Title: **Calling C Library Routines from Java**
Using the Java Native Interface

Summary: This paper presents a technique for calling C library routines directly from Java, saving you the trouble of rewriting code in Java and gaining portability via Java Virtual Machines.

The NAG C Library from the Numerical Algorithms Group (www.nag.com) is a mathematical and statistical library containing routines for linear algebra, optimization, quadrature, differential equations, regression analysis, and time-series analysis. Although written in C, the library's functionality can be accessed from other languages. On PCs, DLL versions of the library can be exploited in many ways, including calling from Microsoft Excel, Visual Basic, or Borland Delphi. Consequently, NAG C Library users often ask if they can call it from Java.

One way to perform numerical computations is by creating Java classes that implement the required functionality. However, this is often difficult and time consuming. In this article, I present a technique for calling C Library routines directly from Java.

Apart from avoiding rewriting numerical code in Java, using existing C code has another advantage. Java is portable -- compiled programs run on any machine with a Java Virtual Machine (VM). To accomplish this, the Java compiler does not compile to a machine-dependent assembler, but to machine-independent bytecode that is interpreted at run time by the VM. Although the interpreter is efficient, no interpreted program runs as fast as programs compiled to assembler code. For applications that perform CPU-intensive operations on large amounts of data, this can be significant, and moving those operations from Java into compiled libraries can cut execution time.

The Java Native Interface

To access a library routine from a Java program, I use the Java SDK's Java Native Interface (JNI), which gives compile- and run-time support for calling native code from a Java program. By native code, I mean code that is not Java, typically C or C++; here I assume C.

At compile time, JNI defines how Java data types correspond to C data types. C programs get this information from JNI header files that come with the Java SDK. Javah, a tool that comes with the SDK, creates application-specific header files that help eliminate programming errors in communication between Java and C routines. At run time, JNI lets Java objects be passed to C code, and lets C code access Java properties and methods. Thus, C code can set properties of Java classes, making it possible to call Java methods from C.

For this article, I used the Java 2 SDK 1.4.1 on: a Linux machine running Red Hat Linux 8.0 with the GNU C

Run-Time Error Messages

If you get a Java error message, such as "interface library CJavaInterface cannot be found" or "*bessely0* function cannot be found" when running your program, you may need to set an environment variable to tell the system where to look. Environment variable names are operating-system dependent.

- Linux machines. The environment variable LD_LIBRARY_PATH is a colon-separated list of directories to be searched for libCJavaInterface.so. For example, if you use the C shell and your library is in /home, the command *setenv LD_LIBRARY_PATH /home* ensures that the current directory /home gets searched.
- Other UNIX machines. The environment variable is likely discussed in the man page for the *ld* command, or in a command referenced by that man page. Common names include LD_LIBRARY_ PATH and SHLIB_PATH.
- Microsoft Windows. There is no separate environment variable used for locating DLLs. The PATH variable is searched, so CJavaInterface.dll must reside somewhere in your path.

—M.P.

compiler, gcc 3.2; a Sun machine running Solaris 8.0 with the Sun Workshop 6 C compiler, cc 5.2; and a PC running Windows 2000 with Visual C++ 5.0. Working on UNIX platforms other than Sun or Linux is similar, the main differences being in the location of Java include files and the method of creating a shared object (dynamic library).

Using Native Code Libraries

When Java programs call native methods (C functions, for instance), extra arguments are prepended to the argument list of the called C function. These arguments give the C code a handle onto Java methods and properties. This may not be a problem if you control and may modify the C code of the function to call—you can add the extra arguments to your native function. However, to call a function that's in object code, you must use an intermediate shared library (UNIX) or DLL (Windows). This library is the interface between the Java code and library code. Your interface library must make its own calls to the underlying library, and send the results back to Java; see Figure 1.

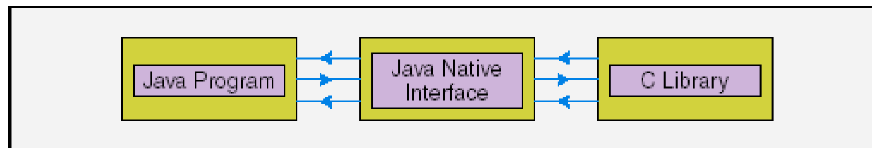

Figure 1: *JNI as a link between Java and a native library.*

Implementing calls from Java to native functions is a three-step process:

1. Write a declaration in Java for the native method. This declaration includes the keyword *native* to signify to the Java compiler that it is implemented externally.
2. Create a header file for use by the native (C) code. This header file contains the declaration of the native method as viewed by the C compiler. It includes the extra arguments required for the C function to access Java methods and properties, and has argument types defined in terms of standard C types.
3. Implement the native method in C. This function uses the header file in Step 2, makes calls to library functions it needs (possibly back to Java methods), and returns results to Java. This C code is compiled to build a shareable library.

After the native shareable library is built, Java code that uses it is still machine-independent even though the native library is not. Thus, you must build the library on all platforms the Java program runs on, although you don't need to edit or rebuild the Java code.

A Simple Function of One Argument

In the simplest example of creating an interface library, I call a function with only one argument and one return value—the $Y_0(x)$ Bessel function routine from the Standard C Math Library. (The various Bessel functions, of which this is one, are named for 18th-century German astronomer Friedrich Wilhelm Bessel.)

The function prototype from the C <math.h> header file is *double y0(double x);*. The Java program declares the method *private native double besselyO(double x);*, which has a *double* argument, returning *double*. The *native* keyword tells the Java compiler that the method is implemented outside Java.

You must build a shared library to contain the native method that interfaces between Java and the math library function $Y_0(x)$. The Java program loads that interface library, CJavaInterface, by passing its name

to a Java *System.LoadLibrary( )* call. Even though the interface library may have a different name (depending on the OS), *LoadLibrary* sorts it out. For example, under Linux or Solaris, the Java *System.loadLibrary("CJavaInterface");* searches for a library named “libCJavaInterface.so.” But under Windows, it searches for “CJavaInterface. dll.”

Listing One *(see page 8)* is the Java program Bessel.java, including the native method declaration and *loadLibrary* call. The *System.loadLibrary* call is placed inside a static initializer so that it is executed when the class gets loaded by Java. The main program gets a value of *x* from the command line, and calls the native method using that argument and nine other arguments derived from it. Compile the Java program with the command *javac Bessel.java.* If all goes well, the compiler produces a file named “Bessel.class.” Note: all source code mentioned in this article is available for download at http://www.nag.com/IndustryArticles/pontjavafiles.zip. With the source code are scripts that you can use to compile and run all the examples under Windows (.bat files) or Linux (.sh files).

Once Bessel.java is compiled, use the Java SDK’s javah tool to create a header file that the C compiler can use. The command *javah -jni Bessel* produces the file Bessel.h; see Listing Two (*page 9*). The header file includes <jni.h> (which comes with the Java SDK); javah extracted this declaration of the native function for use by the C program:

```
JNIEXPORT jdouble JNICALL
      Java_Bessel_bessely0
            (JNIEnv *, jobject, jdouble);
```

The name *Java_Bessel_bessely0* shows the Java class in which it is declared, as well as the name *bessely0* chosen for the native function. The macros *JNIEXPORT* and *JNICALL* are defined via <jni.h> and affect the Windows calling convention (on UNIX, the macros disappear). The types *JNIEnv*, *jobject*, and *jdouble* are also defined via <jni.h>, in terms of machine-dependent C types. For example, the type *jdouble* is the Java *double* type, which equates to the C *double* type.

From the C point of view, the first two arguments, of types *JNIEnv** and *jobject*, give C code access to the Java environment. In this case, the third argument—the argument *x* of the $Y_0(x)$ Bessel function—passes to the C Math Library.

Once Bessel.h is created, you can write a C implementation of *Java_Bessel_bessely0*; see Listing Three (*page 9*). You can write any C code you like here but, in this case, all I do is to call the Y_0 function from the Standard Math Library.

Building the Shareable Library or DLL

Building the shareable library or DLL is operating-system dependent.

To compile under Linux, assuming Java is installed in /usr/java/j2sdk1.4.0_02 (if yours is elsewhere, modify accordingly), you first use the GNU C compiler to compile BesselImp.c:

```
gcc -c -I/usr/java/j2sdk1.4.0_02/include
      -I/usr/java/j2sdk1.4.0_02/include/linux BesselImp.c
```

The *-I* switches tell the C preprocessor where to look for header files. The first directory, */usr/java/j2sdk1.4.0_02/include*, locates the jni.h header file. The second directory, */usr/java/j2sdk1.4.0_02/include/ linux*, is machine dependent and used by jni.h to find type definitions. The *linux* element of this name changes if you use another operating system.

When BesselImp.c compiles, turn it into a shareable object using *ld -G BesselImp.o -o libCJavaInterface.so -lm -lc -lpthread.* The *-G* flag means "create a shareable object." The *-o* flag names the shareable library as libCJavaInterface.so, the name needed by the *loadLibrary( )* call in the Java code. Finally, the *-lm*, *-lc*, and *-lpthread* flags ensure that you link with required system math and run-time libraries.

Under Windows, assuming Java is installed in c:\j2sdk1.4.0_01 (modify accordingly if not), compile and build the DLL in one step:

```
cl /Ic:\j2sdk1.4.1_01\include
      /Ic:\j2sdk1.4.1_01\include\win32
/Gz /LD BesselImp.c /FeCJavaInterface.dll
```

As with UNIX, the two */I* switches tell the C compiler where to look for header files.

Without the */Gz* compiler option (meaning "use the _ _*stdcall* calling convention"), the code may compile and link— and even start running—but eventually may cause an access violation. The */LD* flag means "build a DLL." The */Fe* flag names the output file as CJavaInterface.dll. You can run the program using *java Bessel 1.0.* Listing Four (*page 9*) is the expected output.

Calling a C Function with Array Arguments

Most native functions return more than a single value. To illustrate, I use a vector addition function to show how to pass array arguments between Java and C. I don't call any auxiliary library functions, but code the entire routine from scratch.

The file VectorAdd.java (also available online) is the Java program in which the declaration of the native function *private native int vectoradd(int n, double[ ] a,double[ ] b, double[ ] c);* includes argument *n*, the length of the vectors to be added, along with input vectors *a* and *b,* and output vector *c*.

Compile the Java program with the command *javac VectorAdd.java*, then use *javah -jni VectorAdd* to create a C header file. The generated header file, VectorAdd.h, contains the function prototype:

```
JNIEXPORT jint JNICALL
      Java_VectorAdd_vectoradd
            (JNIEnv *, jobject, jint, jdoubleArray, jdoubleArray,
            jdoubleArray);
```

From the C point of view, the function has extra arguments of type *JNIEnv* * and *jobject*. This time, they are needed; see VectorAddImp.c (*available for download at http://www.nag.com/IndustryArticles/pontjavafiles.zip*). You cannot access the elements of array arguments *a, b,* and *c* directly because they are Java-style (not C-style) arrays of type *jdoubleArray.* Trying to access the array elements directly leads to catastrophe. Instead, convert them to C-style double arrays, using the JNI function *GetDoubleArrayElements*. Header file jni.h declares this function:

```
jdouble * (JNICALL *GetDoubleArrayElements)
(JNIEnv *env, jdoubleArray array, jboolean *isCopy);
```

GetDoubleArrayElements is accessed through the *JNIEnv* pointer, **env*. Given the array of type *jdoubleArray*, it returns a pointer to an array of elements of type *jdouble,* which can be safely manipulated by C. The output argument *isCopy* tells you whether Java made a copy of the array, or passed a pointer

© The Numerical Algorithms Group

to the elements in situ.

The C program, therefore, makes three calls of *GetDoubleArrayElements*, one for each array argument. It adds each element of array *a* to array *b,* putting the results in array *c*. Then it tells Java that it is finished with the array pointers using three calls of *ReleaseDoubleArrayElements*, declared in jni.h as:

```
void (JNICALL *ReleaseDoubleArrayElements)
(JNIEnv *env, jdoubleArray array, jdouble *elems, jint mode);
```

This ensures that results get copied back to the appropriate Java arrays, and that Java garbage collection can work properly.

Compile this code into an interface library in a similar way to the first example. Under Windows:

```
cl /Ic:\j2sdk1.4.1_01\include
      /Ic:\j2sdk1.4.1_01\include\win32
            /Gz /LD VectorAddImp.c
                  /FeCJavaInterface.dll
```

Run the program with *java VectorAdd.*

Calling a C Function with a Function Argument

Assume that you have a C library containing a root-finding function *rootfinder*, with prototype *double rootfinder(double (* f)(double x), double a, double b, int *nits, int *fail);*. The function is designed to find a simple root of an algebraic equation —a point *x* where the function *f(x)* evaluates to zero. For example, *x=3* is a root of the function *f(x)=x*x–5*x+6*. The first argument of *rootfinder* is a pointer to the function of which a root is sought. The arguments *a* and *b* are user-supplied points that the caller asserts are bounds on the root, such that *f(a)* has the opposite sign to *f(b).*

If the search for a root is successful, *rootfinder* returns the root, along with the number of iterations of the algorithm that were required to find the root (via argument *nits*). The output argument *fail* returns as zero if a root was found, and nonzero otherwise.

Because you probably don't have a library containing the *rootfinder* function, rootlib.c (*available for download at http://www.nag.com/IndustryArticles/pontjavafiles.zip*) contains a simple bisection method implementation of *rootfinder*. Under Windows, compile it into rootlib.dll (cl /I. /Gz /LD rootlib.c /Ferootlib.dll). The file RootFinder.java (*also available for download*) calls the *rootfinder* function. In the Java program, I declare the *rootfinder* function as a method: *private native int rootfinder(String funName, double a, double b);*. I use the *int* return value to send back any error code from the native function.

Since it isn't possible to pass a function argument directly from Java to C, I pass the name of a function via the *String* argument *funName*. Also, the Java declaration does not contain any of the output arguments of the C function *rootfinder*. Instead, I use a different way to pass the information that the output arguments contain back to Java.

Compile the Java program and generate a C header file with the commands *javac RootFinder.java* and *javah -jni RootFinder.*

RootFinderImp.c is the C interface library. *(RootFinderImp.c is available for download at http://www.nag.com/IndustryArticles/pontjavafiles.zip).*

The function *Java_RootFinder_rootfinder* is the C implementation of the Java-declared method *rootfinder*. Since you cannot pass the Java method that evaluates *f(x)* directly to the rootlib C Library function *rootfinder*, you need to wrap it in a C function such as *rootFun*. Its prototype is *double rootFun(double x);* and it has the argument type and return type required by the *rootlib* library function. Inside *rootFun,* I only call the Java method to evaluate the function. The trick is in knowing how to make this call to Java.

You do this using the JNI function *CallDoubleMethod*, which is declared in jni.h (there are similar functions—*CallVoidMethod, CallIntMethod*, and others—for methods with different return types). *Call-DoubleMethod* needs several arguments, including the *JNIEnv* pointer argument *env* and the Java object argument, both of which were passed to *Java_RootFinder_ rootfinder*. It also needs the argument *methodID*, which is the ID of the Java method to be called. These arguments are known (or can be obtained) by *Java_RootFinder_rootfinder*, but are not directly known by the function *rootFun*. Instead, I give these arguments to *rootFun* via global variables, which I declare like this in C:

```
JNIEnv *globalJavaEnv;
jobject globalJavaObject;
jmethodID globalMid;
```

Because these variables are global, they can be accessed by both *Java_RootFinder_ rootfinder* and *rootFun*.

Besides these arguments, *rootFun* also passes to *CallDoubleMethod* the actual arguments that the Java method needs to evaluate the function *f(x). CallDoubleMethod* can accept any number of these arguments, but in this case there is only argument *x*.

I could have written the evaluation code in C (and would not have needed to use *CallDoubleMethod* and the routines associated with it) instead of calling the Java method from *rootFun* to evaluate the function *f(x).* However, an advantage to the method I use is that once the interface library is built, you need never rebuild it even if the evaluation function changes—just supply a different Java evaluation function.

Java_RootFinder_rootfinder first copies its arguments *env* and *obj* to global variables *globalJavaEnv* and *globalJavaObject*. Next, take the name of the Java method passed as the *jstring* argument *funName* and convert it into a method ID. Use the JNI function *GetStringUTFChars* to convert the *jstring* into a C *char* pointer named "*functionName"* because the *jstring* cannot be safely accessed directly. Then the JNI functions *GetObjectClass* and *GetMethodID* are used to get hold of the method ID of the Java evaluation function:

```
functionName = (*env)->GetStringUTFChars (env, funName, 0);
...
cls = (*env)->GetObjectClass(env, obj);
globalMid = (*env)->GetMethodID(env, cls, functionName, "(D)D");
```

(Note that in the example program, the evaluation function is either the method *myFunction* or *myFunction2.*) *GetMethodID's* second argument, of type *jclass*, is the class containing the method; the third argument is the name of the method, and the fourth argument is the signature of the method. In this case, the signature (*D)D* means a method with one *double* argument that returns a value of type *double*.

Once you have the method ID to be used by *rootFun*, you no longer need the C string *functionName*, so free it via a call to JNI function *ReleaseStringUTFChars* to avoid memory leaks. At this point, you have everything you need to call the C rootlib library function *rootfinder*. Figure 2 illustrates what happens at run time.

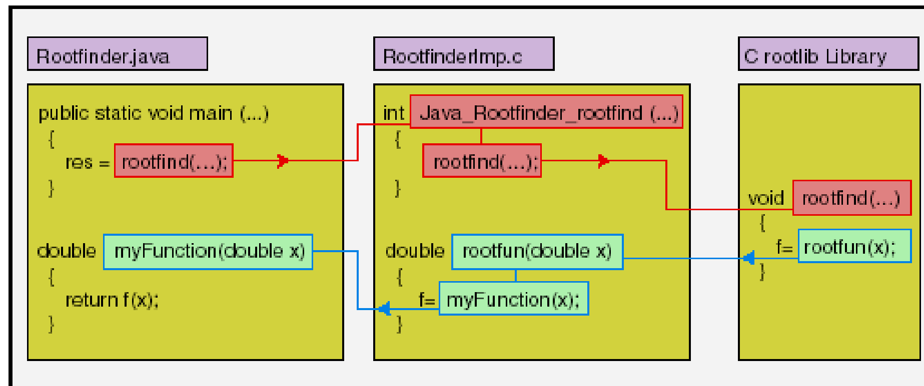

Figure 2: *At run time, Java calls C interface, calls C rootlib library, calls C interface, calls Java.*

Returning Results to Java

After returning from the C rootlib library, results must be returned to Java using JNI functions. Notice that the Java class contains the properties (variables) *result* and *niterations*.

Given the name and signature of one of these variables, the JNI function *GetFieldID* returns its "field ID," which you can pass to another JNI function to set its value. For example, the function *SetDoubleField* sets the value of a *double* property given its field ID. The lines

```
fid = (*env)->GetFieldID(env, cls, "result", "D");
/* Set the result value via the ID */
(*env)->SetDoubleField(env, obj, fid, result);
```

get the field ID of property *result*, and set its value to that of the variable contained in the C code *result*. Similarly, the lines:

```
fid = (*env)->GetFieldID(env, cls, "niterations", "I");
(*env)->SetIntField(env, obj, fid, nits);
```

get the field ID of int property *niterations*, with signature *I*, and set its value to *nits*, the number of iterations taken by *rootfinder* in searching for the root.

Conclusion

With the techniques presented here, you can pass information between C and Java. Furthermore, you should be able to reuse some of the source code presented here to create interfaces to your own routines, written in C or in a precompiled library.

By Mick Pont. *Mick works in the Development Division of the Numerical Algorithms Group and can be contacted at mick@nag.co.uk.*

Originally published by Dr. Dobb's Journal (www.ddj.com), July 2003.

Numerical Algorithms Group
www.nag.com
infodesk@nag.com

Listing One

```
// The Bessel.java file
public class Bessel
{
  // Declaration of the Native (C) function
  private native double bessely0(double x);
  static
    {
      // The runtime system executes a class's static initializer
      // when it loads the class.
      System.loadLibrary("CJavaInterface");
    }
  // The main program
  public static void main(String[] args)
    {
      double x, y;
      int i;
      /* Check that we've been given an argument */
      if (args.length != 1)
        {
          System.out.println("Usage: java Bessel x");
          System.out.println("  Computes Y0 Bessel function of argument x");
          System.exit(1);
        }
      // Create an object of class Bessel
      Bessel bess = new Bessel();
      /* Convert the command line argument to a double */
      x = new Double(args[0]).doubleValue();
      System.out.println();
      System.out.println("Calls of Y0 Bessel function routine bessely0");
      for (i = 0; i < 10; i++)
        {
          /* Call method bessely0 of object bess */
          y = bess.bessely0(x);
          System.out.println("Y0(" + x + ") is " + y);
          /* Increase x and repeat */
          x = x + 0.25;
        }
    }
}
```

Listing Two

```
/* The Bessel.h file generated from the Bessel class by the javah tool. */
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Bessel */
#ifndef _Included_Bessel
#define _Included_Bessel
#ifdef __cplusplus
extern "C" {
#endif
/* Class:     Bessel
 * Method:    bessely0
 * Signature: (D)D
 */
JNIEXPORT jdouble JNICALL Java_Bessel_bessely0
  (JNIEnv *, jobject, jdouble);
#ifdef __cplusplus
}
#endif
#endif
```

Listing Three

```
/* The BesselImp.c file, which implements the native function */
#include <jni.h>      /* Java Native Interface headers */
#include "Bessel.h"   /* Auto-generated header created by javah -jni */
#include <math.h>     /* Include math.h for the prototype of function y0 */

/* Our C definition of the function bessely0 declared in Bessel.java */
JNIEXPORT jdouble JNICALL
Java_Bessel_bessely0(JNIEnv *env, jobject obj, jdouble x)
{
  double y;
  /* Call Y0(x) Bessel function from standard C mathematical library */
  y = y0(x);
  return y;
}
```

Listing Four

```
// Output when running the Java Bessel program
Calls of Y0 Bessel function routine bessely0
Y0(1.0) is 0.08825696421567694
Y0(1.25) is 0.2582168515945407
Y0(1.5) is 0.38244892379775886
Y0(1.75) is 0.465492628646906
Y0(2.0) is 0.5103756726497451
Y0(2.25) is 0.5200647624572782
Y0(2.5) is 0.4980703596152316
Y0(2.75) is 0.4486587215691318
Y0(3.0) is 0.3768500100127903
Y0(3.25) is 0.2882869026730869
```