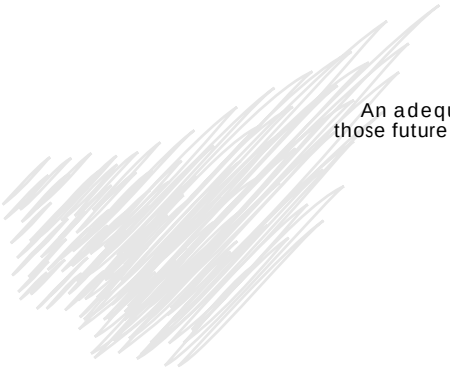
Chapter 2 HELLO JNI WORLD



An adequate sense of tradition manifests itself in a grasp of those future possibilities which the past has made available to the present.

Alasdair MacIntyre
After Virtue

Introduction

In order to situate this book within the short, yet illustrious, tradition of programming books, it seems appropriate to begin with an example from the relatively ancient days of programming languages.

So, with a nod of respect to Messrs. Kernighan and Ritchie, this book begins with the ever useful "Hello World" example. Never mind the obvious. Never mind that you may never need to do such a trivial task. There is much to be learned from this example and that is why it has lasted nearly twenty-five years.

Unlike pure Java code, using the JNI involves much more than simply creating a Java source file and using `javac` to build a class file. An application combining Java code with some platform-specific code requires additional steps to build a native library and interface it with the Java code. Further, some of those steps differ in details depending on the platform for which the application is to be targeted.

The Big Picture

An overview of the steps employed to integrate native code with a Java application will be helpful before we look at the individual steps.

A `native` method in Java is identified using the `native` keyword to modify the method declaration. When the `native` keyword is used to identify a Java method, no body is defined for that method within the Java class in which the method is declared. Instead, the body of the `native` method is contained in a separate C/C++ source file.

For each `native` method declared, a C/C++ function needs to be written. A tool, `javah`, provided with the JDK, takes as input a Java class file and generates an ASCII C function prototype for each `native` method declaration. Each prototype defines the calling protocol the JVM uses to execute a `native` method. The prototype also constrains how the native function is written by defining its input (formal arguments) and output (return value).

With this overview in mind, let's look at the individual steps involved in writing `native` method code.

The Steps

In this section the steps necessary for integrating `native` method code with a Java class will be described. As we get into the details of actually generating the appropriate files, a simple "Hello World" program will serve as an example.

Identify Native Functionality

You are interested in the JNI for one of two reasons. Either you have existing code which you do not want to rewrite and you want to use within a Java application, or you are writing a Java application, some parts of which are best written in C/C++.

Although the JNI does not limit you to C or C++, you will see that `javah` assumes a C/C++ calling interface between Java and native code. If you choose to implement native code in another language, you will at least have to guarantee that arguments passed from the JVM to a `native` method honor your platform's C function call protocol. Alternatively, you could implement the C function prototypes as wrappers to calls into another language environment. In either case, it is your responsibility to get the call stack correct when calling into another language.

This books limits all discussion of `native` method programming to C and C++.

Describing the Interface to the Native Code

If you want to interface Java code with legacy applications, you presumably understand the interface to your existing code. You can describe, probably using ANSI C function prototypes, an API to your existing code. The challenge when integrating legacy code with Java is mapping the formal arguments of the Java `native` methods onto the formal arguments of your existing API. In this case, you generally have two choices:

- Model legacy data in Java and define the appropriate methods for manipulating this data. The `native` method implementation then serves as a wrapper that, possibly, massages the arguments to the `native` method and sends them to the appropriate legacy call(s).
- Map the existing interface directly onto your Java `native` method declarations.

Of these two approaches, the latter is the easiest to implement, but the former offers an opportunity to enhance the design of older applications that may not have been designed using object-oriented techniques and could benefit from Java's encouragement of these techniques.

Each of these approaches is treated in detail in later chapters. Chapter 9 discusses the former as it applies to the use of legacy C++ object libraries. Chapter 10 discusses the second as it applies to the conversion of C structures for use with Java objects.

If you are writing native code from scratch, that is, Java development is driving the design, you have much more freedom in your native code implementation. The `native` method declaration will define the API to which your native code has to be programmed. The function prototypes generated by `javah` ensure this.

Writing the Java Code

Declaring a `native` method within a Java class is as simple as the use of the `native` attribute keyword. A class can signify that any of its methods will be implemented with native code by simply preceding the method name with the `native` keyword and not supplying the body of the method. From a syntactical perspective, the `abstract` keyword and the `native` keyword are identical. Both defer the implementation of the method. In the case of an `abstract` method, of course, a subclass defines a method. In the case of a `native` method, the method is defined within a C/C++ source file.

Because a code snippet is worth a thousand words of English prose, here is an example that illustrates the point of the above paragraph. Note the use of the `native` keyword in line [1].

Listing 2.1 ***A*** `native` ***Method Declaration***

```
// File: AClassWithNativeMethods.java
// A really simple example of a class containing
// a native method.
public class AClassWithNativeMethods {
[1]    public native void theNativeMethod();
   public void aJavaMethod() {
      theNativeMethod();
   }
}
```

Beyond the `native` keyword, there is nothing special about this Java source code when defining and invoking a `native` method.

Using `javac` to Generate Class Files

Once your classes are defined in Java source files, you need to run `javac` to generate the class files. For example, to generate Java bytecode for a java class `Clazz`, the following command is used.

• User Input **Running** `javac`

```
% javac Clazz.java
```

The next step in writing `native` methods requires the class files as input, so you can't proceed without first creating them.

Using `javah` to Generate Include Files

Once the Java class files that include `native` method declarations are generated, ANSI C function prototypes for that method need to be generated. These prototypes are generated using the `javah` tool that comes with the JDK. You will want to run `javah` on all the class files that contain `native` method declarations.

`javah` takes as input the names of Java class files that contain `native` method declarations. The files named must reside in a directory named in the `CLASSPATH` environment variable or by the `-classpath` command line option of `javah`.

• User Input **Using** `javah` **to Generate Function Prototypes**

```
% javah -jni AClassWithNativeMethods
```

The `-jni` option in the above example tells `javah` to generate a JNI style function prototype. This may be stating the obvious, so a little history is in order. The `native` method support in JDK 1.0 is quite different from the support introduced in JDK 1.1. In fact, the term JNI specifically refers to an API defined by JDK 1.1 and extended in JDK 1.2. `javah`, however, still provides the ability to generate function prototypes that adhere to the `native` method programming model defined by JDK 1.0. To generate old-style function prototypes, use the `-stubs` option to `javah` and consult Appendix B for more details.

As output, `javah` produces a C header file that defines an ANSI C function prototype for each `native` method declared in the input class file.

By default, the output file name is determined by the fully-qualified class name of the input class. For a class named `Clazz`, defined in the package `myPkg.tools`, the name of the generated header file is `myPkg_tools_Clazz.h`. If `Clazz` is not a member of a package, the header file generated would be `Clazz.h`. The `-o` option may be used with `javah` to explicitly name the header file.

The generated header file is then available for inclusion (i.e. `#include`) by the C/C++ source file containing the implementation of the `native` methods. The output file from our example `javah` invocation above will be placed in a file called `AClassWithNativeMethods.h`.

Just to put a little meat on this bony description, the contents of this file are shown below.

Listing 2.2 `javah`***-Generated File*** `AClassWithNativeMethods.h`

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
#ifndef _Included_AClassWithNativeMethods
#define _Included_AClassWithNativeMethods
#ifdef __cplusplus
extern “C” {
#endif
/* Class: AClassWithNativeMethods
 * Method:    theNativeMethod
 * Signature: ()V
 */
JNIEXPORT void JNICALL
   Java_AClassWithNativeMethods_theNativeMethod(
      JNIEnv *, jobject);
#ifdef __cplusplus
}
#endif
#endif
```

Each function prototype will contain at least two arguments, always preceding any additional formal arguments to the `native` method. The first of these, the Java execution environment, is an opaque reference to the JVM execution context in which the `native` method is running. This value is of type `JNIEnv`. The second argument, sometimes an object reference, sometimes a class reference, can be thought of as a *this* reference. It refers to the object instance or class of which the method is a member. You can see that it does so using a special type `jobject` in the case of an object instance reference. In the case of a class reference, the formal argument will be of type `jclass`. All other arguments to the `native` method defined in the declaration appear after the *this* reference. There will be more on the arguments to `native` methods later.

A few other things are worth noting.

First, the generated file includes the file `jni.h`. This file contains all the relevant JNI definitions.

Second, the file is generated such that it suppresses C++ name mangling. The `#ifdef __cplusplus` takes care of this. Of course, the function bodies themselves can be written in C++. All the `#ifdef` does is inform the compiler that these functions will be called using the C language naming convention.

Third, the comments contain a line introduced by `Signature:`. What follows is a description, called a *signature*, of the function, its arguments and return value, in terms understandable by the JNI.

Fourth, even though the Java declaration of `theNativeMethod` has no arguments, the generated function prototype has two. As mentioned above, these two arguments will appear in every native function prototype generated by `javah`.

Fifth, two `#define` macros, `JNIEXPORT` and `JNICALL`, appear in the function prototype. The definitions of these are platform-dependent. At this point, you should trust that JNI does the right thing. The curious can look in `$JDK_HOME/include/`*platform*`/jni_md.h` where *platform* is, for example, `solaris` or `win32`.

Finally, in tones most ominous, you are warned not to edit a file generated by `javah`. Being a product of Catholic schools, I take seriously such warnings. You will have to buy another book if you want to learn what happens if you disregard this advice.[1]

If `javah` is rerun with the same arguments, the previously generated header file would be overwritten without warning.

---

1. Okay, I confess, I have edited a `javah`-generated file by hand. But only after having read and thoroughly understood this entire book.

Writing Native Code

Up to this point, all the steps have been platform independent. Now comes the time when you need to write a C/C++ function. That you are even reading this book means you have some job to do that most likely can not be done in Java. Chances are you have requirements specific to a particular platform, whether it be manipulating a serial port on a UNIX machine, starting a Java application using NT's Service Manager or using legacy code from an existing application. In any case, it is time to do some platform-specific work.

The output of the previous step, a header file with ANSI C function prototypes, of course, purposely constrains the writing of the function implementation. By providing a tool such as `javah`, Sun is not only making the job of writing `native` methods easier, but it is enforcing an interface between the world of Java code, the JVM and the world of C/C++ code. Writing to this interface is the task of the native code author. This interface imposes three constraints:

- Every function implementing a `native` method takes a `JNIEnv` pointer as its first argument and an object reference as its second argument.
- The types of the input arguments are defined by the JNI.
- The type of the return value is defined by the JNI.

In our example, the header file which enforces this interface and contains the function prototypes is `AClassWithNativeMethods.h`. This file must be included in the file which implements the `native` methods declared by the Java class `AClassWithNativeMethods`.

The implementation of our `native` method is shown below. Notice the inclusion of the java-generated file `AClassWithNativeMethods.h`. Also, notice that all our `native` method does is print, as promised, `Hello JNI World`.

Listing 2.3 ***Hello JNI World***

```
/* File: theNativeMethod.c
 * Implementes theNativeMethod of class
 * aClassWithNativeMethods
*/
#include <stdio.h>
#include “AClassWithNativeMethods.h”
JNIEXPORT void JNICALL
Java_AClassWithNativeMethods_theNativeMethod(
   JNIEnv* env, jobject thisObj) {
     printf(“Hello JNI World\n”);
}
```

Notice that the two macros `JNIEXPORT` and `JNICALL` find their way into the file that defines the `native` method implementation.

Building a Library

This final step in merging Java and C/C++ code is where all the beautiful machine virtuality provided by Java and all the portability provided by C/C++ can be bid farewell. In the world of compilers, linkers, and library archival tools, the purity of thought one maintains while building those elegant, pattern-like Java classes becomes sullied. One is quickly reminded of the purpose of all those computer science courses.

Okay, it is not that bad, but this is where platform specificity really gets specific. Suffice it to say at this point that you need to:

- Compile your C/C++ code, pointing to all the right places for inclusion of `jni.h`.
- Build a library containing the object files of your native function implementations.

In the Solaris world, for example, this would be a Shared Object Library. In the Windows and NT world, this would be a Dynamic Link Library. In the latter case, at least, there are integrated development environments which do a fair job of hiding some of the subtleties of this task.

These two types of libraries are identified on their respective platforms with special suffixes.

Table 2.1 ***Native Library Descriptions***

Platform	Library Description	Suffix
Solaris	Shared Object Library	`.so`
Win32	Dynamic Link Library	`.dll`

For our "Hello World" example, we will build a Shared Object Library for Solaris. First, let's assume that the environment variable `JDK_HOME` points to the directory in which the JDK is installed. This will allow `cc` to find the required include files. In the example below, note the mention of two directories using the `-I` flag. The first is sufficient to include `jni.h`. The second is required to get at `jni_md.h`, a JDK include file that defines some machine-dependent values. This file is included by virtue of including `jni.h`.

Building a Native Library on Solaris

```
% cc -I$JDK_HOME/include -I$JDK_HOME/include/solaris \
                -G -o libNative.so \
                  theNativeMethod.c
```

This `cc` command, with the `-G` option, will generate a Shared Object Library. The `lib` prefix and `so` suffix are added to satisfy both a Solaris convention and a Java requirement. In the case of multiple C sources, they would all be included on the command line.

The Win32 analog to using the `-I` flag on Solaris platforms is the `/I` option. Most integrated development environments (IDE) also provide a mechanism for supplying preprocessor directories. For the example, the Microsoft Visual Studio allows for setting preprocessor directories from the `Preprocessor Category` under the `Project->Settings->C/C++` tab.

In a similar fashion, most IDEs will explicitly support the construction of a DLL through a specially configured project. The Win32 analog to the `-G` option above is the `/dll` option to the Microsoft link-editor (`LINK.EXE`).

Loading and Invoking the native Method

As mentioned earlier, invoking a `native` method looks exactly like invoking a regular Java method. There is, however, an extra step required to load the `native` method library into the JVM.

To load native code, the class method `System.loadLibrary` must be invoked. A typical way of doing this is by placing a `static` initializer block within the class that declares the `native` methods. This is a sure way to guarantee the library is loaded and the `native` method name resolved when the `native` method is invoked. According to the *Java Language Specification* (12.4.1), code within a `static` initialization block gets executed when the class is *initialized*. A class is initialized at its first *active* use. An active use includes the invocation of a method declared by the class. This would include a `native` method invocation and, therefore, you can be guaranteed that the library containing the code that implements the `native` methods will be loaded if the loading is done within the class declaring the `native` methods.

If you elect to put the `System.loadLibrary` in the body of another class, you must be certain that class is initialized before your application tries to invoke a `native` method. Failure to load the native library before invoking a `native` method will result in an `java.lang.UnsatisfiedLinkError` being thrown by the JVM at the time of attempted `native` method invocation. If this occurs, the `getMessage` method of the thrown object will return the name of the `native` method that could not be found.

In our example, we provide the `static` block within the class `Main` that contains the `public static` method main, the first method invoked at application start-up.

Listing 2.4 ***Loading Native Library and Invoking*** `native` ***Method***

```
public class Main {
static {
[2]    System.loadLibrary(“Chap2example1”);
}
   public static void main(String[] args) {
      AClassWithNativeMethods
         c = new AClassWithNativeMethods();
      c.theNativeMethod();
   }
}
```

On UNIX systems, the Java run-time adds `lib` to the front of and appends `.so` to the argument to `System.loadLibrary`. In line [2] you can see that these strings do not appear in the argument to `System.loadLibrary`. Likewise, on Win32 systems, the suffix `.dll` is appended to the value named in the `System.loadLibrary` method call. This allows the name to be specified in a platform-independent way. The following table makes clear the translation for UNIX and Win32 platforms.

Table 2.2 ***Native Library Naming***

Platform	`System.loadLibrary` **Argument (e.g.)**	**Library Name**
UNIX	`Example`	`libExample.so`
Win32	`Example`	`Example.dll`

In order for `System.loadLibrary` to find your native library, the directory in which the Shared Object Library file resides must appear in the `LD_LIBRARY_PATH` environment variable. On UNIX systems, the `LD_LIBRARY_PATH` environment variable is a colon-separated list of directory names. This value can be verified using the `echo` command.

• User Input **Verifying `LD_LIBRARY_PATH` on Solaris**

```
% echo $LD_LIBRARY_PATH
```

On Win32 systems the `PATH` environment variable is used to locate dynamic link libraries. The `PATH` value is a list of directory names separated by semicolons. On Win32 systems, the `PATH` value can be verified using the DOS shell echo command, with a slightly different syntax.

• User Input **Verifying `PATH` value on Win32**

```
C:\> echo %PATH%
```

On both platforms, the list is searched in order for a file which matches the modified argument to `System.loadLibrary`.

Returning to Listing 2.4 we will look at how the `static` block containing the `System.loadLibrary` [2] can be modified a bit to provide more robustness when loading a native library. By wrapping the call to `System.loadLibrary` with a `try` block and a supporting `catch` clause, the program can test whether or not a library was successfully loaded and take some appropriate action. In the following example, the `try-catch` block is used to test the availability of a specific version of a native library.

Listing 2.5 *`try-catch` When Loading Native Library*

```
public class MainWithTry {
static {
   try {
[3]       System.loadLibrary("Chap2example1beta");
[4]   }  catch (Error e) {
          System.out.println("Using current version");
[5]       System.loadLibrary("Chap2example1");
   }
}
   public static void main(String[] args) {
      AClassWithNativeMethods
         c = new AClassWithNativeMethods();
      c.theNativeMethod();
   }
}
```

If the first `System.loadLibrary` fails in [3] a `LinkageError` is thrown. After being caught in [4], a second attempt to load the native library is made [5]. Note the need for catching a `java.lang.Error` object. Failure

to find a library or an error during loading a library results in a `LinkageError` being thrown which extends `java.lang.Error` and not `java.lang.Exception`.

Alternative Loading Strategy

The above example, and all of the examples in this book, show the native library being loaded in a `static` initialization block within the class in which the `native` methods are defined. This is not a requirement. The only requirement is that the `native` methods be available (i.e., loaded by the JVM) before you attempt to use any of them.

That leaves open the possibility of loading a native library within the mainline execution stream of a program. Listing 2.6 illustrates placing a call to `System.loadLibrary` in the `main` method.

Listing 2.6 ***Loading at Runtime***

```
public class MainRTLoad {
   public static void main(String[] args) {
[6]       System.loadLibrary("Chap2example1");
          AClassWithNativeMethods
             c = new AClassWithNativeMethods();
[7]       c.theNativeMethod();
   }
}
```

This approach works as long as the load [6] of the class containing the `native` methods, in this case, `AClassWithNativeMethods`, takes place before the reference to the method [7].

Loading Problems

Probably the most frustrating obstacle to early success with the JNI are problems involved in loading `native` methods properly into your Java application. In an attempt to avoid these frustrating experiences, a discussion of some of the most common problems and possible solutions will be undertaken.

Incorrect lookup path If your native library can not be found by the JVM, you will get one of two possible errors.

- If you attempt to load a library in a `static` block and this fails, a `java.lang.UnsatisfiedLinkError` will be thrown at the first attempt to invoke a method presumably in that library. The `getMessage` method of the thrown object will return the name of the method your code was trying to invoke.
- If you are loading in the main-line of your application, a `java.lang.UnsatisfiedLinkError` will be thrown as soon as you attempt to load the library. As in the above case, the `getMessage` method of the thrown object will return a message to the effect that the library could not be found.

If either of these conditions occur, the first culprit is, on UNIX systems, your `LD_LIBRARY_PATH` environment variable and, on Win32 systems, your `PATH` environment variable. Check to make sure the value contains the directory in which your library lives.

Improper library name This problem causes the same errors as above. As far as the JVM knows, it can't find the library you asked for. Check the argument to `System.loadLibrary` to make sure it is what you expect. Also make sure any `.so` or `.dll` suffixes do not appear.

One very specific example of this problem is seen when running a Java application under `java_g`. `java_g` is the version of the JVM for use with debuggers. When `java_g` is executing, `System.loadLibrary` appends `_g` to the name of the native library passed as an argument. If you are on a Win32 platform and run `java_g` and the following load is issued:

```
System.loadLibrary("myNativeLib");
```

the VM will look for a file named `myNativeLib_g.dll` in the directories named in the `PATH` environment variable. On UNIX platforms, the JVM will try to load `libmyNativeLib_g.so` from the directories named in the `LD_LIBRARY_PATH` environment variable.

Improper signature match If the number and types of the formal parameters in your `native` method declaration do not match the number and types of the parameters in the function prototype, your `native` method name will not get correctly resolved and a `java.lang.UnsatisfiedLinkError` will be thrown. This condition can occur if you have changed the `native` method declaration without using `javah` to regenerate the function prototypes. Or, if you have been naughty, this problem can arise if you manually edited the function prototypes. One particular manifestation of this problem is testing classes outside a `package` and then placing them in a `package` without using `javah` to regenerate the `native` method function prototypes.

When any of these above problems is confronted, there are some tools helpful in getting at the specific cause. On UNIX platforms, both the `dump` and `nm` tools report symbol information about a library. A analogous tool, `dumpbin`, is available on Win32 systems. These tools are most helpful in allowing you to match your expectations for `native` method names against what the library has recorded.

Building and Running Your Java Application

Hopefully, with all the potential problems behind us, we can now print `Hello JNI World` from native code invoked by a Java application.

Compiling and Running a Java Application

```
% javac Main.java
% java Main
```

Running on Solaris

```
Hello JNI World
```

Feels good, doesn't it?

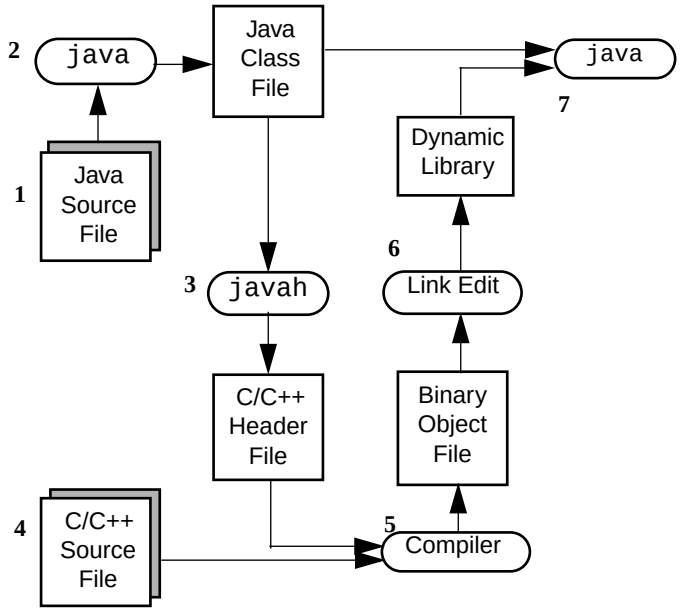
The Steps in Pictures

Now the preceding 1000 or so words will be turned into a picture.

In the diagram below, the files you produce are shaded. All others are automatically created. Tools you run appear in ovals. All other symbols represent data files.

The compile and link-edit steps have been labelled generically and shown separately. Although this is technically correct, these steps are often combined into one by the development tools supported within different environments.

Figure 2-1 ***Steps for Integrating Native Code***



The condensed version of the steps described in the preceding section and the above figure is provided below.

1. Write Java source file.
2. Run Java source file through `javac` to produce class file.
3. Run Java class file through `javah` to produce function prototypes.
4. Write C/C++ source to functions prototypes generated by `javah`.
5. Compile C/C++ source file to produce object files.
6. Run link editor on object file to produce a dynamically loadable library.
7. Run `java` on class file produced in (1) ensuring that this class file invokes Java method `System.loadLibrary` naming library created in step (6).

The following table shows a bit of the specifics for UNIX and Win32 platforms.

Table 2.3 *Platform-Specific Build Tools*[a]

Description	Solaris	Win32
Compile	`cc`	`cl`
Link-Edit	`ld`	`link`

a. Visual Age C++ from IBM uses `icc/ilink` for these steps. Borland products use `bcc/blink`.

Many compilers allow options intended for the link-edit to be named on the compiler command line. When building libraries that are intended to be loaded dynamically, these flags can be quite helpful. These flags differ across platforms. Required flags for Solaris and Win32 platforms are shown below.

Table 2.4 *Flags for Dynamic Library*

Description	Solaris	Win32
Compile	`-G`	`/LD`
Link-Edit	`-G`	`/dll`

Of course, on Win32 platforms with a feast of IDEs from which to choose, much of this detail is hidden from view. For instance, in the Microsoft Visual Studio, all you need to do is build your `native` method code in a project explicitly configured for DLL support.

Summary

Take a deep breath. If you have been paying attention, you have successfully completed writing your first Java application using `native` methods. You are at the end of Chapter 2 and you can now do something useful.

Granted, we did not touch the input from the JVM—the arguments to the native function—nor did we return any value. We have deftly sidestepped many of the details of JNI. Instead, we have focussed on the process of incorporating `native` methods into a Java application. We have gone through all the steps necessary to create a Java application which uses C/C++ code.

Additionally, the keyword `native` has been introduced and rendered innocuous. We have met `javah`, that handy little utility that turns `native` method declarations into ANSI C function prototypes. We even gave you a glimpse of how we are going to help you with a variety of platforms.

In the next chapter we will get a little more dirty with the details and see how arguments passed from Java are manipulated in a C/C++ routine.