A JavaGeeks.com
White Paper

Ted Neward
http://www.javageeks.com/~tneward
tneward@javageeks.com
15September2001

Multiple Java Homes

Giving Java Apps Their Own JRE

Abstract

With the exponential growth of Java as a server-side development language has come an equivablent exponential growth in Java development tools, environments, frameworks, and extensions. Unfortunately, not all of these tools play nicely together under the same Java VM installation. Some require a Servlet 2.1-compliant environment, some require 2.2. Some only run under JDK 1.2 or above, some under JDK 1.1 (and no higher). Some require the "com.sun.swing" packages from pre-Swing 1.0 days, others require the "javax.swing" package names.

Worse yet, this problem can be found even within the corporate enterprise, as systems developed using Java from just six months ago may suddenly "not work" due to the installation of some Java Extension required by a new (seemingly unrelated) application release. This can complicate deployment of Java applications across the corporation, and lead customers to wonder precisely why, five years after the start of the infamous "Installing-this-app-breaks-my-system" woes began with Microsoft's DLL schemes, we still haven't progressed much beyond that. (In fact, the new .NET initiative actually seeks to solve the infamous "DLL-Hell" problem just described.)

This paper describes how to configure a Java installation such that a given application receives its own, private, JRE, allowing multiple Java environments to coexist without driving customers (or system administrators) insane.

Problem discussion

The current state of affairs-the CLASSPATH. Java projects (and products) depend heavily upon the Java CLASSPATH for proper execution. If classes are not found along the CLASSPATH (or within the Extensions directory; see [sbjp] , [jdk] for details), then the application cannot execute certain pieces of functionality at best, or will simply not execute at all at worst. Traditionally, when installing a new Java application, the installation instructions will ask the user/installer to place the application's .jar file on the CLASSPATH, and run "java com.fooware.FooApp" to execute the application. This is, unfortunately, the tried-and-true way for Java apps.

The problem with this approach is fairly simple: not only does the CLASSPATH begin to grow to unwieldy proportions, but certain environments (most notably, NT and Windows 95/98) have an upper limit on the size of the allowed value for an environment variable. Worse, CLASSPATH settings are fragile things, easily corruptible by users and/or other application installations. Certain Java install toolkits, for example, were known to overwrite, rather than append to, the user's CLASSPATH setting, leading to confusing results for users-installing FooWare 1.0 would render JBlah 2.0 (which the user installed last week) inoperable.

Sun, in its JDK 1.2 release, recently made some strides towards correcting this problem, by allowing developers to mark a .jar file as "executable", via the "-jar" option to the "java" interpreter. If a developer places a "Main-Class" entry in the jar's associated manifest file (see [sbjp] , [jdk] for details), then the user may "execute" the jar file by typing "java -jar FooWare.jar" at the command-line, but this is a behavior specific to the java.exe interpreter shipping with the JDK, and is not a "standard" Java feature specified within any of the Sun Specification documentation. What's worse, implementing this option requires tool vendors to examine the java.c file, in the src/launcher directory of the src.jar file that ships with the JDK. (See [sbjp] for discussion.)

Is there, in fact, a problem with this arrangement? After all, if the CLASSPATH is becoming too large, there's nothing preventing a Java user from creating a batch file under NT that reads something like

```
java -classpath C:/fooware/lib/fooware.jar;%CLASSPATH% com.fooware.FooApp
```

So, what precisely is the problem here?

If the above were the only problem with the current approach, then we could simply shrug our collective shoulders, say, "It's not pretty, but it works", and leave it at that. Unfortunately, this all begins to fall apart when our user installs a *second* Java application on the machine.

Version dependencies. It's not uncommon, within the quickly-evolving World Of Java, to find applications that require older versions of the JDK or particular libraries/frameworks (such as Swing) that aren't present within the current version of the JDK. As a concrete example of this, Borland/Inprise's JBuilder2 makes use of the Swing libraries before the 1.0 release, when the packages were named "com.sun.swing" instead of "javax.swing". While it's true that JBuilder2 is not the latest release of JBuilder, corporations may not want to accept the risk of upgrading tool versions in the middle of the project; this in turn means that their code will depend on Swing being named "com.sun.swing" instead of "javax.swing".

While the application will run perfectly well on the developers' machines, as soon as the application is deployed onto a machine already running a later version of the JDK 1.2, using the "javax.swing" classes, it will fail, because the Swing classes on the target machine aren't in the "com.sun.swing" package.

Developer version separation. Even worse, what goes for users goes doubly so for developers. Developers are constantly making use of a variety of tools, applications and/or class libraries that may have varying support requirements. In some cases, it will be for a simple evaluation period, while they struggle to determine if the product or code fits the needs of the project.

At the same time, developers need to keep their development environments free of this sort of version-contention, if for no other reason than because the last thing a developer needs is to chase down a bug in the project that turns out to be due to a mismatch between the version of JavaHelp she *thought* she was using, and the version she actually was using. For the same reason, developers cannot upgrade tools in the middle of the project, lest there be an incompatibility that drives the ship date back weeks (or, in particularly bad cases, even months) due to versioning mismatch.

This problem will only grow worse as OpenSource projects dominate more and more of the development landscape, as OpenSource projects are usually much less concerned with installation and/or support issues as commercial products. Already a number of OpenSource projects are dated enough to require JDK 1.1, even though JDK 1.3 has been out for a year, and JDK 1.4 is "just around the corner".

Solution discussion

The JRE. While few developers may be familiar with it, the Java Runtime Environment, or the JRE, provides a complete runtime-only (that is, no development tools provided) tool suite for executing Java applications. The JRE provides more than just the tools, however-in fact, the directory in which the JRE is installed can serve as a standalone root for a single Java installation, complete with its own Extensions directory.

Given this, it becomes trivial to maintain an entirely separate Java installation-complete with its own interpreter, its own runtime library, and its own Extensions subdirectory. More importantly, with a simple directory structure, an application can be installed in such a way that will not conflict with any other Java application on the system, regardless of installed Extensions elsewhere.

For example, assuming that we have a Java application deployed in fooware.jar, with an appropriate Main-Class manifest setting, we can deploy it in its own directory, C:\FooWare (or /usr/local/fooware). Now, by executing a recursive-copy from the JDK's JRE directory (such as "xcopy /s/e C:\jdk1.2.2\jre*.* C:\FooWare" or "cp -r /usr/local/java/jdk1.2.2/jre/* /usr/local/fooware" under UNIX), two subdirectories, "bin" and "lib" will be created. Inside of "bin" will be the java runtime tools, such as "java", "rmid", and "tnameserv", and inside of the "lib" directory will be the usual runtime library-support files, like "rt.jar" and the "ext" directory, the usual location for installed Extensions.

If, from the C:\FooWare or /usr/local/fooware directory, the command "bin/java -jar fooware.jar" is fired,

only the code in the *fooware*/lib/ext Extensions directory and the fooware.jar file will be executed, regardless of the CLASSPATH setting. If additional files need to be present, they can be dropped into the *fooware*/lib/ext directory, without affecting any other installation of Java on the system. Because the file was executed directly from within the "bin" directory, any installed JDK tools on the PATH will not be picked up-remember, most shells guarantee that the PATH is searched only in the event that the requested file cannot be found in the directory specified. End result: an application or library can be evaluated without affecting any other JDK or JRE installation on the system.

The key to understanding how this can work is to examine the C code used to create the "java" executable. Contrary to what many developers may believe, this code is provided by Sun free of any other licensing restrictions than that of the normal JDK-it ships as part of the JDK 1.2 download bundle, buried inside the "launcher" directory in the "src.jar" file. Four files, java.c, java.h, java_md.c and java_md.h are located here.

JNI, or the Java Native Interface, contains an API for creating a JVM from within standard C code, known as JNI Invocation. The java process uses this API to create the JVM, load the appropriate class (specified on the command-line), either from the CLASSPATH or from the jar given by the "-jar" option.

Java.c contains the majority of the code, where java_md.c contains the platform-dependent aspects of creating the JVM.

To start with, let's take a look at the code in question, the initial Invocation code. As with any standard C application, execution starts in `main`:

```
/*
 * Entry point.
 */
int
main(int argc, char **argv)
{
    JavaVM *vm = 0;
    JNIEnv *env = 0;
    char *jarfile = 0;
    char *classname = 0;
    char *s = 0;
    jclass mainClass;
    jmethodID mainID;
    jobjectArray mainArgs;
    int ret;
    InvocationFunctions ifn;
    char *jvmtype = 0;
    jboolean jvmspecified = JNI_FALSE;   /* Assume no option specified. */
    jlong start, end;
    if (getenv("_JAVA_LAUNCHER_DEBUG") != 0) {
        debug = JNI_TRUE;
        printf("----_JAVA_LAUNCHER_DEBUG----\n");
    }
```

Notice the "_JAVA_LAUNCHER_DEBUG" environment-variable check; this is an easy way to do some coarse level diagnosis regarding the VM.

```
    /* Did the user pass a -classic or -hotspot as the first option to
     * the launcher? */
    if (argc > 1) {
        if (strcmp(argv[1], "-hotspot") == 0) {
            jvmtype = "hotspot";
            jvmspecified = JNI_TRUE;
        } else if (strcmp(argv[1], "-classic") == 0) {
            jvmtype = "classic";
```

```
            jvmspecified = JNI_TRUE;
        }
    }
    ifn.CreateJavaVM = 0; ifn.GetDefaultJavaVMInitArgs = 0;
    if (!LoadJavaVM(jvmtype, &ifn))
        return 1;
```

This is what supports the JVM's "drop-in" JIT support-if the "-hotspot" option is provided, the interpreter will attempt to use the HotSpot JIT interpreter available from Sun; if not, it will look for the classic, non-JIT interpreter.

The key to what makes the separate-JRE work, however, lies in the call to `LoadJavaVM`. This function, defined in java_md.c, provides the basic functionality for finding-and loading-the actual VM into the application's process space.

Win32 LoadJavaVM. The code for `LoadJavaVM` looks like this; notice the references to `HINSTANCE` and other Win32-specific code. Remember, because this file changes with each and every platform Java is ported to, this file will look differently on other platforms.

```
#ifdef DEBUG
#define JVM_DLL "jvm_g.dll"
#define JAVA_DLL "java_g.dll"
#else
#define JVM_DLL "jvm.dll"
#define JAVA_DLL "java.dll"
#endif
/* . . . other code snipped . . . */
/*
 * Load JVM of "jvmtype", and intialize the invocation functions. Notice
 * that if jvmtype is NULL, we try to load hotspot VM as the default.
 * Maybe we need an environment variable that dictates the choice of
 * default VM.
 */
jboolean
LoadJavaVM(char *jvmtype, InvocationFunctions *ifn)
{
    char home[MAXPATHLEN], javadll[MAXPATHLEN], jvmdll[MAXPATHLEN];
    HINSTANCE handle;
    struct stat s;
    /* Is JRE co-located with the application? */
    if (GetApplicationHome(home, sizeof(home))) {
        sprintf(javadll, "%s\\bin\\" JAVA_DLL, home);
        if (stat(javadll, &s) == 0)
            goto jrefound;
    }
    /* Does this app ship a private JRE in <apphome>\jre directory? */
    sprintf(javadll, "%s\\jre\\bin\\" JAVA_DLL, home);
    if (stat(javadll, &s) == 0) {
        strcat(home, "\\jre");
        goto jrefound;
    }
    /* Look for a public JRE on this machine. */
    if (!GetPublicJREHome(home, sizeof(home))) {
        return JNI_FALSE;
    }
```

These three blocks are the core of the whole multiple-JRE trick. To start with, `LoadJavaVM` tried to see if the JRE is "co-located" with the application starting it. To do so, it calls `GetApplicationHome`,

another function in java_md.c:

```
/*
 * If app is "c:\foo\bin\javac", then put "c:\foo" into buf.
 */
jboolean
GetApplicationHome(char *buf, jint bufsize)
{
    char *cp;
    GetModuleFileName(0, buf, bufsize);
    *strrchr(buf, '\\') = '\0'; /* remove .exe file name */
    if ((cp = strrchr(buf, '\\')) == 0) {
        /* This happens if the application is in a drive root, and
         * there is no bin directory. */
        buf[0] = '\0';
        return JNI_FALSE;
    }
    *cp = '\0';  /* remove the bin\ part */
    return JNI_TRUE;
}
```

As you can see, `GetApplicationHome` simply strips the path of the executable's full pathname down two subdirectories; as the comment implies, if the application's fully-qualified pathname is "C:\foo\bin\java.exe", it places "C:\foo" into the buffer. Going back to `LoadJavaVM`, the code then calls the `stat` function to see if it can find a DLL in a "bin" subdirectory under that directory. So, going with the example above, if this code lives in "C:\foo\bin\java.exe", it looks for "java.dll" in "C:\foo\bin"; if it is found, we jump to the "jrefound" symbol.

Under the "FooWare" example we've been using, however, C:\FooWare doesn't have a direct "bin" subdirectory, so we continue on.

The second block in `LoadJavaVM` next looks to see if this application has a "private" JRE installed underneath it; it uses the buffer returned from `GetApplicationHome` to look for the "java.dll" file in the "/jre/bin" subdirectory. Under the "FooWare" example, this will test true, we jump to the jrehome label, and continue execution from there.

Notice, however, what happens if, under most normal installations, no "/jre/bin/java.dll" file is present-we now examine the Registry, in `GetPublicJREHome`, to find the location of the "public" installed JRE. This is key knowledge for developers-when running code, and the JDK\bin directory is on the PATH, the JRE used is the one that lives inside of the JDK, not the one installed under "Program Files" by a standard JDK installation. This is important, in that Extensions installed under the "public" JRE's Extensions directory will not be picked up by the JDK's JRE.

At this point, the rest of `LoadJavaVM` is fairly anticlimactic. It now looks to see if HotSpot has been installed in this JRE, and uses it. If not, it falls back on the classic VM. Once the VM type to load has been determined, it uses the Win32 `LoadLibrary` function to bring the DLL into the process' space, and calls `GetProcAddress` to obtain the function-pointer values of the Invocation functions `JNI_CreateJavaVM` and `JNI_GetDefaultJavaVMInitArgs`. Once these are obtained, it returns.

```
    /* Now we know where JRE is -- the value in the "home" variable. */
jrefound:
    /* Determine if Hotspot VM is installed. */
    if (jvmtype == NULL) {
        sprintf(jvmdll, "%s\\bin\\hotspot\\" JVM_DLL, home);
        if (stat(jvmdll, &s) < 0) {
            jvmtype = "classic";
        } else {
            jvmtype = "hotspot";
        }
```

```
    }
    /* We now know what jvmtype should be */
    sprintf(jvmdll, "%s\\bin\\%s\\" JVM_DLL, home, jvmtype);
    if (debug) {
        printf("Path to JVM is %s\n", jvmdll);
    }
    /* Load the Java VM DLL */
    if ((handle = LoadLibrary(jvmdll)) == 0) {
        fprintf(stderr, "Error loading: %s\n", jvmdll);
        return JNI_FALSE;
    }
    /* Now get the function addresses */
    ifn->CreateJavaVM =
        (void *)GetProcAddress(handle, "JNI_CreateJavaVM");
    ifn->GetDefaultJavaVMInitArgs =
        (void *)GetProcAddress(handle, "JNI_GetDefaultJavaVMInitArgs");
    if (ifn->CreateJavaVM == 0 ||
        ifn->GetDefaultJavaVMInitArgs == 0) {
        fprintf(stderr, "Can't find JNI interfaces in: %s\n", jvmdll);
        return JNI_FALSE;
    }
    return JNI_TRUE;
}
```

Solaris LoadJavaVM. The code for the Solaris (and Linux) version of `LoadJavaVM` looks and behaves radically differently:

```
jboolean
LoadJavaVM(char *jvmtype, InvocationFunctions *ifn)
{
    ifn->CreateJavaVM = JNI_CreateJavaVM;
    ifn->GetDefaultJavaVMInitArgs = JNI_GetDeafaultJavaVMInitArgs;
    return JNI_TRUE;
}
```

The reason for this radical simplification is that most Unixes simply don't have the notion of a Registry, so there is no way in turn to have a "public" JRE installation. Instead, the Unix version of `GetApplicationHome` looks like this:

```
/*
 * If app is "/foo/bin/sparc/green_threads/javac", then put "/foo" into
 * buf.
 */
jboolean
GetApplicationHome(char *buf, jint bufsize)
{
#ifdef USE_APPHOME
    char *apphome = getenv("APPHOME");
    if (apphome) {
        strncpy(buf, apphome, bufsize-1);
        buf[bufsize-1] = '\0';
        return JNI_TRUE;
    } else {
        return JNI_FALSE;
    }
#else
    Dl_info dlinfo;
    dladdr((void *)GetApplicationHome, &dlinfo);
```

```
    strncpy(buf, dlinfo.dli_fname, bufsize - 1);
    buf[bufsize-1] = '\0';
    *(strrchr(buf, '/')) = '\0';  /* executable file      */
    *(strrchr(buf, '/')) = '\0';  /* green|native threads */
    *(strrchr(buf, '/')) = '\0';  /* sparc|i386           */
    *(strrchr(buf, '/')) = '\0';  /* bin                  */
    return JNI_TRUE;
#endif
}
```

As you can see, the UNIX side of this can either (depending on the macro USE_APPHOME at time of compilation):

1. use the environment variable APPHOME to determine where the application's home directory is, or
2. it will make use of UNIX's dynamic linking facilities (`dladdr` and company) to obtain the address of the module containing `GetApplicationHome` (that is, the module in which this code itself resides), get the filename of this module, and strip out the last four directory names. This is necessary because we want jre, where the full directory path to the java executable file is jre/bin/green_threads/sparc/java. Curious readers will then wonder why putting jdk/bin on the PATH allows the executable form of java to be executed; in fact, this magic occurs because each of the java "executables" in the Solaris distribution are actually symbolic links to a file called ".java_wrapper", which ultimately calls into the appropriate version of "java" in the aforementioned directory. Within that script, .java_wrapper looks for the same co-located or dependent JRE installation, by looking for a "jre" directory in either the "JREHOME" or the "APPHOME" environment variable, the same as Windows does.

Ultimately, however, even though the internals of the actual JVM-creation call are wildly different across platforms, this notion of a "private" JVM still holds. This means that if I take the directory "jre" and copy it and its contents over to another directory, regardless of platform, these two JVM installations will be entirely idempotent-neither one will see code in the other, unless explicitly told to look there (via a CLASSPATH directive, for example).

Consequences

By creating a "private" or "co-located" JRE with your Java application, you're accepting a certain set of consequences.

To begin with, installing and running out of a private JRE means a surrender of a certain amount of system-wide control. Because not all Java code is executing out of the same JRE, as is the case in a "global" JRE installation setup, developers cannot install an Extension into the one-and-only JRE Extension directory and have it picked up everywhere. Instead, anywhere where an Extension is used, it must be installed in that application's JRE's Extensions directory. This in turn means more work and complexity for any given user system.

At the same time, however, this same force means that applications will be protected against system-wide installations of Extensions, so a given application can guarantee (to a limited degree, anyway) that the Extension it expects to find will be the version of the Extension it depends on. This in turn means that private-JRE applications are protected against other Java application installations-and as Java applications gain in popularity on client systems, this will become more and more critical.

Further, within corporate intranet systems or in collections of related client apps, such as an "application suite" or "Garden of Applications" system, they can all share the same JRE by having a common root directory from which the JRE directory root begins. This means that Extensions can be installed and picked up across the entire collection of applications, without breaking (or being broken in turn by) other applications.

Summary

Typical Java application installations make one of two fundamental assumptions regarding the JVM: one, that a copy of the JRE is already installed on the target system in some known "global" location, or two, that a copy of the JRE will need to be installed on the target system because no such "global" copy currently exists. This is fine for naive installations, but will begin to fail catastrophically as more and more Java applications come to the end-user system and depend on various versions of the JDK.

One practical solution to this problem is fairly simple: install a JRE within the application's own installation tree. This provides a number of benefits:

- *Separation of concerns.* By running your application out of its own private JRE, you separate your application's dependent components (most notably, the JVM itself) from any other Java application on the system.
- *Less end-user confusion.* Because end-users no longer have to understand which version of Java they need to run in order to run your application, there is less confusion on their part as to whether their current version of the JVM will run your code. Instead, your application depends on its own JVM, without any "external" dependencies.
- *Less dependence on CLASSPATH.* The CLASSPATH environment variable unfortunately represents one of the weakest aspects of the Java platform. Because it is an environment variable, it is malleable, even to the most basic of Java users. Instead of depending on CLASSPATH for class-file location information, instead use Extensions to hold 3rd-party components. For application code, use the "-jar" option of the JDK, allowing you to bundle your application code into a single .jar file and still not have to place it on the CLASSPATH setting. In the ideal world, your application install should not have to modify the CLASSPATH setting at all in order to run-anything less than that and you need to revisit your deployment scheme.

Notes

Bibliography

- **[sbjp]** *Server-Based Java Programming* , by Ted Neward. Manning (ISBN: 1884777716)
- **[jdk]** "JDK 1.2 documentation bundle.", by Sun Microsystems (http://www.javasoft.com).

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 $.