developerWorks®

Java programming with JNI

Scott Stricker

March 26, 2002

This tutorial describes and demonstrates the basic and most commonly used techniques of the Java Native Interface -- calling C or C++ code from Java programs, and calling Java code from C or C++ programs -- to help you develop your own JNI solutions quickly and efficiently.

About this tutorial

The Java Native Interface (JNI) is a native programming interface that is part of the Java Software Development Kit (SDK). JNI lets Java code use code and code libraries written in other languages, such as C and C++. The Invocation API, which is part of JNI, can be used to embed a Java virtual machine (JVM) into native applications, thereby allowing programmers to call Java code from within native code.

This tutorial deals with the two most common applications of JNI: calling C/C++ code from Java programs, and calling Java code from C/C++ programs. We'll cover both the essentials of the Java Native Interface and some of the more advanced programming challenges that can arise.

Should I take this tutorial?

This tutorial will walk you through the steps of using the Java Native Interface. You'll learn how to call native C/C++ code from within a Java application and how to call Java code from within a native C/C++ application.

All the examples use Java, C, and C++ code, and are written to be portable to both Windows and UNIX-based platforms. To follow the examples, you must have some experience programming in the Java language. In addition, you will also need some experience programming in C or C++. Strictly speaking, a JNI solution could be broken down between Java programming tasks and C/C++ programming tasks, with separate programmers doing each task. However, to fully understand how JNI works in both programming environments, you'll need to be able to understand both the Java and C/C++ code.

We'll also cover a number of advanced topics, including exception handling and multithreading with native methods. To get the most out of this part of the tutorial, you should be familiar with the Java platform's security model and have some experience in multithreaded application development.

The section on Advanced topics is separate from the more basic step-by-step introduction to JNI. Beginning Java programmers may benefit from taking the first two parts of the tutorial now and returning to the advanced topics at a later time.

Tools and components

To run the examples in this tutorial, you will need the following tools and components:

- A **Java compiler**: `javac.exe` ships with the SDK.
- A **Java virtual machine (JVM)**: `java.exe` ships with the SDK.
- A **native method C file generator**: `javah.exe` ships with the SDK.
- **Library files and native header files** that define JNI. The jni.h C header file, jvm.lib, and jvm.dll or jvm.so files all ship with the SDK.
- A **C and C++ compiler** that can create a shared library. The two most common C compilers are Visual C++ for Windows and `cc` for UNIX-based systems.

Although you may use any development environment you like, the examples we'll work with in this tutorial were written using the standard tools and components that ship with the SDK. This tutorial specifically addresses Sun's implementation of JNI, which should be regarded as the standard for JNI solutions. The details of other JNI implementations are not addressed in this tutorial.

Additional considerations

In the Java 2 SDK, the JVM and run-time support are located in the shared library file named jvm.dll (Windows) or libjvm.so (UNIX). In the Java 1.1 JDK, the JVM and run-time support were located in the shared library file named javai.dll (Windows) or libjava.so (UNIX). The version 1.1 shared libraries contained the runtime and some native methods for the class libraries, but in version 1.2, the runtime is removed and the native methods are in java.dll and libjava.so. This change is important in Java code that:

- Is written using non-JNI native methods (as with the old native method interface from the JDK 1.0 JDK)
- Uses an embedded JVM through the JNI Invocation Interface

In both cases, you'll need to relink your native libraries before they can be used with version 1.2. Note that this change should not affect JNI programmers implementing native methods -- only JNI code that invokes a JVM through the Invocation API.

If you use the jni.h file that comes with the SDK/JDK, that header file will use the default JVM in the JDK/SDK installation directory (jvm.dll or libjvm.so). Any implementation of a Java platform that supports JNI should do the same thing, or allow you to specify a JVM shared library; however the details of how this is done may be specific to that Java Platform/JVM implementation. Indeed, many implementations of JVMs do not support JNI at all.

Calling C/C++ code from Java programs

Overview

JNI allows you to use native code when an application cannot be written entirely in the Java language. The following are typical situations where you might decide to use native code:

- You want to implement time-critical code in a lower-level, faster programming language.
- You have legacy code or code libraries that you want to access from Java programs.
- You need platform-dependent features not supported in the standard Java class library.

Six steps to call C/C++ from Java code

The process of calling C or C ++ from Java programs consists of six steps. We'll go over each step in depth in the sections that follow, but let's start with a quick look at each one.

1. **Write the Java code**. We'll start by writing Java classes to perform three tasks: declare the native method we'll be calling; load the shared library containing the native code; and call the native method.
2. **Compile the Java code**. We must successfully compile the Java class or classes to bytecode before we can use them.
3. **Create the C/C++ header file**. The C/C++ header file will declare the native function signature that we want to call. This header will then be used with the C/C++ function implementation (see Step 4) to create the shared library (see Step 5).
4. **Write the C/C++ code**. This step consists of implementing the function in a C or C++ source code file. The C/C++ source file must include the header file we created in Step 3.
5. **Create the shared library file**. We'll create a shared library file from the C source code file we created in Step 4.
6. **Run the Java program**. We'll run the code and see if it works. We'll also go over some tips for dealing with the more commonly occurring errors.

Step 1: Write the Java code

We'll start by writing the Java source code file, which will declare the native method (or methods), load the shared library containing the native code, and actually call the native method.

Here's our example Java source code file, called Sample1.java:

```
public class Sample1
{
  public native int intMethod(int n);
  public native boolean booleanMethod(boolean bool);
  public native String stringMethod(String text);
  public native int intArrayMethod(int[] intArray);

  public static void main(String[] args)
  {
    System.loadLibrary("Sample1");
    Sample1 sample = new Sample1();
    int     square = sample.intMethod(5);
    boolean bool   = sample.booleanMethod(true);
    String  text   = sample.stringMethod("JAVA");
    int     sum    = sample.intArrayMethod(
                        new int[]{1,1,2,3,5,8,13} );

    System.out.println("intMethod: " + square);
    System.out.println("booleanMethod: " + bool);
    System.out.println("stringMethod: " + text);
    System.out.println("intArrayMethod: " + sum);
  }
}
```

What's happening in this code?

First of all, note the use of the `native` keyword, which can be used only with methods. The `native` keyword tells the Java compiler that a method is implemented in native code outside of the Java class in which it is being declared. Native methods can only be *declared* in Java classes, not implemented, so a native method cannot have a body.

Now, let's look at the code line by line:

- In lines 3 through 6 we declare four `native` methods.
- On line 10 we load the shared library file containing the implementation for these native methods. (We'll create the shared library file when we come to Step 5.)
- Finally, in lines 12 through 15 we call the native methods. Note that this operation is no different from the operation of calling non-native Java methods.

Note: Shared library files on UNIX-based platforms are usually prefixed with "lib". In this case, line 10 would be `System.loadLibrary("libSample1");` . Be sure to take notice of the shared library file name that you generate in Step 5: Create the shared library file.

Step 2: Compile the Java code

Next, we need to compile the Java code down to bytecode. One way to do this is to use the Java compiler, `javac`, which comes with the SDK. The command we use to compile our Java code to bytecode is:

```
javac Sample2.java
```

Step 3: Create the C/C++ header file

The third step is to create a C/C++ header file that defines native function signatures. One way to do this is to use the native method C stub generator tool, javah.exe, which comes with the SDK. This tool is designed to create a header file that defines C-style functions for each `native` method it finds in a Java source code file. The command to use here is:

```
javah Sample1
```

Results of running `javah.exe` on `Sample1.java`

Sample1.h, below, is the C/C++ header file generated by running the javah tool on our Java code:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Sample1 */

#ifndef _Included_Sample1
#define _Included_Sample1
#ifdef __cplusplus
extern "C" {
#endif

JNIEXPORT jint JNICALL Java_Sample1_intMethod
  (JNIEnv *, jobject, jint);

JNIEXPORT jboolean JNICALL Java_Sample1_booleanMethod
```

```
  (JNIEnv *, jobject, jboolean);

JNIEXPORT jstring JNICALL Java_Sample1_stringMethod
 (JNIEnv *, jobject, jstring);

JNIEXPORT jint JNICALL Java_Sample1_intArrayMethod
 (JNIEnv *, jobject, jintArray);

#ifdef __cplusplus
}
#endif
#endif
```

About the C/C++ header file

As you've probably noticed, the C/C++ function signatures in Sample1.h are quite different from the Java `native` method declarations in Sample1.java. `JNIEXPORT` and `JNICALL` are compiler-dependent specifiers for export functions. The return types are C/C++ types that map to Java types. These types are fully explained in Appendix A: JNI types

The parameter lists of all these functions have a pointer to a `JNIEnv` and a `jobject`, in addition to normal parameters in the Java declaration. The pointer to `JNIEnv` is in fact a pointer to a table of function pointers. As we'll see in Step 4, these functions provide the various faculties to manipulate Java data in C and C++.

The `jobject` parameter refers to the current object. Thus, if the C or C++ code needs to refer back to the Java side, this `jobject` acts as a reference, or pointer, back to the calling Java object. The function name itself is made by the "`Java_`" prefix, followed by the fully qualified class name, followed by an underscore and the method name.

Step 4: Write the C/C++ code

When it comes to writing the C/C++ function implementation, the important thing to keep in mind is that our signatures must be exactly like the function declarations from Sample1.h. We'll look at the complete code for both a C implementation and a C++ implementation, then discuss the differences between the two.

The C function implementation

Here is Sample1.c, an implementation written in C:

```
#include "Sample1.h"
#include <string.h>

JNIEXPORT jint JNICALL Java_Sample1_intMethod
  (JNIEnv *env, jobject obj, jint num) {
   return num * num;
}

JNIEXPORT jboolean JNICALL Java_Sample1_booleanMethod
  (JNIEnv *env, jobject obj, jboolean boolean) {
  return !boolean;
}

JNIEXPORT jstring JNICALL Java_Sample1_stringMethod
  (JNIEnv *env, jobject obj, jstring string) {
    const char *str = (*env)->GetStringUTFChars(env, string, 0);
    char cap[128];
    strcpy(cap, str);
    (*env)->ReleaseStringUTFChars(env, string, str);
    return (*env)->NewStringUTF(env, strupr(cap));
}

JNIEXPORT jint JNICALL Java_Sample1_intArrayMethod
  (JNIEnv *env, jobject obj, jintArray array) {
    int i, sum = 0;
    jsize len = (*env)->GetArrayLength(env, array);
    jint *body = (*env)->GetIntArrayElements(env, array, 0);
    for (i=0; i<len; i++)
    { sum += body[i];
    }
    (*env)->ReleaseIntArrayElements(env, array, body, 0);
    return sum;
}

void main(){}
```

The C++ function implementation

And here's Sample1.cpp, the C++ implementation:

```
#include "Sample1.h"
#include <string.h>

JNIEXPORT jint JNICALL Java_Sample1_intMethod
 (JNIEnv *env, jobject obj, jint num) {
  return num * num;
}

JNIEXPORT jboolean JNICALL Java_Sample1_booleanMethod
  (JNIEnv *env, jobject obj, jboolean boolean) {
  return !boolean;
}

JNIEXPORT jstring JNICALL Java_Sample1_stringMethod
  (JNIEnv *env, jobject obj, jstring string) {
    const char *str = env->GetStringUTFChars(string, 0);
    char cap[128];
    strcpy(cap, str);
    env->ReleaseStringUTFChars(string, str);
    return env->NewStringUTF(strupr(cap));
}

JNIEXPORT jint JNICALL Java_Sample1_intArrayMethod
  (JNIEnv *env, jobject obj, jintArray array) {
    int i, sum = 0;
    jsize len = env->GetArrayLength(array);
    jint *body = env->GetIntArrayElements(array, 0);
    for (i=0; i<len; i++)
    { sum += body[i];
    }
    env->ReleaseIntArrayElements(array, body, 0);
    return sum;
}

void main(){}
```

C and C++ function implementations compared

The C and C++ code is nearly identical; the only difference is the method used to access JNI functions. In C, JNI function calls are prefixed with "`(*env)->`" in order to de-reference the function pointer. In C++, the `JNIEnv` class has inline member functions that handle the function pointer lookup. This slight difference is illustrated below, where the two lines of code access the same function but the syntax is specialized for each language.

C syntax:	`jsize len = (*env)->GetArrayLength(env,array);`
C++ syntax:	`jsize len =env->GetArrayLength(array);`

Step 5: Create the shared library file

Next, we create a shared library file that contains the native code. Most C and C++ compilers can create shared library files in addition to machine code executables. The command you use to create the shared library file depends on the compiler you're using. Below are the commands that will work on Windows and Solaris systems.

Windows:	`cl -Ic:\jdk\include -Ic:\jdk\include \win32 -LD Sample1.c -FeSample1.dll`
Solaris:	`cc -G -I/usr/local/jdk/include -I/user/local/jdk/include/solaris Sample1.c -o Sample1.so`

Step 6: Run the Java program

The last step is to run the Java program and make sure that the code works correctly. Because all Java code must be executed in a Java virtual machine, we need to use a Java runtime environment. One way to do this is to use the Java interpreter, `java`, which comes with the SDK. The command to use is:

```
java Sample1
```

When we run the `Sample1.class` program, we should get the following result:

```
PROMPT>java Sample1
intMethod: 25
booleanMethod: false
stringMethod: JAVA
intArrayMethod: 33

PROMPT>
```

Troubleshooting

You can run into many problems when using JNI to access native code from Java programs. The three most common errors you'll encounter are:

- **A dynamic link cannot be found**. This results in the error message: `java.lang.UnsatisfiedLinkError`. This usually means that either the shared library cannot be found, or a specific native method inside the shared library cannot be found.
- **The shared library file cannot be found**. When you load the library file using the file name with the `System.loadLibrary(String libname)` method, make sure that the file name is spelled correctly and that you do *not* specify the extension. Also, make sure that the library file is accessible to the JVM by ensuring that the library file's location is in the classpath.

- **A method with the specified signature cannot be found**. Make sure that your C/C++ function implementation has a signature that is identical to the function signature in the header file.

Conclusion

Calling C or C++ native code from Java, while not trivial, is a well-integrated function in the Java platform. Although JNI supports both C and C++, the C++ interface is somewhat cleaner and is generally preferred over the C interface.

As you have seen, calling C or C++ native code requires that you give your functions special names and create a shared library file. When taking advantage of existing code libraries, it is generally not advisable to change the code. To avoid this, it is common to create *proxy code*, or a proxy class in the case of C++, that has the specially named functions required by JNI. These functions, then, can call the underlying library functions, whose signatures and implementations remain unchanged.

Calling Java code from C/C++ programs

Overview

JNI allows you to invoke Java class methods from within native code. Often, to do this, you must create and initialize a JVM within the native code using the Invocation API. The following are typical situations where you might decide to call Java code from C/C++ code:

- You want to implement platform-independent portions of code for functionality that will be used across multiple platforms.
- You have code or code libraries written in the Java language that you need to access in native applications.
- You want to take advantage of the standard Java class library from native code.

Four steps to call Java code from a C/C++ program

The four steps in the process of calling Java methods from C/C++ are as follows:

1. **Write the Java code**. This step consists of writing the Java class or classes that implement (or call other methods that implement) the functionality you want to access.
2. **Compile the Java code**. The Java class or classes must be successfully compiled to bytecode before they can be used.
3. **Write the C/C++ code**. This code will create and instantiate a JVM and call the correct Java methods.
4. **Run the native C/C++ application**. We'll run the application to see if it works. We'll also go over some tips for dealing with common errors.

Step 1: Write the Java code

We start by writing the Java source code file or files, which will implement the functionality we want to make available to the native C/C++ code.

Our Java code example, Sample2.java, is shown below:

```
 1. public class Sample2
 2. {
 3.   public static int intMethod(int n) {
 4.       return n*n;
 5.   }
 6.
 7.   public static boolean booleanMethod(boolean bool) {
 8.        return !bool;
 9.   }
10. }
```

Note that Sample2.java implements two `static` Java methods, `intMethod(int n)` and `booleanMethod(boolean bool)` (lines 3 and 7 respectively). `static` methods are class methods that are not associated with object instances. It is easier to call `static` methods because we do not have to instantiate an object to invoke them.

Step 2: Compile the Java code

Next, we compile the Java code down to bytecode. One way to do this is to use the Java compiler, `javac`, which comes with the SDK. The command to use is:

```
javac Sample1.java
```

Step 3: Write the C/C++ code

All Java bytecode must be executed in a JVM, even when running in a native application. So our C/C++ application must include calls to create a JVM and to initialize it. To assist us, the SDK includes a JVM as a shared library file (jvm.dll or jvm.so), which can be embedded into native applications.

We'll start with a look at the complete code for both the C and C++ applications, then compare the two.

A C application with embedded JVM

Sample2.c is a simple C application with an embedded JVM:

```
#include <jni.h>

#ifdef _WIN32
#define PATH_SEPARATOR ';'
#else
#define PATH_SEPARATOR ':'
#endif

int main()
{
  JavaVMOption options[1];
  JNIEnv *env;
  JavaVM *jvm;
  JavaVMInitArgs vm_args;
  long status;
  jclass cls;
  jmethodID mid;
  jint square;
  jboolean not;

  options[0].optionString = "-Djava.class.path=.";
  memset(&vm_args, 0, sizeof(vm_args));
  vm_args.version = JNI_VERSION_1_2;
  vm_args.nOptions = 1;
  vm_args.options = options;
  status = JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);

  if (status != JNI_ERR)
  {
    cls = (*env)->FindClass(env, "Sample2");
    if(cls !=0)
    { mid = (*env)->GetStaticMethodID(env, cls, "intMethod", "(I)I");
      if(mid !=0)
      { square = (*env)->CallStaticIntMethod(env, cls, mid, 5);
   printf("Result of intMethod: %d\n", square);
      }

      mid = (*env)->GetStaticMethodID(env, cls, "booleanMethod", "(Z)Z");
      if(mid !=0)
  { not = (*env)->CallStaticBooleanMethod(env, cls, mid, 1);
        printf("Result of booleanMethod: %d\n", not);
      }
    }

    (*jvm)->DestroyJavaVM(jvm);
    return 0;
  }
  else
  return -1;
}
```

A C++ application with embedded JVM

Sample2.cpp is a C++ application with an embedded JVM:

```
#include <jni.h>

#ifdef _WIN32
#define PATH_SEPARATOR ';'
#else
#define PATH_SEPARATOR ':'
#endif

int main()
{
  JavaVMOption options[1];
  JNIEnv *env;
  JavaVM *jvm;
  JavaVMInitArgs vm_args;
  long status;
  jclass cls;
  jmethodID mid;
  jint square;
  jboolean not;

  options[0].optionString = "-Djava.class.path=.";
  memset(&vm_args, 0, sizeof(vm_args));
  vm_args.version = JNI_VERSION_1_2;
  vm_args.nOptions = 1;
  vm_args.options = options;
  status = JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);

  if (status != JNI_ERR)
  {
    cls = (*env)->FindClass(env, "Sample2");
    if(cls !=0)
    {   mid = (*env)->GetStaticMethodID(env, cls, "intMethod", "(I)I");
```

```
        if(mid !=0)
        {  square = (*env)->CallStaticIntMethod(env, cls, mid, 5);
        printf("Result of intMethod: %d\n", square);
        }

        mid = (*env)->GetStaticMethodID(env, cls, "booleanMethod", "(Z)Z");
        if(mid !=0)
        {  not = (*env)->CallStaticBooleanMethod(env, cls, mid, 1);
        printf("Result of booleanMethod: %d\n", not);
        }
    }

    (*jvm)->DestroyJavaVM(jvm);
   return 0;
  }
  else
    return -1;
}
```

C and C++ implementations compared

The C and C++ code are nearly identical; the only difference is the method used to access JNI functions. In C, JNI function calls are prefixed with `(*env)->` in order to de-reference the function pointer. In C++, the `JNIEnv` class has inline member functions that handle the function pointer lookup. Thus, these two lines of code access the same function, but the syntax is specialized for each language, as shown below.

C syntax:	**`cls = (*env)->FindClass(env, "Sample2");`**
C++ syntax:	`cls = env->FindClass("Sample2");`

A closer look at the C application

We've just produced a lot of code, but what does it all do? Before we move on to Step 4, let's take a closer look at the code for our C application. We'll walk through the essential steps of preparing a native application to process Java code, embedding a JVM in a native application, then finding and calling a Java method from within that application.

Include the jni.h file

We start by including the jni.h C header file in the C application, as shown in the code sample below.

```
#include <jni.h>
```

The jni.h file contains all the type and function definitions we need for JNI on the C side.

Declare the variables

Next, we declare all the variables we want to use in the program. The `JavaVMOption options[]` holds various optional settings for the JVM. When declaring variables, be sure that you declare the `JavaVMOption options[]` array large enough to hold all the options you want to use. In this case, the only option we're using is the classpath option. We set the classpath to the current directory because in this example all of our files are in the same directory. You can set the classpath to point to any directory structure you would like to use.

Here's the code to declare the variables for Sample2.c:

```
JavaVMOption options[1];
JNIEnv *env;
JavaVM *jvm;
JavaVMInitArgs vm_args;
```

Notes:

- `JNIEnv *env` represents JNI execution environment.
- `JavaVM jvm` is a pointer to the JVM. We use this primarily to create, initialize, and destroy the JVM.
- `JavaVMInitArgs vm_args` represents various JVM arguments that we can use to initialize our JVM.

Set the initialization arguments

The `JavaVMInitArgs` structure represents initialization arguments for the JVM. You can use these arguments to customize the runtime environment before you execute your Java code. As you can see, the options are one argument and the Java version is another. We set these arguments as follows:

```
vm_args.version = JNI_VERSION_1_2;
vm_args.nOptions = 1;
vm_args.options = options;
```

Set the classpath

Next, we set the classpath for the JVM, to enable it to find the required Java classes. In this particular case, we set the classpath to the current directory, because the Sample2.class and Sample2.exe are located in the same directory. The code we use to set the classpath for Sample2.c is shown below:

```
options[0].optionString = "-Djava.class.path=.";
// same text as command-line options for the java.exe JVM
```

Set aside memory for vm_args

Before we can use `vm_args` we need to set aside some memory for it. Once we've set the memory, we can set the version and option arguments, as shown below:

```
    memset(&vm_args, 0, sizeof(vm_args));  // set aside enough memory for vm_args
    vm_args.version = JNI_VERSION_1_2;         // version of Java platform
    vm_args.nOptions = 1;                      // same as size of options[1]
    vm_args.options = options;
```

Create the JVM

With all the setup taken care of, we're ready to create a JVM. We start with a call to the method:

```
JNI_CreateJavaVM(JavaVM **jvm, void** env, JavaVMInitArgs **vm_args)
```

This method returns zero if successful or `JNI_ERR` if the JVM could not be created.

Find and load the Java classes

Once we've created our JVM, we're ready to begin running Java code in the native application. First, we need to find and load our Java class, using the `FindClass()` function, shown here:

```
cls = (*env)->FindClass(env, "Sample2");
```

The `cls` variable stores the result of the `FindClass()` function. If the class is found, the `cls` variable represents a handle to the Java class. If the class cannot be found, `cls` will be zero.

Find a Java method

Next, we want to find a method inside of the class using the `GetStaticMethodID()` function. We want to find the method `intMethod`, which takes an `int` parameter and returns an `int`. Here's the code to find `intMethod`:

```
mid = (*env)->GetStaticMethodID(env, cls, "intMethod", "(I)I");
```

The `mid` variable stores the result of the `GetStaticMethodID()` function. If the method is found, the `mid` variable represents a handle to the method. If the method cannot be found, `mid` will be zero.

Remember that in this example, we are calling `static` Java methods. That is why we're using the `GetStaticMethodID()` function. The `GetMethodID()` function does the same thing, but it is used to find instance methods.

If we were calling a constructor, the name of the method would have been "<init>". To learn more about calling a constructor, see Error handling. To learn more about the code used to specify parameter types and about how JNI types map to the Java primitive types, see Appendices .

Call a Java method

Finally, we call the Java method, as shown below:

```
square = (*env)->CallStaticIntMethod(env, cls, mid, 5);
```

The `CallStaticIntMethod()` method takes `cls` (representing our class), `mid` (representing our method), and the parameter or parameters for the method. In this case the parameter is `int` 5.

You will also run across methods of the types `CallStaticXXXMethod()` and `CallXXXMethod()`. These call static methods and member methods, respectively, replacing the variable (`XXX`) with the return type for the method (for example, `Object`, `Boolean`, `Byte`, `Char`, `Int`, `Long`, and so on).

Step 4: Run the application

Now we're ready to run the C application and make sure that the code works correctly. When you run Sample2.exe you should get a result like the following:

```
PROMPT>Sample2
Result of intMethod: 25
Result of booleanMethod: 0

PROMPT>
```

Troubleshooting

JNI's Invocation API is somewhat cumbersome because it is defined in C, a language with minimal object-oriented programming support. As a result, it is easy to run into problems. Below is a checklist that may help you avoid some of the more common errors.

- Always ensure that references are properly set. For example, when creating a JVM with the `JNI_CreateJavaVM()` method, make sure it returns a zero. Also make sure that references set with the `FindClass()` and `GetMethodID()` methods are not zero before you use them.
- Check to see that your method names are spelled correctly and that you properly mangled the method signature. Also be sure that you use `CallStaticXXXMethod()` for static methods and `CallXXXMethod()` for member methods.
- Make sure you initialize the JVM with any special arguments or options your Java class may need. For example, if your Java class requires a great deal of memory, you may need to increase the maximum heap size option.
- Always be sure to set the classpath properly. A native application using an embedded JVM must be able to find the jvm.dll or jvm.so shared library.

Conclusion

Calling Java methods from C is relatively straightforward for experienced C programmers, although it does require fairly advanced quasi-object-oriented programming techniques. Although JNI supports both C and C++, the C++ interface is slightly cleaner and is generally preferred over the C interface.

One important point to remember is that a single JVM can be used to load and execute multiple classes and methods. Creating and destroying a JVM every time you interact with Java from native code can waste resources and decrease performance.

Advanced topics

Overview

Calling native code from within a Java program compromises the Java program's portability and security. Although the compiled Java bytecode remains highly portable, the native code must be recompiled for each platform on which you intend to run the application. The native code also executes outside of the JVM, so it is not necessarily constrained by the same security protocols as Java code.

Calling Java code from within a native program is also complicated. Because the Java language is object-oriented, calling Java code from a native application typically involves object-oriented techniques. In native languages that have no support or limited support for object-oriented programming, such as C, calling Java methods can be problematic. In this section, we'll explore some of the complexities that arise when working with JNI, and look at ways to work around them.

Java strings versus C strings

Java strings are stored as sequences of 16-bit Unicode characters, while C strings are stored as sequences of 8-bit null-terminated characters. JNI provides several useful functions for converting between and manipulating Java strings and C strings. The code snippet below demonstrates how to convert C strings to Java strings:

```
/* Convert a C string to a Java String. */
char[]  str  = "To be or not to be.\n";
jstring jstr = (*env)->NewStringUTF(env, str);
```

Next, we'll look at the code to convert Java strings to C strings. Note the call to the `ReleaseStringUTFChars()` function on line 5. You should use this function to release Java strings when you're no longer using them. Be sure you always release your strings when the native code no longer needs to reference them. Failure to do so could cause a memory leak.

```
1. /* Convert a Java String into a C string. */
2. char buf[128;
3. const char *newString = (*env)->GetStringUTFChars(env, jstr, 0);
4. ...
5. (*env)->ReleaseStringUTFChars(env, jstr, newString);
```

Java arrays versus C arrays

Like strings, Java arrays and C arrays are represented differently in memory. Fortunately, a set of JNI functions provides you with pointers to the elements in arrays. The image below shows how Java arrays are mapped to the JNI C types.

The C type `jarray` represents a generic array. In C, all of the array types are really just type synonyms of `jobject`. In C++, however, all of the array types inherit from `jarray`, which in turn inherits from `jobject` . See Appendix A: JNI types for an inheritance diagram of all the C type objects.

s

Working with arrays

Generally, the first thing you want to do when dealing with an array is to determine its size. For this, you should use the `GetArrayLength()` function, which returns a `jsize` representing the array's size.

Next, you'll want to obtain a pointer to the array's elements. You can access elements in an array using the `GetXXXArrayElement()` and `SetXXXArrayElement()` functions (replace the `XXX` in the method name according to the type of the array: `Object`, `Boolean`, `Byte`, `Char`, `Int`, `Long`, and so on).

When the native code is finished using a Java array, it must release it with a call to the function `ReleaseXXXArrayElements()`. Otherwise, a memory leak may result. The code snippet below shows how to loop through an array of integers and up all the elements:

```
/* Looping through the elements in an array. */
int* elem = (*env)->GetIntArrayElements(env, intArray, 0);
for (i=0; I < (*env)->GetIntArrayLength(env, intArray); i++)
   sum += elem[i]
(*env)->ReleaseIntArrayElements(env, intArray, elem, 0);
```

Local versus global references

When programming with JNI you will be required to use references to Java objects. By default, JNI creates local references to ensure that they are liable for garbage collection. Because of this, you may unintentionally write illegal code by trying to store away a local reference so that you can reuse it later, as shown in the code sample below:

```
 1. /* This code is invalid! */
 2. static jmethodID mid;
 3.
 4. JNIEXPORT jstring JNICALL
 5. Java_Sample1_accessMethod(JNIEnv *env, jobject obj)
 6. {
 7.    ...
 8.  cls = (*env)->GetObjectClass(env, obj);
 9.    if (cls != 0)
10.       mid = (*env)->GetStaticMethodID(env, cls, "addInt", "(I)I");
11.    ...
12. }
```

This code is not valid because of line 10. `mid` is a `methodID` and `GetStaticMethodID()` returns a `methodID`. The `methodID` returned is a local reference, however, and you should not assign a local reference to a global reference. And `mid` is a global reference.

After the `Java_Sample1_accessMethod()` returns, the `mid` reference is no longer valid because it was assigned a local reference that is now out of scope. Trying to use `mid` will result in either the wrong results or a crash of the JVM.

Creating a global reference

To correct this problem, you need to create and use a global reference. A global reference will remain valid until you explicitly free it, which you must remember to do. Failure to free the reference could cause a memory leak.

Create a global reference with `NewGlobalRef()` and delete it with `DeleteGlobalRef()`, as shown in the code sample below:

```
/* This code is valid! */
static jmethodID mid;

JNIEXPORT jstring JNICALL
Java_Sample1_accessMethod(JNIEnv *env, jobject obj)
{
   ...
 cls = (*env)->GetObjectClass(env, obj);
   if (cls != 0)
   {
      mid1 = (*env)->GetStaticMethodID(env, cls, "addInt", "(I)I");
      mid = (*env)->NewGlobalRef(env, mid1);
   ...
}
```

Error handling

Using native methods in Java programs breaks the Java security model in some fundamental ways. Because Java programs run in a controlled runtime system (the JVM), the designers of the Java platform decided to help the programmer by checking common runtime errors like array indices, out-of-bounds errors, and null pointer errors. C and C++, on the other hand, use no such runtime error checking, so native method programmers must handle all error conditions that would otherwise be caught in the JVM at runtime.

For example, it is common and correct practice in Java programs to report errors to the JVM by throwing an exception. C has no exceptions, so instead you must use the exception handling functions of JNI.

JNI's exception handling functions

There are two ways to throw an exception in the native code: you can call the `Throw()` function or the `ThrowNew()` function. Before calling `Throw()`, you first need to create an object of type `Throwable`. By calling `ThrowNew()` you can skip this step because this function creates the object for you. In the example code snippet below, we throw an `IOException` using both functions:

```
/* Create the Throwable object. */
jclass cls = (*env)->FindClass(env, "java/io/IOException");
jmethodID mid = (*env)->GetMethodID(env, cls, "<init>", "()V");
jthrowable e = (*env)->NewObject(env, cls, mid);

/* Now throw the exception */
(*env)->Throw(env, e);
...

/* Here we do it all in one step and provide a message*/
(*env)->ThrowNew(env,
                 (*env)->FindClass("java/io/IOException"),
                 "An IOException occurred!");
```

The `Throw()` and `ThrowNew()` functions do not interrupt the flow of control in the native method. The exception will not actually be thrown in the JVM until the native method returns. In C you cannot use the `Throw()` and `ThrowNew()` functions to immediately exit a method on error conditions, as you can in Java programs by using the throw statement. Instead, you need to use a return statement right after the `Throw()` and `ThrowNew()` functions to exit the native method at a point of error.

JNI's exception catching functions

You may also need to catch exceptions when calling Java from C or C++. Many JNI functions throw exceptions that you may want to catch. The `ExceptionCheck()` function returns a `jboolean` indicating whether or not an exception was thrown, while the `ExceptionOccured()` method returns a `jthrowable` reference to the current exception (or `NULL` if no exception was thrown).

If you're catching exceptions, you may be handling exceptions, in which case you need to clear out the exception in the JVM. You can do this using the `ExceptionClear()` function. The `ExceptionDescribed()` function is used to display a debugging message for an exception.

Multithreading in native methods

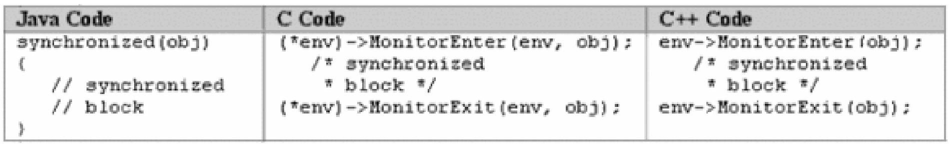
One of the more advanced issues you'll face when working with JNI is multithreading with native methods. The Java platform is implemented as a multithreaded system, even when running on platforms that don't necessarily support multithreading; so the onus is on you to ensure that your native functions are thread safe.

In Java programs, you can implement thread-safe code by using `synchronized` statements. The syntax of the `synchronized` statements allows you to obtain a lock on an object. As long as you're in the `synchronized` block, you can perform whatever data manipulation you like without fear that another thread may sneak in and access the object for which you have the lock.

JNI provides a similar structure using the `MonitorEnter()` and `MonitorExit()` functions. You obtain a monitor (lock) on the object you pass into the `MonitorEnter()` function and you keep this lock until you release it with the `MonitorExit()` function. All of the code between the `MonitorEnter()` and `MonitorExit()` functions is guaranteed to be thread safe for the object you locked.

Synchronization in native methods

The table below shows how to synchronize a block of code in Java, C, and C++. As you can see, the C and C++ functions are similar to the `synchronized` statement in the Java code.



Java Code	C Code	C++ Code
`synchronized(obj)`	`(*env)->MonitorEnter(env, obj);`	`env->MonitorEnter(obj);`
`{`	`   /* synchronized`	`   /* synchronized`
`   // synchronized`	`    * block */`	`    * block */`
`   // block`	`(*env)->MonitorExit(env, obj);`	`env->MonitorExit(obj);`
`}`		

Using `synchronized` with native methods

Another way to ensure that your native method is synchronized is to use the `synchronized` keyword when you declare your `native` method in a Java class.

Using the `synchronized` keyword will ensure that whenever the `native` method is called from a Java program, it will be `synchronized`. Although it is a good idea to mark thread-safe native methods with the `synchronized` keyword, it is generally best to always implement synchronization in the native method implementation. The primary reasons for this are as follows:

- The C or C++ code is distinct from the Java native method declaration, so if the method declaration changes (that is, if the `synchronized` keyword is ever removed) the method may suddenly no longer be thread safe.
- If anyone ever codes other native methods (or other C or C++ functions) that use the function, they may not be aware that native implementation isn't thread safe.
- If the function is used outside of a Java program as a normal C function it will not be thread safe.

Other synchronization techniques

The `Object.wait()`, `Object.notify()`, and `Object.notifyAll()` methods also support thread synchronization. Since all Java objects have the `Object` class as a parent class, all Java objects have these methods. You can call these methods from the native code as you would any other method, and use them in the same way you would use them in Java code to implement thread synchronization.

Wrap-up

Summary

The Java Native Interface is a well-designed and well-integrated API in the Java platform. It is designed to allow you to incorporate native code into Java programs as well as providing you a way to use Java code in programs written in other programming languages.

Using JNI almost always breaks the portability of your Java code. When calling native methods from Java programs, you will need to distribute native shared library files for every platform on which you intend to run your program. On the other hand, calling Java code from native programs can actually improve the portability of your application.

Appendices

Appendix A: JNI types

JNI uses several natively defined C types that map to Java types. These types can be divided into two categories: primitive types and pseudo-classes. The pseudo-classes are implemented as structures in C, but they are real classes in C++.

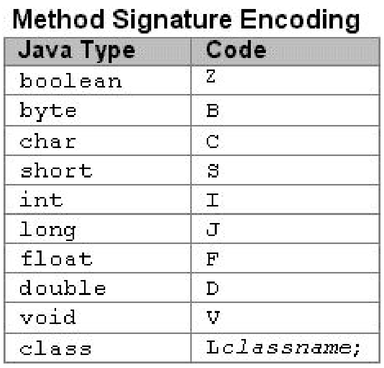
The Java primitive types map directly to C platform-dependent types, as shown here:

The C type `jarray` represents a generic array. In C, all of the array types are really just type synonyms of `jobject`. In C++, however, all of the array types inherit from `jarray`, which in turn inherits from `jobject`. The following table shows how the Java array types map to JNI C array types.

Here is an object tree that shows how the JNI pseudo-classes are related.

Appendix B: JNI method signature encoding

Native Java method parameter types are rendered, or *mangled*, into native code using the encoding specified in the table below.



Method Signature Encoding

Java Type	Code
`boolean`	`Z`
`byte`	`B`
`char`	`C`
`short`	`S`
`int`	`I`
`long`	`J`
`float`	`F`
`double`	`D`
`void`	`V`
`class`	`L`*`classname`*`;`

Notes:

- The semicolon at the end of the class type L expression is the terminator of the type expression, not a separator between expressions.
- You must use a forward slash (/) instead of a dot (.) to separate the package and class name. To specify an array type use an open bracket ([). For example, the Java method:

  ```
  boolean print(String[] parms, int n)
  ```

  has the following mangled signature:

  ```
  ([Ljava/lang/Sting;I)Z
  ```

Related topics

- The C++ Programming Language, Third Edition
- Jamsa's C/C++ Programmer's Bible
- Mastering Object-Oriented Design in C++
- Structured and Object-Oriented Techniques: An Introduction Using C++
- Complete source files, j-jni-source.zip, for this tutorial