Custom Class Loaders Using Java

by Kelly Campbell

Comment on this article

Get the Acrobat PDF of this article

- [Download the Source for this article(4KB)]

One of the neatest things to come out of the Java revolution is the applet. Prior to applets web pages were static - no animation, no scrolling marquees, no dynamic content. Today web pages are all about dynamic content. This has changed the web experience from one of static text and images to mini-applications embedded in a web browser.

When your browser encounters a Java applet on a web page, it pulls the applet across the wire and starts it up. Your browser is literally executing code it has never seen before. This is a key feature of Java: the ability to dynamically load and execute code. When I first started working with Java the notion of having an application execute new code without re-compiling, or re-linking, or even re-installing struck me as pretty powerful.

So what allows your browser to dynamically load and execute an applet? The answer is a custom class loader. By default, the Virtual Machine loads classes from the file system. Using custom class loader, a browser is able to load new classes, in the form of an applet, through the Internet.

In this article we'll create a custom class loader and use it to load and execute new classes dynamically. To keep things clean and simple our custom class loader will load its classes from the file system.

Starting Out

Java provides an abstract class, java.lang.ClassLoader, as a starting point for building custom class loaders. We'll derive our class, FileClassLoader, from ClassLoader. The bulk of our code will be in the overridden method loadClass. This method is called by the Virtual Machine to resolve a string name into a Class object.

There are a couple of rules a well-behaved class loader must follow:

- Don't load system classes. Java is pretty security conscious. It wouldn't do to allow a custom class loader to provide a rogue, or even a "better" version of a system class.
- Don't load the same class twice. It's not a good idea to have two classes, or different versions of the same class, floating around in memory. We'll have to keep a reference to each class we load. If we're asked to load it again, we'll just return the reference.
- Don't fool around with the byte-code. The Virtual Machine will be checking behind our backs to make sure that the byte-code we provide for our new class is valid. We'll simply use the output from a Java compiler, a .class file, to make sure we're passing in valid byte-code.

Class Prototype

Here is the prototype for the FileClassLoader class.

```
public class FileClassLoader extends ClassLoader
 {
    // called by the Virtual Machine to convert name into a Class.
    public synchronized Class loadClass( String name, boolean resolve )
        throws ClassNotFoundException

    // return the current path used for loading .class files.
    public String getCurrentPath()

    // set the path used for loading .class files.
    public void setCurrentPath( String s )

    // internal method used to load data from a .class file.
    private byte loadClassData( String filename )[]

    // stores references to the classes we load.
    private Hashtable cache = new Hashtable();

    // path to use when loading .class files.
    private String sCurrentPath = null;
}
```

As you might suspect, the interesting code takes place in the loadClass and loadClassData methods. Once a class has been successfully loaded, we'll store a reference to it in the cache. This reference will be returned if the class is requested again. The set and get methods are used to maintain the path to the .class files. The expectation is that users of FileClassLoader will ask for classes by name, without any leading path or package information.

Using the FileClassLoader

Now that we have a custom class loader, the next step is to put it to work. There are five basic steps for using the FileClassLoader.

1. Create an instance of the FileClassLoader.
2. Call the loadClass method, storing the returned Class object.
3. Call newInstance to create an instance of the new class. This will return an Object reference.
4. Cast the Object reference into an appropriate interface.
5. Access the newly loaded class through the interface.

Step 3 needs a bit of elaboration. Why do we need to cast the instantiated Object into an "appropriate" interface? And what is an "appropriate" interface anyway? The call to classRef.newInstance returns a reference to an Object instance. We can't really do anything interesting from an application standpoint using the methods on Object. This is where the "appropriate" interface comes in. We will simply define an interface having methods that are useful in our application and require each class we load to implement this interface. Instances of the newly loaded class can then be cast to this interface. The interface is "appropriate" because we will define the methods we need to get the job done.

Below is the definition of a simple interface that we can use for testing purposes.

```
/* ----------------------- *
 * I T e s t . j a v a
 * ----------------------- */
public interface ITest
{
        public void test();
}
```

Thus, we are stating that each class we load dynamically will implement the test method of the ITest interface. The code below shows one possible implementation. Compile Test.java using javac to create Test.class, the file to be loaded by the FileClassLoader.

```
/* ----------------------- *
 * I T e s t . j a v a
 * ----------------------- */
import ITest;

public class Test implements ITest
{
    public void test()
    {
        System.out.println( "* ------------------- *" );
        System.out.println( "* Test method called! *" );
        System.out.println( "* ------------------- *" );
    }
}
```

We now have a custom class loader, an interface, and a .class file to load. All that's left is a driver class to put all the pieces together.

```
/* ------------------------------------------------------------------------ *
 * M a i n A p p . j a v a
 * Shell to launch the application.
 * ------------------------------------------------------------------------ */

// The interface we will use to access the classes loaded via the
// FileClassLoader.  While the FileClassLoader doesn't use this interface,
// we have to have some way to call methods on the classes it returns.
import ITest;

/* ------------------------------------------------------------------------ *
 * Wrapper class for main.
 * ------------------------------------------------------------------------ */
public class MainApp
{
    static public void main( String args[] )
    {
        try
        {
            /* ------------------------------------------------------------ *
             * Create an instance of our custom class loader and point
             * it to the directory containing the .class files to load.
             * Assume the .class files are in a subdirectory called
             * 'classes'.
             * ------------------------------------------------------------ */
            FileClassLoader loader = new FileClassLoader();
            loader.setCurrentPath( "classes\\" );

            /* ------------------------------------------------------------ *
             * Load the class named "Test" using the FileClassLoader.
             * Call newInstance() to create a new instance of the class.
             * newInstance() returns a reference to a Java Object.
             * ------------------------------------------------------------ */
            Class  classRef = loader.loadClass( "Test" );
            Object objectRef = classRef.newInstance();
            ITest testClass = null;

            /* ------------------------------------------------------------ *
             * Verify that the newly created class instance implements the
             * ITest interface.  If it does, call the test() method.
             * ------------------------------------------------------------ */
            if( objectRef instanceof ITest )
            {
                testClass = (ITest) objectRef;
                testClass.test();
            }
            else
            {
                System.out.println( "ERROR:  Loaded class does not " +
                    "implement interface ITest." );
            }

            /* ------------------------------------------------------------ *
             * Do it again...This time "Test" will come from the cache.
             * ------------------------------------------------------------ */
            classRef = loader.loadClass( "Test" );
            objectRef = classRef.newInstance();

            if( objectRef instanceof ITest )
            {
                testClass = (ITest) objectRef;
                testClass.test();
            }
            else
            {
                System.out.println( "ERROR:  Loaded class does not " +
                    "implement interface ITest." );
            }
        }
        catch( Exception e )
        {
            System.out.println( "Exception:  " + e.getMessage() );
        }
    }
}
```

Organizing and Compiling the Code

The code should be organized in a structure similar to the one shown below.

```
FileClassLoader
|- FileClassLoader.java
|- MainApp.java
|- ITest.java
|- Classes (sub-directory)
|- ITest.java
|- Test.java
```

From the Classes subdirectory, run javac Test.java. This will create two class files, ITest.class and Test.class.

From the FileClassLoader directory, run javac MainApp.java. This will create FileClassLoader.class, MainApp.class, and ITest.class.

Running the Application

Type java MainApp to run the application.

Running the application produces the following results on my machine. The numbers to the left were inserted to make the explanation below easier to follow.

```
C:\Campbell\FileClassLoader
     >java MainApp
1 Class:  Test
   Loading...
     Path:  classes\Test.class
2 Class:  java.lang.Object
     System class.
3 Class:  ITest
     System class.
4 Class:  java.lang.System
     System class.
5 Class:  java.io.PrintStream
     System class.
  * ------------------- *
  * Test method called! *
  * ------------------- *
6 Class:  Test
     Found in cache.
  * ------------------- *
  * Test method called! *
  * ------------------- *
```

Look carefully at the output above. From our MainApp driver class, we called loadClass twice. Yet the output above shows that loadClass was called a total of six times! Where did the four extra calls come from? The answer has to do with the call to resolveClass in the loadClass method.

Recall that resolveClass causes the Virtual Machine to load any classes the newly loaded class depends upon by making subsequent calls to loadClass. All dependant classes must be loaded before the new class can be used. Give that this is the case let's look back at the output and piece together what's happening. The numbers in the output and the list below coincide with the calls to loadClass.

1. The first call to loadClass was made by us from our MainApp driver class. The Test class is loaded from the .class file in the classes subdirectory.
2. All classes in Java are derived from Object, and class Test is no exception. The Virtual Machine makes the second call to loadClass to load Object, the base class for Test.
3. Class Test implements the ITest interface. The Virtual Machine makes the third call to loadClass to load ITest.
4. Calls four and five are a result of the print statements embedded in the code for debugging and informational purposes. Class Test makes calls to System.out.println. In call four to loadClass, the Virtual Machine is loading the System class.
5. Here the Virtual Machine is loading the PrintStream class for writing text to the screen.

6. The final call to loadClass was made by us from our MainApp driver class. Notice that no loading took place since Test was found in the cache.

Conclusion

We've learned how to implement a custom class loader and put it to use. We saw that even though classes can be loaded dynamically, we still have to know something about the interface they implement. And finally, the output of the driver application gave us insight as to what the Virtual Machine goes through to load a new class. Hopefully this article has provided you with insight into the inner workings of the Virtual Machine and given you a starting point for creating more dynamic Java applications.

Developed Under:
JDK 1.2