Java™ Native Methods

Rex Jaeschke

Java Native Methods

Edition: 2.0 (matches JDK1.6/Java 2)

The training materials associated with this book are available for license. Interested parties should contact the author at the address below.

Please address comments, corrections, and questions to the author:

Rex Jaeschke
2051 Swans Neck Way
Reston, VA 20191-4023
+1 (703) 860-0091
+1 (703) 860-3008 (fax)
www.RexJaeschke.com
rex@RexJaeschke.com

Preface

While Java is the language of choice for certain kinds of applications, it doesn't do some things well, efficiently, or at all. There also exists a myriad of existing commercial and homegrown libraries, primarily written in C or C++, which people wish to use in new development. And often there are low-level platform-specific routines that need to be called.

These issues were anticipated during the design of Java and resulted in support for what are known as *native methods*; that is, for methods written in some language other than Java. The original, platform-specific, approach provided with JDK1.0—now referred to as the *Native Method Interface* (NMI) —was superseded in JDK1.1 by the more general *Java Native Interface* (JNI). Note, however, that while JNI was designed to allow the same native binary code to work correctly with *any* Java Virtual Machine (JVM) implementation on the *same* platform, it makes no promises about porting the native source code between different native language implementations on the same or different platforms, or using virtual machines on different platforms. The basic intent of JNI is that both Java and native methods be able to create, share, access, and update Java objects.

According to "The Java Language Specification", 3rd edition, by James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, ISBN 0-321-24678-0, Addison-Wesley, 2005: *A method that is native is implemented in platform-dependent code, typically written in another programming language such as C, C++, FORTRAN, or assembly language.* While that's all the formal definition of Java says about interfacing with the non-Java world, a companion volume "Java Native Interface: Programmer's Guide and Specification", by Sheng Liang, ISBN 0-201-32577-2, Addison-Wesley 1999, provides a detailed discussion.

All the examples we will see have been developed and run using Sun's JDK1.2 through to .6, and Microsoft's Visual C++ V6 through to Visual C++ .NET 2008, running on Windows NT V4.0 through to Vista. If you are using a different development environment, you'll have to extrapolate as necessary; however, the basic approach should be the same.

As we will see, the machinery needed to call a native method from Java is such that a Java method cannot directly call existing non-Java methods written without regard to the possibility of their being called from Java. One of the fundamental questions to consider when contemplating using native methods is whether the native methods you write will also need to be callable by languages other than Java. By far the simplest model is to have native methods that are written for use only in conjunction with Java code.

Reader Assumptions

To fully understand and exploit the material, you should be conversant with the following concepts and the syntax required to express them in Java:

- Basic Language Elements
- Looping and Testing
- Methods
- References, Strings, and Arrays

Java Native Methods

- Classes
- Inheritance
- Exception handling
- Input and Output
- Packages
- Interfaces

A working knowledge of C (or at least the C subset of C++) is also necessary.

Source Code

The programs shown in the text are made available electronically in a directory tree named Source.

The Java Development Kit

Sun's initial production release of Java was the Java Development Kit (JDK) version 1.0. Versions 1.1 through 1.5 contained numerous bug fixes, and language and library enhancements. The latest version can be downloaded from Sun's website (www.sun.com).

While the language has remained very stable, along the way, several features were added along with numerous new packages and classes and new methods to existing classes. Also, some existing method names have been changed. In these latter cases, the old names continue to be acceptable, but are flagged by the compiler as *deprecated*, meaning that support for them might well be removed in future versions. If your compiler issues such a warning, consult Sun's on-line documentation to find the recommended replacement.

From an internationalization (I18N) viewpoint, one of the most significant additions made by V1.1 was the completion of support for dealing with non-US, non-English environments, including those involving very large alphabets and non-Latin writing systems.

Rex Jaeschke, September 2009

1. A Simple Example

To see the general approach used, we'll start by having a Java program call a C function. This function, `doubler`, takes one integer argument, and returns a result of the same type whose value is double that of the argument passed in. While it's not very exciting, neither is it unnecessarily complicated (see Nt01.java):

```
class Nt01
{
/*1*/   private static native int doubler(int arg);

        static
        {
/*2*/           System.loadLibrary("Nt01");
        }

        public static void main(String[] args)
        {
                for (int i = -100; i <= 100; i += 50) {
                        System.out.println("i = " + i + ", doubler("
/*3*/                           + i + ") = " + doubler(i));
                }
        }
}
```

In case 1, we declare that `doubler` is a native method by giving it the modifier `native`. Being a native method, `doubler` has no body.[1] In all other respects, this method looks like any method implemented in Java, so it should come as no surprise that calls to it, as in case 3, look like any other method call. Whether a native method is public, protected, private, or package-private, is up the Java programmer, and has no impact on the native method itself.

A Java compiler generates architecture-neutral bytecodes, which are then interpreted by the host system's JVM. As such, unlike most other languages, Java programs do not go through the static linking stage. Instead, Java must somehow be linked dynamically to object modules written in some other language. The mechanism used for this is some sort of *shared library*. (On Microsoft systems, this is called a dynamically linked library, or DLL.) Therefore, all native methods must reside in one or more shared libraries, and those shared libraries must be loaded explicitly by the Java program using one of two methods. The first method, used in case 2, is `System.loadLibrary`. The second method is `System.load`, which is not used in this example. The basic difference is that `System.load` needs the full path/filename of the shared library while `System.loadLibrary` requires only the shared library's name, in which case, that shared library is assumed to be in one of the places

---

[1] Even though a native method has no body, all formal parameters must have names; that is, the name `arg` cannot be omitted.

searched by the system. If either method fails to find the required library, an exception of type `java.lang.UnsatisfiedLinkError` is thrown.

When we compile this file, file Nt01.class is produced. We then input this class file to a utility that produces a header suitable for use with a C/C++ function. The generic name for this utility is javah, a utility shipped with the JDK. We can input multiple class files to javah. By default, it will generate a separate header for each; however, if we use the –o option, we can get all output concatenated into one header. Here's the output from javah when applied to Nt01.class (see Nt01.h):

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Nt01 */

#ifndef _Included_Nt01
#define _Included_Nt01
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     Nt01
 * Method:    doubler
 * Signature: (I)I
 */
JNIEXPORT jint JNICALL Java_Nt01_doubler
  (JNIEnv *, jclass, jint);
#ifdef __cplusplus
}
#endif
#endif
```

The header `jni.h` comes with the JDK and provides some helper macros and structure definitions. (It is very likely this header will contain directly, or will include another header that contains, platform-specific macros and type synonyms.) The main point of interest here is the declaration of the native method `doubler`.

```
JNIEXPORT jint JNICALL Java_Nt01_doubler(JNIEnv *, jclass, jint);
```

When placing C/C++ object modules in a shared library, some systems require these modules to have certain attributes. Such attributes can be provided via the macros `JNIEXPORT` and `JNICALL`.[1] Of course, if either macro is not really needed, it can have an empty definition; that is, we should always use these macros, even if they expand to nothing.

Although the argument and return type of `doubler` in Nt01.java are both `int`, we see that they are `jint` in the C/C++ function prototype. In Java, an `int` is required to have exactly 32 bits while in C/C++ the size of an `int` is

---

[1] On 32-bit Windows systems, JNIEXPORT is defined as `__declspec(dllexport)` while JNICALL is defined as `__stdcall`.

implementation-defined, but must be at least 16 bits. On the other hand, in C/C++, a `long int` must have at least 32 bits. We'll discuss primitive type mapping further in §1; for now, let's simply acknowledge that we must be careful to get the mappings correct.

While Java considers the native method's name to be `doubler`, the C/C++ name for it is `Java_Nt01_doubler`. This name is created from the following components: the prefix `Java`, the parent package name (if any), the parent class name (in this case, `Nt01`), and the Java method name, with adjacent component names being separated by an underscore. For example, if this method and class were in a package called COM.demo, the resulting native function name would be `Java_COM_demo_Nt01c_doubler` (see Nt01c.java and COM_demo_Nt01c.h).

The number of arguments in the C/C++ function is always two more than in the corresponding Java declaration. The first argument, of type `JNIEnv *`, provides access to a structure containing a set of function pointers (or, in the case of C++, a set of inline functions) that give us access to the Java environment. The second argument has type `jclass`, indicating our native method is static; if it were non-static, this argument's type would be `jobject` instead. As these two arguments are not used in this example, we'll discuss them in later sections as the need arises.

To complete the program, here's the C implementation of `doubler` (see Nt01.c):

```
#include "Nt01.h"


JNIEXPORT jint JNICALL Java_Nt01_doubler(JNIEnv *penv, jclass cl, jint i)
{
        return i * 2;
}
```

As we should expect, the function is defined just as it was declared in the header. Now all we need to do is to build Nt01.c into a shared library, place that shared library into one of the locations searched by the system, and run the Java program. The output produced is:

```
i = -100, doubler(-100) = -200
i = -50, doubler(-50) = -100
i = 0, doubler(0) = 0
i = 50, doubler(50) = 100
i = 100, doubler(100) = 200
```

A C++ version of Nt01.c is provided as Nt01b.cpp; however, its contents are identical.

Note that the mangling of external names is disabled in C++ mode; therefore, we cannot have overloaded native C++ methods since these have no equivalent in C. Note, however, that we *can* define overloaded methods in Java, some or all of which are also native. For example, when the following Java code (see Nt01d.java) is given to javah:

3. Passing and Returning Strings and Arrays

The program defined in class `Nt03` in Nt03.java calls a number of native methods that take arguments and/or return values having type `String`, arrays thereof, and arrays of primitives. For each example within this program, we'll see the native method's declaration and then the corresponding C implementation. All the native methods are defined in Nt03.c, and the corresponding header in Nt03.h. The Java code resides in the universal unnamed package.

Method `displayString` simply displays the String passed to it, both as a multibyte string and as a Unicode string. Here is its declaration:

```
private static native void displayString(String str);
```

and here is its C implementation:

```
JNIEXPORT void JNICALL
Java_Nt03_displayString(JNIEnv *penv, jclass jc, jstring jstr)
{
/*1a*/  const char *pstr1 = (*penv)->GetStringUTFChars(penv, jstr, NULL);

/*2a*/  printf("Received string >%s< having length %lu\n",
                pstr1, (unsigned long)(*penv)->GetStringUTFLength(penv, jstr));

/*3a*/  (*penv)->ReleaseStringUTFChars(penv, jstr, pstr1);
}
```

We see that the String passed in has the abstract type `jstring`. Before we can process a String from within C/C++, we must convert it to a suitable form. We do this in case 1a,[1] by calling GetStringUTFChars via the Java environment pointer argument, passing it the Java environment and the `jstring` argument. When this conversion is done, it might result in the String's being copied. If this does not happen, the object is considered *pinned*. In any event, by passing in a non-null third argument of type `jboolean *`, we can find out whether or not a copy was made by comparing the resulting `jboolean` value with the macros JNI_TRUE and JNI_FALSE. The resultant string must be treated as being read-only, hence the use of the `const` qualifier in the declaration of `pstr1`.

In Java, a String can contain any 16-bit Unicode character, yet `pstr1` is a pointer to a (probably 8-bit) plain `char`. Rather than trafficking in single-byte characters, `GetStringUTFChars` actually converts to multibyte characters. Of course, if the String passed contains characters whose upper 8 bits are clear, this method results in characters consisting of one byte only. If the String passed contains characters whose values exceeded 255, you'll have to

---

[1] In C++, we can use the simpler notation `penv->` instead of the `(*penv)->` required in C code.

make sure you compile and build the C code with multibyte support enabled. (In fact, since you never know just what characters will be passed, you should always compile in that mode just to be safe.)

`GetStringUTFLength` returns the length of the string in bytes; however, its return type is the abstract type `jsize`. Since no guarantee is made about just which C/C++ type this maps to (although we are told that it's the same as `jint`), we must be careful to deal with it in an abstract manner; for example, we cast it to `unsigned long` in case 2a before passing it to `printf`. (The cast is unnecessary in C++ when I/O is performed, since the compiler knows `jsize`'s type exactly.)

The call to `ReleaseStringUTFChars` in case 3a informs the JVM that we are finished with the converted string, allowing it to clean up after us. Failure to release such resources results in a memory leak.

If you prefer to deal with Unicode characters directly instead of multibyte characters, you can use a different family of functions; for example:

```
/*1b*/   const wchar_t *pstr2 = (*penv)->GetStringChars(penv, jstr, NULL);


/*2b*/   wprintf(L"Received wide string >%s< having length %lu\n",
                 pstr2, (unsigned long)(*penv)->GetStringLength(penv, jstr));


/*3b*/   (*penv)->ReleaseStringChars(penv, jstr, pstr2);
```

Of course, this assumes that `wchar_t` maps exactly to a Unicode character. Since we are writing out a wide character string in case 2b, we use `wprintf` and a wide character string literal.

The second native method, `calcAverage`, is passed an array of `int`. If that array has any elements, `calcAverage` returns the average element value; otherwise, it returns zero. The function assumes that the array reference passed in is non-null. Here is the method's declaration:

```
private static native int calcAverage(int[] intArray);
```

and here is its C implementation:

```
JNIEXPORT jint JNICALL
Java_Nt03_calcAverage(JNIEnv *penv, jclass jc, jintArray jarray)
{
/*11*/  jsize elementCount = (*penv)->GetArrayLength(penv, jarray);
/*12*/  jint *elements;
/*13*/  jint sum = 0;
/*14*/  jsize i;

        if (elementCount == 0) {
                return 0;
        }
```

```
/*15*/  elements = (*penv)->GetIntArrayElements(penv, jarray, NULL);

        printf("\ncalcAverage received ");
        for (i = 0; i < elementCount; ++i) {
/*16*/          printf("%ld ", (long)elements[i]);
                sum += elements[i];
/*17*/          elements[i] += 1000;
        }
        putchar('\n');

/*18*/  (*penv)->ReleaseIntArrayElements(penv, jarray, elements, 0);

        return sum/elementCount;
}
```

The length of any Java array can be determined by calling `GetArrayLength`, as shown in case 11.

To convert the in-coming array to a form suitable for use by C/C++, we call `GetIntArrayElements` in case 15. This function returns the address of the first element in the contiguous array of integers. (By substituting another primitive type name for `Int`, we can likewise handle arrays of any other primitive type.) As with a String argument, we can tell if a copy has been made by passing a non-null third argument.

Since the array passed has elements of the abstract type `jint`, we convert to type `long int` before passing each element's value to `printf` in case 16.

In case 17, we increment each element's value by 1000.

Just as we must free memory for strings, we must also free it for arrays, and we do that in case 18 by calling the corresponding type release method, `ReleaseIntArrayElements`. The fourth argument is a mode flag that has three possible values:

- `0` – Copy back the array's contents and free the buffer.
- `JNI_COMMIT` – Copy back the array's contents but do not free the buffer.
- `JNI_ABORT` – Free the buffer without copying back the array's contents.

As we want the new element values to be seen by the Java code when we return, and we no longer need the local buffer, we use mode `0`.

In the approach shown above, the whole array is fetched and released. If we have large arrays and we only need read or write access to a small contiguous set of elements, we can use the functions Get*Type*`ArrayRegion` and Set*Type*`ArrayRegion`, respectively. To use these, we must specify a starting index and length. We must also supply our own buffer to hold the region. As a result, the copy of the region is under our own management, so we need not call any special JNI release function to free it.

The third native method, `findLongestString`, is passed an array of Strings. It looks for the longest String in this array and then creates a new String that is an exact copy of that String and returns the copy. If the in-coming array

4.   Accessing Static Fields and Methods

Program Nt04.java defines class `Nt04` to contain a number of static fields, some of which are `final` while others are not. Here's an extract in which all fields are `final`, have some primitive type, and have a compile-time constant initializer:

```
private final static boolean    finalClassField1a = true;
private final static boolean    finalClassField1b = false;
private final static char       finalClassField2 = 'A';
private final static byte       finalClassField3 = 100;
private final static short      finalClassField4 = -200;
private final static int        finalClassField5 = 300;
private final static float      finalClassField6 = -1.234F;
```

The corresponding output from `javah` is as follows (see Nt04.h):

```
#undef Nt04_finalClassField1a
#define Nt04_finalClassField1a 1L
#undef Nt04_finalClassField1b
#define Nt04_finalClassField1b 0L
#undef Nt04_finalClassField2
#define Nt04_finalClassField2 65L
#undef Nt04_finalClassField3
#define Nt04_finalClassField3 100L
#undef Nt04_finalClassField4
#define Nt04_finalClassField4 -200L
#undef Nt04_finalClassField5
#define Nt04_finalClassField5 300L
#undef Nt04_finalClassField6
#define Nt04_finalClassField6 -1.234f
```

As we can see, these fields get turned into object-like macros whose names are prefixed with their parent class name and an underscore; however, type information is lost, since all of the integer and `boolean` field value macros expand to expressions of type `long int`. The `float` field value macro does expand to an expression of type `float` based on the explicit suffix provided. (We'll talk about fields of type `long` and `double` at the end of this section. For now, suffice it to say that some versions of `javah` aren't quite getting things right.)

We can use these macros in the usual way from any C/C++ native function.

The following three fields are also `final` and only two have a primitive type. The important difference here is that one has array type and none has a compile-time constant initializer:

```
private final static int        finalClassField7 = init(5);

private static int init(int x)
{
        int sum = 0;

        for (int i = 1; i <= x; ++i) {
                sum += i;
        }

        return sum;
}

private final static int[]      finalClassField8 = new int[6];
private final static int        finalClassField9;

static {
        int total = 0;

        for (int i = 0; i < finalClassField8.length; ++i) {
                finalClassField8[i] = i * i;
                total += finalClassField8[i];
        }
        finalClassField9 = total/finalClassField8.length;
}
```

As a result, javah is unable to turn these fields into object-like macros. In fact, the file Nt04.h contains the following:

```
/* Inaccessible static: finalClassField7 */
/* Inaccessible static: finalClassField8 */
/* Inaccessible static: finalClassField9 */
```

Here's how we can access these fields from a static native method implemented as a C function (see Nt04.c):

```
JNIEXPORT void JNICALL Java_Nt04_test1(JNIEnv *penv, jclass cl)
{
        jfieldID fid;
        jint intVar;
        jshort shortVar;
        jstring jstr, jstr2;
        const char *str;
        jintArray jintAry;
        jsize elementCount, i;
        jint *elements;
```

```
/*1*/    fid = (*penv)->GetStaticFieldID(penv, cl, "finalClassField7", "I");
         if (fid == NULL) {
                 printf("Cannot locate finalClassField7\n");
         }
         else {
/*2*/            intVar = (*penv)->GetStaticIntField(penv, cl, fid);
                 printf("C:finalClassField7 = %d\n", intVar);
         }
```

In this fragment, we access and display the value of the `final static int` field `finalClassField7`. In case 1, we call `GetStaticFieldID` to get the ID of the field of interest. We pass it four arguments: the Java environment, the `jclass` of which we are a static method, the exact name of the static field we wish to access, and that field's type. The following table shows how field type strings must be written. (Note that the field name is a multibyte character string, so names containing letters from other than the Latin alphabet are supported.)

Table 4-1: JVM Type Signatures

byte	B
char	C
double	D
float	F
int	I
short	S
class	L*classname*;
long	J
boolean	Z

Note that *classname* must be fully qualified (as we shall see later when using the `String` type). Arrays have a field type string of the form "[*type*" with one left bracket for each dimension. (The format for objects and arrays is exactly the same as that produced by `Object.toString` when it is not overridden.)

If the `jfieldID` returned is null, the field could not be found. (In subsequent calls, we'll ignore this possibility.)

Once we have a field's ID we can extract its value by calling `GetStaticIntField` in case 2. As we asked for the value of a Java `int`, we store the result in a `jint`. To access a static field of some other primitive type simply replace the `Int` with the corresponding type name making sure the first latter is capitalized.

Even though we saw that certain `final static` primitive fields have corresponding macros, we can also use the machinery just demonstrated, to access them as well. While we'll get exactly the same values that the macros provide, we're able to deal with these fields by their Java type rather than seeing them all as `long ints`.

Let's see how to access the `final static` array of `int`, `finalClassField8`. Note that while the reference to the array is final, the elements of the array themselves are not. The native method accesses and displays the value of each element, and then increases that value by 1000: