.net;

public abstract class URLStreamHandler {
    //always override this method:
    abstract protected URLConnection openConnection(URL u)
        throws IOException;
    //override these only if your URL syntax differs from
    //standard URLs:
    protected void parseURL(URL u, String spec,
        int start, int limit);
    protected String toExternalForm(URL u);
    //NOT SHOWN: several other methods you might override
}

package java.net;
//All listed methods can throw IOException
public abstract class URLConnection {
    //probably override these
    abstract public void connect();
    public InputStream getInputStream();
    public OutputStream getOutputStream();
    //remainder omitted for brevity
}
```

The `URLStreamHandler` class does two things. First, it parses the protocol string into a URL in the `parseURL` method. Then, it creates a connection for that URL in the `openConnection` method. The `URLStreamHandler` class should always be named `Handler`, and its full package name should be

```
{arbitrarypkgs.}protocolname.Handler
```

If your URL protocol has syntax similar to HTTP URLs, then the handler is trivial to implement; all it needs to do is return your custom connection class. The handler for the `not` protocol is shown in Listing 5–11.

Listing 5-11 Handler for the not Protocol

```
package test.not;

import java.io.*;
import java.net.*;

public class Handler extends URLStreamHandler {
    protected URLConnection openConnection(URL u)
        throws IOException
    {
        return new NotConnection(u);
    }
}
```

The `URLConnection` subclass for a protocol handler provides connection semantics. Because the design is inspired by HTTP, the notion of connection semantics feels very much like HTTP. The `URLConnection` class is full of methods that are relevant to setting and extracting common HTTP headers, such as `getContentType`, `getExpiration`, `setExpiration`, and `getDate`. These methods will often be irrelevant in a custom connection.

The three key methods of `URLConnection` were shown previously in Listing 5-10. The `connect` method should actually begin a communication with the resource. If the resource is across the network, this will typically involve opening a socket, speaking the correct wire protocol, and verifying that somebody is listening. If the resource is local, then `connect` may do nothing. The `getInputStream` and `getOutputStream` methods enable two-way communication with the resource. If the resource is across the network, then these calls may return the socket streams directly, or they may preprocess them in some way, such as by reading and interpreting headers first. If the resource is local, then these streams might be file streams or some custom stream class.

The `NotConnection` implementation is straightforward and is shown in Listing 5-12. Because the `not` URL does not access a network resource, the `connect` method does nothing. Also, communication is unidirectional; there is no need to send anything to a `not` URL. All the information to locate a class is in the URL itself, so there is no need to implement `getOutputStream`. The interesting method is `getInputStream`, which opens a file stream and then wraps it with a filter stream `NotInputStream` that inverts each byte.

Listing 5-12 The NotConnection Class

```
package test.not;
import java.io.*;
import java.net.*;

public class NotConnection extends URLConnection {
    private InputStream is;
    private Object lock = new Object();
    public NotConnection(URL u) {
        super(u);
    }
    public void connect() throws IOException {
    }
    public InputStream getInputStream() throws IOException {
        synchronized (lock) {
            if (is == null) {
                String file = getURL().getFile();
                FileInputStream fis = new FileInputStream(file);
                is = new NotInputStream(fis);
            }
            return is;
        }
    }
}
```

5.3.2 Installing a Custom Handler

Once you have created a handler and supporting classes, the trick is to get the runtime to recognize them. By default, the Java SDK uses only handlers that have a package prefix `sun.net.www.protocol`, a package suffix name matching the protocol, and the name `Handler`. For example, when you use an `http` URL, the runtime uses reflection to load the class

```
sun.net.www.protocol.http.Handler
```

and create an instance. Then the runtime casts the instance to type `URLConnectionHandler` and uses it to parse and connect to the URL.

The Java license forbids creating your own classes in the `sun.*` namespace, so do not bother trying to install your handlers this way. Instead, there are two hooks you can use to install your own custom handler. The `URLConnection` class has a `setURLConnectionHandler` factory that allows you to install your own

mapping from protocols to handlers. It takes as its argument an instance of `URLStreamHandlerFactory`, an interface whose one method takes a protocol name and returns a handler.

The `URLStreamHandlerFactory` code is shown in Listing 5–13. You can install a stream handler factory once per VM, and the factory will be used prior and in addition to consulting the standard handlers.⁸ In Listing 5–14, the `NotURLReader` class installs a factory that understands the `not` protocol, and then it outputs a hex dump of the data found at a URL passed on the command line.

Listing 5–13 `URLStreamHandlerFactory`

```
//from java.net.URLStreamHandlerFactory
public interface URLStreamHandlerFactory {
    public URLStreamHandler createURLStreamHandler(
        String prot);
}

//from java.net.URL
public static void
setURLStreamHandlerFactory(URLStreamHandlerFactory fac);
```

Listing 5–14 The `NotURLReader` Class

```
package test;

import com.develop.util.*;
import java.io.*;
import java.net.*;
public class NotURLReader implements URLStreamHandlerFactory
{
    public URLStreamHandler createURLStreamHandler(String prot)
    {
        if (prot.equals("not")) {
            return new com.develop.handlers.not.Handler();
        }
        return null;
    }
}
```

8. Because your custom handler is consulted first, you could replace the standard handlers for `http` et al. if you wanted to.

```

public static void main(String [] args) throws Exception {
    if (args.length != 1) {
        System.out.println("usage: test.NotURLReader url");
        System.exit(-1);
    }
    URL.setURLStreamHandlerFactory (new NotURLReader());
    URL u = new URL(args[0]);
    InputStream is = u.openStream();
    byte[] buf = new byte[4096];
    int length = 0;
    while (0 < (length = is.read(buf))) {
        System.out.println(HexFormatter.convertBytesToString(
            buf, 0, length, 16, true));
    }
}

```

Though it is possible to install new handlers from within Java code as shown above, it is typically more convenient to give control of handlers over to an administrator. To this end, you can set a property that specifies where to look for custom handlers. The `java.protocol.handler.pkgs` property contains a list of “|”- delimited package prefixes. The `URL` class attempts to create handlers based on custom package prefixes after checking the installed `URLStreamHandlerFactory` but before checking the standard `sun.net.www.protocol` handlers.

Given a command line as shown in Listing 5–15’s Example 1, the `URL` class would try the following steps until one succeeded:

1. If a `URLStreamHandlerFactory` is installed, see if it supports `not`.
2. Try to use an instance of `foo.not.Handler`.
3. Try to use an instance of `bar.not.Handler`.
4. Try to use an instance of `sun.net.www.protocol.not.Handler`.
5. Report a `MalformedURLException: unknown protocol: not`.

Listing 5–15, Example 2, shows the correct command line to locate the `not` handler from Listing 5–11.



Listing 5–15 Specifying URLStreamHandlers on the Command Line

```
Example 1.  
java -Djava.protocol.handler.pkgs=foo|bar MainClass  
  
Example 2.  
java -Djava.protocol.handler.pkgs=test.not  
MainClass
```

5.3.3 Choosing between Loaders and Handlers

At first glance, writing your own protocol handler may appear to be a lot more work than just writing your own custom class loader. After all, you have to write at least two classes, and probably more. The simple `not` handler required the `Handler`, `NotConnection`, and `NotInputStream` classes. However, these classes are mostly boilerplate code, and they factor the process of locating a resource process into the following distinct pieces:

1. Protocol handlers parse URLs and return connections.
2. Connections manage communication and return streams.
3. Streams read and write data and possibly apply transformations.

Moreover, stream handlers can be used to connect to any resource, whereas custom class loaders can be used only to load classes and other co-located resources. Finally, stream handlers leverage the security features already built into the `URLClassLoader`. And, because they can be installed on the command line, you can make the presence of stream handlers completely transparent to the rest of your code.

The only downside of stream handlers comes when you try to handle a URL whose string format is radically different from `http` syntax. For example, consider the hypothetical URLs shown in Listing 5–16. These URLs do not map to the standard

```
protocol://host:port/file
```

format, and to implement them, you would have to hack around the fact that the `URL` class pretty much assumes this format. Your stream handler would be much more complex, implementing at least the `parseURL` method and possibly several others. Except in cases like these where the semantics of your class



loader are very difficult to express as a URL, you should prefer stream handlers to custom class loaders.

Listing 5-16 Some Hypothetical Custom URLs

```
For connecting to a database:
db://hostname:port/user=stu;pwd=hmp;table=ORDERS
A URL that applies a transform to data from a wrapped URL:
xform://xformtype/xformargs/http://localhost/file
xform://xformtype/xformargs/http://localhost/file
```

5.4 Getting Past Security to the Loader You Need

In the situations discussed so far, only two levels of trust are involved in class loading. Trusted code launches the process and chooses security settings, and then it instantiates class loaders to load less trusted code. Less trusted code may have a greater or lesser degree of permissions, but it will almost never have permission to create a `ClassLoader` instance. If it did, it might lie about the `ProtectionDomain` of classes that it loaded, thereby subverting the security model.

To give your less-trusted code the ability to use class loaders, authors of trusted code (such as J2EE containers) need to provide a callback mechanism whereby you can request a specific class loader. The trusted code can then create a class loader that meets your specifications for how class bytes are located. Note that this does not compromise security in the slightest. The security is not in locating the class, but in assigning its `ProtectionDomain`. The trusted code keeps this prerogative for itself.

As an example of where this might be useful, consider a servlet container run by an application hosting company. The container is the process owner. It is highly trusted and will activate Java security to protect itself (and other customers) from damage that your code might cause. Your code is less trusted than the container code, but you still might want to customize class loading. For example, you might write a custom class loader that checks for new versions of classes on your development server and then makes them available to the servlet container based on some criteria you define. You cannot simply create a class loader in your code because the application hosting company will not give you the necessary security permission.

You can see the problem by simply turning on Java security for any program that creates a class loader. If you try this, the program will fail as soon as it attempts to instantiate the `ClassLoader`, as shown in Listing 5–17. Fortunately, the `URLClassLoader` class includes code specifically designed to address this problem. You should rarely instantiate a `URLClassLoader` directly. Instead, use one of the static factory methods named `newInstance`.

Listing 5–17 Instantiating a Class Loader in a Secure Process

```
java -Djava.security.manager UseAClassLoader
java.security.AccessControlException: access denied
(java.lang.RuntimePermission createClassLoader)
...
at java.lang.ClassLoader.<init>(ClassLoader.java:234)
...
```

The code for `newInstance` is shown in Listing 5–18. Without delving too deeply into the security model, you can see the basic idea. The field `acc` saves away the protection domains that are current when the call begins. The code inside the `PrivilegedAction` runs with the permissions of the `URLClassLoader` class, ignoring possibly untrusted classes higher on the call stack. This makes it possible to create the class loader, and it is secure because the privileged action has been carefully coded not to do anything that would compromise security.

Listing 5–18 Swapping Access Control Contexts

```
package java.net;
public class URLClassLoader extends SecureClassLoader {
    public static URLClassLoader
    newInstance(final URL[] urls, final ClassLoader parent)
    {
        // Save the caller's context
        AccessControlContext acc = AccessController.getContext();
        // Need a privileged block to create the class loader
        URLClassLoader ucl = (URLClassLoader)
        AccessController.doPrivileged(new PrivilegedAction() {
            public Object run() {
                return new FactoryURLClassLoader(urls, parent);
            }
        });
    }
}
```



```
    }  
  });  
  // Now set the context on the loader using the one we saved,  
  // not the one inside the privileged block...  
  ucl.acc = acc;  
  return ucl;  
}  
//remainder omitted for brevity  
}
```

Remember that *using* a class loader does not open a security hole, but *instantiating* one does. Once the class loader has been created, it is assigned the set of protection domains `acc` that were on the stack when `newInstance` was called. When you later attempt to use this class loader, it will have to pass essentially two separate security checks: the one implied by `acc`, plus whatever classes are on the call stack at the time of the call.

The important point here is that you should use the `newInstance` method to create `URLClassLoaders`; otherwise, you are likely to get an `AccessControlException` when somebody attempts to execute your code in a secured environment. If you write your own custom class loader, you will need to add a factory method similar to `newInstance` if you want untrusted code to be able to create an instance of your loader.

5.5 Reading Custom Metadata

The Java binary class format provides a standard set of extensions for adding custom data to binary classes. However, to take advantage of any custom class data at runtime, you must write a custom class loader that is aware of your extensions to the class format. In this section, you will see a custom extension to the class format that includes extra version information, and you will see a custom class loader that uses this information to locate the correct versions of classes that it will load.

Before you jump into the example or apply this technique in your own code, you need to be very careful that your additions to the class do not violate the Java license agreement. You cannot create modified classes that *require* a special class loader or virtual machine because this would violate Java's "Prime

Directive”—code must be able to run on any compliant Java platform. The relevant section of [LY99] begins:

Compilers are permitted to define and emit class files containing new attributes in the attributes tables of class file structures. Java virtual machine implementations are permitted to recognize and use new attributes found in the attributes tables of class file structures. However, any attribute not defined as part of this Java virtual machine specification must not affect the semantics of class or interface types. Java virtual machine implementations are required to silently ignore attributes they do not recognize.

Any semantics that your custom metadata enables must be *optional* semantics, that is, your classes could function just fine without them. The versioning information I am going to introduce here is a good example of this. If a virtual machine does not recognize the custom version information in the binary class, it will still be able to load and execute the class.

5.5.1 Example: Version Attributes

The versioning problem is a fundamental one. Java’s class loading architecture does not attempt to verify the version of a class being loaded. If class `A` references class `B`, then `A`’s class loader delegation will attempt to find a definition for `B`. Standard class loaders such as `URLClassLoader` will load the *first* matching definition of `B`. This will cause deployment headaches if different components rely on different versions of `B`.

JAR sealing (§3.6.3) can help some. If you deploy all of your classes in sealed JAR files, and you are careful with your build process, you can guarantee that all the classes in a package are from the same build. However, this does not help with cross-package dependencies. When JAR files are sealed, the runtime can identify potential version problems by throwing an exception, but it cannot automatically locate the correct version.

Package reflection (§3.6) also helps some, but not enough. After a class `B` is loaded, you may be able to use package reflection to discover the version of `B` you have found. Unfortunately, this information arrives too late to be of use. If `B`

turns out to be the wrong version, that's too bad. You have already loaded it, and you cannot now unload it short of creating a new class loader and starting over entirely. As with JAR sealing, package reflection simply identifies the problem, and leaves you to solve it.

Here are design goals for a simple version authority that can automatically locate the correct versions of Java classes:

1. The version authority tracks version information for a loaded class. This includes the version of the class and the versions of any other classes that the class depends on.
2. Before a class is loaded, the version authority checks the candidate class's version against the requirements of all the classes already loaded by this loader. If the version does not match, the candidate is rejected and the class loader can continue to search for additional matches.
3. The binary format of the version information and the definition of a version "match" can be customized.
4. The presence of version information is transparent at runtime. Code that uses version information does not look any different from code that does not.

Java's custom metadata is suitable for implementing such a design. The version information is stored in a custom class attribute, which can be accessed via command-line tools during development. At runtime, a special class loader reads and caches the version information and then uses it to rule out classes that do not match.

5.5.2 Serializable Classes as Attributes

The version information needs to take two different forms. The binary class format stores version information as a byte array, but other Java code accesses the version information as a Java object. The obvious approach to writing a Java object that can be converted to and from a byte array is to simply use a `Serializable` object. The `SerializableAttribute` class shown in Listing 5-19 represents a class attribute that is also a Java object.

Listing 5-19 The SerializableAttribute Class

```
package com.develop.classfile;

import com.develop.util.*;
import java.io.*;

public class SerializableAttribute extends Attribute
{
    private final Object info;
    private final byte [] packet;

    public SerializableAttribute(Object info)
        throws IOException {
        super("ser." + info.getClass().getName());
        this.info = info;
        packet = writePacket();
    }
    private SerializableAttribute(String name, short index,
        Object info, byte[] packet) {
        super(name, index);
        this.info = info;
        this.packet = packet;
    }

    public static Attribute read(String name, ClassFile cf,
        short name_index, int length)
        throws IOException
    {
        DataInputStream dis = cf.getStream();
        byte[] packet = new byte[length];
        dis.readFully(packet);
        ObjectInputStream ois = new ObjectInputStream(new
            ByteArrayInputStream(packet));
        Object info = null;
        try {
            info = ois.readObject();
        }
        catch (ClassNotFoundException cnfe) {
            return new CustomAttribute(name, packet);
        }
        return new SerializableAttribute(name, name_index,
            info, packet);
    }
}
```

```

private byte[] writePacket() throws IOException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    ObjectOutputStream oos = new ObjectOutputStream(baos);
    oos.writeObject(info);
    return baos.toByteArray();
}
public Object getObject() {
    return info;
}
public int getLength() {
    return packet.length;
}
public void writeToStream(DataOutputStream ds)
    throws IOException {
    super.writeToStream(ds);
    ds.write(packet);
}
public String toString() {
    return "Attribute " + getName() + "\n" + info;
}
}

```

The `SerializableAttribute` class simply holds two different representations of the same information: `info` holds a Java object, and `packet` holds the serialized form of that same object.

`SerializableAttribute`, and the rest of the code in this section, comes from the Java Class File Editor, an open source project developed by the author (see [JCFC] for details). Classes that are not listed in full here, such as `SerializableAttribute`'s base class `Attribute`, are JCFC classes that deal generically with the binary class format, and they are not specific to the current discussion.

The `SerializableAttribute` class, though originally written for this example, is entirely generic and can store any `Serializable` Java object as a custom class attribute. To construct a `SerializableAttribute` that stores version information, you will pass in an instance of `VersionInfo`, shown in Listing 5-20. The `version` field contains the version of a particular class, in whatever format you find meaningful. The `requiredVersions` field records the versions of other packages that this class depends on. The keys are package names, and the values are version objects in some format that you choose.

Listing 5-20 The VersionInfo Class

```
package com.develop.version;

import java.io.*;
import java.util.*;

public class VersionInfo implements Serializable {
    private final Object version;
    private final Map requiredVersions;

    public VersionInfo(Object version, HashMap requiredVersions)
    {
        this.version = version;
        this.requiredVersions = requiredVersions;
    }
    public VersionInfo(Object version, String pkgRequired,
                       Object versionRequired) {
        this.version = version;
        requiredVersions = new HashMap();
        requiredVersions.put(pkgRequired, versionRequired);
    }
    public Object getVersion() {
        return version;
    }
    public Map getRequiredVersions() {
        return Collections.unmodifiableMap(requiredVersions);
    }
    public String toString() {
        StringBuffer result = new
            StringBuffer("VersionInfo: ").append(version);
        Set s = requiredVersions.entrySet();
        for (Iterator it = s.iterator(); it.hasNext(); ) {
            Map.Entry e = (Map.Entry) it.next();
            result.append("\t").append(e.getKey())
                .append(": ").append(e.getValue());
        }
        return result.toString();
    }
}
```

5.5.3 Reading Attributes during Class Loading

To take advantage of this version information at runtime, you need a class loader that reads the version attribute *before* loading a class and then compares that version with the `requiredVersions` of all previously loaded classes. Figure 5–2 demonstrates the idea. A `VersioningLoader` searches multiple code sources, and it rejects class versions that do not match the version required by the client.

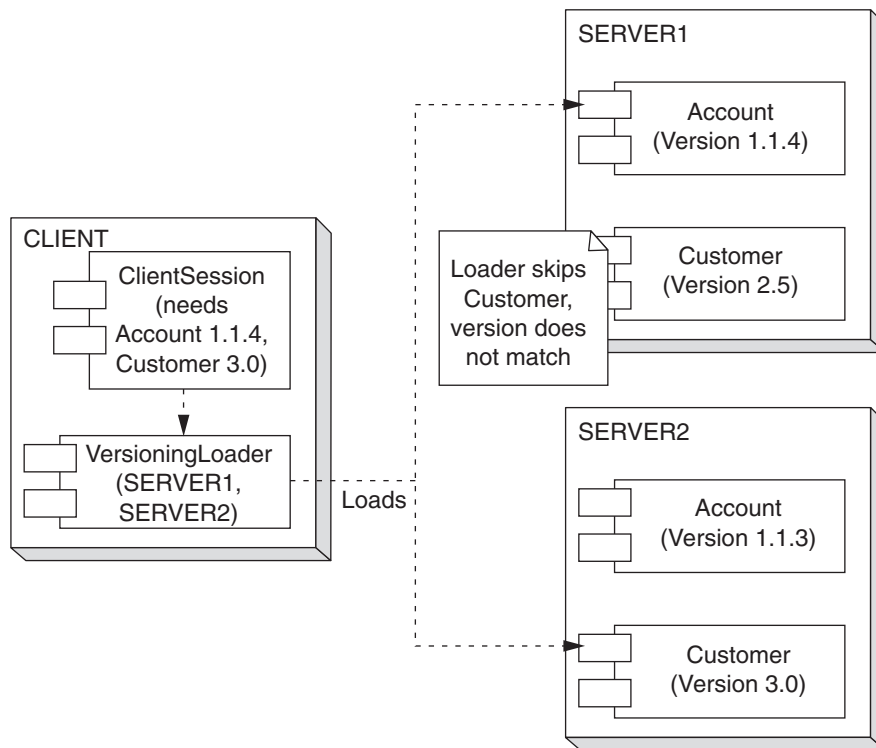


Figure 5–2 Applying a versioning policy

Listing 5–21 shows the `VersioningLoader` class. The meat of the class is the `findClass` method. Rather than loading a class from a single URL that matches the desired class name, the `VersioningLoader` uses `URLClassLoader`'s `findResources` method to return an enumeration of all potential matches, possibly one for each URL searched by the loader. Then, the loader

compares each of these classes with the version requirements of previously loaded classes.

Listing 5-21 The VersioningLoader Class

```
package com.develop.version;

import com.develop.classfile.*;
import java.io.*;
import java.net.*;
import java.security.CodeSource;
import java.util.*;

public class VersioningLoader extends URLClassLoader {
    private static boolean auditHit;
    private static boolean auditMiss;
    private static boolean auditFail;
    static {
        String audit =
            System.getProperty("com.develop.version.audit");
        if (audit != null) {
            if (-1 != audit.indexOf("hit")) auditHit = true;
            if (-1 != audit.indexOf("miss")) auditMiss = true;
            if (-1 != audit.indexOf("fail")) auditFail = true;
        }
    }
    private final VersionMatcher vm;
    private static class RequiredVersions {
        HashMap pkgToVersion = new HashMap();
        public List getPackageRequirements(String pkgName,
                                           boolean create) {
            List l = (List) pkgToVersion.get(pkgName);
            if (l == null && create == true) {
                l = new ArrayList();
                pkgToVersion.put(pkgName, l);
            }
            return l;
        }
    }
    RequiredVersions rv = new RequiredVersions();
    public VersioningLoader(URL[] urls, VersionMatcher vm)
    {
        super(urls);
        this.vm = vm;
    }
}
```



```

    }
    public VersioningLoader(URL[] urls, ClassLoader parent,
                           VersionMatcher vm) {
        super(urls, parent);
        this.vm = vm;
    }

    public VersioningLoader(URL[] urls, ClassLoader parent,
                           URLStreamHandlerFactory fact, VersionMatcher vm)
    {
        super(urls, parent, fact);
        this.vm = vm;
    }

    private URL getURLBase(URL url) {
        URL[] urls = getURLs();
        int length = urls.length;
        String stringForm = url.toExternalForm();
        for (int n=0; n<length; n++) {
            if (stringForm.startsWith(urls[n].toExternalForm())) {
                return urls[n];
            }
        }
        return null;
    }

    private byte[] getClassBytes(URL url) throws IOException {
        InputStream is = url.openStream();
        if (is == null)
            return null;
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        for (int ch=0; -1 != (ch=is.read()); )
            baos.write(ch);
        return baos.toByteArray();
    }

    private Class defineClass(String name, URL url,
                             byte[] bytes) {
        URL urlBase = getURLBase(url);
        if (urlBase == null) {
            throw new Error("url has no base");
        }
        return defineClass(name, bytes, 0, bytes.length,
                           new CodeSource(urlBase, null));
    }

    private static VersionInfo getVersionInfo(ClassFile cf) {
        SerializableAttribute sa = (SerializableAttribute)cf

```

```

        getAttribute("ser.com.develop.version.VersionInfo");
        if (sa == null) return null;
        return (VersionInfo)sa.getObject();
    }
    private boolean versionMatches(String name, VersionInfo vi)
    {
        int lastDot = name.lastIndexOf('.');
        String pkgName = (lastDot == -1) ? "" :
            name.substring(0,lastDot);
        List l = rv.getPackageRequirements(pkgName, false);
        Object version = vi.getVersion();
        if (l == null) return true;
        for (Iterator it = l.iterator(); it.hasNext(); ) {
            if (!vm.verify(it.next(), version)) {
                return false;
            }
        }
        return true;
    }
    private void updateVersionInfo(VersionInfo newInfo) {
        if (newInfo == null) return;
        Set newEntries = newInfo.getRequiredVersions()
            .entrySet();
        for (Iterator it = newEntries.iterator(); it.hasNext(); )
        {
            Map.Entry entry= (Map.Entry) it.next();
            List l=rv.getPackageRequirements(
                (String)entry.getKey(),
                true);
            l.add(entry.getValue());
        }
    }
    protected Class findClass(final String name)
        throws ClassNotFoundException
    {
        String className = name.replace('.', '/') + ".class";
        try {
            for (Enumeration e = findResources(className);
                e.hasMoreElements() ; ) {
                URL url = (URL) e.nextElement();
                byte[] bytes = getClassBytes(url);
                ClassFile cf = new ClassFile(bytes);
                VersionInfo vi = getVersionInfo(cf);
                if (versionMatches(name, vi)) {

```

```

        //must update version info before resolving class
        updateVersionInfo(vi);
        Class cls = defineClass(name, url, bytes);
        if (auditHit) {
            String vers = (vi == null) ? "" :
                vi.getVersion().toString();
            System.out.println("VL: Loading " + name + " "
                + vers + " from " + url);
        }
        return cls;
    }
    if (auditMiss) {
        System.out.println("VL: Missed match " + name
            + " at URL " + url);
    }
}
}
catch (IOException ioe) {}
if (auditFail) {
    System.out.println("VL: could not load " + name);
}
return null;
}
}

```

For each candidate class, the loader uses the helper method `getVersion-Info` to extract the class's custom version attribute. Next, the `version-Matches` method compares the candidate's version information with the requirements of previously loaded classes. The `RequiredVersions` nested class manages the cache of requirements. Note that there can be more than one requirement and that the candidate class must match all requirements. If the candidate is satisfactory, then its requirements are added to the cache via a call to `updateVersionInfo`. Finally, the class is loaded into the VM by `define-Class`. The ordering of these last two steps is very important. The version information must be cached before the new class is loaded because loading the class will probably trigger requests for other classes.

The `VersioningLoader` supports a flexible notion of "matching" versions. The actual work of matching version metadata is performed by an implementation of the `VersionMatcher` interface. The interface, plus a trivial

implementation that requires an exact match, is shown in Listing 5–22. You can write your own `VersionMatcher` implementations for other common versioning strategies. For example, you can store a standard dotted version number and always insist upon the highest-numbered version, or you can require a specific major version but accept any minor version. The JCFE functions for adding version attributes to a binary class could easily be integrated into a build process, which would make the use of version metadata transparent to application developers.

Listing 5–22 The `VersionMatcher` Interface

```
package com.develop.version;
public interface VersionMatcher {
    public boolean verify(Object requiredVersion,
                        Object matchVersion);
}

package com.develop.version;
public class ExactMatcher implements VersionMatcher {
    public boolean verify(Object requiredVersion,
                        Object matchVersion) {
        return requiredVersion.equals(matchVersion);
    }
}
```

5.5.4 Debugging Support

One final aspect of the `VersioningLoader` deserves mention. During its static initializer, the loader inspects the system property `com.develop.version.audit` looking for the strings `hit`, `miss`, and/or `fail`. If the property includes any of these strings, debugging output is sent to `System.out`. The `hit` output logs classes that are loaded, and the `miss` output logs classes that are skipped because their version data does not match. The `fail` output logs a complete failure to load a class, which means that either no classes were found, or that none of them were of the correct version. This debugging information is very helpful to application developers because the only time they are likely to encounter this infrastructure code is when something goes wrong. The idiom of using a system property to turn on various logging options is borrowed from other infrastructure projects in the Java platform itself, including both security and RMI.

Listing 5–23 shows audit output from the `VersioningLoader`. In this example, the loader is searching five URLs. First, the loader finds the `Caller` class with version information “Version1” at `loc1`. Then, the loader begins to seek for a matching version of a class named `Callee`. Each of the five URLs has a `Callee` class, but only the final URL at `loc5` has the correct version. So, the loader considers and rejects the other `Callee` binaries, finally accepting the correct one at `loc5`.

Listing 5–23 Audit Output from `VersioningLoader`

```
VL: Loading test.version.Caller Version0 from
file:/E:/jcfe/testout/loc0/test/version/Caller.class
VL: Missed match test.version.Callee at URL
file:/E:/jcfe/testout/loc0/test/version/Callee.class
VL: Missed match test.version.Callee at URL
file:/E:/jcfe/testout/loc1/test/version/Callee.class
VL: Missed match test.version.Callee at URL
file:/E:/jcfe/testout/loc2/test/version/Callee.class
VL: Missed match test.version.Callee at URL
file:/E:/jcfe/testout/loc3/test/version/Callee.class
VL: Loading test.version.Callee Version0 from
file:/E:/jcfe/testout/loc4/test/version/Callee.class
```

While the `VersioningLoader` is valuable in itself, it is intended to be illustrative rather than definitive. There are many other ways to approach versioning on the Java platform. You might want to use JAR metadata in addition to or instead of class metadata. You would almost certainly want to consider versioning relationships across multiple class loaders. The important point of this example is that *custom class metadata provides a way to extend the platform itself*. Rather than working around problems in the Java platform, you may be able to augment the Java platform to fit the way you work.

5.6 Onward

Custom class loaders allow you to define your own strategy for locating code and resources. You can make your own rules, as long as you can take a Java class name and turn it into a byte array in the binary class format. With custom class loaders you might load classes from an object database, from source control, or over a custom network protocol.

Protocol handlers split the responsibility of class loading into two distinct tasks. A `URLClassLoader` instance performs security work, assigning the `CodeSource` that will be used to associate the class with its runtime permissions. A protocol handler defines a URL protocol for locating resources that you can plug in anywhere the Java platform uses URLs. For many custom class loading tasks, you could use either a custom loader or a protocol handler, but protocol handlers have the advantage of being used to locate other resources besides classes.

Custom class loaders have advantages as well. A custom loader can process custom attributes added to a binary class. You can use custom attributes to extend the behavior of the Java classes in arbitrary ways, which are not directly correlated to the Java language itself. The `VersioningLoader` of this chapter is a powerful example; it shows that you can add valuable customizations to the process of Java class loading without changing the appearance of the platform to application developers.

5.7 Resources

There are several resources that cover different aspects of custom class loading. For a thorough treatment of the Java 2 security model, see [Gon99]. If you are writing custom class loaders, [New00] has some interesting examples. If you are writing a custom protocol handler, look at [Mas01], which describes a protocol handler for the Win32 registry.

If you are interested in custom attributes, and the loaders that read them, download the source code at [JCFE], which is a generic architecture for extending class loading and reflection to handle custom attributes.