

Environments and Debugging

James W. Cooper

Sometimes you just can't help yourself and you have to go native. You just can't stand on the ceremony of Java's formal elegance, and you have to wade into the unmapped thickets of native C++ or (ick!) C code. Pointers! Include files! Aaaaah! Well, going native doesn't always mean that you've lost your mind. Sometimes there are perfectly good reasons for it. If there is a large, useful library of code that has already been written and is being maintained in one of these post-Big Iron-age languages, you might want to make use of it by writing a little wrapper to get from Java (aaah!) to C (gaaah!).

The Java Native Interface or JNI is a relatively simple way of making calls to C code. While the C itself will be platform and processor dependent, the code can usually be compiled for all major platforms without much trouble. I'm going to run through how you build a simple JNI interface, not because there aren't good books on this (there are) but in order to show you that you can debug both the Java and the C at the same time. This is a pretty neat trick my colleague Herb Chong showed me, and I thought you might find it useful, too.

In early releases of Java, there was a `getenv` method in the `System` class that allowed you to get the values of environment variables. This has now been deprecated, because not all platforms support environment variables. So in order to hatch a simple-minded but useful example, we'll build a little library that calls the C-language `getenv` function and returns the result to Java.

Now the only convenient way to communicate between C and Java is using the standard primitive types: `int`, `float`, `double`, and `String`. These first three are passed into your C code as variables of type `jint`, `jfloat` and `jdouble`. These are identical to `long`, `float` and `double`. However, the `String` in Java is represented much differently than the C `char*` variable, so Java provides a function to make the conversion back and forth. Further, to prevent memory leaks and allow garbage collection, you always have to release any `String` resources after using them, as we'll show below.

When you design functions that are to be called in C from Java, you first write the Java declarations of each function you will need, and then write the C code to match it. The reason we do it in this order is so we can use the `javah` program to generate our C header files for us. In this simple example, we only need one C-function that returns the value of an environment variable:

```
public native String getEnv(String envName);
```

Note that we declare the method name and arguments without any method body, much like an *interface* declaration, but with the addition of the *native* keyword.

Now when we actually call these native methods, they of course have to be part of a class. In addition, the library (such as a Windows DLL) has to be loaded. So, I put all the native methods in a single class and load the library. While I initially loaded the library in the constructor, we want to avoid having multiple copies loaded, so I put the `loadLibrary` call in a static method as follows:

```

package com.javapro.javatecture;
//A simple library to make system calls
public class SysCalls {
    //always load the dll
    private static String libName = "envd";
    static {
        System.loadLibrary (libName);
    }
    //declarations of native functions
    public native String getEnv(String envName);
}

```

In this example code, I made the class part of a package called com.javapro.javatecture. We enter this code compile it along with our main program that calls this class. In this case, I am using the Eclipse development environment, and it automatically creates the com\javapro\javatecture directories under the main project directory, here d:\eclipse\workspace\JNIDemo.

The next thing we need to do is to generate the C header file. To do this, we open up a command window and go to the main project directory:

```
cd\eclipse\workspace\JNIDemo
```

We invoke the javah program to create the header file from this directory, with the com directory just below the current directory, and type:

```
javah com.javapro.javatecture.SysCalls
```

This will create a file at this to level directory, called

```
com_javapro_javatecture_SysCalls.h
```

and its contents are the following declaration.

```
JNIEXPORT jstring JNICALL Java_com_javapro_javatecture_SysCalls_getEnv
(JNIEnv *, jobject, jstring);
```

Heading Out to C

Now, we write the C code to invoke the system function. When we set up the C development environment (I am using Visual Studio.NET here), we need to make sure to copy the header file we just generated to the include directory. Visual Studio creates a project directory. I then manually create an \include subdirectory under it, and copy this header file we just generated into it. We also need to copy the jni.h file from the JDK \include directory, or make that directory part of our include path.

When Java calls one of these functions, it passed in a pointer to the Java environment. This environment includes functions for converting between Java and C strings and for releasing strings that are no longer in use. We wrap these functions in simpler calls within our C program as follows:

```

//-----
//converts a Java String to the char * string
//-----
const char *convJString(JNIEnv *env, jstring js) {

```

```

        return env->GetStringUTFChars( js, 0);
    }
    //-----
    //function to create a new Java String
    //-----
    JNIEXPORT jstring makeJstring(JNIEnv *env, const char *buf) {
        return env->NewStringUTF(buf);
    }
    //-----
    //function to release space used by converted string
    //It is very important that you call this
    //for each converted string
    //-----
    void releaseString(JNIEnv *env, jstring js, const char* s) {
        env->ReleaseStringUTFChars( js, s);
    }
}

```

Then, our actual function that calls the C getenv function uses these and is only 5 lines of code:

```

JNIEXPORT jstring JNICALL Java_com_javapro_javatecture_SysCalls_getEnv
(JNIEnv *env , jobject, jstring jenvName) {
    //convert the Java string to a C string
    const char* evar = convJString(env, jenvName);
    //get the environment variable
    char* etag = getenv(evar);
    //convert the C string to a Java string
    jstring jetag = makeJstring(env, etag);
    //release the input Java string
    releaseString(env, jenvName, evar);
    return jetag; //return the result
}

```

When we compile this C program, it makes a DLL called envd.dll. We have to make sure that that DLL is available in a directory pointed to by your systems PATH variable.

A Visual Demo

Our visual interface to this class creates an instance of the SysCalls class

```

public class ShowEnv extends JFrame
    implements ActionListener{

    private SysCalls sCalls;
    private JButton btGet;
    private JTextField inField, outField;
    //-----
    public ShowEnv() {
        super("Environment demo");
        //create instance of interface class
        sCalls = new SysCalls();
        setGUI();
    }
}

```

and when the “Get” button is clicked, it looks up the environment variable you type in the upper entry field:

```

public void actionPerformed(ActionEvent e){
    String env = sCalls.getEnv(inField.getText());
    outField.setText(env);
}

```

}

The running program is shown in Figure 1.

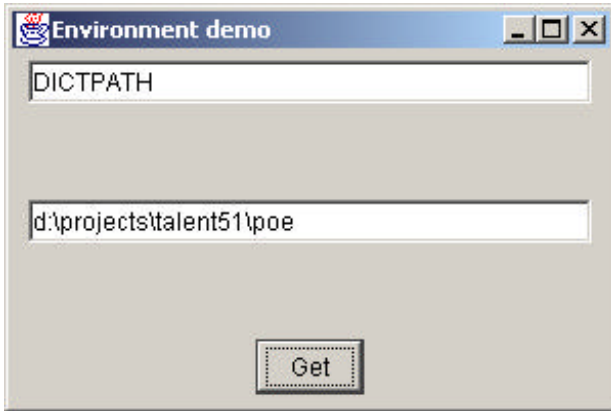


Figure 1 – The running ShowEnv program.

Getting the Bugs Out of the C

Now as we all know, Java is so easy to write, that it is always bug free right from the start (right?). But the interface to C is somewhat trickier, and can require some debugging. Let's suppose we actually have to debug our C code while Java is running. How can we do it? In this example, I'll use Eclipse and Visual Studio.NET, but this will work with other systems as well.

First, place a breakpoint just before your code loads the C library. In Eclipse (and a number of other systems) you just click in the margin of the code until the breakpoint bullet shows, and start the system in debugging mode by clicking on the Bug icon. WE show this in Figure 2.

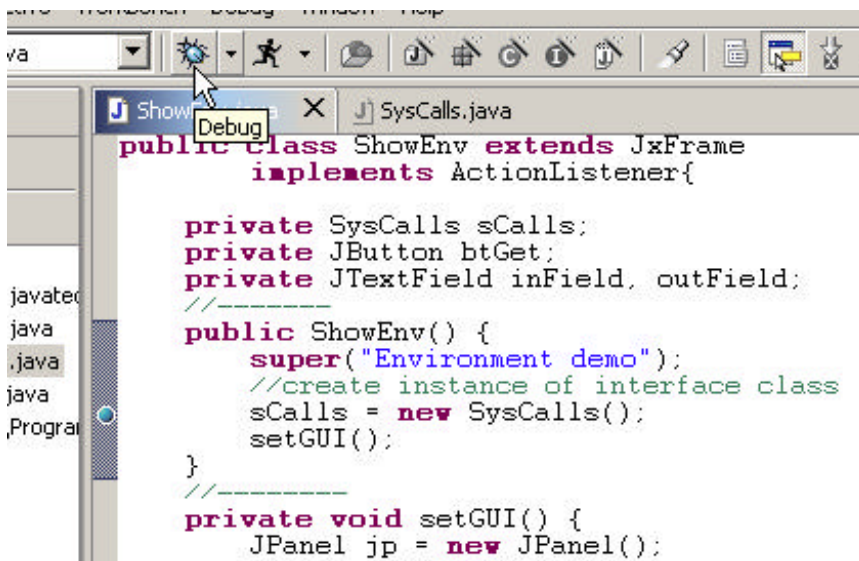


Figure 2 – Setting the debug breakpoint and starting the debugger.

Then, start the Java program in debug mode and let it run to the breakpoint as shown in Figure 3

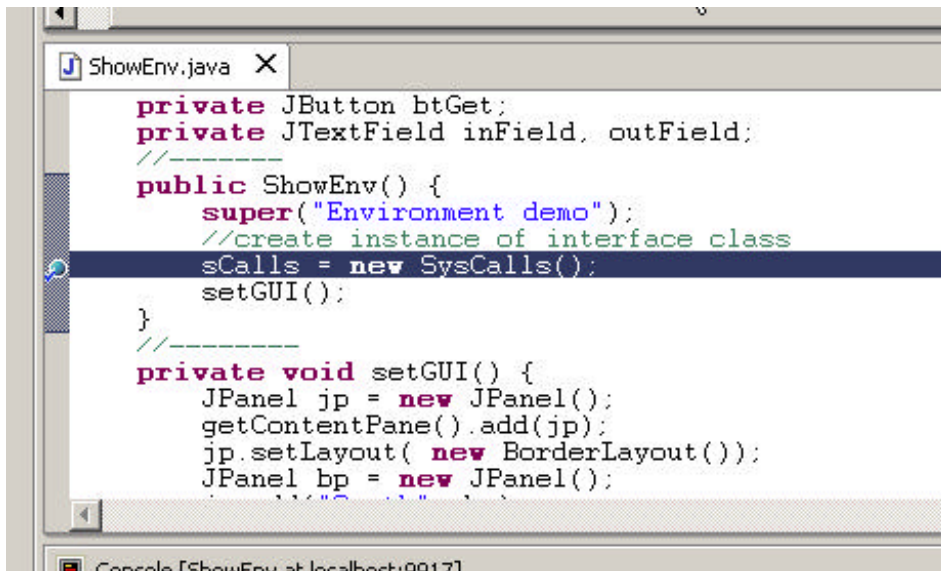


Figure 3 – The ShowEnv program stopped at the breakpoint.

Now, start up Visual Studio and load the envd DLL project. Place a breakpoint within the code, somewhere early in the function you want to debug (Figure 4).

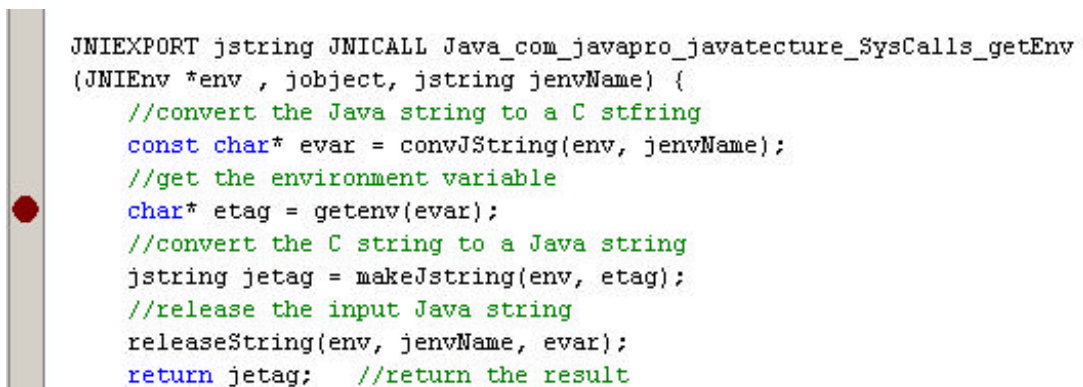


Figure 4 – Putting a breakpoint in the C JNI code.

Now, here is the neat part. In Visual Studio, click on Debug and select Processes (Figure 5).

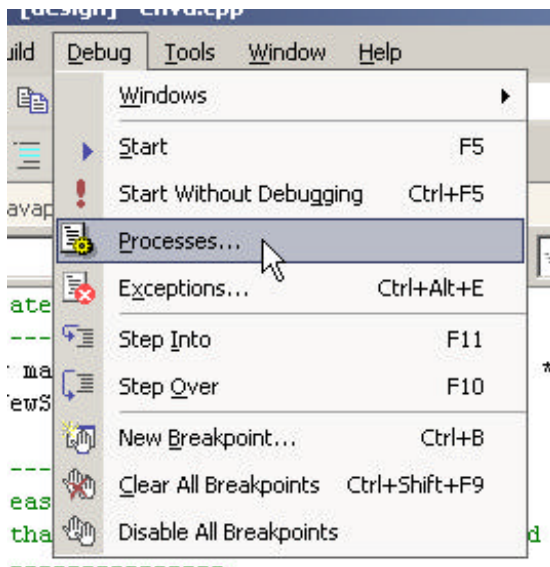


Figure 5 – Selecting the process to debug from within Visual Studio.

This will bring up a list of processes. Select your running Java process. In this case, since we are using Eclipse, which is itself a Java process, we select the next one after the Eclipse process (Figure 6).

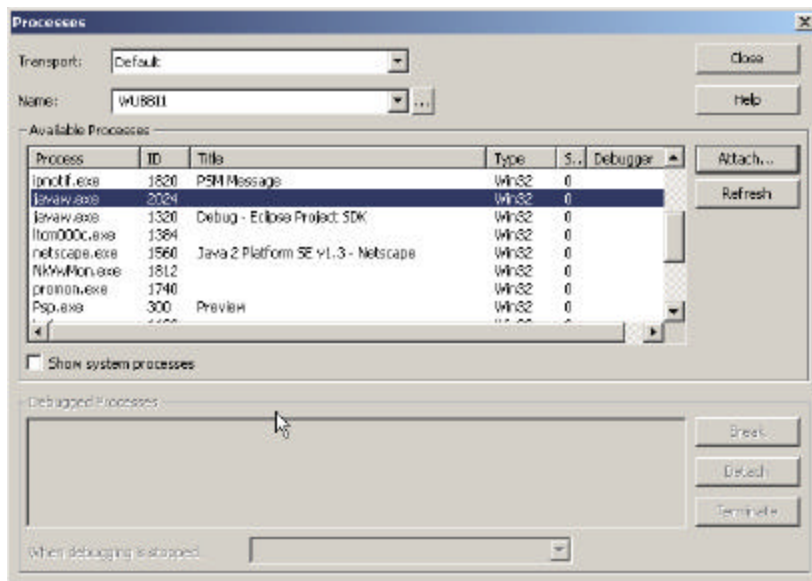


Figure 6 – Selecting the Java process to debug.

Click on **Attach**. This brings up one last window (Figure 7), where you select the fact that you only want to debug the native C code.

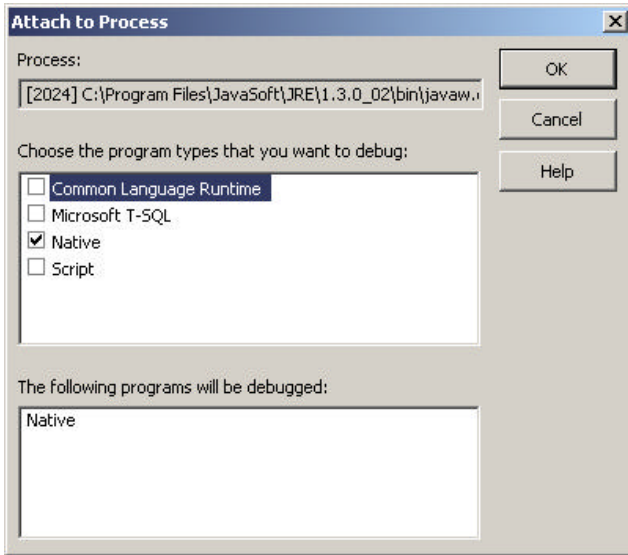


Figure 7 – Selecting the native C code process to debug.

Now, press the Continue function in the Eclipse environment: either the green arrow or function key F8. The ShowEnv program will start. Enter a path name and click on the Get button. This will cause a breakpoint in the C code, as we see in Figure 8

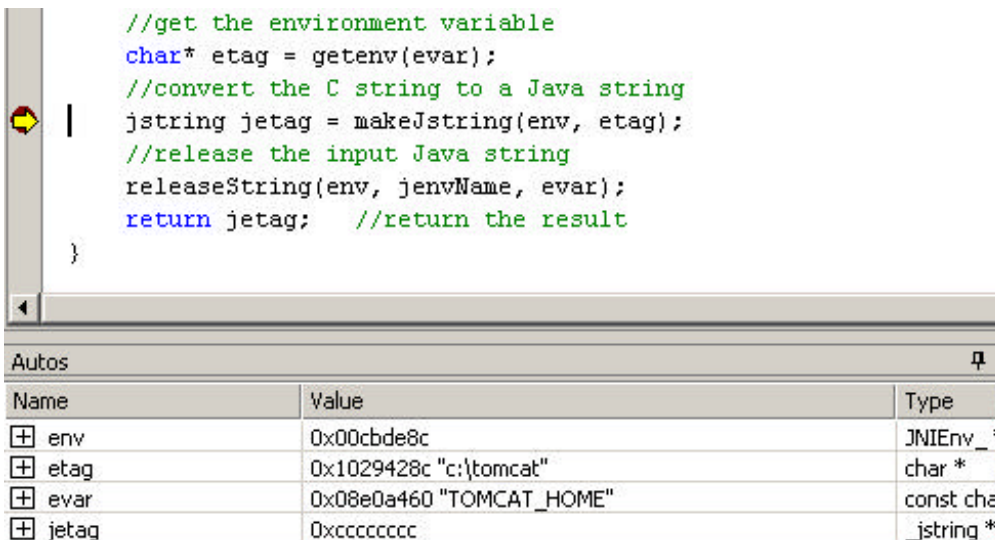


Figure 8 – A breakpoint in the C code during debugging of the JNI library.

Conclusions

So, you see, JNI is convenient when you have to connect to an already written library from Java. The advantage is that you can write most of your code in Java, which is always much faster and less error-prone, while still making use of existing code libraries. And furthermore, debugging of these libraries is really very convenient.

We illustrated this example using a really simple C function. But there is no reason why you can't do the same thing in C++. While the actual interface functions have to be in C, they can communicate with instances of C++ classes just as well as with other C.

So write your code in Java when you can, but even if some of it is below C level, you won't drown.

References

1. Sheng Liang, *The Java Native Interface*, Addison-Wesley, 1999.