Exhibit B

Understanding the Java ClassLoader

Skill Level: Introductory

Greg Travis (mito@panix.com)
Programmer

24 Apr 2001

This tutorial provides an overview of the Java ClassLoader and takes you through the construction of an example ClassLoader that automatically compiles your code before loading it. You'll learn exactly what a ClassLoader does, and what you need to do to create your own.

Section 1. Tutorial tips

Should I take this tutorial?

The Java ClassLoader is a crucial, but often overlooked, component of the Java run-time system. It is the class responsible for finding and loading class files at run time. Creating your own ClassLoader lets you customize the JVM in useful and interesting ways, allowing you to completely redefine how class files are brought into the system.

This tutorial provides an overview of the Java ClassLoader and takes you through the construction of an example ClassLoader that automatically compiles your code before loading it. You'll learn exactly what a ClassLoader does and what you need to do to create your own.

A basic understanding of Java programming, including the ability to create, compile, and execute simple command-line Java programs, as well as an understanding of the class file paradigm is sufficient background to take this tutorial.

Upon completion of this tutorial, you will know how to:

© Copyright IBM Corporation 2001. All rights reserved.

- Expand the functionality of the JVM
- Create a custom ClassLoader
- Learn how to integrate a custom ClassLoader into your Java application
- Modify your ClassLoader to accommodate the Java 2 release

Section 2. Introduction

What is a ClassLoader?

Among commercially popular programming languages, the Java language distinguishes itself by running on a Java virtual machine (JVM). This means that compiled programs are expressed in a special, platform-independent format, rather than in the format of the machine they are running on. This format differs from traditional executable program formats in a number of important ways.

In particular, a Java program, unlike one written in C or C++, isn't a single executable file, but instead is composed of many individual class files, each of which corresponds to a single Java class.

Additionally, these class files are not loaded into memory all at once, but rather are loaded on demand, as needed by the program. The ClassLoader is the part of the JVM that loads classes into memory.

The Java ClassLoader, furthermore, is written in the Java language itself. This means that it's easy to create your own ClassLoader without having to understand the finer details of the JVM.

Why write a ClassLoader?

If the JVM has a ClassLoader, then why would you want to write another one? Good question. The default ClassLoader only knows how to load class files from the local filesystem. This is fine for regular situations, when you have your Java program fully compiled and waiting on your computer.

But one of the most innovative things about the Java language is that it makes it easy for the JVM to get classes from places other than the local hard drive or network. For example, browsers use a custom ClassLoader to load executable content from a Web site.

© Copyright IBM Corporation 2001. All rights reserved.

There are many other ways to get class files. Besides simply loading files from the local disk or from a network, you can use a custom ClassLoader to:

- Automatically verify a digital signature before executing untrusted code
- Transparently decrypt code with a user-supplied password
- Create dynamically built classes customized to the user's specific needs

Anything you can think of to write that can generate Java bytecode can be integrated into your application.

Custom ClassLoader examples

If you've ever used the appletviewer included in the JDK or any Java-enabled browser, you've almost certainly used a custom ClassLoader.

When Sun initially released the Java language, one of the most exciting things was watching how this new technology executed code that it had loaded on the fly from a remote Web server. (This was before we'd realized something more exciting -- that Java technology provided a great language for writing code.) There was just something thrilling about it executing bytecode that had just been sent through an HTTP connection from a distant Web server.

What made this feat possible was the ability of the Java language to install a custom ClassLoader. The appletviewer contains a ClassLoader that, instead of looking in the local filesystem for classes, accesses a Web site on a remote server, loads the raw bytecode files via HTTP, and turns them into classes inside the JVM.

The ClassLoaders in browsers and appletviewers do other things as well: they take care of security and keep different applets on different pages from interfering with each other.

Echidna by Luke Gorrie is an open-source software package that allows you to safely run multiple Java applications inside a single virtual machine. (See Further reading and references.) It uses a custom ClassLoader to prevent the applications from interfering with each other, by giving each application its own copy of the class files.

Our example ClassLoader

After you have a good idea of how a ClassLoader works and how one is written, we'll create our own custom ClassLoader called CompilingClassLoader (CCL). CCL compiles our Java code for us, in case we didn't bother to do it ourselves. It's basically like having a simple "make" program built directly into our run-time system.

© Copyright IBM Corporation 2001. All rights reserved.

Note: Before we go any further, it's important to note that some aspects of the ClassLoader system have been improved in JDK version 1.2 (also known as the Java 2 platform). This tutorial was written with JDK versions 1.0 and 1.1 in mind, but everything in it works under later versions as well.

ClassLoader changes in Java 2 describes the changes in Java version 1.2 and provides details for modifying our ClassLoader to take advantage of these changes.

Section 3. The ClassLoader structure

ClassLoader structure overview

A ClassLoader's basic purpose is to service a request for a class. The JVM needs a class, so it asks the ClassLoader, by name, for this class, and the ClassLoader attempts to return a `Class` object that represents the class.

By overriding different methods corresponding to different stages of this process, you can create a custom ClassLoader.

In the remainder of this section, you'll learn about the critical methods of the Java ClassLoader. You'll find out what each one does and how it fits into the process of loading class files. You'll also find out what code you'll need to write when creating your own ClassLoader.

In the next section, you'll put that knowledge to work with our example ClassLoader, the CompilingClassLoader.

Method loadClass

`ClassLoader.loadClass()` is the entry point to the ClassLoader. Its signature is as follows:

```
Class loadClass( String name, boolean resolve );
```

The `name` parameter specifies the name of the class that the JVM needs, in package notation, such as `Foo` or `java.lang.Object`.

The `resolve` parameter tells the method whether or not the class needs to be resolved. You can think of class resolution as the task of completely preparing the class for execution. Resolution is not always needed. If the JVM needs only to

© Copyright IBM Corporation 2001. All rights reserved.

determine that the class exists or to find out what its superclass is, then resolution is not required.

In Java version 1.1 and earlier, the `loadClass` method is the only method that you need to override to create a custom ClassLoader. (ClassLoader changes in Java 2 provides information about the `findClass()` method available in Java 1.2.)

Method defineClass

The `defineClass` method is the central mystery of the ClassLoader. This method takes a raw array of bytes and turns it into a `Class` object. The raw array contains the data that, for example, was loaded from the filesystem or across the network.

`defineClass` takes care of a lot of complex, mysterious, and implementation-dependent aspects of the JVM -- it parses the bytecode format into a run-time data structure, checks for validity, and so on. But don't worry, you don't have to write it yourself. In fact, you couldn't override it even if you wanted to because it's marked as final.

Method findSystemClass

The `findSystemClass` method loads files from the local filesystem. It looks for a class file in the local filesystem, and if it's there, turns it into a class using `defineClass` to convert raw bytes into a `Class` object. This is the default mechanism for how the JVM normally loads classes when you are running a Java application. (ClassLoader changes in Java 2 provides details on changes to this process in Java version 1.2.)

For our custom ClassLoader, we'll use `findSystemClass` only after we've tried everything else to load a class. The reason is simple: our ClassLoader is responsible for carrying out special steps for loading classes, but not for *all* classes. For example, even if our ClassLoader loads some classes from a remote Web site, there are still plenty of basic Java libraries on the local machine that must also be loaded. These classes aren't our concern, so we ask the JVM to load them in the default way: from the local filesystem. This is what `findSystemClass` does.

The procedure works as follows:

- Our custom ClassLoader is asked to load a class.
- We check the remote Web site, to see if the class is there.
- If it is, fine; we grab the class and we're done.
- If it's not there, we assume that this class is one from the basic Java

© Copyright IBM Corporation 2001. All rights reserved.

libraries and call `findSystemClass` to load it from the filesystem.

In most custom ClassLoaders, you would want to call `findSystemClass` first to save time spent looking on the remote Web site for the many Java library classes that are typically loaded. However, as we'll see in the next section, we don't want to let the JVM load a class from the local filesystem until we've made sure that we've automatically compiled our application's code.

Method resolveClass

As I mentioned previously, loading a class can be done partially (without resolution) or completely (with resolution). When we write our version of `loadClass`, we may need to call `resolveClass`, depending on the value of the `resolve` parameter to `loadClass`.

Method findLoadedClass

`findLoadedClass` serves as a cache: when `loadClass` is asked to load a class, it can call this method to see if the class has already been loaded by this ClassLoader, saving the trouble of reloading a class that has already been loaded. This method should be called first.

Putting it all together

Let's see how all these methods fit together.

Our example implementation of `loadClass` carries out the following steps. (We won't specify here what special technique will be used to get the class file -- it might be loaded from the Net, or pulled out of an archive, or compiled on the fly. Whatever it is, it's the special magic that gets us our raw class file bytes.)

- Call `findLoadedClass` to see if we have already loaded the class.
- If we haven't loaded the class, we do special magic to get the raw bytes.
- If we have the raw bytes, call `defineClass` to turn them into a `Class` object.
- If we don't have the raw bytes, then call `findSystemClass` to see if we can get the class from the local filesystem.
- If the `resolve` parameter is `true`, call `resolveClass` to resolve the `Class` object.
- If we still don't have a class, throw a `ClassNotFoundException`.

© Copyright IBM Corporation 2001. All rights reserved.

- Otherwise, return the class to the caller.

Taking stock

Now that you have a working knowledge of ClassLoaders, it's time to build one. In the next section, we'll bring CCL to life.

Section 4. The CompilingClassLoader

CCL revealed

The job of our ClassLoader, CCL, is to make sure our code is compiled and up to date.

Here is a description of how it works:

- When a class is requested, see if it exists on disk, in the current directory, or in the appropriate subdirectory.
- If the class is not available, but the source is, call the Java compiler to generate the class file.
- If the class file does exist, check to see if it is older than its source code. If it is older than the source, call the Java compiler to regenerate the class file.
- If the compilation fails, or if for any other reason the class file could not be generated from the existing source, throw a `ClassNotFoundException`.
- If we still don't have the class, maybe it's in some other library, so call `findSystemClass` to see if that will work.
- If we still don't have the class, throw a `ClassNotFoundException`.
- Otherwise, return the class.

How Java compilation works

Before we get too far into our discussion, we should back up a bit and talk about

© Copyright IBM Corporation 2001. All rights reserved.

Java compilation. Generally, the Java compiler doesn't just compile the classes you ask it to. It also compiles other classes, if those classes are needed by the classes you've asked it to compile.

The CCL will compile each class in our application, one by one, that needs to be compiled. But, generally speaking, after the compiler compiles the first class, the CCL will find that all the other classes that needed to be compiled have in fact been compiled. Why? The Java compiler employs a rule similar to the one we are using: if a class doesn't exist or is out of date with respect to its source, then it needs to be compiled. In essence, the Java compiler is one step ahead of the CCL, and takes care of most of the work for it.

The CCL reports on what application classes it is compiling as it compiles them. In most cases, you'll see it call the compiler on the main class in your program, and that will be all it does -- a single invocation of the compiler is enough.

There is a case, however, in which some classes don't get compiled on the first pass. If you load a class by name, using the `Class.forName` method, the Java compiler won't know that this class is needed. In this case, you'll see the CCL run the Java compiler again to compile this class. The example in The source code illustrates this process.

Using the CompilationClassLoader

To use the CCL, we have to invoke our program in a special way. Instead of running the program directly, like this:

```
% java Foo arg1 arg2
```

we run it like this:

```
% java CCLRun Foo arg1 arg2
```

CCLRun is a special stub program that creates a CompilingClassLoader and uses it to load up the main class of our program, ensuring that the entire program will be loaded through the CompilingClassLoader. CCLRun uses the Java Reflection API to call the main method of the specified class and to pass the arguments to it. For more details, see The source code .

Example run

Included with the source is a set of small classes that illustrate how things work. The main program is a class called `Foo`, which creates an instance of class `Bar`. Class `Bar` creates an instance of another class called `Baz`, which is inside a package

© Copyright IBM Corporation 2001. All rights reserved.

called `baz` in order to illustrate that the CCL works with code in subpackages. `Bar` also loads a class by name, namely class `Boo`, to illustrate this ability also works with the CCL.

Each class announces that it has been loaded and run. Use The source code and try it out now. Compile CCLRun and CompilingClassLoader. Make sure you don't compile the other classes (`Foo`, `Bar`, `Baz`, and `Boo`) or the CCL won't be of any use, because the classes will already have been compiled.

```
% java CCLRun Foo arg1 arg2
CCL: Compiling Foo.java...
foo! arg1 arg2
bar! arg1 arg2
baz! arg1 arg2
CCL: Compiling Boo.java...
Boo!
```

Note that the first call to the compiler, for `Foo.java`, takes care of `Bar` and `baz.Baz` as well. `Boo` doesn't get called until `Bar` tries to load it by name, at which point our CCL has to invoke the compiler again to compile it.

Section 5. ClassLoader changes in Java 2

Overview

The ClassLoader facility has been improved in Java versions 1.2 and later. Any code written for the old system will work, but the new system can make your life a bit easier.

The new model is a *delegation* model. A ClassLoader can delegate a request for a class to its parent. The default implementation calls on the parent *before* trying to load the class itself, but this policy can be changed. At the root of all ClassLoaders is the system ClassLoader, which loads classes the default way -- that is, from the local filesystem.

Default implementation of loadClass

A custom-written `loadClass` method generally tries several things to load a requested class, and if you write a lot of ClassLoaders, you'll find yourself writing variations on the same, fairly complicated method over and over again.

© Copyright IBM Corporation 2001. All rights reserved.

The default implementation of `loadClass` in Java 1.2 embodies the most common approach to finding a class and lets you customize it by overriding the new `findClass` method, which `loadClass` calls at the appropriate time.

The advantage of this approach is that you probably don't have to override `loadClass`; you only have to override `findClass`, which is less work.

New method: findClass

This new method is called by the default implementation of `loadClass`. The purpose of `findClass` is to contain all your specialized code for your ClassLoader, without having to duplicate the other code (such as calling the system ClassLoader when your specialized method has failed).

New method: getSystemClassLoader

Whether you override `findClass` or `loadClass`, `getSystemClassLoader` gives you direct access to the system ClassLoader in the form of an actual `ClassLoader` object (instead of accessing it implicitly through the `findSystemClass` call).

New method: getParent

This new method allows a ClassLoader to get at its parent ClassLoader, in order to delegate class requests to it. You might use this approach when your custom ClassLoader can't find a class using your specialized method.

The parent of a ClassLoader is defined as the ClassLoader of the object containing the code that created that ClassLoader.

Section 6. The source code

CompilingClassLoader.java

Here is the source code for CompilingClassLoader.java

```
// $Id$
```


© Copyright IBM Corporation 2001. All rights reserved.

```
import java.io.*;

/*

A CompilingClassLoader compiles your Java source on-the-fly.  It
checks for nonexistent .class files, or .class files that are older
than their corresponding source code.

*/

public class CompilingClassLoader extends ClassLoader
{
  // Given a filename, read the entirety of that file from disk
  // and return it as a byte array.
  private byte[] getBytes( String filename ) throws IOException {
    // Find out the length of the file
    File file = new File( filename );
    long len = file.length();

    // Create an array that's just the right size for the file's
    // contents
    byte raw[] = new byte[(int)len];

    // Open the file
    FileInputStream fin = new FileInputStream( file );

    // Read all of it into the array; if we don't get all,
    // then it's an error.
    int r = fin.read( raw );
    if (r != len)
      throw new IOException( "Can't read all, "+r+" != "+len );

    // Don't forget to close the file!
    fin.close();

    // And finally return the file contents as an array
    return raw;
  }

  // Spawn a process to compile the java source code file
  // specified in the 'javaFile' parameter.  Return a true if
  // the compilation worked, false otherwise.
  private boolean compile( String javaFile ) throws IOException {
    // Let the user know what's going on
    System.out.println( "CCL: Compiling "+javaFile+"..." );

    // Start up the compiler
    Process p = Runtime.getRuntime().exec( "javac "+javaFile );

    // Wait for it to finish running
    try {
      p.waitFor();
    } catch( InterruptedException ie ) { System.out.println( ie ); }

    // Check the return code, in case of a compilation error
    int ret = p.exitValue();

    // Tell whether the compilation worked
    return ret==0;
  }

  // The heart of the ClassLoader -- automatically compile
  // source as necessary when looking for class files
  public Class loadClass( String name, boolean resolve )
      throws ClassNotFoundException {

    // Our goal is to get a Class object
    Class clas = null;
```


© Copyright IBM Corporation 2001. All rights reserved.

```
    // First, see if we've already dealt with this one
    clas = findLoadedClass( name );

    //System.out.println( "findLoadedClass: "+clas );

    // Create a pathname from the class name
    // E.g. java.lang.Object => java/lang/Object
    String fileStub = name.replace( '.', '/' );

    // Build objects pointing to the source code (.java) and object
    // code (.class)
    String javaFilename = fileStub+".java";
    String classFilename = fileStub+".class";

    File javaFile = new File( javaFilename );
    File classFile = new File( classFilename );

    //System.out.println( "j "+javaFile.lastModified()+" c "+
    //  classFile.lastModified() );

    // First, see if we want to try compiling.  We do if (a) there
    // is source code, and either (b0) there is no object code,
    // or (b1) there is object code, but it's older than the source
    if (javaFile.exists() &&
         (!classFile.exists() ||
          javaFile.lastModified() > classFile.lastModified())) {

      try {
        // Try to compile it.  If this doesn't work, then
        // we must declare failure.  (It's not good enough to use
        // and already-existing, but out-of-date, classfile)
        if (!compile( javaFilename ) || !classFile.exists()) {
          throw new ClassNotFoundException( "Compile failed: "+javaFilename );
        }
      } catch( IOException ie ) {

        // Another place where we might come to if we fail
        // to compile
        throw new ClassNotFoundException( ie.toString() );
      }
    }

    // Let's try to load up the raw bytes, assuming they were
    // properly compiled, or didn't need to be compiled
    try {

      // read the bytes
      byte raw[] = getBytes( classFilename );

      // try to turn them into a class
      clas = defineClass( name, raw, 0, raw.length );
    } catch( IOException ie ) {
      // This is not a failure!  If we reach here, it might
      // mean that we are dealing with a class in a library,
      // such as java.lang.Object
    }

    //System.out.println( "defineClass: "+clas );

    // Maybe the class is in a library -- try loading
    // the normal way
    if (clas==null) {
      clas = findSystemClass( name );
    }

    //System.out.println( "findSystemClass: "+clas );

    // Resolve the class, if any, but only if the "resolve"
```


© Copyright IBM Corporation 2001. All rights reserved.

```
    // flag is set to true
    if (resolve && clas != null)
      resolveClass( clas );

    // If we still don't have a class, it's an error
    if (clas == null)
      throw new ClassNotFoundException( name );

    // Otherwise, return the class
    return clas;
  }
}
```

CCRun.java

Here is the source code for CCRun.java

```
// $Id$

import java.lang.reflect.*;

/*

CCLRun executes a Java program by loading it through a
CompilingClassLoader.

*/

public class CCLRun
{
  static public void main( String args[] ) throws Exception {

    // The first argument is the Java program (class) the user
    // wants to run
    String progClass = args[0];

    // And the arguments to that program are just
    // arguments 1..n, so separate those out into
    // their own array
    String progArgs[] = new String[args.length-1];
    System.arraycopy( args, 1, progArgs, 0, progArgs.length );

    // Create a CompilingClassLoader
    CompilingClassLoader ccl = new CompilingClassLoader();

    // Load the main class through our CCL
    Class clas = ccl.loadClass( progClass );

    // Use reflection to call its main() method, and to
    // pass the arguments in.

    // Get a class representing the type of the main method's argument
    Class mainArgType[] = { (new String[0]).getClass() };

    // Find the standard main method in the class
    Method main = clas.getMethod( "main", mainArgType );

    // Create a list containing the arguments -- in this case,
    // an array of strings
    Object argsArray[] = { progArgs };
```


© Copyright IBM Corporation 2001. All rights reserved.

```
    // Call the method
    main.invoke( null, argsArray );
  }
}
```

Foo.java

Here is the source code for Foo.java

```
// $Id$

public class Foo
{
  static public void main( String args[] ) throws Exception {
    System.out.println( "foo! "+args[0]+" "+args[1] );
    new Bar( args[0], args[1] );
  }
}
```

Bar.java

Here is the source code for Bar.java

```
// $Id$

import baz.*;

public class Bar
{
  public Bar( String a, String b ) {
    System.out.println( "bar! "+a+" "+b );
    new Baz( a, b );

    try {
      Class booClass = Class.forName( "Boo" );
      Object boo = booClass.newInstance();
    } catch( Exception e ) {
      e.printStackTrace();
    }
  }
}
```

baz/Baz.java

© Copyright IBM Corporation 2001. All rights reserved.

Here is the source code for baz/Baz.java

```
// $Id$

package baz;

public class Baz
{
  public Baz( String a, String b ) {
    System.out.println( "baz! "+a+" "+b );
  }
}
```

Boo.java

Here is the source code for Boo.java

```
// $Id$

public class Boo
{
  public Boo() {
    System.out.println( "Boo!" );
  }
}
```

Section 7. Wrapup

Wrapup

As you have seen in this short tutorial, knowing how to create a custom ClassLoader can really help you get at the guts of the JVM. The ability to load class files from any source, or even to generate them on the fly, can extend the reach of your JVM and allow you to do some really interesting and powerful things.

Other ClassLoader ideas

As I mentioned earlier in this tutorial, custom ClassLoaders are crucial to programs

© Copyright IBM Corporation 2001. All rights reserved.

like Java-enabled browsers and appletviewers. Here are a few other ideas for interesting ClassLoaders:

- **Security.** Your ClassLoader could examine classes before they are handed off to the JVM to see if they have a proper digital signature. You can also create a kind of "sandbox" that disallows certain kinds of method calls by examining the source code and rejecting classes that try to do things outside the sandbox.
- **Encryption.** It's possible to create a ClassLoader that decrypts on the fly, so that your class files on disk are not readable by someone with a decompiler. The user must supply a password to run the program, and the password is used to decrypt the code.
- **Archiving.** Want to distribute your code in a special format or with special compression? Your ClassLoader can pull raw class file bytes from any source it wants.
- **Self-extracting programs.** It's possible to compile an entire Java application into a single executable class file that contains compressed and/or encrypted class file data, along with an integral ClassLoader; when the program is run, it unpacks itself *entirely in memory* -- no need to install first.
- **Dynamic generation.** They sky's the limit here. You can generate classes that refer to other classes that haven't been generated yet -- create entire classes on the fly and bring them into the JVM without missing a beat.

© Copyright IBM Corporation 2001. All rights reserved.

Resources

- Read the online documentation for the ClassLoader class on the Sun Web site.
- The Java Developer Connection has a custom ClassLoader tutorial.
- Learn about class file loading in The Java Language Specification.
- The Java Virtual Machine Specification includes information about ClassLoaders.
- The JDK version 1.3 documentation has a list of core Java library classes that use ClassLoaders.
- " Create a Java 1.2-style custom ClassLoader " (*JavaWorld* March 2000) provides insight on building a ClassLoader under JDK 1.2.
- " Make classes from XML data " (developerWorks, August 2000) describes using a custom ClassLoader to create new classes on the fly.
- If you are new to the Java platform, Java language essentials (developerWorks, November 2000) provides a thorough guide to the platform fundamentals.

About the author

Greg Travis

Greg Travis is a freelance programmer living in New York City. His interest in computers can be traced back to that episode of "The Bionic Woman" where Jamie is trying to escape a building whose lights and doors are controlled by an evil artificial intelligence, which mocks her through loudspeakers. Greg is a firm believer that, when a computer program works, it's a complete coincidence.

© Copyright IBM Corporation 2001. All rights reserved.