Introduction to Native Calls

Mark Allen Weiss
Copyright 1999

1

Outline of Topics

- What Is A Native Call
- Native Call Pros and Cons
- General Setup
- Accessing methods and fields of an object
- Accessing static methods and fields of a class
- Strings
- Arrays
- Exceptions
- Invocation API

2

What Is A Native Method

- A native method is code written in another language (usually C) and called from a Java program.
- Generally you cannot use your own native methods in applets.
- The JNI (Java Native Interface) specifies a communication protocol.

3

Why Use Native Methods

- **You already have lots of tricky code already written and debugged in another language (for instance, numerical analysis libraries). You'd rather not rewrite it in Java.**
- **You need access to system devices. At some point, parts of the Java I/O library make native calls.**
- **You think Java might be too slow (probably a lame excuse)**

4

Problems With Native Methods

- **You lose portability. You must provide a native library for each supported environment.**
- **You lose safety. Native methods do not have same checks as Java methods. If your native method has a bad bug (e.g. a corrupt pointer) you are in trouble. The VM can get completely lost.**
- **Generally, a native call is untrusted.**
- **The JNI binding is not a work of beauty. Coding is cumbersome.**

5

Basic Ideas

- **A Java class declares that a particular method has a native implementation, by using the `native` keyword.**
- **A C or C++ function (the *stub*) is written to implement the keyword, using a JNI protocol that we will discuss.**
- **The stub is compiled into a shared library (DLL on Windows).**
- **The Java VM loads the library. Calls to the native method are handled by calling the stub.**

6

The Complications

- **Basic ideas are simple, as next example will illustrate.**
- **There are complications:**
  - **How are method calls made (C has no classes)?**
  - **How are parameters passed?**
  - **How is a value returned?**
  - **What about function overloading?**
  - **How can the stub throw an exception?**
  - **How do we differentiate between static and non-static members?**
  - **What about strings and arrays?**

7

Simple Stuff First

- **Example is a function called `hello`, that is called from a Java program.**
- **`hello` is declared as a native method**
- **`HelloNative` (the DLL containing `hello`) is loaded prior to first use.**

```
class HelloNative {
  native public static void hello( );
  static {
    System.loadLibrary( "HelloNative" );
  }
  public static void main( String[] args ) {
    hello( );
  }
}
```

8

Generating the Stub Specification

- **After you compile the java code, you generate the header file for the stub. This will tell you what function(s) to implement. It uses a bizarre encoding. Run (from MS-DOS window)**

  `javah HelloNative`

- **What you get is (approximately)**

```
#include <jni.h>
JNIEXPORT void JNICALL Java_HelloNative_hello
  (JNIEnv *, jclass);
```

- **There are also comments and `#ifdefs` that allow you to write both C and C++ code.**

9

Mangling

- The stub corresponds to the native method name, signature, and return type.
- There is a rule to figure it all out, but why bother? Rule takes into account:
  - Class name, Method name, weird characters in identifiers
  - Parameter types, overloading
- Remember that `javah` requires a class file generated by `javac`.

10

Some Details

- The parameters (especially `env`) in the stub are useful when the native method has parameters, return types, exceptions, or if we want to create strings and arrays. That's a future example.
- We need to implement the stub in a .c file, then compile it into a shared DLL.

```
#include <stdio.h>
#include "HelloNative.h"

JNIEXPORT void JNICALL
Java_HelloNative_hello(JNIEnv *env, jclass cls) {
  printf( "Hello world\n" );
}
```

11

More Details

- Remember to add `jdk??/include` and `jdk??/include/win32` to include path for C or C++ compilation.
- Move `.dll` file up to same directory as Java project.

12

Parameters

- Primitive parameters and return values in the Java native declaration have C equivalents in the stub.

```
int     -->  jint
double  -->  jdouble
boolean -->  jboolean
```

- Use these for portability. (A `jint` will always be 32 bits)
- `JNI_TRUE` and `JNI_FALSE` are also defined

13

Strings

- Use the `jstring` type in your stubs.
- May need to convert between C-style `char*` strings and `jstring`.
- Use `NewStringUTF` to create a new `jstring` from a C-style string. Useful for returning a `jstring`. `NewStringUTF` is accessed through `env` with a funky call. First parameter is `env`, second parameter is a C-style string.

```
JNIEXPORT jstring JNICALL
Java_Class1_GetHelloWorld(JNIEnv *env, jclass c)
{return (*env)->NewStringUTF(env,"Hello world");}
```

14

Strings as Parameters

- Need to get info about the string, and most likely, get a C-style equivalent.
- Problem: If a C-style equivalent holds a reference to the string, then garbage collector won't reclaim it.
- C-style equivalent must release its hold.
- These methods are accessed via `(*env)->`

```
const jbyte* GetStringUTFChars( JNIENV *env,
          jstring str, jboolean *isCopy );
void ReleaseStringUTFChars( JNIENV *env,
          jstring str, const jbytes *bytes );
```

15

Here's a String Concatenation

- **This is very prone to bugs, with bad pointers and memory leaks. Also, watch constness.**

```
JNIEXPORT jstring JNICALL Java_StringAdd_add (JNIEnv *env,
   jclass cl, jstring a, jstring b) {
  const char *a1 = (*env)->GetStringUTFChars( env, a, NULL );
  const char *b1 = (*env)->GetStringUTFChars( env, b, NULL );
  char *c = (char *) malloc( strlen(a1) + strlen(b1) + 1 );
  jstring result;
  strcpy( c, a1 ); strcat( c, b1 );
  result = (*env)->NewStringUTF( env, c );
  (*env)->ReleaseStringUTFChars( env, a, a1 );
  (*env)->ReleaseStringUTFChars( env, b, b1 );
  free( c );
  return result;
}
```

16

Same Code in C++

- **In C++, change `(*env)` to `env`, and remove `env` as first parameter in most calls.**

```
JNIEXPORT jstring JNICALL Java_StringAdd_add (JNIEnv *env,
   jclass cl, jstring a, jstring b) {
  const char *a1 = env->GetStringUTFChars( a, NULL );
  const char *b1 = env->GetStringUTFChars( b, NULL );
  char *c = new char[ strlen(a1) + strlen(b1) + 1 ];

  strcpy( c, a1 ); strcat( c, b1 );
  jstring result = env->NewStringUTF( c );
  env->ReleaseStringUTFChars( a, a1 );
  env->ReleaseStringUTFChars( b, b1 );
  delete [] c;
  return result;
}
```

17

Same Code in Safer C++

- **Can use C++ library classes.**

```
JNIEXPORT jstring JNICALL Java_StringAdd_add (JNIEnv *env,
   jclass cl, jstring a, jstring b) {
  const char *a1 = env->GetStringUTFChars( a, NULL );
  const char *b1 = env->GetStringUTFChars( b, NULL );

  string c = a1;
  c += a2;

  jstring result = env->NewStringUTF( c.c_str( ) );
  env->ReleaseStringUTFChars( a, a1 );
  env->ReleaseStringUTFChars( b, b1 );

  return result;
}
```

18

Accessing Instance Members

- **Not all that convoluted (the standards are going down!)**
  - **Use the second parameter (`jobject`) in the stub**
  - **Need to get a `jclass` object for the `jobject`**
  - **Need to get either a `fieldID` or `methodID`; this involves more convoluted mangling**
  - **Need to then use either `GetXXXField` or `SetXXXField` or `CallXXXMethod` (`XXX` is `int` or `double` or `Object`, etc.)**
- **On second thought, this is convoluted!**

19

Getting Fields

- **Consider an `Employee` class with private `int` fields `month`, `day`, and `year`.**
- **Here's code to print out the month:**

```
JNIEXPORT void JNICALL Java_Date_printMonth
  (JNIEnv * env, jobject obj)
{
  jint month;
  jclass class = (*env)->GetObjectClass( env, obj );
  jfieldID id_month = (*env)->GetFieldID( env, class,
                                     "month", "I" );
  month = (*env)->GetIntField( env, obj, id_month );
  printf( "Month is %d\n", month );
}
```

20

Some Details

- **Use `GetObjectClass` to obtain the `jclass` object. Pass in `env` and the object.**
- **Use `GetFieldID` to obtain the `fieldID`. Parameters are `env`, the `jclass` object, the field name, and the mangled type of the field.**
- **Use `javap -s -private ClassName` to get the mangled info. Again, there's a formula for this, but why bother. You must be exceptionally accurate with the mangling.**
- **Use `GetXXXField` to get a field. Parameters are `env`, the object, and the `fieldID`.**

21

Getting String fields

- **Use `GetObjectField`; you must typecast down to a `jstring`.**

Calling Methods

- **Here's code to print out a month by calling its `getMonth` method:**

```
JNIEXPORT void JNICALL Java_Date_printMonth
     (JNIEnv * env, jobject obj ) {
  jclass class = (*env)->GetObjectClass( env, obj );
  jmethodID id_getMonth = (*env)->GetMethodID( env,
                          class, "getMonth", "()I" );
  jint month = (*env)->CallIntMethod( env, obj, id_getMonth );
  printf( "Month = %d", month );
}
```

- **If method takes parameters, they are additional parameters to `CallXXXMethod`**
- **Last parameter to `GetMethodID` is mangled**

Accessing Static Members

- **Use `FindClass` instead of `GetObjectClass` to obtain jclass reference**

```
jclass class_math = (*env)->FindClass( env,
  "java/lang/math" );
```

- **Use `GetStaticXXXField` and `SetStaticXXXField` to access static fields.**
  - **Second parameter to access field is `jclass` instead of `jobject`.**
- **Use `GetStaticMethodID` and `CallStaticXXXMethod` to access static methods. Again, use `jclass` when invoking.**

Calling a Constructor

- **Invoke a constructor by calling `NewObject`:**

```
jobject obj = (*env)->NewObject( env, class,
  methodID, param1, param2, ... );
```

- **Get the class by using `FindClass`.**
- **To get the `methodID` pass four parameters:**
  - **`env` (as usual)**
  - **The class obtained above**
  - **`"<init>"` as the method name**
  - **The usual mangling stuff that contains the signature**

25

Arrays

- **Java arrays have corresponding C types.**

```
double[] ->    jdoubleArray
int[]    ->    jintArray
Object[] ->    jobjectArray
```

- **Can use `GetXXXArrayElement` and `SetXXXArrayElement` to access elements:**

```
jint x = (*env)->GetIntArrayElement( env, arr, 3 );
```

- **Syntax is annoying, to say the least**
- **Can call `NewXXXArray` to create a new array.**

```
jintArray a = (*env)->NewIntArray( env, 40 );
```

26

Getting a C-style Array

- **Can get a C-style array using `GetXXXArrayElements` (note pluralization).**
- **This gives a pointer to an array.**
- **This may be a copy of the array, but the copy can be copied back to the original when you call `ReleaseXXXArrayElements`.**
  - **last param is 0, `JNI_COMMIT`, or `JNI_ABORT`**
- **If you don't call `Release...`, there is no guarantee that any changes stick.**

```
jint *a = (*env)->GetIntArrayElements(env, arr, NULL);
...
(*env)->ReleaseIntArrayElements( env, arr, 0 );
```

27

Exceptions

- **Can throw an exception with `ThrowNew`. The exception will be thrown when the native method eventually returns.**
- **`ThrowNew` does not terminate the method.**

```
(*env)->ThrowNew(env, (*env)->FindClass(env,
   "java/io/IOException"), "IO error" );
```

- **Native method can check if an exception occurred by calling `ExceptionOccured`:**

```
jthrowable e = (*env)->ExceptionOccurred(env);
```

- **`e` is `NULL` if no exception; otherwise should return and let VM propagate exception**

28

Some Details

- **Object references (`jstring`, `jobject`, etc) are valid for duration of method call only, and in same thread only**
  - **use global references if you need longer duration**
    - **Create with**

```
globref = env->NewGlobalReference( ref );
```

    - **Must then eventually use**

```
env->DeleteGlobalReference( globref );
```

- **Can also get and release monitors**
  - **Obtain monitor with `env->MonitorEnter`, release with `env->MonitorExit`**

29

Summary

- **With native code, you can write pretty much anything you want to.**
- **Very difficult debugging; there's little help.**
- **Not portable.**
- **Don't use it unless you have to.**

30