white paper

DEVELOPMENTOR℠

Understanding *Class.forName*

Loading Classes Dynamically from within Extensions

Abstract

Dynamic loading of Java classes at runtime provides tremendous flexibility in the development of enterprise systems. It provides for the basis of “application servers”, and allows even simpler, lighter-weight systems to accomplish some of the same ends. Within Java, dynamic loading is typically achieved by calling the *forName* method on the class java.lang.Class; however, when *Class.forName* is naïvely called from within an Extension, strange errors can occur. This paper describes why those errors occur, and how Java2 provides two facilities: one a modification to the *forName* syntax, and the other, called the “Thread context ClassLoader”, to avoid them.

This paper assumes you are passingly familiar with Java Reflection, and have at least a general idea of what Java ClassLoaders are and how they are used within Java code.

KEYWORDS: Java ClassLoaders Extensions Threads

Problem

Dynamic runtime loading. One of Java's strongest features is its ability to dynamically load code given the name of the class to load, without having to know the actual classname until runtime. This allows Java developers to build flexible, dynamic systems that can grow and change without requiring complete recompilation. For example, the following code executes the *main* method of a class given on the command-line to the class DynamicLoader:

```
public class DynamicLoader
{
    public static void main(String[] args)
        throws Exception
    {
        Class toRun = Class.forName(args[0]);

        String[] newArgs = scrubArgs(args);

        Method mainMethod = findMain(toRun);
        mainMethod.invoke(null, new Object[] { newArgs });
    }
    private static String[] scrubArgs(String[] args)
    {
        String[] toReturn = new String[args.length-1];
        for (int i=1; i<args.length; i++)
        {
            toReturn[i-1] = args[i].toLowerCase();
        }
        return toReturn;
    }
    private static Method findMain(Class clazz)
        throws Exception
    {
        Method[] methods = clazz.getMethods();

        for (int i=0; i<methods.length; i++)
        {
            if (methods[i].getName().equals("main"))
                return methods[i];
        }

        return null;
    }
}
```

As you can see, DynamicLoader takes the arguments (except the first one) given to it, and passes them directly on to the *main* method of the loaded class, after first making all the command-line arguments lower case. This means we can run any compiled Java class through DynamicLoader without having to change DynamicLoader's code in the slightest. In fact, we can run any executable Java code, from Swing-based client apps to console-based Web servers—DynamicLoader really doesn't care. For example, when we run the following class:

```
public class Echo
{
    public static void main (String args[])
    {
        for (int i=0; i<args.length; i++)
        {
            System.out.println("Echo arg"+i+" = "+args[i]);
        }
    }
}
```

through DynamicLoader, like this:

```
> java DynamicLoader Echo ONE TWO THREE
```

we get the following output:

```
Echo arg0 = one
Echo arg1 = two
Echo arg2 = three
```

As you can see, DynamicLoader created an instance of the Echo class, and called its *main* method, all without having any direct reference to Echo itself. In O-O parlance, this means DynamicLoader is completely *decoupled* from Echo; there are no explicit dependencies between these two classes.

This use of dynamic runtime loading is the heart of Java Application Servers like the Java2 Enterprise Edition reference implementation, Enterprise JavaBeans, and the Servlet Specification. In each one of these architectures, at the time the application server is compiled, it knows nothing about the code that will be attached to it. Instead, it simply asks the user for a classname to load, loads the class, creates an instance of the class, and starts making method calls on that instance. (It does so either through Reflection, or by requiring that clients implement a particular interface or class, like GenericServlet in the Servlet spec, or EJBObject in the EJB spec.)

Use of dynamic runtime loading isn't restricted solely to server-side architectures, however. Web browsers must use it in order to be able to load applets specified by name by a remote web page, as well. RMI uses it to dynamically load stubs and/or skeletons for remote objects, and JNDI and JDBC will use it to load, at runtime, the exact driver necessary (given by the user's appropriate ".properties" file) to perform the work asked. In each case, a ClassLoader is asked to load a class by name that may not have existed within the JVM before this moment.

Enterprise application architects are also starting to discover ways to put dynamic runtime loading to work for them, as well. For example, several papers and articles have already been published describing how to use XML to create GUI dialogs and/or windows [6], including the ability to instantiate custom classes that didn't exist when the basic system was built.

Class.forName. In most of these systems, the code to do the runtime loading comes through the method *forName* on the class java.lang.Class; its use is demonstrated in the previous DynamicLoader code. *Class.forName* attempts to load the class whose name is given in its only argument, and returns the Class instance representing that class. In the event that the Class could not be found, resolved, verified, or loaded, *Class.forName* throws one of several different Exceptions, all of which are listed in the javadoc page for java.lang.Class [1].

Assume for the moment that this DynamicLoader class we've created is a useful bit of functionality, and several other developers have expressed an interest in making use of it. In accordance with the new Java2 emphasis on Extensions, we package DynamicLoader up into a .jar file, and drop it into the Extensions directory of our Java execution environment[1]. (If you're not familiar with Java Extensions, see [1] or [5] for details.)

The problem comes, however, when we try to execute the Echo class found on the CLASSPATH from within DynamicLoader[2]:

```
> java DynamicLoader Echo ONE TWO THREE
Exception in thread "main" java.lang.ClassNotFoundException: Echo
        at java.net.URLClassLoader$1.run(URLClassLoader.java:202)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:191)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:280)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:237)
        at java.lang.Class.forName0(Native Method)
        at java.lang.Class.forName(Class.java:124)
        at DynamicLoader.main(DynamicLoader.java:8)
```

[1] Either <JDK>/jre/lib/ext or the <JRE>/lib/ext directories, depending on whether the executing version of *java* is running out of the <JDK>/bin directory or <JRE>/bin.
[2] This assumes you've jar'ed DynamicLoader.class into a .jar and placed the .jar in the <JAVA_HOME>/jre/lib/ext directory; if you haven't, do that now.

We can verify that the class, Echo, does in fact exist, either by running Echo directly, or by running the *javap* utility to list its attributes:

```
> javap Echo
Compiled from Echo.java
public class Echo extends java.lang.Object {
    public Echo();
    public static void main(java.lang.String[]);
}
```

So what gives? Why is DynamicLoader failing to load Echo? What changed between DynamicLoader-on-the-CLASSPATH and DynamicLoader-as-an-Extension?

Problem Analysis

In order to understand precisely what's going on here, you have to understand some of the basics of the Java ClassLoader model and what changed in Java2 from JDK 1.1.

Java2 ClassLoader Delegation Model. As described in [2], the Java2 ClassLoader model is a "delegating parent" model, meaning that when a ClassLoader is asked to load a class, it first delegates the opportunity to load the requested class to its parent ClassLoader. Only if the parent (a) hasn't already loaded the Class, and (b) can't load the Class does the child ClassLoader get the opportunity to provide the Class requested, if it can. This means that ClassLoaders form a hierarchical tree, with the "bootstrap" ClassLoader (the only ClassLoader in the JVM with a "value" of *null*) as the root of the tree. (More on the bootstrap ClassLoader later.) Any ClassLoader constructed within user code[3] must have a parent ClassLoader; if none is provided in the ClassLoader constructor call, the ClassLoader constructors assume this ClassLoader's parent to be the "system" or ClassLoader. (More on the "system ClassLoader" later.)

Implicitly associated with each loaded Class in a JVM is the ClassLoader that loaded it. It is this ClassLoader that is returned from the Class method *getClassLoader*. Each Class is associated with one-and-only-one ClassLoader, and this is not mutable—Class objects can't change the ClassLoader reference to point to another ClassLoader.

A ClassLoader's *loadClass* method is called (either directly by the programmer or implicitly by the JVM) when a client wishes to load a class through that ClassLoader. Under the JDK 1.0 and 1.1 models, the ClassLoader directly overrode *loadClass* to undertake the necessary actions to load the code. Under JDK 1.2, however, this is no longer the case; instead, the *loadClass* method calls its ClassLoader parent/delegate's *loadClass* to see if the parent recognizes the class. Only if the parent (and, by extensions, all of its parental ancestors, as well) fails to load the class does this ClassLoader get the chance to load the code. To do that, it calls the *findClass* method, which is the method customized ClassLoaders are now supposed to override.

[3] That is, non-JVM code.

As an example of how this works, consider a JVM embedded inside of a Web browser. When a Web page comes in with an <APPLET …> tag on it, the Web browser constructs an instance of its (vendor-specific) AppletClassLoader. This ClassLoader is responsible for obtaining the compiled bytecode from the Website that served the page in the first place; the Web browser simply does something similar to the following:

```
// inside the Web browser somewhere...
AppletClassLoader cl = new AppletClassLoader(appletCodebase);
Applet applet = (Applet)cl.loadClass(appletClassName).newInstance();
```

where *appletClassName* is given within the <APPLET …> HTML tag. AppletClassLoader, in its *loadClass* method, first checks with its parent, which under most JVMs will be the primordial ClassLoader. Let's track how this applet's Class gets loaded.

Because the system ClassLoader is the parent to the AppletClassLoader, the system ClassLoader is given the opportunity to find the Class. Because the Class code can't be found along the CLASSPATH, in any of the Extensions directories, or in the core library classes (it's an applet, remember the code resides up on the server), the system ClassLoader signals failure. This means it's now AppletClassLoader's turn, and it examines the <APPLET …> tag for any information, like CODEBASE attributes, on the whereabouts of the Class file requested. If AppletClassLoader finds a compiled .jar or .class file, it downloads the code as a binary file and loads it into the JVM (via the *defineClass* method on ClassLoader) from there. The Web browser can now construct an instance of the Applet class, because now it has the requested Class file loaded into its JVM: It can now call the *start*, *stop*, and other methods on Applet.

This parent-child relationship of Java2 now releases Java ClassLoader developers from having to remember to check the system ClassLoader to see if the class has already been loaded. More importantly, however, it permits a "chain-loading" of classes from a variety of sources.

For example, in a hypothetical enterprise system, I might create a JDBCClassLoader, whose parent is a URLClassLoader pointing to an HTTP server inside my corporate firewall. That URLClassLoader in turn is a child of the primordial ClassLoader. This provides me a three-layered, code-loading facility, where rapidly-changing code, such as business rules (sales promotionals and the like) can be stored in the RDBMS, potentially on a user-by-user basis, if necessary. The less-mutable code is stored on my HTTP server, and the system and bootstrap ClassLoaders are responsible for the core Java classes, as usual. This approach allows the more stable code to "override" more mutable behavior; if you desire the opposite, simply reverse the placement of the JDBCClassLoader and the URLCLassLoader. In each case, because the nature of the ClassLoading mechanism is "parent-first", whichever ClassLoader is highest up on the hierarchy has "classloading priority" over its children.

Because each child delegates to its parent, we will load code in top-down order; in the fictitious aforementioned example, the system ClassLoader gets the first chance, followed by the URLClassLoader, followed by the JDBCClassLoader. Understanding how the system ClassLoader is architected, along with this understanding of the "parent-first" nature of delegating ClassLoaders, yields discovery of the problem.

Java2/JDK1.2 Default ClassLoader architecture. The Java2 system provides for loading code from one of three places by default: the core library, the Extensions directory (or directories, if you modify the *java.ext.dirs* property to include multiple subdirectories), and from the directories and/or .jar/.zip files found along the *java.class.path* property, which in turn comes from the CLASSPATH environment variable. Each of these three locations is in turn covered by its own ClassLoader instance: the core classes, by the *bootstrap* ClassLoader, the Extensions directory/directories by the *extension* ClassLoader, and the CLASSPATH by the *system* or *application* ClassLoader[4].

Before we delve too deeply into this, let's run a quick test. Compiling and executing this code:

```
public class EchoClassLoader
{
    public static void main (String args[])
    {
        ClassLoader current =
            new EchoClassLoader().getClass().getClassLoader();

        while (current != null)
        {
            System.out.println(current.getClass());
            current = current.getParent();
        }
    }
}
```

produces this rather interesting result:

```
> java EchoClassLoader
class sun.misc.Launcher$AppClassLoader
class sun.misc.Launcher$ExtClassLoader
```

The result highlights something *very* interesting. I've been speaking of the system ClassLoader as a singular entity, but there are *two* parents to our class' ClassLoader: one called Launcher$AppClassLoader, and one called Launcher$ExtClassLoader, both of which live in the sun.misc package.

The *bootstrap* ClassLoader is implemented as part of the VM itself, and cannot be instantiated by Java code. It is this ClassLoader that brings the core Java classes into the VM, allowing the rest of the JVM to load itself. (Take a second to think about this—in order to construct a ClassLoader, which extends Object, the JVM must load the Object class. But in order to do this, it requires a ClassLoader with which to load it!) Normally, this means loading code from the "rt.jar" file in the jdk/jre/lib subdirectory, but under the Sun JVM, the *boot.class.path* property actually controls it[5].

[4] At one point, during the JDK 1.2 Beta3 release, it was also called the *base* ClassLoader.
[5] Why this could be important is beyond the scope of this paper.

The *extension* ClassLoader, the first child of the boostrap ClassLoader, is implemented in pure Java code. As already mentioned, the extension ClassLoader's primary responsibility is to load code from the JDK's *extension directories*. This in turn provides users of Java the ability to simply "drop in" new code extensions (hence the name "Extension directory"), such as JNDI or JSDT, without requiring modification to the user's CLASSPATH environment variable.

The *system*, or *application*, ClassLoader is the ClassLoader returned from the static method *ClassLoader.getSystemClassLoader*. This is the ClassLoader responsible for loading code from the CLASSPATH, and by default will be the parent to any user-created or user-defined ClassLoader in the system.

(As an aside, both of these classes are direct descendants of URLClassLoader. ExtClassLoader reads the *java.ext.dirs* property for a list of directories. It takes each directory in turn, iterates over each file in the directory, builds a URL out of it, and stores that URL in the URLClassLoader-parent portion of itself. ExtClassLoader also overrides the *findLibrary* method to search the directory given by the property *os.arch* under each extension directory for the library matching the requested name. This allows extension directories to also silently support native-libraries in the same "drop-in" manner as Extensions themselves. AppClassLoader reads the *java.class.path* property, and for each *File.pathSeparator*-separated entry, builds a URL out of it. Once these two classes are created, however, they behave in all respects like a standard URLClassLoader. These two classes, along with URLClassLoader, are described more fully in [5].)

So now we have some deeper insight into the architecture of the ClassLoader system in Java2. Let's take a moment and see, step-by-step, why *Class.forName*, when called on a class found within the CLASSPATH, fails when called from within an Extension.

To summarize the "normal" ClassLoader tree:

<u>*Bootstrap ClassLoader*</u>
Type: none; implemented within the VM
Parent: none
Loads core runtime library code from *sun.boot.class.path*

<u>*Extension ClassLoader*</u>
Type: *sun.misc.Launcher$ExtClassLoader*
Parent: *null* (bootstrap ClassLoader)
Loads code found in extension directories (given by *java.ext.dirs*)

<u>*System (or Application) ClassLoader*</u>
Type: *sun.misc.Launcher$AppClassLoader*
Parent: Extension ClassLoader
Returned from *ClassLoader.getSystemClassLoader*
Loads code found on *java.class.path* (the CLASSPATH variable)

Calling *Class.forName* within an Extension. Go back to the original problem: this code:

```
public class DynamicLoader
{
    public static void main(String[] args)
        throws Exception
    {
        Class toRun = Class.forName(args[0]);

        String[] newArgs = scrubArgs(args);

        Method mainMethod = findMain(toRun);

        mainMethod.invoke(null, new Object[] { newArgs });
    }
    // . . .
}
```

when called from within an Extension, fails with:

```
> java DynamicLoader Echo ONE TWO THREE
Exception in thread "main" java.lang.ClassNotFoundException: Echo
        at java.net.URLClassLoader$1.run(URLClassLoader.java:202)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:191)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:280)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:237)
        at java.lang.Class.forName0(Native Method)
        at java.lang.Class.forName(Class.java:124)
        at DynamicLoader.main(DynamicLoader.java:8)
```

Walking through the *Class.forName* call highlights the problem. *Class.forName* obtains the caller's ClassLoader, and uses that ClassLoader instance to load the Class into the VM. Written in pseudoc ode, *Class.forName* looks like:

```
public Class forName(String classToLoad)
    throws . . .
{
    ClassLoader cl = ClassLoader.getCallerClassLoader();
    return cl.loadClass(classToLoad);
}
```

It's fairly easy, now, to spot what the problem is. When a Class is loaded from the CLASSPATH, its associated ClassLoader is the AppClassLoader instance. When that Class invokes *Class.forName*, it can find other classes along the CLASSPATH or in the Extensions directory because AppClassLoader first delegates to its parent, the ExtClassLoader. Only if the requested class is not found as an Extension is the CLASSPATH searched.

When the calling Class is loaded as an Extension, however, its associated ClassLoader is the ExtClassLoader. Thus, when the Extension-loaded class calls *Class.forName*,

1. The caller's ClassLoader is obtained; in this case, that ClassLoader is the ExtClassLoader instance.

2. The ExtClassLoader, being a good child, gives its parent, the bootstrap ClassLoader, the opportunity to load the class.

3. Because the bootstrap ClassLoader fails to find it in jdk/jre/lib/rt.jar, it returns without a loaded Class.

4. ExtClassLoader's *loadClass* method then attempts to load the Class from within its list of URLs (given, as described earlier, in the *java.ext.dirs* property). Because the class is not within the list of URLs associated with ExtClassLoader, the load fails, and a ClassNotFoundException is thrown.

Put simply, because Extension-loaded classes don't have the AppClassLoader instance anywhere on their parent-ClassLoader chain, they can't load any classes off of the CLASSPATH.

Solution

Fundamentally, there are two solutions available to you: use a different form of the *forName* method on Class, or use a ClassLoader associated with each specific Thread, called the Thread's *context ClassLoader*. As with everything, each approach has its own consequences.

Use Thread.getContextClassLoader. A new concept introduced in Java2, each Thread now has a *context ClassLoader* associated with it. Looking at the javadocs[1] for the Thread methods *getContextClassLoader* and *setContextClassLoader* reveals that

> *Thread.getContextClassLoader:* Returns the context ClassLoader for this Thread. The context ClassLoader is provided by the creator of the thread for use by code running in this thread when loading classes and resources. If not set, the default is the ClassLoader context of the parent Thread. The context ClassLoader of the primordial thread is typically set to the class loader used to load the application.
>
> *Thread.setContextClassLoader:* Sets the context ClassLoader for this Thread. The context ClassLoader can be set when a thread is created, and allows the creator of the thread to provide the appropriate class loader to code running in the thread when loading classes and resources.

The Launcher constructor, after constructing both the ExtClassLoader and AppClassLoader, sets the thread context ClassLoader to be the AppClassLoader instance. This means, unless your code (or other code loaded by your application) changes it, the thread's context ClassLoader is the AppClassLoader instance. This also means that now the AppClassLoader instance is available to Extension classes:

```
import java.lang.reflect.*;

public class FixedDynamicLoader
{
    public static void main(String[] args)
        throws Exception
    {
        // Instead of
        // Class toRun = Class.forName(args[0]);
        // use:
        Thread t = Thread.currentThread();
        ClassLoader cl = t.getContextClassLoader();
        Class toRun = cl.loadClass(args[0]);

        String[] newArgs = scrubArgs(args);

        Method mainMethod = findMain(toRun);

        mainMethod.invoke(null, new Object[] { newArgs });
    }
    private static String[] scrubArgs(String[] args)
    {
        String[] toReturn = new String[args.length-1];
        for (int i=1; i<args.length; i++)
        {
            toReturn[i-1] = args[i].toLowerCase();
        }
        return toReturn;
    }
    private static Method findMain(Class clazz)
        throws Exception
    {
        Method[] methods = clazz.getMethods();

        for (int i=0; i<methods.length; i++)
        {
            if (methods[i].getName().equals("main"))
                return methods[i];
        }

        return null;
    }
}
```

After compiling, placing into a .jar and dropping that .jar into the Extensions directory, we achieve success:

```
> java FixedDynamicLoader Echo ONE TWO THREE
Echo arg0 = one
Echo arg1 = two
Echo arg2 = three
```

So the short answer is to use the ClassLoader associated with the current Thread to call *loadClass* to load the class desired, instead of using *Class.forName*.

Those readers familiar with multi-threaded environments may be wondering about Threads created by user-code—what ClassLoader is set to be the context ClassLoader for user-created Threads? A Thread, when created, inherits the context ClassLoader of the Thread that created it, so all Threads, unless modified, will have AppClassLoader set as their context ClassLoader.

Use the 3-arg form of *Class.forName*. Sun introduced a new form of the *forName* method that takes 3 arguments, instead of just one. In addition to the name of the Class to load, callers pass a boolean parameter indicating whether to initialize the loaded Class or not, and a ClassLoader parameter to load the code through. This method performs the same steps as the 1-arg *Class.forName*; in fact, the 1-arg version of Class.forName calls directly into the 3-arg version, passing "true" and the caller's ClassLoader as the second and third parameters, respectively.

Class.forName(classname, init_flag, classloader) will load the code "through" the ClassLoader passed to it; crudely put, it functions somewhat similar to the following:

```
Class forName(String classnm, boolean init, ClassLoader loader)
    throws . . .
{
    // . . .

    loader.loadClass(classnm);

    // . . .
}
```

This is a gross oversimplification, but the point is clear—instead of using the *caller's* ClassLoader to do the load, it uses the ClassLoader instance passed in. This now means, to avoid the problem that started this paper, FixedDynamicLoader could also do:

```
public class FixedDynamicLoader
{
    public static void main(String[] args)
        throws Exception
    {
        //Class toRun = Class.forName(args[0]);
        Class toRun =
            Class.forName(args[0],
                          true,
                          ClassLoader.getSystemClassLoader());

        String[] newArgs = scrubArgs(args);

        Method mainMethod = findMain(toRun);

        mainMethod.invoke(null, new Object[] { newArgs });
    }
    // . . .
}
```

This code has the advantage of being perhaps a bit more explicit about what the intent of the code is, as opposed to the slightly more obscure use of the Thread's context ClassLoader. Other (technical) differences between the two are more profound, however.

Class.forName vs *ClassLoader.loadClass*. There are some subtle differences between these two APIs. The method call **CL.loadClass(C)**, where CL is our ClassLoader and C is the name of the class to load, queries the ClassLoader directly for the class by name. This in turn relies on ClassLoader delegation to ask the bootstrap ClassLoader to determine if the class has already been loaded. Conversely, the method call **Class.forName(C, false, CL)** uses the VM's internal class resolution mechanism to do the actual loading. Among other things, this allows *Class.forName* to support the loading of *arrays of Classes* as a type; for example, calling *CL.loadClass("[C;")* will result in failure, where doing the same with *Class.forName("[C;", false, CL)* will succeed.

Consequences

As with any discussion, there are consequences to each of the solutions set forth within this white paper.

? Versioning: The 3-arg *Class.forName*, *Thread.getContextClassLoader*, and *Thread.setContextClassLoader* are Java2/JDK1.2 APIs. This means any Java API or class library that wants to remain compatible or usable with JDK 1.1 will not be able to be loaded. Remember, the VM, during its resolution step of classloading (see [4]), will verify that all methods referenced inside of a class actually exist, and will throw an Exception if this is not the case. How, then can code avoid this ClassLoader trap in Java2 code, while remaining compatible with 1.1 code?

One solution is to eliminate your JDK 1.1 compatibility requirement. It sounds a bit draconian, but Java2/JDK 1.2 has been out for approximately a year at the time of this writing, Java2/JDK 1.3 is in the final beta stages. More to the point, almost no new development is being done in JDK 1.1 (except for applets, since 1.1 is as far as most browsers have gone in JDK support).

However, for many APIs and/or design teams, this is an unacceptable solution. Too many JDK 1.1-based systems exist to simply write off JDK 1.1 environments entirely, and developing a system that fails to work properly in a JDK 1.2 / Java2 environment is simply unthinkable. Fortunately, a convenient middle ground is possible.

Because Java uses a lazy dynamic-loading system (see [2], [4], or [5] for details), classes aren't loaded into the VM until the last possible moment. This means it's possible to use a Strategy-pattern approach towards classloading, based on the version of the VM your code is executing within:

```
package com.javageeks.lang.classloader;

interface VMClassLoader
{
    public Class loadClass(String cls)
        throws ClassNotFoundException;
}

public class ClassLoaderHelper
```

```
{
    private static VMClassLoader vmClassLoader;
    static
    {
        String vmVersion = System.getProperty("java.version");
        if (vmVersion.startsWith("1.2"))
        {
            //System.out.println("Loading 1.2 VMClassLoader");
            vmClassLoader = new VMClassLoader()
            {
                public Class loadClass(String cls)
                    throws ClassNotFoundException
                {
                    Thread t = Thread.currentThread();
                    ClassLoader cl = t.getContextClassLoader();
                    return cl.loadClass(cls);
                }
            };
        }
        else if (vmVersion.startsWith("1.1") ||
                 vmVersion.startsWith("1.0"))
        {
            //System.out.println("Loading 1.1/1.0 VMClassLoader");
            vmClassLoader = new VMClassLoader()
            {
                public Class loadClass(String cls)
                    throws ClassNotFoundException
                {
                    return Class.forName(cls);
                }
            };
        }
        else
        {
            // ???
        }
    }

    public static Class loadClass(String cls)
        throws ClassNotFoundException
    {
        return vmClassLoader.loadClass(cls);
    }


    /**
     * Test driver.
     */
    public static void main(String[] args)
        throws Exception
    {
        //Class toRun = Class.forName(args[0]);
        Class toRun =
            ClassLoaderHelper.loadClass(args[0]);

        String[] newArgs = scrubArgs(args);

        java.lang.reflect.Method mainMethod =
            findMain(toRun);

        mainMethod.invoke(null, new Object[] { newArgs });
        }
        private static String[] scrubArgs(String[] args)
        {
            String[] toReturn = new String[args.length-1];
            for (int i=1; i<args.length; i++)
            {
                toReturn[i-1] = args[i].toLowerCase();
            }
            return toReturn;
        }
        private static java.lang.reflect.Method findMain(Class clazz)
            throws Exception
        {
            java.lang.reflect.Method[] methods = clazz.getMethods();

            for (int i=0; i<methods.length; i++)
            {
                if (methods[i].getName().equals("main"))
                    return methods[i];
            }

            return null;
        }
    }
```

The key here is in the static block of the ClassLoaderHelper class. When loaded in a 1.2 VM, an anonymous instance of VMClassLoader is created, which uses the Thread contextClassLoader methods to obtain the ClassLoader and call *loadClass*. When loaded into a 1.1 (or 1.0, although this is, as of this writing, untested) VM, the anonymous VMClassLoader instance falls back on *Class.forName*, since 1.1 VMs don't have the problem described in this paper.

Running this code within a 1.2 VM yields the following:

```
    > java -version
    java version "1.2"
    Classic VM (build JDK-1.2-V, native threads)

    > java ClassLoaderHelper Echo ONE TWO THREE
    Loading 1.2 VMClassLoader
    Echo arg0 = one
    Echo arg1 = two
    Echo arg2 = three
```

Running this code within a 1.1 VM yields the following:

```
    > java -version
    java version "1.1.7"

    > java ClassLoaderHelper Echo ONE TWO THREE
    Loading 1.1/1.0 VMClassLoader
    Echo arg0 = one
    Echo arg1 = two
    Echo arg2 = three
```

By using *ClassLoaderHelper.loadClass* instead of *Class.forName*, code can continue to support both JDK 1.1 and JDK 1.2 (and beyond) VMs without having to maintain two (or more) separate codebases.

(ClassLoaderHelper could also be modified to use the 3-arg version of *Class.forName* instead of the Thread's context ClassLoader. However, it would require a ClassLoader instance to be passed in, to avoid making the assumption that it should use the system ClassLoader.)

? Using *Thread.getContextClassLoader* relies on the Thread's context ClassLoader to be appropriately set. What if the Thread's current context ClassLoader *isn't* the one expected, but is instead set by an arbitrary third-party package (like RMI or JNDI)?

In fact, there's not much you *can* do to prevent this. Because *Thread.setContextClassLoader* makes a Security check, however, you can take some small comfort in the fact that only those systems that have "setContextClassLoader" permission will be allowed to modify a Thread's context ClassLoader.

Practically speaking, this means in an enterprise system, you can modify your policy file such that only your codebase is permitted to modify the Thread's context ClassLoader. This doesn't prevent Sun-sponsored APIs (like RMI, which is "part" of the runtime library, or JNDI, which is part of the runtime library starting in JDK 1.3) from being able to modify it, but at least you can prevent rogue third-parties from doing so.

? Knowing your loading ClassLoader. In order to use the 3-arg form of *Class.forName*, callers must have the ClassLoader instance they want to load through available to them in order to pass it; under certain circumstances, simply calling *ClassLoader.getSystemClassLoader()* here will be wildly inappropriate. For example, a Servlet will usually be loaded under its own ClassLoader, and if the Servlet calls *Class.forName* with the system ClassLoader as the third argument, code loaded under the Servlet's ClassLoader won't be found. This situation can arise more commonly than one might expect—many Servlets are bundled together with supporting code directly in the "servlets" directory, placing the supporting code under the Servlet ClassLoader's domain.

Under those circumstances, using the Thread's context ClassLoader can be the only appropriate parameter; you can either pass it as the third parameter to the 3-arg form of *Class.forName*, or else call its *loadClass* method directly. But, as pointed out in the aforementioned point, using it relies on the context ClassLoader being set correctly by the code that loaded your class.

It may seem that callers will always know the ClassLoader they should call through. However, keep in mind that the ExtClassLoader/AppClassLoader pair is not the only situation in which ClassLoaders further "up the tree" are going to look to load code further "down" the ClassLoader tree. For example, sophisticated security mechanisms within a Java application might make use of an instance of URLClassLoader to load code on a per-user-role basis; certain parts of the application, however, will be constant, and therefore installed as an Extension. Those constant classes cannot simply assume that the ClassLoader that loaded *them* will be the correct ClassLoader to pass into *Class.forName* (it won't—we're now right back to the original problem), nor will simply passing *ClassLoader.getSystemClassLoader()* be correct, either. Instead, they will have to trust that somebody will set the current Thread's context ClassLoader to be that custom URLClassLoader instance, and use that to dynamically load the code.

Summary

Java2, with the release of JDK 1.2, subtly changed the nature of classloading. When Sun added the "Extens ion" capability to the language/environment, they split the responsibility for loading Extensions code and "Application" (that is, CLASSPATH-based) code into two separate ClassLoaders. Under "normal" circumstances, this change will be invisible to most Java developers, but those working under dynamic-loading systems need to be aware of this change and what it means for them.

Many Java developers may believe that the circumstances described in this paper won't apply to them. "I'm not making use of any of this", they mutter to themselves. "Why do I care?" It's a more relevant concern than readers might wish. Several key Java technologies, most notably RMI and JNDI, make use of the Thread context ClassLoaders for precisely this reason—the core classes for both technologies are loaded high up in the CllassLoader tree, but need to be able to load code stored along the CLASSPATH. Worse yet, as more enterprise systems are built under EJB servers, which may well be installed as Extensions, this problem could become more and more common.

As Java2 Security becomes more and more well-understood and implemented in Java projects, this issue will again rear its ugly head, since developers will slowly begin to adopt codebase permissions on a per-ClassLoader basis. Multiple ClassLoaders means the potential for code further *up* the chain looking for code further *down* the chain, and we're right back to where we started with all this.

Fortunately, as JDK 1.1 systems slowly phase out, it should become more and more comfortable to use *Thread.currentThread().getContextClassLoader().loadClass()* or the 3-arg version of *Class.forName()* directly; until that time; however, the ClassLoaderHelper class should provide a measure of portability across JVM versions for doing dynamic classloading from within an Extension.

Bibliography/Additional Reading

[1] JDK 1.2 documentation bundle. See http://www.javasoft.com for downloading.

[2] << Liang/Bracha OOPSLA paper >>

[3] *Java Language Specification*.

[4] *Java Virtual Machine Specification, 2nd Edition*.

[5] *Server-Side Java*, by Ted Neward. Manning Publishing, Feb 2000.

[6] Personal communication with Peter Jones of Sun Microsystems; used with permission.

Copyright