Using The Java Native Interface

by S. Fouzi Husaini

Abstract:

The Java Native Interface (JNI) comes with the standard Java Development Kit (JDK) from Sun Microsystems. It permits Java programmers to integrate native code (currently C and C++) into their Java applications. This article will focus on how to make use of the JNI and will provide a few examples illustrating the usefulness of this feature. Although a native method system was included with the JDK 1.0 release, this article is concerned with the JDK 1.1 JNI which has several new features, and is much cleaner than the previous release. Also, the examples given will be specific to JDK 1.1 installed on the Solaris Operating System.

Motivation

Recently I found myself in the position of needing to place a user-friendly Java Graphical User Interface (GUI) on a very bland, keyboard input menu-driven program that I had written in C. The C program was just a wrapper which made calls to a very rich C library. Having had some experience with Java, I knew the basics of creating a simple GUI (creating a frame and adding panels to the frame consisting of buttons, text fields etc.). I needed to find a way I could interface the C program and the Java GUI. I was certainly not about to rewrite the meat of the C program just for the sake of the interface. Then I came to know about the Java Native Interface, or JNI, which allows Java programmers to make calls to native code.

What is Different About the Native Interface From JDK 1.0 to 1.1?

The biggest change which came with JDK 1.1 is the ability for a native application to interface with the Java Virtual Machine (JVM). In the native method system from JDK 1.0, an application programmer could call native code but the native code could not invoke methods from the Java application/applet. Not being able to work with the JVM meant that the *interface* was very one-sided. This placed restrictions on anyone doing native interface programming which required more from the Java side than just calling out native methods [1]. Sun solved this problem in JDK 1.1 with the introduction of the JNI. Using the JNI, a programmer can have the Java application make a native call, then have the native method make a call back to a Java method, and so on. The JNI also supports invocation of the JVM. What this means is that a programmer can start up the JVM, call methods and create Java objects all within native code. This is done by running the native code executable after compiling and linking the native code, the libraries shipped with the JDK and the JNI function library.

The other major change deals with portability. With the older system, inconsistencies occurred when moving native method code to different machines because of platform dependent features such as data sizes [1]. With

the current version, the creators of the JNI have provided for a clean, uniform definition of all data sizes, so that the application programmer does not need to make any source level changes during a port [2].

Rather than writing Hello World programs (which can be very beneficial), lets put some of the features of the JNI to use for an actual problem.

Problem Statement

A powerful program which simulates fluid pressure along a pipeline is used by an engineering team to perform numerical analysis and design analysis upon the structure. Unfortunately, to input data into the simulator one must edit text files and manually perform repetitive, and archaic commands. Rewriting this legacy code is not an option because it is not only functional, but is very complex, and makes use of many other legacy code libraries. A front end GUI is desired for the simulator. This GUI will be written in Java and will interface with the simulator which was written in C.

Command Line Arguments Example

First, let us assume that the pipeline simulator accepts command line input for various initialization settings. Our first task is to find a way of having those inputs sent to the simulator. Here is a sample program which takes in command line arguments given to a Java program, and makes a native call with those arguments.

- First write the Java Program called Arguments.java as below. (The code can be found online at http://www.csc.calpoly.edu/~fouzi/crossroads/Arguements.java)

```
class Arguments
{
   private native void setArgs (String[] javaArgs);
   public static void main (String args[])
   {
      Arguments A = new Arguments();
      newArgs[] = A.setArgs(args);
   }
   static
   {
      System.loadLibrary("MyArgs");
   }
}
```

There are 3 points of interest here.

- Notice the keyword *native* appearing before the declaration of the *setArgs* method. Also notice that *setArgs* ends with a semi-colon. When the keyword *native* appears before a method return type, this

lets the Java compiler know that this method will be defined later on as native code. The semi-colon at the end of the method makes it look a lot like a prototype in C. Surprisingly enough, it is the same idea.

- The *setArgs* method is referenced just like any other method. The class name and then the period followed by the method name. Nothing special at all about this.
- Last, there appears a static block at the end of this class. The JNI works by compiling native code into a shared library (DLL for Windows 95/NT) and then loading that library within the class where the native code is first declared. In UNIX, shared library files usually have the prefix *lib* and the suffix *.so*. The JNI assumes this to be the case, so when the program runs it will try to load a library by the name **lib***MyArgs***.so**

Compiling

The Java program is compiled just like any other non-JNI Java program. For our example we would do the following:

```
javac Arguments.java
```

Header File

Before jumping into writing the implementation for the native method, we can take advantage of the *javah* utility. This will create a header file that can be used as a guide when writing the implementation of the method in native code. To create the header file we would do the following:

```
javah -jni Arguments
```

This will create a file named Arguments.h which can be found online at http://www.csc.calpoly.edu/~fouzi/crossroads/Arguments.h.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Arguments */

#ifndef _Included_Arguments
#define _Included_Arguments
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     Arguments
 * Method:    setArgs
 * Signature: ([Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL Java_Arguments_setArgs
  (JNIEnv *, jobject, jobjectArray);
```

```
#ifdef __cplusplus
}
#endif
#endif
```

- The first two parameters of *Java_Arguments_setArgs* appear before every native method. The first is a pointer to the object, and the second is the object itself. Later on when we look into the implementation, it will become clearer how and why these are used.
- The third parameter of type *jobjectArray* is what is supposed to represent the *String[]* object we intend to pass to the pipeline simulator. The JNI has converted this into a generic type of an object array. Since it would be ridiculous to have a type for every possible Array type, the developers of the JNI included the primitive data types. For all other objects there is the *jobject*, and for an array of any object type, the *jobjectArray*.
- Notice how our method name *setArgs* now has a much longer name. All native methods are prefixed with *Java_ClassName_*. From a very painful experience I recommend to NEVER use underscores as a naming convention for your native methods. This throws off the parser, and things just plain won't work.

Native Method Implementation

Now we are ready to begin writing the implementation for our *setArgs* native method. Since we are receiving a *Strings[]* object we need to convert that into an *argv, argc* data type to pass to the pipeline simulator. We must also keep in mind the fact that Java arguments begin with the first argument provided, while C arguments begin with the program name itself. Here is an implementation of Aguments.c, also found online at http://www.csc.calpoly.edu/~fouzi/crossroads/Arguments.c in which the *jobjectArray* is converted into an argv, and the arguments are printed to the screen.

```
#include "/usr/local/java/include/jni.h"
#include "Arguments.h"
#include <stdio.h>

JNIEXPORT void JNICALL
Java_Arguments_setArgs (JNIEnv *jenv, jobject job, jobjectArray oarr)
{
   /* obtain the size the array with a call to the JNI function
      GetArrayLength()                                          */
   jsize argc = (*jenv)->GetArrayLength(jenv, oarr);

   /* Declare a char array for argv */
   char const* argv[128];
   int i;

   for (i = 1; i < argc + 1; i++)
```

```
   {
      /* obtain the current object from the object array */
      jobject myObject = (*jenv)->GetObjectArrayElement(jenv, oarr, i-1);

      /* Convert the object just obtained into a String */
      const char *str = (*jenv)->GetStringUTFChars(jenv,myObject,0);

      /* Build the argv array */
      argv[i] = str;

      /* Free up memory to prevent memory leaks */
      (*jenv)->ReleaseStringUTFChars(jenv, myObject, str);

      /* print the argv array to the screen */
      printf ("argv[%i] = %s\n",i,argv[i]);
   }

  /* Increment argc to adjust the difference between Java and C arguments
      argc++;

      Call a pipeline simulator function which uses command line arguments
      initializePipeline(argc,argv);
  */

   return;
}
```

- The very first line of our implementation makes a JNI call to *GetArrayLength*. Here we can see how the jenv pointer is used to make JNI function calls. We pass a pointer to the object, and the object Array to *GetArrayLength* which then returns something of type *jsize*, which is simply typedefed as a *jint*.
- To extract the data from within the array, we call the JNI function *GetObjectArrayElements*. Passing it the pointer to the object, the object array, and an index into the object array, it returns something of type jobject. Since we know what type of object array this is that we are dealing with *String* we can treat it as such.
- Now to convert the object which we have stored in *myObject* we call the JNI function *GetStringUTFChars*. Passing it, again the pointer to the object, the object containing the string, and a boolean value for copying purposes, it returns *const char**. So we simply add this to our *argv* array.
- To prevent from having memory leaks, or other performance problems, the JNI provides us with a function that frees up memory that gets used up when getting strings. The *ReleaseUTFChars* serves this purpose, and it should be used every time you make a call to *GetStringUTFChars*.
- The commented section of the code, is there if we really had this pipeline simulator. Supposing there was an initialization function called *initializePipeline* we could then send it the argc, and argv values.

Compiling

To compile the native implementation we need to create a shared library file (recall the static block in the Java file). Assuming that Java is installed in /usr/local/java our compile line will look like this:

```
cc -G -I/usr/local/java/include -I/usr/local/java/include/solaris
        Arguments.c -o libMyArgs.so
```

NOTE: Substituting *gcc* for *cc* will also work.

Before running the program, you must make sure that the shared library file is placed in a directory which is contained within the LD_LIBRARY_PATH environment variable. Otherwise it will not run, and will result in an error.

Running the Program

```
java Arguments firstargument secondArgument thirdargument
```

Program Output

```
argv[1] = firstargument
argv[2] = secondArgument
argv[3] = thirdargument
```

Calling Java Methods

To illustrate the usefulness of the new feature in the JNI, we will demonstrate a situation where calling a Java method from within native code is desired. Based on some user input which is fed to the simulator via a native call, the next method which is invoked depends on the results of the processed data. The only way to make this decision is to have some sort of two way communication with the native method.

Java Program

Here we have an example of a compressor in Compressor.java within the pipeline. The Java GUI is supposed to draw either an increase or a decrease, depending upon the value set for the compression.

```
class Compressor
{
   private native void setCompression(double pressure);
   public static void main (String args[])
   {
      Compressor C = new Compressor();
      double pressure = 5.0;
      C.setCompression(pressure);
```

```
   }

   public void drawCompressionIncrease(double amount)
   {
      System.out.println ("Increase: Amount = " + amount);
   }
   public void drawCompressionDecrease(double amount)
   {
      System.out.println ("Decrease: Amount = " + amount);
   }
   static
   {
      System.loadLibrary("MyCompressor");
   }
}
```

- Just like the previous example, we have a native method here as well. This native method *setCompression* is expected to make a call to the pipeline simulator and set the compression rate. For this we pass this method a double.
- There also exist two other methods here, *drawCompressionIncrease* and *drawCompressionDecrease*. In a real situation, these two methods would be doing GUI related drawing things, but here we are just printing the value that they are called with.

Compiling

```
javac Compressor.java
```

Header File and Javap

Header File

We perform the same process for obtaining the header file Compressor.h (found at http://www.csc.calpoly.edu/~fouzi/crossroads/Compressor.h).

```
javah -jni Compressor
```

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Compressor */

#ifndef _Included_Compressor
#define _Included_Compressor
#ifdef __cplusplus
extern "C" {
```

```
#endif
/*
 * Class:     Compressor
 * Method:    setCompression
 * Signature: (D)V
 */
JNIEXPORT void JNICALL Java_Compressor_setCompression
  (JNIEnv *, jobject, jdouble);

#ifdef __cplusplus
}
#endif
#endif
```

Method Signatures

In order to make a method call from within native code, we need to know the method signatures of the methods we intend on calling. To do so, we can use the *javap* utility.

```
javap -s -p Compressor
```

The -s flag tells javap to output signatures instead of Java types, and the -p forces that private methods be included as well [2]. This gives us the following:

```
Compiled from Compressor.java
synchronized class Compressor extends java.lang.Object
    /* ACC_SUPER bit set */
{
    private native void setCompression(double);
        /*   (D)V   */
    public static void main(java.lang.String[]);
        /*   ([Ljava/lang/String;)V   */
    public void drawCompressionIncrease(double);
        /*   (D)V   */
    public void drawCompressionDecrease(double);
        /*   (D)V   */
    Compressor();
        /*   ()V   */
    static static {};
        /*   ()V   */
}
```

- The method signatures for *drawCompressionIncrease* and *drawCompressionDecrease* are both *"(D)V"*. These are simply symbols from JVM data types. The *D* represents a *double* and the *V* indicates that

the method has a return type of *void*.

Native Method Implementation

Now with both the header file and the function signatures, we can begin to write the native implementation of *setCompression*. Find this Compressor.c file online at http://www.csc.calpoly.edu/~fouzi/crossroads/Compressor.c

```
#include "/usr/local/java/include/jni.h"
#include "Compressor.h"
#include <stdio.h>

JNIEXPORT void JNICALL
Java_Compressor_setCompression (JNIEnv *jenv, jobject job, jdouble amount)
{
   /* obtain the class of the "this" object */
   jclass myClass = (*jenv)->GetObjectClass(jenv, job);

   /* obtain the method ID of the drawCompressionIncrease Java Method */
   jmethodID increaseMID = (*jenv)->GetMethodID(jenv,myClass,
                            "drawCompressionIncrease", "(D)V");

   /* obtain the method ID of the drawCompressionDecrease Java Method */
   jmethodID decreaseMID = (*jenv)->GetMethodID(jenv,myClass,
                           "drawCompressionDecrease", "(D)V");
   double change;

   /* set the change to a value returned by a pipeline simulator call
      change = pipeline_set_compression(amount);                    */

  /* method invocation */

   change = 1.0;

   if (change > 0)
   (*jenv)->CallVoidMethod(jenv,job,increaseMID,change);
   else
   (*jenv)->CallVoidMethod(jenv,job,decreaseMID,change);

   return;
}
```

- In order to make a Java method call, the native code must know the class, the method name, and the signature of the method. The class is obtained by calling the JNI function *GetObjectClass* which

returns type jclass.

- The first method we define is for the *drawCompressionIncrease* method. From the *javap* output we determined that the signature for this method is *"(D)V"*. Calling the JNI function *GetMethodID* we can store the method ID into a variable called *increaseMID*. We do the same for *drawCompressionDecrease*
- The commented code, is a function call to the pipeline simulator. For the purpose of this example we comment that code, and hardcode the value of change to 1.0
- The Java methods are then invoked using the JNI function *CallVoidMethod* by passing in the method ID as well as whatever parameters the method requires. We expect the *drawCompressionIncrease* method to be called.

Compiling

Again, we must create a shared library file.

```
cc -G -I/usr/local/java/include
-I/usr/local/java/include/solaris
Compressor.java -o libMyCompressor.so
```

Running the Program

```
java Compressor
```

Program Output

```
Increase: Amount = 1.0
```

Platform Independence

An article written on Java that does not mention platform independence is probably an incomplete article. How does platform independence play into the JNI? Well, with languages like C and C++, software builders find themselves porting their software to various systems. In our example we can assume that the legacy pipeline simulator had been ported to both a UNIX machine such as a Sun Sparc, and a DOS/Windows PC. With some other non-platform independent language we would have been forced to port the GUI along with the legacy code. With Java that is not necessary. Our GUI can simply be moved over to the PC and plugged into that simulator with a few minor modifications made to the shared library. No new code, no hassle!

Conclusion

The examples I have provided above are just a few reasons for why someone would want to use the JNI. The need to interface with legacy code will always be there. By providing programmers with the JNI, Java makes itself more flexible and available to a wider range of applications.

My expectations before looking into the JNI were that it would be a 3 week intensive lesson just trying to figure out how to get it to work. It turned out to be quite the opposite. After reading through the tutorials (http://java.sun.com/docs/books/tutorial/native1.1) available from Sun Microsystems, I was up and running within a few hours. The developers of the JNI have done an excellent job of providing Java programmers with a helpful, and easy to use tool. On the downside though, there is not very much literature out there about the JNI. The tutorial by Beth Stearns at the Sun Microsystems web page [1] is very helpful, but more needs to be written about it. Hopefully in writing this article, I will not only inform some people about the JNI but it will prompt others to do more in-depth research and provide us all with more literature about this great feature of Java.

References

1

Shiffman, Hank. *Java Grows Up: JNI, RMI, Beans & More*. Silicon Graphics Inc. http://www.disordered.org/Java-At-One.html

2

Stearns, Beth. *Integrating Native Code and Java Programs*. Sun Microsystems Inc. http://java.sun.com/docs/books/tutorial/native1.1/index.html

---

Fouzi Husaini is senior level undergraduate in Computer Science at California Polytechnic State University, San Luis Obispo, CA. He interned at Sun Microsystems under the supervision of Software Simulation Engineer, Assana Fard from June till September 1997 where he developed an ASIC verification tool in Java using the JNI.