JAVAWORLD

JAVA TIPS
By John D. Mitchell

About |
The continuation of JavaWorld's most popular long-running series, featuring new and classic Java programming tips from your peers in the Java developer community.

HOW-TO

Java Tip 23: Write native methods

Here's a step-by-step recipe for writing and using native methods on the Linux platform

JavaWorld | Jan 1, 1997 12:00 AM PT

The ability to write just one set of code in Java and have it run on every system with a Java run-time is one of Java's primary strengths. But this platform independence has one key drawback: What do we do with the vast amount of existing code? The trick is to use the so-called *native method interface*.

Writing native methods involves importing C code into your Java application. In this tip I'll walk you through the basic recipe for creating native methods and using them in a Java application.

Seven steps to native method nirvana The steps to creating native methods are as follows:

- Write Java code
- Compile Java code
- Create C header (*.h* file)
- Create C *stubs* file
- Write C code
- Create shared code library (or DLL)
- Run application

Our exercise is to write some text to the console from inside the native method. The specifics of this example will be geared toward a Unix-like system, specifically Linux. I'll point out the couple of spots where the details differ for other platforms.

Write Java code

Write your Java code as you normally would. To use native methods in your Java code, you must do two things. First, write a native method declaration for each native method that you want to use. This is just like writing the declaration of a normal Java method interface, but you must specify the *native* keyword, as follows:

```
public native void printText ();
```

The second hoop to jump through is you must explicitly load the native code library. (We will create this later.) We do this by loading the library in a class static block:

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

To put these pieces together for our example, create a file called `Happy.java` with the following contents:

```
class Happy
    {
    public native void printText ();
    static
   {
   System.loadLibrary ("happy");   /* Note lowercase of classname! */
   }
    public static void main (String[] args)
   {
   Happy happy = new Happy ();
   happy.printText ();
   }
    }
```

Compile Java code

Compile the Happy.java file:

```
% javac Happy.java
```

Create a C header file

There are various magic incantations that must be made available so that our C code can be used as a native method. The javah functionality of the Java compiler will generate the necessary declarations and such from our Happy class. This will create a Happy.h file for us to include in our C code:

```
% javah Happy
```

Create a C stubs file

In a manner reminiscent of the mangling that C++ translators do to the names of C++ methods, the Java compiler has a similar madness. To ease the pain of having to write a lot of tedious code so that our C code can be invoked from the Java run-time system, the Java compiler can generate the necessary trampoline code automatically for us:

```
% javah -stubs Happy
```

Write C code

Now, let's write the actual code to print out our greeting. By convention we put this code in a file named after our Java class with the string "Imp" appended to it. This results in HappyImp.c. Place the following into HappyImp.c:

```
#include &ltStubPreamble.h>      /* Standard native method stuff. */
#include "Happy.h"         /* Generated earlier. */
#include &ltstdio.h>         /* Standard C IO stuff. */
void Happy_printText (struct HHappy *this)
    {
    puts ("Happy New Year!!!");
    }
```

In interfacing your C code with Java, many other aspects are involved -- such as how to pass and return the myriad types. For more information, see the Java tutorial or the Hermetica Native Methods Paper (see the Resources section for URLs).

Create a shared library

This section is the most system-dependent. It seems like every platform and each compiler/linker combination has a different method of creating and using shared libraries. For folks using any of the various Microsoft Windows platforms, check the documentation for your C compiler for the nitty-gritty details.

For you Linux folks, here's how to create a shared library using GCC. First, compile the C source files that we have already created. You have to tell the compiler where to find the Java native method support files, but the main trick here is that you have to explicitly tell the compiler to produce ***P**osition **I**ndependent **C**ode*:

```
% gcc -I/usr/local/java/include -I/usr/local/java/include/genunix -
fPIC -c Happy.c HappyImp.c
```

Now, create a shared library out of the resulting object (.o) files with the following magical incantation:

```
% gcc -shared -Wl,-soname,libhappy.so.1 -o libhappy.so.1.0 Happy.o
HappyImp.o
```

Copy the shared library file to the standard short name:

```
% cp libhappy.so.1.0 libhappy.so
```

Finally, you may need to tell your dynamic linker where to find this new shared library file. Using the *bash* shell:

```
% export LD_LIBRARY_PATH=`pwd`:$LD_LIBRARY_PATH
```

Execute the application

Run the Java application as usual:

```
% java Happy
```

Well, that's all there is to it. Thanks to Tony Dering for passing on the Linux-specific incantations.

A quick design note

Before rushing off to write native methods for all of that legacy code, I would caution all of us to look carefully at the existing systems and see if there are better ways to connect them to Java. For instance, there are Java Database Connectivity (JDBC) and even higher-level solutions for accessing databases from Java. So, look at all of the tricks in your bag and use what makes sense for the project at hand.

Learn more about this topic

- JavaSoft Native Method Tuturial http://www.javasoft.com/books/Series/Tutorial/native/implementing/index.html
- Hermetica Native Methods Paper http://www.hermetica.com/technologia/java/native/