JNI Examples for Android

Jurij Smakov
jurij@wooyd.org

April 25, 2009

Contents

1 Licence

This document and the code generated from it are subject to the following licence:

2 Introduction

While JNI is a pretty exciting way to greatly extend Android functionality and port existing software to it, to date there is not a lot of detailed documentation on how to create the native libraries and interface with them from the Android's Java Virtual Machine (JVM). This document aims at filling this gap, by providing a comprehensive example of creating a native JNI library, and using it from Java.

This document has been generated from source using noweb, a literate programming tool. The JNIExample.nw is the source in noweb format. It can be used to generate the document output in a variety of formats (for example, PDF), as well as generate the JNI example source code.

The complete Android project, including the source code generated from JNIExample.nw is available for download. So, if you are impatient, just grab it and check out the "Building the native library" section 5, which describes prerequisites for the build and the build procedure itself.

This document is not a replacement for other general JNI documentation. If you are not familiar with JNI, you may want to have a look at the following resources:

- Sun's Java Native Interface guide
- Java Native Interface: Programmer's Guide and Specification

Also, there are a couple of blog entries, which contain some bits of useful information:

- JNI in Android
- How to add a new module to Android

If you notice any errors or omissions (there are a couple of known bugs and unresolved issues 7), or have a suggestion on how to improve this document, feel free to contact me using the email address mentioned above.

3 Java interface

We start by defining a Java class `JNIExampleInterface`, which will provide the interface to calling the native functions, defined in a native (C++) library. The native functions corresponding to Java functions will need to have matching call signatures (i.e. the count and types of the arguments, as well as return type). The easiest way to get the correct function signatures in the native library is to first write down their Java prototypes, and then use the `javah` tool to generate the native JNI header with native function prototypes. These can be cut and pasted into the C++ file for implementation.

The Java functions which are backed by the corresponding native functions are declared in a usual way, adding a `native` qualifier. We also want to demonstrate how we could do the callbacks, i.e. calling the Java code from native code. That leads to the following high-level view of our interface class:

2 ⟨*JNIExampleInterface.java* 2⟩≡

```
package org.wooyd.android.JNIExample;

import android.os.Handler;
import android.os.Bundle;
import android.os.Message;
import org.wooyd.android.JNIExample.Data;

public class JNIExampleInterface {
    static Handler h;
    ⟨Example constructors 3a⟩
    ⟨Example native functions 3b⟩
    ⟨Example callback 3c⟩
}
```

This code is written to file `JNIExampleInterface.java`.

One valid question about this definition is why we need a `Handler` class attribute. It turns out that it will come in handy in situations, when the native library wants to pass some information to the Java process through a callback. If the callback will be called by a native thread (for extended discussion see "Calling Java functions" section 4.2), and then will try to modify the application's user interface (UI) in any way, an exception will be thrown, as Android only allows the thread which created the UI (the UI thread) to modify it. To overcome this problem we are going to use the message-passing interface provided by `Handler` to dispatch the data received by a callback to the UI thread, and allow it to do the UI modifications. In order for this to work, we are going to accept a `Handler` instance as an argument for non-trivial constructor (reasons for keeping trivial one will become apparent later), and save it in a class attribute, and that's pretty much the only task for the constructor:

3a ⟨*Example constructors* 3a⟩≡ (2)
```
public JNIExampleInterface() {}
public JNIExampleInterface(Handler h) {
    this.h = h;
}
```

To illustrate various argument-passing techniques, we define three native functions:

- `callVoid()`: takes no arguments and returns nothing;
- `getNewData()`: takes two arguments and constructs a new class instance using them;
- `getDataString()`: extracts a value from an object, which is passed as an argument.

3b ⟨*Example native functions* 3b⟩≡ (2)
```
public static native void callVoid();
public static native Data getNewData(int i, String s);
public static native String getDataString(Data d);
```

The callback will receive a string as an argument, and dispatch it to the `Handler` instance recorded in the constructor, after wrapping it in a `Bundle`:

3c ⟨*Example callback* 3c⟩≡ (2)
```
public static void callBack(String s) {
    Bundle b = new Bundle();
    b.putString("callback_string", s);
    Message m = Message.obtain();
    m.setData(b);
    m.setTarget(h);
    m.sendToTarget();
}
```

We also need a definition of a dummy `Data` class, used purely for illustrative purposes:

4a ⟨*Data.java* 4a⟩≡

```
package org.wooyd.android.JNIExample;

public class Data {
    public int i;
    public String s;
    public Data() {}
    public Data(int i, String s) {
        this.i = i;
        this.s = s;
    }
}
```

This code is written to file `Data.java`.

After the source files `Data.java` and `JNIExampleInterface.java` are compiled, we can generate the JNI header file, containing the prototypes of the native functions, corresponding to their Java counterparts:

```
$ javac -classpath /path/to/sdk/android.jar \
        org/wooyd/android/JNIExample/*.java
$ javah -classpath . org.wooyd.android.JNIExample.JNIExampleInterface
```

4 Native library implementation

At a high level, the Java library (consisting, in this case, of a single source file `JNIExample.cpp`) will look like that:

4b ⟨*JNIExample.cpp* 4b⟩≡

```
⟨JNI includes 5a⟩
⟨Miscellaneous includes 5b⟩
⟨Global variables 5c⟩
#ifdef __cplusplus
extern "C" {
#endif
⟨callVoid implementation 6⟩
⟨getNewData implementation 9b⟩
⟨getDataString implementation 10⟩
⟨initClassHelper implementation 12a⟩
⟨JNIOnLoad implementation 11⟩
#ifdef __cplusplus
}
#endif
```

This code is written to file `JNIExample.cpp`.

4.1 Headers and global variables

The following includes define the functions provided by Android's version of JNI, as well as some useful helpers:

5a ⟨*JNI includes* 5a⟩≡ (4b)

```
#include <jni.h>
#include <JNIHelp.h>
#include <android_runtime/AndroidRuntime.h>
```

Various other things which will come in handy:

5b ⟨*Miscellaneous includes* 5b⟩≡ (4b)

```
#include <string.h>
#include <unistd.h>
#include <pthread.h>
```

It is useful to have some global variables to cache things which we know will not change during the lifetime of our program, and can be safely used across multiple threads. One of such things is the JVM handle. We can retrieve it every time it's needed (for example, using `android::AndroidRuntime::getJavaVM()` function), but as it does not change, it's better to cache it.

We can also use global variables to cache the references to required classes. As described below, it is not always easy to do class resolution in native code, especially when it is done from native threads (see "Calling Java functions" section 4.2 for details). Here we are just providing the global variables to hold instances of `Data` and `JNIExampleInterface` class objects, as well as defining some constant strings which will come in handy:

5c ⟨*Global variables* 5c⟩≡ (4b)

```
static JavaVM *gJavaVM;
static jobject gInterfaceObject, gDataObject;
const char *kInterfacePath = "org/wooyd/android/JNIExample/JNIExampleInterface";
const char *kDataPath = "org/wooyd/android/JNIExample/Data";
```

4.2 Calling Java functions from Java and native threads

The `callVoid()` function is the simplest one, as it does not take any arguments, and returns nothing. We will use it to illustrate how the data can be passed back to Java through the callback mechanism, by calling the Java `callBack()` function.

At this point it is important to recognize that there are two distinct possibilities here: the Java function may be called either from a thread which originated in Java or from a native thread, which has been started in the native code, and of which JVM has no knowledge of. In the former case the call may be performed directly, in the latter we must first attach the native thread to the JVM. That requires an additional layer, a native callback handler, which will do the right thing in either case. We will also need a function to create the native thread, so structurally the implementation will look like this:

6 ⟨*callVoid implementation* 6⟩≡ (4b)
  ⟨*Callback handler* 7⟩
  ⟨*Thread start function* 8⟩
  ⟨*callVoid function* 9a⟩

Native callback handler gets the JNI environment (attaching the native thread if necessary), uses a cached reference to the `gInterfaceObject` to get to `JNIExampleInterface` class, obtains `callBack()` method reference, and calls it:

7 ⟨*Callback handler 7*⟩≡ (6)

```
static void callback_handler(char *s) {
    int status;
    JNIEnv *env;
    bool isAttached = false;

    status = gJavaVM->GetEnv((void **) &env, JNI_VERSION_1_4);
    if(status < 0) {
        LOGE("callback_handler: failed to get JNI environment, "
             "assuming native thread");
        status = gJavaVM->AttachCurrentThread(&env, NULL);
        if(status < 0) {
            LOGE("callback_handler: failed to attach "
                 "current thread");
            return;
        }
        isAttached = true;
    }
    /* Construct a Java string */
    jstring js = env->NewStringUTF(s);
    jclass interfaceClass = env->GetObjectClass(gInterfaceObject);
    if(!interfaceClass) {
        LOGE("callback_handler: failed to get class reference");
        if(isAttached) gJavaVM->DetachCurrentThread();
        return;
    }
    /* Find the callBack method ID */
    jmethodID method = env->GetStaticMethodID(
        interfaceClass, "callBack", "(Ljava/lang/String;)V");
    if(!method) {
        LOGE("callback_handler: failed to get method ID");
        if(isAttached) gJavaVM->DetachCurrentThread();
        return;
    }
    env->CallStaticVoidMethod(interfaceClass, method, js);
    if(isAttached) gJavaVM->DetachCurrentThread();
}
```

A few comments are in order:

- The JNI environment, returned by the JNI `GetEnv()` function is unique for each thread, so must be retrieved every time we enter the function. The `JavaVM` pointer, on the other hand, is per-program, so can be cached (you will see it done in the `JNI_OnLoad()` function), and safely used across threads.
- When we attach a native thread, the associated Java environment comes with a bootstrap class loader. That means that even if we would try to get a class reference in the function (the normal way to do it would be to use `FindClass()` JNI function), it would trigger an exception. Because of that we use a cached copy of `JNIExampleInterface` object to get a class reference (amusingly, we cannot cache the reference to the class itself, as any attempt to use it triggers an exception from JVM, who thinks that such reference should not be visible to native code). This caching is also done in `JNI_OnLoad()`, which might be the only function called by Android Java implementation with a functional class loader.
- In order to retrieve the method ID of the `callBack()` method, we need to specify its name and JNI signature. In this case the signature indicates that the function takes a `java.lang.String` object as an argument, and returns nothing (i.e. has return type `void`). Consult JNI documentation for more information on function signatures, one useful tip is that you can use `javap` utility to look up the function signatures of non-native functions (for native functions the signature information is already included as comments into the header, generated by `javah`).
- Someone more paranoid than me could use locking to avoid race conditions associated with setting and checking of the `isAttached` variable.

In order to test calling from native threads, we will also need a function which is started in a separate thread. Its only role is to call the callback handler:

8 ⟨*Thread start function* 8⟩≡ (6)
```
void *native_thread_start(void *arg) {
    sleep(1);
    callback_handler((char *) "Called from native thread");
}
```

We now have all necessary pieces to implement the native counterpart of the callVoid() function:

9a ⟨*callVoid function* 9a⟩≡ (6)

```
/*
 * Class:     org_wooyd_android_JNIExample_JNIExampleInterface
 * Method:    callVoid
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_org_wooyd_android_JNIExample_JNIExampleInterface_callVoid
  (JNIEnv *env, jclass cls) {
    pthread_t native_thread;

    callback_handler((char *) "Called from Java thread");
    if(pthread_create(&native_thread, NULL, native_thread_start, NULL)) {
        LOGE("callVoid: failed to create a native thread");
    }
}
```

4.3 Implementation of other native functions

The getNewData() function illustrates creation of a new Java object in the native library, which is then returned to the caller. Again, we use a cached Data object reference in order to obtain the class and create a new instance.

9b ⟨*getNewData implementation* 9b⟩≡ (4b)

```
/*
 * Class:     org_wooyd_android_JNIExample_JNIExampleInterface
 * Method:    getNewData
 * Signature: (ILjava/lang/String;)Lorg/wooyd/android/JNIExample/Data;
 */
JNIEXPORT jobject JNICALL Java_org_wooyd_android_JNIExample_JNIExampleInterface_getNewData
  (JNIEnv *env, jclass cls, jint i, jstring s) {
    jclass dataClass = env->GetObjectClass(gDataObject);
    if(!dataClass) {
        LOGE("getNewData: failed to get class reference");
        return NULL;
    }
    jmethodID dataConstructor = env->GetMethodID(
        dataClass, "<init>", "(ILjava/lang/String;)V");
    if(!dataConstructor) {
        LOGE("getNewData: failed to get method ID");
        return NULL;
    }
    jobject dataObject = env->NewObject(dataClass, dataConstructor, i, s);
    if(!dataObject) {
        LOGE("getNewData: failed to create an object");
        return NULL;
    }
    return dataObject;
}
```

The getDataString() function illustrates how a value stored in an object's attribute can be retrieved in a native function.

10 ⟨*getDataString implementation* 10⟩≡ (4b)

```
/*
 * Class:     org_wooyd_android_JNIExample_JNIExampleInterface
 * Method:    getDataString
 * Signature: (Lorg/wooyd/android/JNIExample/Data;)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_org_wooyd_android_JNIExample_JNIExampleInterface_getDataString
  (JNIEnv *env, jclass cls, jobject dataObject) {
    jclass dataClass = env->GetObjectClass(gDataObject);
    if(!dataClass) {
        LOGE("getDataString: failed to get class reference");
        return NULL;
    }
    jfieldID dataStringField = env->GetFieldID(
        dataClass, "s", "Ljava/lang/String;");
    if(!dataStringField) {
        LOGE("getDataString: failed to get field ID");
        return NULL;
    }
    jstring dataStringValue = (jstring) env->GetObjectField(
        dataObject, dataStringField);
    return dataStringValue;
}
```

4.4 The `JNI_OnLoad()` function implementation

The `JNI_OnLoad()` function must be provided by the native library in order for the JNI to work with Android JVM. It will be called immediately after the native library is loaded into the JVM. We already mentioned a couple of tasks which should be performed in this function: caching of the global `JavaVM` pointer and caching of the object instances to enable us to call into Java. In addition, any native methods which we want to call from Java must be registered, otherwise Android JVM will not be able to resolve them. The overall structure of the function thus can be written down as follows:

11 ⟨*JNIOnLoad implementation* 11⟩≡ (4b)

```
jint JNI_OnLoad(JavaVM* vm, void* reserved)
{
    JNIEnv *env;
    gJavaVM = vm;
    LOGI("JNI_OnLoad called");
    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        LOGE("Failed to get the environment using GetEnv()");
        return -1;
    }
    ⟨Class instance caching 12b⟩
    ⟨Native function registration 13⟩
    return JNI_VERSION_1_4;
}
```

We need some way to cache a reference to a class, because native threads do not have access to a functional classloader. As explained above, we can't cache the class references themselves, as it makes JVM unhappy. Instead we cache *instances* of these classes, so that we can later retrieve class references using `GetObjectClass()` JNI function. One thing to remember is that these objects must be protected from garbage-collecting using `NewGlobalRef()`, as that guarantees that they will remain available to different threads during JVM lifetime. Creating the instances and storing them in the global variables is the job for the `initClassHelper()` function:

12a ⟨*initClassHelper implementation* 12a⟩≡ (4b)

```
void initClassHelper(JNIEnv *env, const char *path, jobject *objptr) {
    jclass cls = env->FindClass(path);
    if(!cls) {
        LOGE("initClassHelper: failed to get %s class reference", path);
        return;
    }
    jmethodID constr = env->GetMethodID(cls, "<init>", "()V");
    if(!constr) {
        LOGE("initClassHelper: failed to get %s constructor", path);
        return;
    }
    jobject obj = env->NewObject(cls, constr);
    if(!obj) {
        LOGE("initClassHelper: failed to create a %s object", path);
        return;
    }
    (*objptr) = env->NewGlobalRef(obj);

}
```

With this function defined, class instance caching is trivial:

12b ⟨*Class instance caching* 12b⟩≡ (11)

```
    initClassHelper(env, kInterfacePath, &gInterfaceObject);
    initClassHelper(env, kDataPath, &gDataObject);
```

In order to register the native functions, we create an array of `JNINativeMethod` structures, which contain function names, signatures (they can be simply copied from the comments, generated by `javah`), and pointers to the implementing functions. This array is then passed to Android's `registerNativeMethods()` function:

13 ⟨*Native function registration* 13⟩≡ (11)

```
JNINativeMethod methods[] = {
    {
        "callVoid",
        "()V",
        (void *) Java_org_wooyd_android_JNIExample_JNIExampleInterface_callVoid
    },
    {
        "getNewData",
        "(ILjava/lang/String;)Lorg/wooyd/android/JNIExample/Data;",
        (void *) Java_org_wooyd_android_JNIExample_JNIExampleInterface_getNewData
    },
    {
        "getDataString",
        "(Lorg/wooyd/android/JNIExample/Data;)Ljava/lang/String;",
        (void *) Java_org_wooyd_android_JNIExample_JNIExampleInterface_getDataString
    }
};
if(android::AndroidRuntime::registerNativeMethods(
    env, kInterfacePath, methods, NELEM(methods)) != JNI_OK) {
    LOGE("Failed to register native methods");
    return -1;
}
```

5 Building the native library

In order to build the native library, you need to include Android's native headers and link against native libraries. The only way I know to get those is to check out and build the entire Android source code, and then build it. Procedure is described in detail at Android Get Source page. Make sure that you use the branch tag matching your SDK version, for example code in the `release-1.0` branch matches Android 1.1 SDK.

For an example of `CXXFLAGS` and `LDFLAGS` you need to use to create a shared library with Android toolchain, check out the `Makefile`, included in the example project tarball. They are derived from `build/core/combo/linux-arm.mk` in Android source.

You will probably want to build the entire example project, so you will need a copy of the SDK as well. This code has been tested to build with Android's 1.1 SDK and run on the currently released version of the phone. Once you downloaded the SDK and the example tarball and unpacked them, you can build the project using the command

```
ANDROID_DIR=/path/to/android/source SDK_DIR=/path/to/sdk make
```

6 Using native functions in Java code

We will now create a simple activity, taking advantage of the JNI functions. One non-trivial task we will have to do in `onCreate()` method of the activity is to load the native JNI library, to make the functions defined there accessible to Java. Overall structure:

14 ⟨*JNIExample.java* 14⟩≡

```
  package org.wooyd.android.JNIExample;
  ⟨Imports 15a⟩
  public class JNIExample extends Activity
  {
      TextView callVoidText, getNewDataText, getDataStringText;
      Button callVoidButton, getNewDataButton, getDataStringButton;
      Handler callbackHandler;
      JNIExampleInterface jniInterface;

      @Override
      public void onCreate(Bundle savedInstanceState)
      {
           super.onCreate(savedInstanceState);
           setContentView(R.layout.main);
           ⟨Load JNI library 16a⟩
           ⟨callVoid demo 16b⟩
           ⟨getNewData demo 17b⟩
           ⟨getDataString demo 17c⟩
      }
  }
```

This code is written to file `JNIExample.java`.

Imports needed to draw the UI and display it to the user:

15a ⟨*Imports* 15a⟩≡ (14) 15b ▷
```
import android.app.Activity;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
```

Imports needed to enable communication between the Java callback and the UI thread:

15b ⟨*Imports* 15a⟩+≡ (14) ◁ 15a 15c ▷
```
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
```

Imports for manipulation with the native library:

15c ⟨*Imports* 15a⟩+≡ (14) ◁ 15b 15d ▷
```
import java.util.zip.*;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.FileOutputStream;
import java.io.File;
```

We will also need access to our JNI interface class and toy `Data` class:

15d ⟨*Imports* 15a⟩+≡ (14) ◁ 15c 15e ▷
```
import org.wooyd.android.JNIExample.JNIExampleInterface;
import org.wooyd.android.JNIExample.Data;
```

Logging utilities will also come in handy:

15e ⟨*Imports* 15a⟩+≡ (14) ◁ 15d
```
import android.util.Log;
```

At this time the only officialy supported way to create an Android application is by using the Java API. That means, that no facilities are provided to easily build and package shared libraries, and automatically load them on application startup. One possible way to include the library into the application package (file with extension `.apk`) is to place it into the `assets` subdirectory of the Android project, created with `activitycreator`. During the package build it will be automatically included into the APK package, however we still will have to load it by hand when our application starts up. Luckily, the location where APK is installed is known, and APK is simply a ZIP archive, so we can extract the library file from Java and copy it into the application directory, allowing us to load it:

16a ⟨*Load JNI library* 16a⟩≡ (14)

```
try {
    String cls = "org.wooyd.android.JNIExample";
    String lib = "libjniexample.so";
    String apkLocation = "/data/app/" + cls + ".apk";
    String libLocation = "/data/data/" + cls + "/" + lib;
    ZipFile zip = new ZipFile(apkLocation);
    ZipEntry zipen = zip.getEntry("assets/" + lib);
    InputStream is = zip.getInputStream(zipen);
    OutputStream os = new FileOutputStream(libLocation);
    byte[] buf = new byte[8092];
    int n;
    while ((n = is.read(buf)) > 0) os.write(buf, 0, n);
    os.close();
    is.close();
    System.load(libLocation);
} catch (Exception ex) {
     Log.e("JNIExample", "failed to install native library: " + ex);
}
```

The rest simply demonstrates the functionality, provided by the native library, by calling the native functions and displaying the results. For the `callVoid()` demo we need to initialize a handler first, and pass it to the JNI interface class, to enable us to receive callback messages:

16b ⟨*callVoid demo* 16b⟩≡ (14) 17a ▷

```
callVoidText = (TextView) findViewById(R.id.callVoid_text);
callbackHandler = new Handler() {
    public void handleMessage(Message msg) {
        Bundle b = msg.getData();
        callVoidText.setText(b.getString("callback_string"));
    }
};
jniInterface = new JNIExampleInterface(callbackHandler);
```

We also set up a button which will call `callVoid()` from the native library when pressed:

17a ⟨*callVoid demo* 16b⟩+≡ (14) ◁16b

```
callVoidButton = (Button) findViewById(R.id.callVoid_button);
callVoidButton.setOnClickListener(new Button.OnClickListener() {
    public void onClick(View v) {
        jniInterface.callVoid();

    }
});
```

For `getNewData()` we pass the parameters to the native function and expect to get the `Data` object back:

17b ⟨*getNewData demo* 17b⟩≡ (14)

```
getNewDataText = (TextView) findViewById(R.id.getNewData_text);
getNewDataButton = (Button) findViewById(R.id.getNewData_button);
getNewDataButton.setOnClickListener(new Button.OnClickListener() {
    public void onClick(View v) {
        Data d = jniInterface.getNewData(42, "foo");
        getNewDataText.setText(
            "getNewData(42, \"foo\") == Data(" + d.i + ", \"" + d.s + "\")");
    }
});
```

And pretty much the same for `getDataString()`:

17c ⟨*getDataString demo* 17c⟩≡ (14)

```
getDataStringText = (TextView) findViewById(R.id.getDataString_text);
getDataStringButton = (Button) findViewById(R.id.getDataString_button);
getDataStringButton.setOnClickListener(new Button.OnClickListener() {
    public void onClick(View v) {
        Data d = new Data(43, "bar");
        String s = jniInterface.getDataString(d);
        getDataStringText.setText(
            "getDataString(Data(43, \"bar\")) == \"" + s + "\"");
    }
});
```

Try pushing the buttons and see whether it actually works!

7 Unresolved issues and bugs

Even though the example is fully functional, there are a couple unresolved issues remaining, which I was not able to figure out so far. Problems appear when you start the activity, then press the Back button to hide it, and then start it again. In my experience, calls to native functions in such restarted activity will fail spectacularly. `callVoid()` simply crashes with a segmentation fault, while calls to `getNewData()` and `getDataString()` cause JVM to abort with an error, because it is no longer happy with the globally cached object reference. It appears that activity restart somehow invalidates our cached object references, even though they are protected with `NewGlobalRef()`, and the activity is running within the original JVM (activity restart does not mean that JVM itself is restarted). I don't have a good explanation on why that happens, so if you have any ideas, please let me know.