A JavaGeeks.com
White Paper

Ted Neward
http://www.javageeks.com/~tneward
tneward@javageeks.com
16March2001

Using the BootClasspath

Tweaking the Java Runtime API

Abstract

Many Java programmers don't realize it, but the Java Runtime Environment [] is an amazingly configurable environment-so much about the Java execution environment can be controlled via options either on the command-line or through the JNI Invocation interface. One such option is the ability to define the location of the Java "bootstrap" classes-java.lang.Object, java.lang.Exception, and so forth-to come from someplace other than the ubiquitous "rt.jar" file in the "jre/lib" directory. In fact, we can use this non-standard JVM option to subvert the Java environment in many powerful ways, giving Java programmers a tremendous amount of power over their environment. But with power comes complexity, and this is no exception: it's powerful, but only if you're willing to accept the risks that go along with it.

Problem Discussion, Part I (Statics)

Under many circumstances, what the Java environment does by default may not be exactly what you want. For example, by default, the Sun JDK VM starts with 1 MB of heap space allocated, and will continue to grow until it consumes a maximum of 64 MB. Or, as an alternate example, the VM will create a ClassLoader delegation chain as described in [sbpj] [classForName] [insideVM] , where classes to be loaded are first loaded from the Runtime API jar file (rt.jar), then the Extensions directory, then the CLASSPATH. This behavior, such as it is, may not be precisely what you desire as a Java implementor or deployer, and the Sun JVM has the flexibility to accommodate you[1].

Within the Sun VM, for example, it is possible to change the initial heap size and maximum heap size through the non-standard "-Xms" and "-Xmx" options, which set the initial and maximum heap sizes, respectively. The Sun VM documents all of the "-X" options as nonstandard options, ones that other VMs are not constrained to support. Running the "java -X" command with no target classname on the JDK 1.2.2 at the command line reveals

```
C:\Test>java -X
    -Xbootclasspath:<directories and zip/jar files separated by ;>
                      set search path for bootstrap classes and resources
    -Xnoclassgc       disable class garbage collection
    -Xms<size>        set initial Java heap size
    -Xmx<size>        set maximum Java heap size
    -Xrs              reduce the use of OS signals
    -Xcheck:jni       perform additional checks for JNI functions
    -Xrunhprof[:help]|[:<option>=<value>, ...]
                      perform heap, cpu, or monitor profiling
    -Xdebug           enable remote debugging
    -Xfuture          enable strictest checks, anticipating future default
The -X options are non-standard and subject to change without notice.
C:\Test>
```

Doing so for the the JDK 1.3.0 VM reveals a similar message:

```
C:\Test>java -X
    -Xmixed           mixed mode execution (default)
    -Xint             interpreted mode execution only
    -Xbootclasspath:<directories and zip/jar files separated by ;>
                      set search path for bootstrap classes and resources
    -Xbootclasspath/a:<directories and zip/jar files separated by ;>
                      append to end of bootstrap class path
    -Xbootclasspath/p:<directories and zip/jar files separated by ;>
                      prepend in front of bootstrap class path
    -Xnoclassgc       disable class garbage collection
    -Xincgc           enable incremental garbage collection
```

```
    -Xms<size>        set initial Java heap size
    -Xmx<size>        set maximum Java heap size
    -Xprof            output cpu profiling data
    -Xrunhprof[:help]|[:<option>=<value>, ...]
                      perform JVMPI heap, cpu, or monitor profiling
    -Xdebug           enable remote debugging
The -X options are non-standard and subject to change without notice.
C:\Test>
```

One of the options supported by both VMs is the "-Xbootclasspath" option. Before it's possible to describe how this option benefits Java programmers, it's necessary to revisit the standard ClassLoader delegation model.

The ClassLoader Delegation Tree, Revisited

As described in [sbpj] and [insideVM] , when the VM first starts up, it begins entirely devoid of any Java classes whatsoever. In fact, if no JNI call to FindClass ever occurs, the VM will in fact never load any Java classes-it uses a lazy-evaluating loader mechanism that only loads classes when demanded to. When the "java.exe" process is given a command-line argument, the native code that makes up the "java.exe" codebase calls FindClass to attempt to load that class into the VM, and execute its main() method, if one is found. As part of loading the class, any dependent classes (base class, implemented interfaces, classes used as part of the implementation of this class, such as String, Exception, or any of the Collections classes (java.util.Vector, for example) must also be loaded.

We have a problem, however. ClassLoaders, the code responsible for the loading of code into the VM, are themselves written in Java. Because ClassLoaders must follow all the standard rules of Java programming, that means that any ClassLoader ultimately extends java.lang.Object. But the ClassLoader, according to the rules set forth in the Java Language Specification and Java Virtual Machine Specification, cannot be fully loaded until its dependent classes have been loaded, all of which must depend on ClassLoader in order to be loaded!

The solution, of course, is to have one ClassLoader that is not, in fact, written in Java, but implemented entirely within the VM as native code-in this way, it can load the necessary bootstrapping code into place, such as the other two ClassLoaders in the delegation tree (ExtClassLoader, for extensions, and AppClassLoader, for loading from the CLASSPATH), and all the normal rules of ClassLoaders can be upheld-each class is in turn associated with the ClassLoader that loaded it, and ClassLoaders are still used to bring the code into the VM.

This bootstrap ClassLoader is the ultimate parent of any delegating ClassLoader, since its representation is that of a null reference. Thus, in the following code

```
public class ListCL {
    public static void main (String args[]) {
        System.out.println(java.lang.Object.class.getClassLoader());
    }
}
```

produces the following result

```
C:\Test>java ListCL
null
C:\Test>
```

In fact, the entire Runtime API will be loaded by the bootstrap ClassLoader, since

1. classes always ask the ClassLoader that loaded them to load any of their dependent classes, and
2. ClassLoaders always delegate up the delegation tree to see if their "parent" ClassLoaders wish to

load the class first.

By combination of these two rules, this means any class shipping with the JDK as part of the "rt.jar" file in the jre/lib directory will be loaded by the bootstrap ClassLoader. Once the bootstrap ClassLoader is able to load one of the necessary classes (such as Throwable), then all corresponding classes will come out of the bootstrap ClassLoader, as well.

As a result, this creates a potentially dangerous situation: any class that maintains static variables will only have those variables be static across the entire VM if-and-only-if that class is loaded by the bootstrap ClassLoader. As described in [7], statics are in fact, delimited by ClassLoader boundaries, and while it's unlikely that this should occur, it's still entirely possible to create a scenario such as:

```
import java.io.*;
import java.net.*;
public class SeparateClassLoader
{
    public static void main (String args[])
        throws Exception
    {
        // Create a URLClassLoader that uses as its parent
        // ClassLoader the bootstrap ClassLoader (indicated
        // by the "null" URLClassLoader constructor's
        // second argument value). This means that this
        // ClassLoader no longer uses the AppClassLoader and
        // ExtClassLoader as parent delegates, and should
        // not pick up code along the CLASSPATH (such as the
        // current directory).
        //
        URL[] urlArray =
        {
            new File("./nonexistent_directory").toURL()
        };
        URLClassLoader cl = new URLClassLoader(urlArray, null);
        cl.loadClass(SeparateClassLoader.class.getName());
    }
}
```

In this code, we create an instance of URLClassLoader that looks solely in a nonexistent directory for code, and uses the bootstrap ClassLoader as its delegating ClassLoader parent. Thus, when this code is executed:

```
C:\Projects\Papers\BootClasspath\src>java SeparateClassLoader
Exception in thread "main" java.lang.ClassNotFoundException: SeparateClassLoader
        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:290)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:247)
        at SeparateClassLoader.main(SeparateClassLoader.java:23)
C:\Projects\Papers\BootClasspath\src>
```

Intriguing, no? Even though SeparateClassLoader, the class, was loaded by the VM and executed, when we ask this new URLClassLoader to load the very same class, it fails. Because the AppClassLoader (which loaded the SeparateClassLoader class the first time, since it was found on the CLASSPATH) isn't part of ucl's ClassLoader delegation chain, and `ucl` itself can't find it, then `ucl` gives up and throws a ClassNotFoundException, as it should.

Modify this code to let the new URLClassLoader instance load out of the current directory, and inspect the value of a static variable when it does so:

```
import java.io.*;
import java.net.*;
public class SeparateClassLoaderWithStatic
{
    public static int instanceCount = 0;
    public SeparateClassLoaderWithStatic()
    {
        System.out.println("This is instance #" + ++instanceCount);
    }
    public static void main (String args[])
        throws Exception
    {
        new SeparateClassLoaderWithStatic();
        // Create a URLClassLoader that uses as its parent
        // ClassLoader the bootstrap ClassLoader (indicated
        // by the "null" URLClassLoader constructor's
        // second argument value). This means that this
        // ClassLoader no longer uses the AppClassLoader and
        // ExtClassLoader as parent delegates, and should
        // not pick up code along the CLASSPATH (such as the
        // current directory).
        //
        URL[] urlArray =
        {
            new File(System.getProperty("user.dir")).toURL()
        };
        URLClassLoader cl = new URLClassLoader(urlArray, null);
        cl.loadClass("SeparateClassLoaderWithStatic").newInstance();
    }
}
```

And the executed result:

```
C:\Projects\Papers\BootClasspath\src>java SeparateClassLoaderWithStatic
This is instance #1
This is instance #1
C:\Projects\Papers\BootClasspath\src>
```

Because the URLClassLoader was not part of the AppClassLoader/ExtClassLoader delegation chain, it didn't find that SeparateClassLoaderWithStatic had already been loaded, and so loaded it into its own namespace, which in turn created a new static value for "instanceCount".

What all of this ultimately boils down to, so far as we are concerned, is that the bootstrap ClassLoader is the first-and-foremost ClassLoader in the delegation tree. As shown by the SeparateClassLoaderWithStatic example, it is possible to create ClassLoaders that don't participate at all in the delegation tree, thus causing code to be reloaded even though it may already exist within the VM as part of another tree. As a result, the bootstrap ClassLoader, as the root of the tree, is the only ClassLoader we can safely guarantee will hold static instances across the entire VM.

This leaves us with a dilemma, then-if the bootstrap ClassLoader is the only ClassLoader we can trust absolutely with static instances, then how do we force code to participate in the bootstrap ClassLoader's namespace?

Enter the "-Xbootclasspath" option, of course.

Solution Discussion, Part I

Recall, from earlier in this text, that the Sun JVM supports the "-Xbootclasspath" option. This option, according to the JDK documentation that comes in the JDK 1.2.2 download "doc" bundle, claims

> Specify a semicolon-separated list of directories, JAR archives, and ZIP archives to search for boot class files. These are used in place of the boot class files included in the JDK 1.2 software.
>
> *--jdk1.2.2/docs/tooldocs/win32/java.html*

Just as the "CLASSPATH" environment variable turns into a Java System property when the VM is executed, so too does the "bootclasspath" option turn into a System property; when the following code is executed, listing all properties on System, we get an interesting response

```
/**
 *
 */
public class Props {
    public static void main (String args[]) {
        System.getProperties().list(System.out);
    }
}
```

```
C:\Projects\Papers\BootClasspath\src>java Props
-- listing properties --
java.specification.name=Java Platform API Specification
awt.toolkit=sun.awt.windows.WToolkit
java.version=1.2.2
java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment
user.timezone=America/Los_Angeles
java.specification.version=1.2
java.vm.vendor=Sun Microsystems Inc.
user.home=C:\WINDOWS
java.vm.specification.version=1.0
os.arch=x86
java.awt.fonts=
java.vendor.url=http://java.sun.com/
user.region=US
file.encoding.pkg=sun.io
java.home=C:\PRG\JDK1.2.2\JRE
java.class.path=.
line.separator=
java.ext.dirs=C:\PRG\JDK1.2.2\JRE\lib\ext
java.io.tmpdir=c:\windows\TEMP\
os.name=Windows 95
java.vendor=Sun Microsystems Inc.
java.awt.printerjob=sun.awt.windows.WPrinterJob
java.library.path=C:\PRG\JDK1.2.2\BIN;.;C:\WINDOWS\SYST...
java.vm.specification.vendor=Sun Microsystems Inc.
sun.io.unicode.encoding=UnicodeLittle
file.encoding=Cp1252
java.specification.vendor=Sun Microsystems Inc.
user.language=en
user.name=unknown
java.vendor.url.bug=http://java.sun.com/cgi-bin/bugreport...
java.vm.name=Classic VM
java.class.version=46.0
java.vm.specification.name=Java Virtual Machine Specification
sun.boot.library.path=C:\PRG\JDK1.2.2\JRE\bin
os.version=4.10
java.vm.version=1.2.2
```

```
java.vm.info=build JDK-1.2.2-W, native threads, sy...
java.compiler=symcjit
path.separator=;
file.separator=\
user.dir=C:\Projects\Papers\BootClasspath\src
sun.boot.class.path=C:\PRG\JDK1.2.2\JRE\lib\rt.jar;C:\PRG...
C:\Projects\Papers\BootClasspath\src>
```

Notice that the "sun.boot.class.path" property[2], when listed out completely, lists not only the "rt.jar", which we would expect, but also a subdirectory that doesn't exist in a standard JDK or JRE installation:

```
public class Props {
    public static void main (String args[]) {
        System.out.println(System.getProperty("sun.boot.class.path"));
    }
}
```

```
C:\Projects\Papers\BootClasspath\src>java Props
C:\PRG\JDK1.2.2\JRE\lib\rt.jar;C:\PRG\JDK1.2.2\JRE\lib\i18n.jar;C:\PRG\JDK1.2.2\
JRE\classes
C:\Projects\Papers\BootClasspath\src>
```

When executed under a JDK 1.3.0 VM, the property gets even more interesting:

```
C:\Projects\Papers\BootClasspath\src>java Props
C:\PRG\JDK1.3\JRE\lib\rt.jar;C:\PRG\JDK1.3\JRE\lib\i18n.jar;C:\PRG\JDK1.3\JRE\li
b\sunrsasign.jar;C:\PRG\JDK1.3\JRE\classes
C:\Projects\Papers\BootClasspath\src>
```

Not only are both VM's expecting to be able to load code into the bootstrap ClassLoader from the nonexistent "classes" directory under the JRE's installation directory, but the 1.3 JVM loads the "i18n.jar" and "sunrsasign.jar" archives as part of the bootclasspath, as well.

The next step is to see if, in fact, we can load code into the bootstrap classloader by simply creating the "classes" directory and seeing if, in fact, the VM will honor the property and load code from there. This means we should be able to drop any arbitrary class into a "classes" subdirectory in the JRE directory, and it should be picked up by the boot ClassLoader:

```
/**
 * Compiled this into the "C:\Prg\JDK1.2.2\JRE\classes" directory, or
 * wherever your JRE lives
 */
public class NewObject
{
}
/**
 * Compile and run anywhere, without C:\Prg\JDK1.2.2\JRE\classes
 * on the CLASSPATH
 */
public class NewObjectLoader {
    public static void main (String args[]) {
        NewObject no = new NewObject();
        System.out.println(
            no.getClass().getName() + " was loaded by " +
            no.getClass().getClassLoader());
    }
}
```

Sure enough, the NewObject class is not only found at compile-time and runtime, but it prints "NewObject was loaded by null", indicating it owes its existence to the bootstrap ClassLoader.

This means that we've solved the first problem, that of making sure statics really are static, all the way across the VM. It also means that, if you run into the kinds of Extension problems described in [classForName] , you can drop the user-level code into this "classes" directory, and Extensions, by virtue of the fact that they delegate to the bootstrap ClassLoader, will find your user code.

Consequences, Part I

Several large caveats go with this, however. To begin with, any and all security policy set by the SecurityManager or the associated Policy implementation (see [8] for details) are completely ignored and entirely unenforced for code loaded from the bootstrap ClassLoader. This is an entirely undesirable situation, since the Java2 SecurityManager will not be allowed to restrict code from doing damaging or potentially unsafe things should untrusted code somehow make it into that "classes" subdirectory. Treat any code that goes there with the utmost of care.

Secondly, the problems described in [classForName] don't truly go away-code loaded by the bootstrap ClassLoader cannot dynamically load code from the CLASSPATH or from the Extensions directory, since the bootstrap ClassLoader is higher up the delegation tree than either of those two ClassLoaders. In fact, code loaded by the bootstrap ClassLoader will never delegate to any other ClassLoader, because it has no parent. So systems that want to allow end-users to "hook in" via user-level APIs will still be back in the same boat they were when the code was loaded under the ExtClassLoader.

There is, however, a more interesting notion that occurs with respect to ClassLoaders.

Problem Discussion, Part II (Runtime API Replacement)

Since we have the ability to specify the source for the Runtime API library, it's entirely possible to enhance, modify, or even replace parts of the API with your own code. By placing a directory ahead of the rt.jar file on the "sun.boot.class.path" property, your classes can get loaded ahead of the Sun-standard API classes.

This ability can be used with tremendous effect when "snooping" throughout the JVM. Curious to know when <something interesting> happens? Copy the source file in question over to the "boot" directory (in the appropriately-named package subdirectories, of course), modify the code, recompile, and run

```
java -Xbootclasspath:./boot;C:\prg\jdk1.2.2\jre\lib\rt.jar;... MyApp
```

And now, when the JVM starts, the bootstrap ClassLoader will first look in the local "boot" directory for code before going to the "rt.jar" file.

As an experiment, let's replace the java.lang.Integer class with one of our own making. Copy the Integer.java file (from src/java/lang under the JDK directory) to the boot/java/lang directory, and modify the Integer constructor like so:

```
/**
 * Constructs a newly allocated Integer object that
 * represents the primitive int argument.
 *
 * @param value the value to be represented by the Integer.
 */
public Integer(int value) {
        System.out.println("Custom Integer");
```

```
            // I added the above line
            //
        this.value = value;
}
```

All we've done is modify the Integer constructor to spit out a line to the console. Testing to make sure our new Integer is used is relatively simple:

```
public class TestInteger
{
    public static void main (String args[])
    {
        new Integer(12);
    }
}
```

```
C:\Projects\Papers\BootClasspath\src>javac TestInteger.java
C:\...\>java -Xbootclasspath:./boot;C:\Prg\jdk1.3\jre\lib\rt.jar TestInteger
Custom Integer
C:\Projects\Papers\BootClasspath\src>
```

As you can see, running the TestInteger with "./boot" prepended to the normal boot classpath forces the bootstrap ClassLoader to pick up the customized Integer class in "boot/java/lang/Integer.class", rather than the Integer class found in the rt.jar file[3].

This has some interesting implications, not only for "snooping" around in the Java VM, but also in the realm of supplanting Sun's code with some of your own. Let's examine a potential example: you want to support the ability of the VM to access environment variables in the process currently executing the VM.

Recall, from the JDK 1.0 days, that the way of obtaining environment variables was to use the `System.getenv()` method. From the javadocs, the `System.getenv()` method has this to say:

> Gets an environment variable. An environment variable is a system-dependent external variable that has a string value.
>
> **Deprecated.** The preferred way to extract system-dependent information is the system properties of the java.lang.System.getProperty methods and the corresponding getTypeName methods of the Boolean, Integer, and Long primitive types. For example:
>
> ```
> String classPath = System.getProperty("java.class.path",".");
> if (Boolean.getBoolean("myapp.exper.mode"))
>     enableExpertCommands();
> ```
>
> *--jdk1.2.2/docs/api/java/lang/System.html*

In JDK 1.1, this method was marked deprecated, mostly because not all JVM hosting operating systems have a notion of "environment variables" (most notably, the MacOS). Unfortunately, it can be a royal pain to have to specify all the environment variables as Java properties on the command-line, so it would be nice to have the ability to call out to the process to get it.

To do so, in fact, is a two-step process: the first step, modifying the java.lang.System class to change its getenv() method implementation from the current no-op it does now, to an implementation that makes the actual call. The second step, by far the more complex one, is to write a native library that, in fact, does the actual retrieval of the environment variable value[4].

I'll not show the details of building the C++ code to do the actual retrieval; the C/C++ code to do so

would involve a simple call to the ANSI C function `getenv`. In the System class, the implementation of getenv() would then have to change from this:

```
public static String getenv(String name) {
        throw new Error("getenv no longer supported, use properties and -D instead: " + name);
    }
```

To something a bit more complex and (we hope) useful:

```
private static boolean getenvSupported = false;
static
{
    try
    {
        System.loadLibrary("GETENV");
        getenvSupported = true;
    }
    catch (Exception x)
    {
        // Nothing we can do here; System gets loaded so early
        // in the process, nothing is available for us to signal
        // the error condition to the VM invoker!
    }
}
private native String native_getenv(String name);
public static String getenv(String name) {
    if (getenvSupported)
        return native_getenv(name);
    else
        throw new Error("getenv not supported on this platform, " +
            "use properties and -D instead: " + name);
}
```

What's going on here is (hopefully) fairly straightforward: when System is first loaded into the VM, the static initializer block attempts to load the native library "getenv" (which translates into "getenv.dll" on a Win32 machine, or "libgetenv.so" on a Unix machine). If that native library is found, then the static boolean field getenvSupported is flagged as "true" to indicate that we have, in fact, environment variable support on this platform. Then, when the System.getenv() method is invoked, if the native library was loaded, we call into the native_getenv() method exported from that native library. If the native library couldn't be loaded, we do what getenv() has done now since JDK 1.1, which is to throw an Error instance with the error message.

Stop and consider, for just a moment, what we're doing here-by specifying our own version of the java.lang.System class ahead of the rt.jar file in the sun.boot.class.path property, we're effectively patching the Sun-shipped JDK implementation. Now, any time the getenv() method of java.lang.System is invoked, our new implementation will first look to see if it has the necessary native-library support, and use it if present.

Consequences, Part II

Suddenly, no Java class in the entire Java Runtime API is safe-we can mix & match classes between Sun's implementation and customized code as the mood suits us. In some cases, this can be used for Good: consider the java.util.Stack class, one of the worst-designed classes in Java history[5]. Now we can replace the Sun-default java.util.Stack with our own java.util.Stack, one which properly delegates element storage to an internal Vector, rather than extending it.

Alternatively, just as this power can be used for Good, it can also be used for Evil: at various times, on

the advanced-java mailing list, list participants have bemoaned the fact that "this method is marked protected" or "that field is marked private", and thus inaccessible to the poster. With this ability to replace classes, nothing prevents a Java hacker from effectively gaining access to fields and methods they shouldn't. By simply taking the associated source class (typically available from the src.jar file that ships with the JDK), modifying the desired protection attribute, and placing it into a directory that comes first on the sun.boot.class.path property[6], the hacker in question can gain access to that field or method. Needless to say, this is not the intended spirit of this paper-remember, a true Programmer uses the Source only for knowledge and defense, never for attack[7].

At the same time, if such things matter to you, realize that doing this is a fast way to lose any sort of Sun branding you might be pursuing-granted, although the code may technically be "100% Pure Java", you're monkeying with the fundamental definition of the JVM. This is not the way to endear yourself-or your product-to the engineers at Sun.

Summary

The Sun JDK implementations provide an option, "-Xbootclasspath[8]", which permit you, the JVM invoker, to specify an alternative location for code to be loaded from the bootstrap ClassLoader. This forces code found along the sun.boot.class.path property into the bootstrap ClassLoader, which, because the bootstrap ClassLoader is the root of the ClassLoader delegation tree, means that this class definition will be seen by the entire JVM and can never be unloaded.

Because of ClassLoader relationships, code may not behave precisely the way you expect-in particular, code which uses static fields may, in fact, be loaded into multiple ClassLoader instances, thereby creating new instances of static fields each time. This can have serious drawbacks in developing code, particularly since there is no way the compiler can assist or help-the ClassLoader relationship is a runtime one. Thus, the only way to ensure a static reference across the entire JVM is to make sure the class is loaded by the bootstrap ClassLoader.

Sun will be the first (and I will be the second) to point out that there are alternatives to using the bootclasspath option, depending on what your exact needs are. Alternatives include

1. Use the Extensions directory to load code. Instead of modifying the bootclasspath to include the DatabasePool class (from the servlet example above), drop the DatabasePool into the JRE Extensions directory and leave it at that. Extensions are loaded by the ExtClassLoader, which is an immediate child of the bootstrap ClassLoader itself, so you're fairly far up the ClassLoader chain. What's more, Extensions are fully documented and should be supported on all Java2 virtual machines, something we can't say with respect to the bootstrap ClassLoader options.
2. Write a custom ClassLoader to load your own code, and have it parented directly to the bootstrap ClassLoader. This has the drawback of not being able to find either installed Extension classes, or classes along the java.class.path, but also ensures that your classes are second only to the Java Runtime API in terms of location along the ClassLoader delegation tree.

The bootstrap ClassLoader is intended for use solely by the JVM--as its name implies, its entire purpose is to simply bring those Java classes into the VM that the VM in turn depends upon. Because of this, you should be warned that any attempt to replace the existing API classes carries with it severe risks that may not be acceptable in a production environment:

- *Security.* Code loaded from the bootstrap ClassLoader has no security policy enforced upon it. That means that anyone can drop a .class file into a subdirectory and have immediate access to any secure operation defined in Java.
- *Stability.* Any time you replace a class from the Java Runtime API, you are making the implicit statement that you know more (or better) than the engineers at Sun. Without debating the truth of the statement, understand that by replacing code like that, you are assuming you know every possible interaction between the class in question and the rest of the Runtime API. As a case in point, for the API-replacement example, I wanted to create a new java.util.Properties class that would echo to the command line any property requested by the System.getProperty() or System.getProperties().get() methods. (The idea was simple: create a definitive list of all properties

requested throughout the lifetime of the VM.) Unfortunately, I ultimately had to give up the idea, since the java.lang.System class gets loaded so early into the VM's lifetime, that attempting to write to System.out resulted in NullPointerExceptions, and attempting to open a file proved equally fruitless, since the FileOutputStream classes in turn depended on the complete System class being loaded into the VM. This was a trivial example-imagine how cranky the VM would get if you tried to replace something vital, like java.lang.Object itself.

- *Robustness.* Let's face it, the code coming out of Sun's doors is going to get a far better workout than code you write to replace the Sun APIs, if only because Sun has thousands of Java developers banging on it daily, and you don't.

If you choose to use the bootclasspath option, be careful out there. Like many things, with power comes responsibility.

Notes

[1] Note-much of what I describe in this paper, while specific to the Sun JVM, could be implemented in other VMs if they use the Sun VM as a strict guide to follow. All of this has only been tested with the Sun JVM, however, and cannot be guaranteed to work on any other VM.
[2] All Sun-specific (that is, non-standard) properties begin with "sun.", to flag them as such.
[3] Like the CLASSPATH (or "java.class.path" property, as it should be more properly called), the first class found will satisfy the loadClass() request, so here, when the boot/java/lang/Integer.class file is found, it stops looking and never finds the .class file in the rt.jar file.
[4] It has to be a native library implementation because the JVM itself provides no way to do this-that's what we're trying to add back in, remember?
[5] For those unfamiliar with it, java.util.Stack extends java.util.Vector, meaning we can easily bypass the Stack's encapsulation of its data by calling any of the Vector methods on it, like elementAt(). The verdict? A horrible design/implementation; whomever spawned this thing should be shot.
[6] Theoretically, nothing would prevent said hacker from simply slipping it into the rt.jar file shipped with the JRE. However, Sun could (and probably should), in a future implementation, sign the rt.jar file to prevent such tampering.
[7] With all apologies to George Lucas
[8] In fact, the JDK 1.3 provides three such options, all of which conspire to allow you to prepend, append, or replace the existing bootclasspath value entirely.

Bibliography

- **[classForName]** "Understanding Class.forName()", by Ted Neward (http://www.javageeks.com/Papers).
- **[JREPaper]** "Multiple JRE Homes", by Ted Neward (http://www.javageeks.com/Papers).
- **[javaStatics]** "Java Statics", by Ted Neward (http://www.javageeks.com/Papers).
- **[sbpj]** *Server-Based Java Programming* , by Ted Neward. Manning Publications (ISBN: ZZZ)
- **[insideVM]** *Inside the Java2 VM*, by Bill Venners. ZZZ (ISBN: ZZZ)
- **[javaSecurity]** *Inside Java2 Security*, by Li Gong. Javasoft Press (ISBN: ZZZ)

Copyright

Colophon

This paper was written using a standard text editor, captured in a custom XML format, and rendered to PDF using the Apache Xalan XSLT engine and the Apache FOP-0.17 XSL:FO engine. For information on the process, contact the author at tneward@javageeks.com. Revision $Revision: 1.1 $, using whitePaper.xsl revision $Revision: 1.1 $.