JAVAWORLD
SOLUTIONS FOR JAVA DEVELOPERS

This story appeared on JavaWorld at
http://www.javaworld.com/jw-10-1996/jw-10-indepth.html

The basics of Java class loaders

The fundamentals of this key component of the Java architecture

By Chuck Mcmanis, JavaWorld.com, 10/01/96

The class loader concept, one of the cornerstones of the Java virtual machine, describes the behavior of converting a named class into the bits responsible for implementing that class. Because class loaders exist, the Java run time does not need to know anything about files and file systems when running Java programs.

What class loaders do

Classes are introduced into the Java environment when they are referenced by name in a class that is already running. There is a bit of magic that goes on to get the first class running (which is why you have to declare the *main()* method as static, taking a string array as an argument), but once that class is running, future attempts at loading classes are done by the class loader.

At its simplest, a class loader creates a flat name space of class bodies that are referenced by a string name. The method definition is:

```
Class r = loadClass(String className, boolean resolveIt);
```

The variable *className* contains a string that is understood by the class loader and is used to uniquely identify a class implementation. The variable *resolveIt* is a flag to tell the class loader that classes referenced by this class name should be resolved (that is, any referenced class should be loaded as well).

All Java virtual machines include one class loader that is embedded in the virtual machine. This embedded loader is called the primordial class loader. It is somewhat special because the virtual machine assumes that it has access to a repository of *trusted classes* which can be run by the VM without verification.

The primordial class loader implements the default implementation of *loadClass()*. Thus, this code understands that the class name **java.lang.Object** is stored in a file with the prefix `java/lang/Object.class` somewhere in the class path. This code also implements both class path searching and looking into zip files for classes. The really cool thing about the way this is designed is that Java can change its class storage model simply by changing the set of functions that implements the class loader.

Digging around in the guts of the Java virtual machine, you will discover that the primordial class loader is implemented primarily in the functions *FindClassFromClass* and *ResolveClass*.

So when are classes loaded? There are exactly two cases: when the new bytecode is executed (for example, **FooClass** *f* = new **FooClass()**;) and when the bytecodes make a static reference to a class (for example, **System**.*out*).

A non–primordial class loader

"So what?" you might ask.

The Java virtual machine has hooks in it to allow a user–defined class loader to be used in place of the primordial one. Furthermore, since the user class loader gets first crack at the class name, the user is able to implement any number of interesting class

repositories, not the least of which is HTTP servers -- which got Java off the ground in the first place.

There is a cost, however, because the class loader is so powerful (for example, it can replace **java.lang.Object** with its own version), Java classes like applets are not allowed to instantiate their own loaders. (This is enforced by the class loader, by the way.) This column will not be useful if you are trying to do this stuff with an applet, only with an application running from the trusted class repository (such as local files).

A user class loader gets the chance to load a class before the primordial class loader does. Because of this, it can load the class implementation data from some alternate source, which is how the **AppletClassLoader** can load classes using the HTTP protocol.

Building a SimpleClassLoader

A class loader starts by being a subclass of **java.lang.ClassLoader**. The only abstract method that must be implemented is *loadClass()*. The flow of *loadClass()* is as follows:

- Verify class name.
- Check to see if the class requested has already been loaded.
- Check to see if the class is a "system" class.
- Attempt to fetch the class from this class loader's repository.
- Define the class for the VM.
- Resolve the class.
- Return the class to the caller.

Some Java code that implements this flow is taken from the file SimpleClassLoader and appears as follows with descriptions about what it does interspersed with the code.

```
public synchronized Class loadClass(String className, boolean resolveIt)
        throws ClassNotFoundException {
        Class result;
        byte classData[];
        System.out.println(" >>>>>> Load class : "+className);
```

```
        /* Check our local cache of classes */
        result = (Class)classes.get(className);
        if (result != null) {
            System.out.println(" >>>>>> returning cached result.");
            return result;
}
```

The code above is the first section of the *loadClass* method. As you can see, it takes a class name and searches a local hash table that our class loader is maintaining of classes it has already returned. It is important to keep this hash table around since you **must** return the same class object reference for the same class name every time you are asked for it. Otherwise the system will believe there are two different classes with the same name and will throw a **ClassCastException** whenever you assign an object reference between them. It's also important to keep a cache because the *loadClass()* method is called recursively when a class is being resolved, and you will need to return the cached result rather than chase it down for another copy.

```
/* Check with the primordial class loader */
try {
    result = super.findSystemClass(className);
    System.out.println(" >>>>>> returning system class (in CLASSPATH).");
    return result;
    } catch (ClassNotFoundException e) {
            System.out.println("        >>>>>> Not a system class.");
    }
```

As you can see in the code above, the next step is to check if the primordial class loader can resolve this class name. This check is essential to both the sanity and security of the system. For example, if you return your own instance of **java.lang.Object** to the caller, then this object will share no common superclass with any other object! The security of the system can be compromised if your class loader returned its own value of **java.lang.SecurityManager**, which did not have the same checks as the real one did.

```
/* Try to load it from our repository */
classData = getClassImplFromDataBase(className);
if (classData == null) {
    throw new ClassNotFoundException();
```

```
}
```

After the initial checks, we come to the code above which is where the simple class loader gets an opportunity to load an implementation of this class. As you can see from the source code, the **SimpleClassLoader** has a method *getClassImplFromDataBase()* which in our simple example merely prefixes the directory `"store\"` to the class name and appends the extension `".impl"`. I chose this technique in the example so that there would be no question of the primordial class loader finding our class. Note that the **sun.applet.AppletClassLoader** prefixes the codebase URL from the HTML page where an applet lives to the name and then does an HTTP get request to fetch the bytecodes.

```
/* Define it (parse the class file) */
result = defineClass(classData, 0, classData.length);
```

If the class implementation was loaded, the penultimate step is to call the *defineClass()* method from **java.lang.ClassLoader**, which can be considered the first step of class verification. This method is implemented in the Java virtual machine and is responsible for verifying that the class bytes are a legal Java class file. Internally, the *defineClass* method fills out a data structure that the JVM uses to hold classes. If the class data is malformed, this call will cause a **ClassFormatError** to be thrown.

```
if (resolveIt) {
    resolveClass(result);
}
```

The last class loader–specific requirement is to call *resolveClass()* if the boolean parameter *resolveIt* was true. This method does two things: First, it causes any classes that are referenced by this class explicitly to be loaded and a prototype object for this class to be created; then, it invokes the verifier to do dynamic verification of the legitimacy of the bytecodes in this class. If verification fails, this method call will throw

a **LinkageError**, the most common of which is a **VerifyError**.

Note that for any class you will load, the *resolveIt* variable will always be true. It is only when the system is recursively calling *loadClass()* that it may set this variable false because it knows the class it is asking for is already resolved.

```
classes.put(className, result);
System.out.println("        >>>>>> Returning newly loaded class.");
        return result;
}
```

The final step in the process is to store the class we've loaded and resolved into our hash table so that we can return it again if need be, and then to return the **Class** reference to the caller.

Of course if it were this simple there wouldn't be much more to talk about. In fact, there are two issues that class loader builders will have to deal with, security and talking to classes loaded by the custom class loader.

Security considerations

Whenever you have an application loading arbitrary classes into the system through your class loader, your application's integrity is at risk. This is due to the power of the class loader. Let's take a moment to look at one of the ways a potential villain could break into your application if you aren't careful.

In our simple class loader, if the primordial class loader couldn't find the class, we loaded it from our private repository. What happens when that repository contains the class **java.lang.FooBar** ? There is no class named **java.lang.FooBar**, but we could install one by loading it from the class repository. This class, by virtue of the fact that it would have access to any package–protected variable in the **java.lang** package, can manipulate some sensitive variables so that later classes could subvert security measures. Therefore, one of the jobs of any class loader is to *protect the system name space*.

In our simple class loader we can add the code:

```
if (className.startsWith("java."))
    throw newClassNotFoundException();
```

just after the call to *findSystemClass* above. This technique can be used to protect any package where you are sure that the loaded code will never have a reason to load a new class into some package.

Another area of risk is that the name passed must be a verified valid name. Consider a hostile application that used a class name of "..\..\..\..\netscape\temp\xxx.class" as its class name that it wanted loaded. Clearly, if the class loader simply presented this name to our simplistic file system loader this might load a class that actually wasn't expected by our application. Thus, before searching our own repository of classes, it is a good idea to write a method that verifies the integrity of your class names. Then call that method just before you go to search your repository.

Using an interface to bridge the gap

The second non–intuitive issue with working with class loaders is the inability to cast an object that was created from a loaded class into its original class. You need to cast the object returned because the typical use of a custom class loader is something like:

```
CustomClassLoader ccl = new CustomClassLoader();
Object o;
Class c;
c = ccl.loadClass("someNewClass");
o = c.newInstance();
((SomeNewClass)o).someClassMethod();
```

However, you cannot cast *o* to **SomeNewClass** because only the custom class loader "knows" about the new class it has just loaded.

There are two reasons for this. First, the classes in the Java virtual machine are considered castable if they have at least one common class pointer. However, classes loaded by two different class loaders will have two different class pointers and no classes in common (except **java.lang.Object** usually). Second, the idea behind having a custom class loader is to load classes *after* the application is deployed so the application does not know a priory about the classes it will load. This dilemma is solved by giving both the application and the loaded class a class in common.

There are two ways of creating this common class, either the loaded class must be a subclass of a class that the application has loaded from its trusted repository, or the loaded class must implement an interface that was loaded from the trusted repository. This way the loaded class and the class that does not share the complete name space of the custom class loader have a class in common. In the example I use an interface named **LocalModule**, although you could just as easily make this a class and subclass it.

The best example of the first technique is a Web browser. The class defined by Java that is implemented by all applets is **java.applet.Applet**. When a class is loaded by **AppletClassLoader**, the object instance that is created is cast to an instance of **Applet**. If this cast succeeds the *init()* method is called. In my example I use the second technique, an interface.

Playing with the example

To round out the example I've created a couple more `.java` files. These are:

```
public interface LocalModule {
    /* Start the module */
    void start(String option);
}
```

This interface is shared between my local class repository (it is in the classpath of my application) and the loaded module. A test class helps illustrate the class loader in operation. The code below shows an example class named **TestClass** that implements

the shared interface. When this class is loaded by the **SimpleClassLoader** class, we can cast instantiated objects to the common interface named **LocalModule**.

```
import java.util.Random;
import java.util.Vector>;
public class TestClass implements LocalModule {
    /*
     * This is an example of a class reference that will be resolved
     * at load time.
     */
    Vector v = new Vector();
    /** This is our start function */
    public void start(String opt) {
        /* This reference will be resolved at run time. */
        Random r;
        System.out.println("Running the Test class, option was                    '"+opt+"'");
        System.out.println("Now initializing a Random object.");
        r = new Random();
        for (int i = 0; i < 5; i++) {
            v.addElement(new Integer(r.nextInt()));
        }
        /* This reference should get the cached copy of random. */
        r = new Random();
        System.out.println("A series of 5 random numbers: ");
        for (int i = 0; i < v.size(); i++) {
            Integer z = (Integer)v.elementAt(i);
            System.out.println(i+": "+z);
        }
    }
}
```

This test class is to be loaded by our simple class loader and then executed by the example application. There are a couple of interesting things to try and do when running the application. First, watch which classes get loaded and when they get loaded. The initialization of *v* and the static reference to **System** cause these classes to be loaded. Furthermore, since *out* is actually a **PrintStream** object, this class gets loaded as well, and perhaps suprisingly the class **java.lang.StringBuffer** gets loaded. This last class is loaded because the compiler automatically changes string expressions into invocations of **StringBuffer** objects.

When you run it, you will notice that the **java.util.Random** class is not loaded until after the statement `"Now initializing a Random object"` is printed. Even though it was referenced in the definition of *r*, it is not actually used until the **new** operator is executed. The last thing to notice is that the second time **java.util.Random** is

referenced, your class loader does not get a chance to load it; it is already installed in the virtual machine's loaded class database. It is this latter feature that will be changed in a future release of Java to support garbage collectible classes.

Our simple application **Example** ties everything together:

```
public class Example {
    public static void main(String args[]) {
        SimpleClassLoader sc = new SimpleClassLoader();
        Object o;
        String tst = "TestClass";
        System.out.println("This program will use SimpleClassLoader.");
             if (args.length != 0)
                 tst = args[0];
             try {
                 o = (sc.loadClass(tst)).newInstance();
                ((LocalModule) o).start("none");
             } catch (Exception e) {
                 System.out.println("Caught exception : "+e);
             }
        }
}
```

This simple application first creates a new instance of **SimpleClassLoader** and then uses it to load the class named **TestClass** (which is shown above). When the class is loaded, the *newInstance()* method (defined in class **Class**) is invoked to generate a new instance of the **TestClass** object; of course, we don't know it's a **TestClass**. However, our application has been designed to accept only classes that implement the **LocalModule** interface as valid loadable classes. If the class loaded did **not** implement the **LocalModule** interface, the cast in line #12 will throw a **ClassCastException**. If the cast does succeed then we invoke the expected *start()* method and pass it an option string of "none."

Winding down, wrapping up

With this introduction you should have enough information to build an application with its own custom class loader. Feel free to base any code you wish on the code in this article (for commercial or non–commercial purposes).

Class loaders provide the mechanism that allows Java applications, whether they are Web browsers or EMACs replacements, to be dynamically extended in a controlled way with additional Java code. The applications of class loaders are bounded only by your imagination. Some interesting experiments for you to run if you download and compile the code.

- Try creating two instances of **SimpleClassLoader** and see what programs loaded by separate class loaders can, and cannot, do to or with each other.
- Think of how you might be able to implement class identity checks with a class loader.
- Try writing a text editor where every key is bound to a dynamically loaded Java class, then send me a copy. :-)

Next month I think I'll focus a bit on application development in Java, building something a bit more significant than spinning pictures or tumbling mascots.

About the author

Chuck McManis is currently the director of system software at FreeGate Corp. FreeGate is a venture-funded start-up that is exploring opportunities in the Internet marketplace. Before joining FreeGate, McManis was a member of the Java group. He joined the Java group just after the formation of FirstPerson Inc. and was a member of the portable OS group (the group responsible for the OS portion of Java). Later, when FirstPerson was dissolved, he stayed with the group through the development of the alpha and beta versions of the Java platform. He created the first "all Java" home page on the Internet when he did the programming for the Java version of the Sun home page in May 1995. He also developed a cryptographic library for Java and versions of the Java class loader that could screen classes based on digital signatures. Before joining FirstPerson, Chuck worked in the operating systems area of SunSoft developing networking applications, where he did the initial design of NIS+.