

# Using The Java Native Interface <sup>1,2</sup>



by Christopher Batty

Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada

Last revised: October 23, 2003

## Overview:

In this tutorial we illustrate how the Java Native Interface (JNI) can be used to permit Java programs to communicate with C programs. We begin by compiling the Java program and the C program at the command prompt (i.e. outside of Eclipse) using the MinGW gcc compiler. Once the programs are compiled and executing correctly, we show how the programs can be integrated into an Eclipse project using the CDT plugin.

## Requirements:

The first portion of the tutorial makes the following assumptions:

- you are using Windows;
- you have installed Java and Eclipse;
- you have installed and setup the MinGW tools as described in the Eclipse Project CDT Plugin Tutorial available from the University of Manitoba (<http://www.cs.umanitoba.ca/~eclipse/7-EclipseCpp.pdf>).

Note: This tutorial makes use of code adapted from Sun's JNI tutorial which is located at <http://java.sun.com/docs/books/tutorial/native1.1/stepbystep/index.html>. If you are interested in learning more about the JNI, this is a useful site to visit.

## A Summary of the JNI

The essential idea behind the JNI is that in Java we sometimes want to access functionality that is unavailable to us through the standard Java class libraries. The functionality may be native to the particular operating system we are using, we might want to avoid re-writing pre-existing libraries that are written in a different language, or we may want to use C to optimize certain slow areas of the code.

---

<sup>1</sup> This work was funded by an IBM Eclipse Innovation Grant.

<sup>2</sup> © Christopher Batty and David Scuse

Whatever the reason for using it, the JNI bridges the gap between Java and C. It does this by accessing shared libraries that can be written in C (or possibly C++). We first write our Java code, identifying certain methods as belong to such a library. Then we write our C code and compile it into a library. We can then run our Java code as usual, and it will transparently access the library and run the native function.

It is important to remember that when we use the JNI, we lose many of the benefits that Java normally provides. In particular, C does not provide automatic garbage collection so extra caution is required to avoid introducing memory leaks.

## The Java side

The first step in using the JNI is to create a Java file that will call our native C code. Create a Java file called Hello.java with the following contents:

```
//File: Hello.java
class Hello {
    public native void sayHello();

    static {
        System.loadLibrary("hello");
    }

    public static void main(String[] args) {
        Hello h = new Hello();
        h.sayHello ();
    }
}
```

There are two items that distinguish this from regular Java code. The first is the **declaration** of a native method using the keyword **native**.

```
public native void sayHello();
```

This tells the compiler that sayHello() will be accessed from an external library, and that it should not look for it in the Java source code. Accordingly, notice that there is no implementation of this method present in the Java code.

The second unusual looking section of code is the following:

```
static {
    System.loadLibrary("hello");
}
```

This will load the external library “hello.” In Windows this file will be named “hello.dll”. The exact name of the file is dependent on the operating system conventions for naming of libraries.

The remainder of the code resembles standard Java code. An instance of the Hello class is created, and one of its methods, sayHello() is called. The fact that the method is native is irrelevant to the caller. Simply compile Hello.java as you would for any other Java source file (using **javac Hello.java**). The result will be a file called Hello.class.

## The C side – Writing the Code

The remainder of the process is concerned with creating the library that will contain the sayHello() method declared in our Java code.

To simplify the process of creating our library, the “javah” program will automatically generate the header file for us. Simply execute the following at the command prompt in the correct folder:

```
javah -jni Hello
```

This produces the file Hello.h. It contains the C declarations for the methods that were declared native in Hello.java. If we open the Hello.h file we will see the following code:

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

#ifndef _Included_Hello
#define _Included_Hello
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:      Hello
 * Method:     sayHello
 * Signature:  ()V
 */
JNIEXPORT void JNICALL Java_Hello_sayHello
    (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

The crucial component is the C declaration for our sayHello method.

```
JNIEXPORT void JNICALL Java_Hello_sayHello
    (JNIEnv *, jobject);
```

Notice that the method name has been prepended with Java\_ and then the full name of the class to which it belongs, followed by another underscore. Naturally, this is to distinguish it from methods that belong to other classes and might have the same name.

When we implement the C function, we need to ensure that we use the exact same method signature as is provided in the header file. It is not auto-generated for us, so we have to be careful in this regard.

Create a file called Hello.c. This will contain the implementation of our sayHello() method. Enter the following code:

```
//File: Hello.c
#include <jni.h>
#include "Hello.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_Hello_sayHello
    (JNIEnv *env, jobject obj)
{
    printf("Hello world!\n");
    return;
}
```

Consider each line of this code in turn. First, we include the <jni.h> header, which is required if we wish to use the JNI. The second line includes our auto-generated header file. The third line includes the C standard I/O functions, so that we can use printf.

Next we see the method signature that we first saw in Hello.h. The only difference between this line and the one in Hello.h is that the parameters to the method now have names specified. Later we will learn how these two parameters allow us to communicate data to and from our Java code. For now, we can simply ignore them.

Finally, we arrive at the traditional “Hello world!” line. Using the C printf function, we display the message on the console, and then return from the function.

## The C side – Compiling the Library

Now that we have all of our C code written, all that remains is to produce the dynamic link library (or DLL) that the Java code accesses, and we can at last run our program.

The first step is to compile the C code into an object (.o) file. On the machine this tutorial was written on, the command looks as follows:

```
gcc -c -I"C:\jdk1.4\include" -I"C:\jdk1.4\include\win32" -o hello.o Hello.c
```

The -c option tells the compiler not to link the program just yet, since we only want to produce an object file and not an executable.

The -I parameter adds two paths to your standard include paths, so the compiler can access the necessary JNI header files. The paths are <java install dir>\include and <java install dir>\include\win32, where <java install dir> is replaced with the directory in which the Java SDK is installed on your machine. If the path name contains internal spaces, it has to be surrounded by double quotes for clarity (both for the reader and the compiler).

If done correctly, this command will produce a file called hello.o.

The next step is to write a .def file, specifying the names of the functions that are to be exported, that is, those that should be visible to the Java code. This file is used by the compiler when it creates the DLL. In our case the .def filename is hello.def. The format for a .def file is:

```
EXPORTS
<function name 1>
<function name 2>
...
<function name n>
```

Therefore hello.def should contain the following two lines:

```
EXPORTS
Java_Hello_sayHello
```

To produce hello.dll, we use gcc one more time.

```
gcc -shared -o hello.dll hello.o hello.def
```

This will issue a warning that looks something like the following:

**Warning: resolving \_Java\_Hello\_sayHello by linking to  
\_Java\_Hello\_sayHello@8**

This informs you that the compiler is automatically dealing with an issue involving the calling conventions in Windows. Behind the scenes, gcc converts functionName to functionName@xx, where xx is the number of bytes passed to the function on the stack. What the compiler is warning us is that it is automatically associating functionName with functionName@xx so that when a program looks for functionName in the library, it will correctly be referred to functionName@xx. For a program as simple as ours, we can essentially ignore this warning since this is the behaviour we expect.

To verify that the program is working correctly, we can run it as we would with any other Java program.

```
java Hello
```

The message displayed on the console should be the expected “Hello world!” If this is what you see, then you have successfully used the JNI to link Java together with native C code.

**A file buildHello.bat has been included in the project folder; this file contains all of the commands required to compile the java and C programs and then to run the java program. This command is executed from the Windows command prompt. It will be necessary to change the paths to the commands if you have installed the JDK or GCC in a different location than those used in the batch file.**

An interesting feature of the JNI is that since the C code is placed into a DLL, the C code need not be recompiled when you change aspects of your Java program that don’t directly change the native method signature. Likewise, you can change the behaviour of your C methods with having to recompile your Java program.

### **Communicating between C and Java**

This Hello application is certainly interesting, but it is rare that you would use a JNI function that neither takes parameters nor returns any value. In the next section, we examine how parameters and return values can be included.

If the declaration of our sayHello method in the Java code had contained parameters, they would have been added to the native method’s signature, in their C equivalents. For example, if the method had taken an int parameter as well:

```
public native void sayHello(int v);
```

then the C declaration would look like:

```
JNIEXPORT void JNICALL Java_Hello_sayHello
    (JNIEnv *, jobject, jint)
```

“jint” is the C name for a Java integer primitive type. All of Java’s primitive types map in a similar way, essentially by pre-pending the letter j to the name of the type in Java. A list of the type mappings is available at:

<http://java.sun.com/docs/books/tutorial/native1.1/integrating/types.html>

These types are simply typedefs to the corresponding C data types, and so they can be used with all of the standard C functions. The following example demonstrates how to create a simple native function to take the square root of a Java double.

```
//File: Sqrt.java
public class Sqrt {
    public native double sqrt(double d);

    static {
        System.loadLibrary("sqrt");
    }

    public static void main(String[] args) {
        Sqrt s = new Sqrt();
        double result = s.sqrt(9.1);
        System.out.println("Square root of 9.1 is " + result);
    }
}
```

We generate a C header file as we did before for the Hello world example. Take note of the form that the function declaration for sqrt takes. Sqrt.c is shown below:

```
//File: Sqrt.c
#include <jni.h>
#include "Sqrt.h"
#include <math.h>

JNIEXPORT jdouble JNICALL Java_Sqrt_sqrt (JNIEnv *env, jobject obj,
jdouble d) {
    return sqrt(d);
}
```

The standard C sqrt function is used to calculate the answer. It acts directly on the jdouble d, since jdouble is simply a typedef for the primitive C type double. Likewise, its return value is just the double returned by sqrt().

As with the Hello world program, a batch file named `buildSqrt.bat` has been included in the project folder. This batch file compiles and executes the program. When the program is run, the following output is generated:

```
>java Sqrt
Square root of 9.1 is 3.0166206257996713
```

## Strings, Arrays, and Objects

First let's consider the declaration of the native method in the hello world example.

```
JNIEXPORT void JNICALL Java_Hello_sayHello
    (JNIEnv *env, jobject obj)
```

When we first came across this line, we simply ignored the first two parameters to the function. Let's now examine them in greater detail.

**JNIEnv\* env** is a pointer which gives us access to a variety of JNI functions. For example, in order to work with strings and arrays in C, we need to call one of these functions to convert the data to a usable C format.

**jobject obj** is a reference to the object that the method has been called on. In the Hello world example, `obj` represented `h`, our instance of the Hello class. In combination with some JNIEnv functions, it can be used to access member data or methods of the class. Also, if we pass non-primitive objects, other than strings, this is the form they will take on in the native code.

Java Strings will appear as the type `jstring` in the C declaration. Unlike the primitive types, a `jstring` cannot simply be used interchangeably with a C-style string. If our `jstring` is called `text`, we call the function

```
const char *str = (*env)->GetStringUTFChars(env, text, 0);
```

which will return a `char*` that can be used with standard C functions (eg. `printf`). When you are finished using the C-string, you must call

```
(*env)->ReleaseStringUTFChars(env, text, str);
```

to release the memory that was allocated for it. There are also functions for getting Unicode strings, for creating new `jstrings` to act as a return value, and methods for determining the length of a particular `jstring`.

Java arrays are passed to native functions as instances of <java type name>Array, such as jintArray or jfloatArray. As with strings, we cannot simply access these arrays directly as if they were standard C arrays. If our jintArray is called iArray, we use:

```
jint* data = (*env)->GetIntArrayElements(env, iArray, 0);
```

To release it we call:

```
(*env)->ReleaseIntArrayElements(env, iArray, data, 0);
```

To get the length of a Java array, use:

```
jsize len = (*env)->GetArrayLength(env, iArray);
```

Included with the code for this tutorial are two additional Java programs. The first, JNIString.java, demonstrates accessing a Java String in C. The second, JNIArray.java, demonstrates accessing an integer array from C. The code is intended to be as straightforward and simple as possible, so the contents of the relevant data structure are only printed.

## JNI Capabilities

We have looked at how to use the JNI in very simple ways, but using these techniques we can achieve a great deal. For example, the Eclipse project's SWT libraries make extensive use of the JNI in order to access the native windowing capabilities of the particular operating system. The JNI comprises numerous other capabilities beyond the scope of this tutorial, such as accessing Java methods and data from C functions, exception handling and multithreading in C, and even launching a Java virtual machine in a stand-alone C program. By employing these techniques selectively and carefully, the JNI can be an extremely useful tool in any Java programmers' toolkit.

## Using Eclipse to Perform the Compilation

We now examine how to set up Eclipse to develop in Java with native C code within a single project. We assume that:

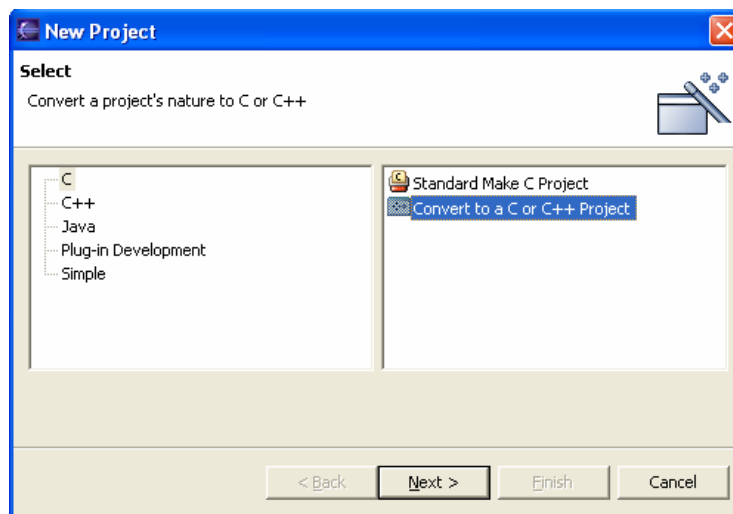
- you have installed Eclipse 2.1, MinGW, and the CDT Plugin for Eclipse, as outlined in the CDT tutorial (also available from the U of M at [www.cs.umanitoba.ca/~eclipse](http://www.cs.umanitoba.ca/~eclipse));
- you have read and understood the JNI material described earlier in this tutorial;
- you possess at least a basic understanding of make and makefiles.

### The Process

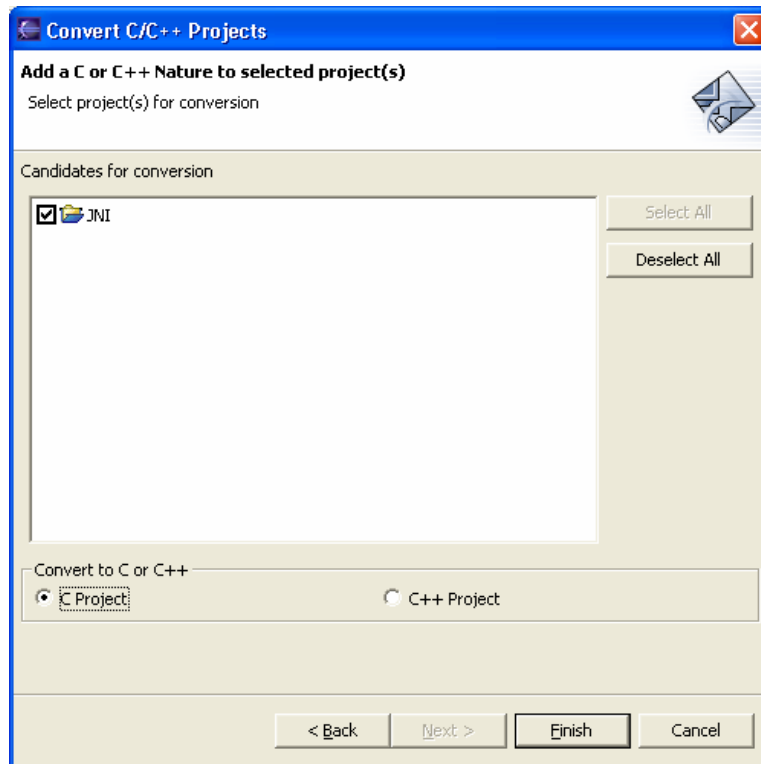
Using the CDT and JDT in combination, we can set up Eclipse to develop applications that use the JNI, all within a single project. We will create a Java project and then use the “Convert to C or C++ Project” facility to add the CDT’s C/C++ project functionality to it. Then whenever we save our code, both the Java and C code will be compiled.

To begin, create a new Java project, and place in it the Hello.java code from the JNI material described earlier in this tutorial. (We will use the project folder as both source and output folder. This setting is the default, so you can safely ignore this warning if it doesn’t make sense to you.)

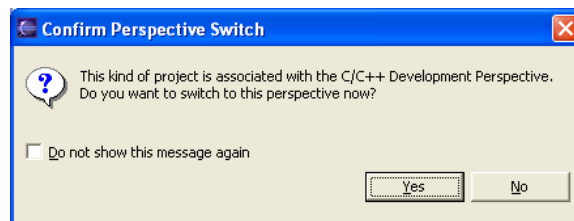
Next we will modify the project to allow C/C++ files to be compiled as in the CDT tutorial. From the File menu, choose “New” and select “Project...” In the tree on the left, select “C”. In the corresponding list on the right, choose “Convert to a C or C++ Project,” and press “Next”.



On the next screen, you select the project you wish to convert by checking its corresponding check box. Choose the Java project you just created. Then select the “C Project” radio button at the bottom of the screen.



When you hit finish, you may be presented with a message like the following:



You may choose “Yes,” or you can switch to the C perspective later on.

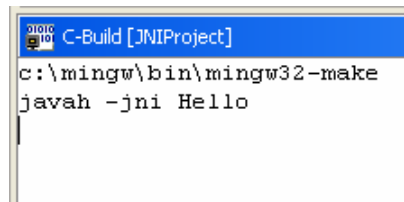
Now that the project has been converted, right-click on its project name, and choose “Properties.” We can see that it now has properties pages for both Java **and** C/C++. Under the “C/C++ Project” properties, set up your build command as outlined in the CDT tutorial.

Next we will write a simple makefile that can build the JNI code, using the same commands that were covered in the JNI tutorial. Create a new file, and name it “makefile”.

Recall that the first thing we need to do to start writing our C implementation is to generate the C header file from our compiled Java class file. Our first make dependency will therefore specify that our header file Hello.h depends on Hello.class. The makefile should look like the following:

```
all : Hello.h
Hello.h : Hello.class
    javah -jni Hello
clean :
    -del Hello.h
```

This makefile says that to build Hello.h, we must run javah on Hello.class. When make runs it will produce the C header file. Output like the following should appear in the C-build view.



The resulting header file (Hello.h) should also appear in the C/C++ Project view. We can now write our C implementation file. However, since we have already written Hello.c, simply place it in the new project. In order to produce the DLL, we also need the hello.def file that specifies the externally visible methods, so copy it into the project as well.

Now that we have all the required source files in place, we just need to modify our makefile so that our DLL gets built. The new makefile looks as follows: (Note: You will have to modify the JNI path names to fit your system.):

```
all : hello.dll

hello.dll : Hello.o hello.def
    gcc -shared -o hello.dll Hello.o hello.def

Hello.o : Hello.c Hello.h
    gcc -I"C:\jdk1.4\include" -I"C:\jdk1.4\include\win32"
        -c Hello.c -o Hello.o

Hello.h : Hello.class
    javah -jni Hello

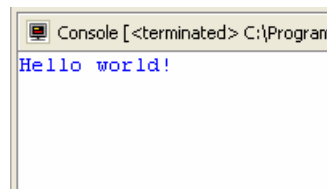
clean :
    -del Hello.h
    -del Hello.o
```

We have added two more dependencies to the makefile. The first specifies how to build `hello.dll` using `Hello.o` and `hello.def`. The second specifies how to build `Hello.o` from `Hello.c` and `Hello.h`. We have also changed the “all” build target to the DLL rather than `Hello.h`. For real projects your makefiles will likely need to be more complex, but the essential commands remain the same.

(Note: By specifying that `Hello.h` must be built whenever `Hello.class` changes, `Hello.h` will likely be regenerated fairly often. If you make manual changes to `Hello.h`, which should be avoided anyhow, they will be overwritten.)

Now when `make` executes, the DLL is built. `hello.dll` and `hello.o` becomes visible in the “C/C++ Projects” view. To run the program, create and run a regular Run Configuration for the Hello Java application. The run configuration is the same as for any other Java application.

The output should look like the following:



Using the techniques outlined above, you can edit your Java and C source code all within Eclipse. Modifications to the native method declarations in your Java code, will cause a new version of `Hello.h` will be generated when `make` is run. Changes to the C code will result in a new DLL being produced. Compile errors in both Java and C++ will appear in the Task view. Aside from having to manually write the `.def` and makefiles, the JNI code can be written and compiled almost seamlessly as a part of a regular Java project.