Legacy Application Access Through Java

By Jeff Jilg and John Dodson

Java provides a mechanism for tying existing applications to platform-neutral Java interfaces. Java applets and applications can also be extended to take advantage of platform-specific (native) features. This article explores the native method interface deployed in Java 1.0 for connecting Java to platform-specific programs. The proposed native method interface in Java 1.1 is also discussed in this context.

Since Java™ is now virtually pervasive on all operating systems, it is quickly becoming a *de facto* programming and runtime environment. The Java model provides near platform independence through the Java programming language. Programs written in Java can be compiled on one platform and executed on any computer that has a Java runtime environment. (For a review of Java on AIX®, see *"Java on AIX—A Strong Brew"* in the November 1996 issue of AIXpert). Figure 1 shows the different types of Java applications that will be discussed in this overview. The focus of this article is on scenario D, connecting Java to legacy applications.

Java Applications

There are a variety of ways to program in Java or access Java functions. It is useful to characterize the different methods that span the spectrum from pure (traditional) Java applets to non-Java applications, which interact with Java programs.

Scenario A: Many Java programs are classified as applets since they run inside a Java-enabled Web browser. A well-known example of this scenario is developing an applet on one platform, such as AIX, and distributing the applet through a Web server to an arbitrary client on the Internet (for example, Windows® 95 running Netscape™).

Scenario B: Many applets already on the Internet demonstrate this example. While this provides a useful outlet for small, independent applets, few full-featured applications currently use this model. Corel's® Java-enabled office suite is just now emerging as one of the first major application sets to exploit Java in this way. Lotus® has publicly announced an upcoming port of SuiteSpot™, which will be enabled through Java. The number of these full-featured applets and application sets is expected to increase dramatically through 1997 and 1998 as Java matures.

Scenario C: Another avenue from which Java applications are occurring is through Web server side Java applets. These Java applets result from a Web client request that is satisfied on the Web server (through Java). An example of this model is a Web client requesting data on the Web server that must be processed through some server side logic. For example, the Web client may request a list of current office supply inventory from the Web server used by an office supply warehouse. In this case, the Web server must have a mechanism for obtaining and formatting the information before forwarding the answer back to the Web client. Different ways of providing this server side logic include Common Gateway Interface (CGI), JavaScript™, and Java applets. The benefit of Java applets in this example is their platform independence.

Scenario D: Corporations and small companies have a huge investment in existing legacy applications. Rewriting these applications into Java just for the sake of Java's benefits is far too

Jeff Jilg

John Dodson

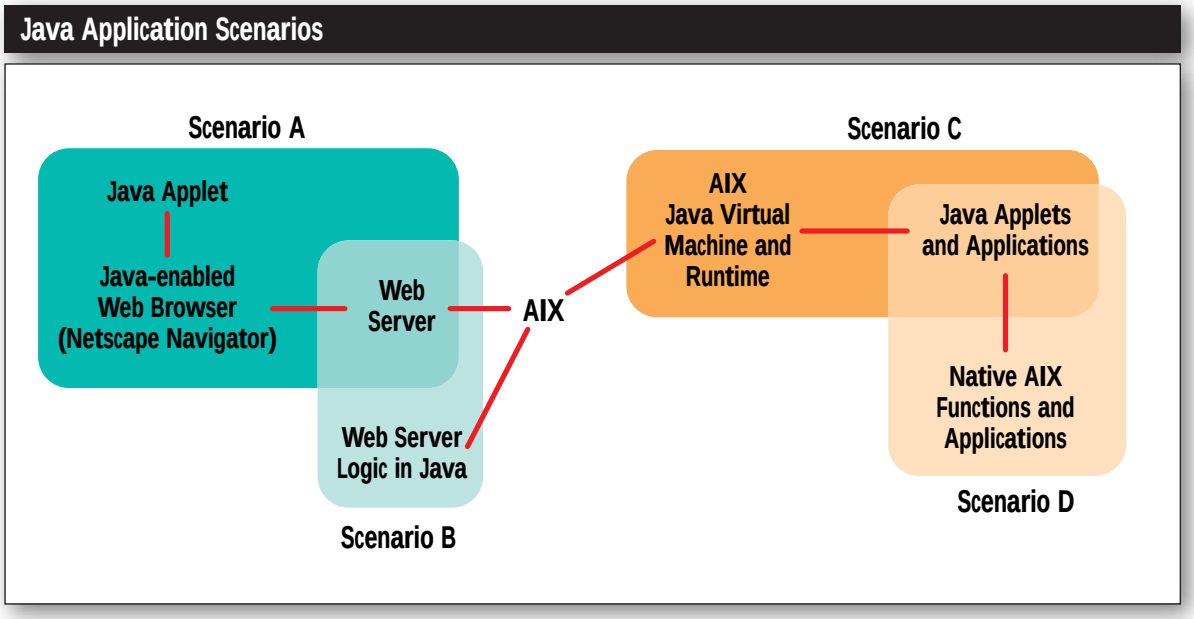

Figure 1. Scenarios of Java applications

expensive. Some applications are being connected through Web servers on intranets and the Internet using the scenarios described above. Another direct mechanism for connecting legacy programs to Java and Java-enabled browsers is through the Java native methods interface. This mechanism is already being adopted across a variety of industries. Native methods will continue to be deployed in the near term rather than the Java application types discussed above. That is because native methods provide a means of accessing legacy programs through Java. Java native method interfaces also can be used to extend the functionality of existing programs through Java applets and applications.

Java native method interfaces can be used in any of the four scenarios described above. This interface allows any Java program to execute a platform-specific binary, such as an AIX program. Thus, a native method can be accessed from the Java applet in scenario A in similar fashion to that shown in scenario D.

The remainder of this article will explore Java native method interfaces in more detail. The next section will describe some advantages, disadvantages, and features provided by this mechanism. The following section shows how native method interfaces can be deployed using the existing Java 1.0 on AIX 4.1 and 4.2. It then examines the similarities and differences between the existing methodology and the new methodology in Java 1.1. The article concludes with a discussion on interoperability and the projected future of Java interfacing.

Features of the Native Method Interface

Contrary to the easy Java applet Web distribution model, native method interfaces have some disadvantages, including the following:

- A native method interface creates instant platform dependence since the corresponding C or C++ program must be recompiled or even modified for each execution platform.
- Java security restrictions are essentially bypassed in the native method since C programs inherit the security properties associated with the user who executes the Java program.
- Coding linkage between Java and C is cumbersome and must be rebuilt. It is then redistributed to all machines that use it (just like current C programs) each time a change occurs to the native method code.

These disadvantages focus on the fact that pure Java programs have distinct advantages that are well-known. However, several key advantages to writing native methods include the following:

- The most important advantage is that native methods allow Java programs to access functions (and whole programs) in the vast array

of existing applications that already support most businesses today.

- Another advantage is the extended functionality provided by native method access to the system. While this is also listed as a disadvantage, the tight security restrictions in Java are sometimes overbearing. For example, the current Java implementation with Netscape Navigator™ Version 3.0 does not allow applets to read or write files on the client. Most production programs must be able to retrieve or store information at some point, so native methods can be used to supplement functionality that is not available in widely used Java applet runtime models.
- Speed and runtime efficiency are critical to the end user perception of all applications. Java on AIX has been supplemented with a Just-In-Time (JIT) compiler that improves runtime performance of Java programs. Mission-critical applications can take advantage of native AIX and other platform Application Programming Interfaces (APIs) to enhance runtime execution on a function-by-function basis from Java using native method interfaces.

The advantages provide insight into how to extend your Java program. Native methods work in both the Java applet and application models. Much of the Java programming over the next two years will use native methods to access existing C-coded function. Java on AIX program execution speed should not be a big issue with the current and future releases of the JIT compiler.

Steps for Native Method Interface on AIX

The steps to implementing a Java native method are straightforward, but must be followed closely for the native method to work properly. The tools required to produce working native methods come standard with the Java Development Kit (JDK) that shipped with AIX 4.2. Access to a C compiler (not part of the JDK) and a linker (provided with AIX) is also required. The following outline can be followed to create native methods.

Steps for Implementing a Java Native Method

Java native methods require six steps to implement.

Step 1. Write the Java code. Subtasks for writing the code include the following:

- Create a Java class that defines the native method. This class should also load the shared library containing the native method implementation.
- Create a Java driver class that will invoke the native method.

Step 2. Compile the Java code using javac. The standard Java compiler, javac, is provided with the JDK. This will create a bytecode class file with the same name as the Java class. Java source file `linkCheck.java`, for example, would result in `linkCheck.class` upon compilation.

Step 3. Create a header file using javah. Also a standard tool, javah is included in the JDK. The usage is `javah <CLASS NAME>`. Note that the actual class name is used, not the class file name. This will create a C header file named `<CLASS>.h`, where `<CLASS>` is the name of the Java class containing the native method definition.

Step 4. Create a C stubs file using javah. Later, the resulting C file will be compiled and linked into a shared library.

Step 5. Write the actual native method. Subtasks for this step include the following:

- The C function signature for the native method including name, parameters, and return type must exactly match the one in the header file created in step 3 above.
- Write the actual native method code as you see fit.

Step 6. Create the AIX shared library. Subtasks include the following:

- The native method must reside in a shared library. Creating this library is platform specific.
- Compile the C code.
- Create the exports file.
- Build the library.

AIX Hints for the Exports File

The steps for implementing native methods are universal. However, there are some AIX-specific items to be aware of. These are detailed in this "hints" section.

Hint 1: The exports file must contain two entries for each native method. The actual function name of the native method from the header file (generated in step 3) and the stub name from the C file (generated in step 4) must

Native methods allow Java programs to access functions (and whole programs) in the vast array of existing applications that already support most businesses today.

be listed on lines by themselves. Note that the exports file requires only the symbol name for the function and does not include function parameters.

Hint 2: The first line in the exports file *must* contain `#!` (on a line by itself).

AIX Hints for Building the Shared Library

The shared library is created using the C compiler and linker. Several AIX-specific items are important to mention:

Hint 1: The library name must be of the form `lib<NAME>.so`, where `<NAME>` is the library name referenced in the Java source file (from step 1) that invokes `System.loadLibrary`. An example library name is `liblinkExample.so`.

Hint 2: Since a shared reentrant load module is required, pass the following flags to the AIX linker: `-bM:SRE -bnoentry`

Hint 3: The Java exports file must also be passed to the linker:

```
-bI:$(JAVA_HOME)/include/java.exp
```

Hint 4: Several libraries must also be linked:

```
-L/usr/lib/threads
-lm -lpthreads -lc_r /usr/lib/libc.a
```

Hint 5: When running the Java program, make sure the `LD_LIBRARY_PATH` (not `LIBPATH`) includes the directory containing the newly created shared library. Example:

```
export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:new_path
```

will add `new_path` to the existing environment variable.

To run the Java application, first ensure the `LD_LIBRARY_PATH` is set to include the shared library created in step 5 above. Then, simply type:

```
java <CLASS>
```

where `<CLASS>` is the name of the driver class created in the second bullet of step 1 above. If the native method is called from a Java applet (instead of an application), then the applet would be referenced from a Web page and loaded through a call to the Web page from appletviewer or a Java-enabled browser such as Netscape Navigator.

The steps to implementing a Java native method are straightforward, but must be followed closely for the native method to work properly.

Java Native Method Example

The following example illustrates an AIX-specific function that is not available in Java. In a similar fashion, native methods can be used as interfaces to existing C applications.

Suppose that you want to determine if a file is a symbolic link. Java has no built-in methods to perform this AIX-oriented function. One solution would be to write a native method in C and invoke that native method from within your Java code. The following example shows how to solve this unique problem using the six steps discussed above.

Step 1: Write the Java code. Figure 1 shows the `linkCheck.java` file. Figure 2 shows the `driver.java` file. Note that this Java code defines a native method interface called `NMisLink` that requires one String parameter, a filename, and returns an int. Also note that a shared library called `linkExample`, which maps to `liblinkExample.so` on AIX, is loaded. Finally, the Java method `isLink` invokes the actual C function. Other Java code can access the native method only through the `isLink` Java method.

Step 2: Compile the Java code. `javac linkCheck.java` creates `linkCheck.class`. `javac driver.java` creates `driver.class`.

Step 3: Create a header file using javah. `javah linkCheck` creates `linkCheck.h`.

Step 4: Create a C stubs file using javah. `javah -stubs linkCheck` creates `linkCheck.c`.

Step 5: Write the actual native method. Figure 3 shows the `NMisLink.c` file. Note that the C function signature exactly matches the prototype in the header file created in step 3.

Step 6: Create the AIX shared library. Three tasks in this step include the following:

- Compile the C code.

```
cc -c NMisLink.c linkCheck.c -I
$JAVA_HOME/include -I
$JAVA_HOME/include/aix_pt
```

- Create the exports file. Figure 4 shows the `linkExample.exp` file.

```
import  java.lang.*;
 class linkCheck
 {
    private native int NMisLink (String fileName);
    static
    {
       System.loadLibrary (“linkExample”);
    }
    public boolean isLink (String fileName)
    {
       if (NMisLink (fileName) == 1) //Call to the native method
          return true;
       else
          return false;
    }
 }
```

Figure 1. linkCheck.java

```
//a “driver” to test calling code implemented on a native method

class driver
{
   public static void main (String [] args)
   {
      linkCheck lc = new linkCheck(); //arg[0] is expected to be a pathname to check

      if (lc.isLink(args[0]) == true)
         System.out.println (“File “ + args[0] + “ IS a link.”);
      else
         System.out.println (“File “ + args[0] + “ is NOT a link.”);
   }
}
```

Figure 2. driver.java

```
#include <stdio.h>
#include <StubPreamble.h>
#include “linkCheck.h”
long linkCheck_NMisLink (struct HlinkCheck *this,
                         struct Hjava_lang_String *javaFileName)
{
   char c_string[PATH_MAX+1];
   struct stat stat_buf;
   int rc;
   javaString2CString(javaFileName, c_string, PATH_MAX);
   rc = lstat (c_string, &stat_buf);
   if (rc == 0 && (stat_buf.st_mode & S_IFMT) == S_IFLNK)
      return 1;     /* IS a link  */
   else
      return 0;     /* NOT a link */
}
```

Figure 3. NMisLink.c

```
#!
linkCheck_NMisLink
Java_linkCheck_NMisLink_stub
```

Figure 4. linkExample.exp

```
ld -o liblinkExample.so NMisLink.o
   -bM:SRE -bnoentry
   -bI:$JAVA_HOME/include/java.exp
   -L /usr/lib/threads
   -lm -lpthreads -lc_r /usr/lib/libc.a
   -bE:linkExample.exp
```

Figure 5. LiblinkExample.so

- Build the shared library using the example in Figure 5. This will create `liblinkExample.so`.

Now, `LD_LIBRARY_PATH` is exported to include the current directory. The program can be run by `Java driver /etc/init`, which shows the following:

```
File /etc/init IS a link.
```

This example is invoked as a Java application. The code for a Java applet would be similar, but the Java code would not have a main method. Instead, `driver.java` code would be encapsulated in the driver class without main wrapper code.

The example shows good programming practice in the Java method call since the native method (in C) was accessed through a Java method that encapsulated the native method. The private method type was used to insulate the implementation from the end user (programmer), similar to standard object-oriented programming.

Java 1.1 JNI and Future Interoperability

Java 1.1 is in a beta state as this article is being published. A new interface has been proposed and implemented in Java 1.1. The new interface, Java Native Interface (JNI), is designed to satisfy the same requirements as the existing interface. JavaSoft™ considered several interfaces in order to design a best-of-breed consolidated interface. Namely, the existing designs for the following were integrated into JNI:

- Java native method interface from Sun®
- Java Runtime Interface (JRI) from Netscape
- Raw Native Interface (RNI) from Microsoft®

A variety of reasons account for the changes, including efficiency, portability, COM interoperability, and functionality. The new design allows more flexibility. It also shows the openness of JavaSoft to consider current limitations and other models with the goal of delivering technology that can grow over time.

The new Java 1.1 implementation will still execute existing native method interfaces (created under the Java 1.0 specification). When Java 1.1 becomes formalized, Sun is recommending that new interfaces be written using the new specification. The draft specification can be found on the JavaSoft home page under developer news under Java 1.1. It is more complex than the current mechanism, but this complexity also allows for more flexibility.

This article outlined common Java programming scenarios. The most prevalent model is connecting existing C (or C++) applications to Java applets and applications. Java native method interfaces can be used to accomplish this scenario. Required steps for programming interfaces were demonstrated through a simple example.

A new Java JNI method has been introduced for Java 1.1. This evolution of Java to incorporate new mechanisms is backward compatible and shows both the openness of the technology and the rapid evolution that Java is undergoing. IBM is delivering a robust Java-compatible port of Java JDK on AIX. Java on AIX is being used in a variety of programming contexts, including the scenario outlined in this article.

Jeff Jilg, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Dr. Jilg currently works in AIX System Architecture on leading-edge technology. His recent work areas include systems management, object-oriented tools, and the Internet. His current responsibilities include release architecture, Java on AIX, Internet integration, and the AIX Bonus Pack. His PhD in Computer Science from Texas A&M complements his master's degree in the same field from the University of Texas at El Paso.*

John Dodson, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Dodson is currently working in AIX Development on the Common Desktop Environment (CDE) and various Java development projects. His current assignments and responsibilities include systems management, Java development, providing JDK on AIX, and CDE development and support. He has a BS in Computer Science from Brigham Young University and is currently pursuing an MBA at Saint Edwards University.*