Android apps are typically written in Java, with its elegant object-oriented design. However, at times, you need to overcome the limitations of Java, such as memory management and performance, by programming directly into Android native interface. Android provides Native Development Kit (NDK) to support native development in C/C++, besides the Android Software Development Kit (Android SDK) which supports Java.

[TODO] more.

NDK is a complex and advanced topics. I assume that your are familiar with:

1. Android, Java and C/C++.
2. Java Native Interface (JNI). Read "[Java Native Interface](http://www3.ntu.edu.sg/home/ehchua/programming/java/JavaNativeInterface.html)".
3. (For Windows) Cygwin. Read "[How to install Cygwin and MinGW](http://www3.ntu.edu.sg/home/ehchua/programming/howto/Cygwin_HowTo.html)".

**2.  Installing the Native Development Kit (NDK)**

The NDK provides all the tools (compilers, libraries, and header files) to build apps that access the device natively. Native code (in C/C++) is necessary for high performance to overcome the limitations in Java's memory management and performance.

Read "Android NDK" @ <http://developer.android.com/tools/sdk/ndk/index.html>.

To install Android NDK:

1. Setting up all the necessary tools for Android programming, such as JDK, Eclipse, Android SDK, Eclipse ADT (Read "[How to install Android SDK and Get Started](http://www3.ntu.edu.sg/home/ehchua/programming/android/Android_HowTo.html)"); and (for Windows Users) Cygwin (Read "[How to install Cygwin](http://www3.ntu.edu.sg/home/ehchua/programming/howto/Cygwin_HowTo.html)" and "[GCC and Make](http://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html)").
2. Download the Android NDK from <http://developer.android.com/tools/sdk/ndk/index.html> (e.g., android-ndk-r8-windows.zip).
3. Unzip the downloaded zip file into a directory of your choice (e.g., d:\myproject). The NDK will be unzipped as d:\myproject\android-ndk-r8. I shall denote the installed directory as <NDK\_HOME>.
4. Include the NDK installed directory in the PATH environment variable.

**3.  Writing a Hello-world Android NDK Program**

**Step 0: Read the Documentation**

Read "Android NDK" @ <http://developer.android.com/tools/sdk/ndk/index.html>.

Read the NDK documentation "documentation.html" @ Android NDK's installed directory. The NDK documentation is kept in the "docs" sub-directory.

The steps in building an Android NDK app are:

1. Create a sub-directory called "jni" and place all the native sources here.
2. Create a "Android.mk" to describe your native sources to the NDK build system.
3. Build your native code by running the "ndk-build" (in NDK installed directory) script from your project's directory. The build tools copy the stripped, shared libraries needed by your application to the proper location in the application's project directory.
4. Finally, compile and run your application using the SDK tools in the usual way. The SDK build tools will package the shared libraries in the application's deployable ".apk" file.

Study the sample programs provided in "samples" directory, in particular the "hello-jni".

**Step 1: Write an Android JNI program**

In this example, we shall create an activity, that calls a native method to obtain a string and displays the string on a TextView.

Create an Android project called "AndroidHelloJNI", with application name "Hello JNI" and package "com.mytest". Create an activity called "JNIActivity" with Layout name "activity\_jni" and Title "Hello JNI".

Replaced the "JNIActivity.java" as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | package com.mytest;    import android.os.Bundle;  import android.app.Activity;  import android.widget.TextView;    public class JNIActivity extends Activity {    static {  System.loadLibrary("myjni"); // "myjni.dll" in Windows, "libmyjni.so" in Unixes  }    // A native method that returns a Java String to be displayed on the  // TextView  public native String getMessage();    @Override  public void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  // Create a TextView.  TextView textView = new TextView(this);  // Retrieve the text from native method getMessage()  textView.setText(getMessage());  setContentView(textView);  }  } |

This JNI program uses a static initializer to load a shared library ("myjni.dll" in Windows or "libmyjni.so" in Unixes). It declares a native method called getMessage(), which returns a String to be as the TextView's message. The onCreate() method declares a TextView, and invokes the native method getMessage() to set its text.

**Step 2: Generating C/C++ Header File using "javah" Utility**

Create a folder "jni" under the project's root (right-click on the project ⇒ New ⇒ Folder). Create a sub-folder "include" under "jni" for storing the header files.

Run "javah" utility (from a CMD shell) to create C/C++ header called "HelloJNI.h":

> **javah --help**

......

// Change directory to <project-root>/jni/include

> **javah -classpath ../../bin/classes;<ANDROID\_SDK\_HOME>\platforms\android-<*xx*>\android.jar**

**-o HelloJNI.h com.mytest.JNIActivity**

* -classpath: in our case, we need the JNIActivity.class which is kept in "<project-root>\bin\classes"; and its superclass Android.app.Activity.class which is kept in android.jar under the Android SDK.
* -o: to set the output filename.
* You need to use the fully-qualified name "com.mytest.JNIActivity".

The header file contains a function prototype:

JNIEXPORT jstring JNICALL **Java\_com\_mytest\_JNIActivity\_getMessage**(JNIEnv \*, jobject);

The native method getMessage() maps to the above header in the native code, in the form of Java\_<*fully-qualified-name*>\_methodName, with dots replaced by underscores.

**Step 2: C Implementation - HelloJNI.c**

Create the following C program called "HelloJNI.c" under the "jni" directory (right-click on the "jni" folder ⇒ New ⇒ File):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #include <jni.h>  #include "include/HelloJNI.h"    JNIEXPORT jstring JNICALL **Java\_com\_mytest\_JNIActivity\_getMessage**  (JNIEnv \*env, jobject thisObj) {  return (\*env)->NewStringUTF(env, "Hello from native code!");  } |

The native program gets and returns a JNI jstring via JNI environment interface function NewStringUTF() with an input C-string "Hello from native code!". Read "[Java Native Interface (JNI)](http://www3.ntu.edu.sg/home/ehchua/programming/java/JavaNativeInterface.html)" for details.

**Step 3: Create an Android makefile - Android.mk**

Create an Android makefile called "Android.mk" under the "jni" directory (right-click on "jni" folder ⇒ New ⇒ File), as follows:

LOCAL\_PATH := $(call my-dir)

include $(CLEAR\_VARS)

LOCAL\_MODULE := myjni

LOCAL\_SRC\_FILES := HelloJNI.c

include $(BUILD\_SHARED\_LIBRARY)

In the above makefile, "myjni" is the name of our shared library (used in System.loadLibrary()), and "HelloJNI.c" is the source file.

**Step 4: Build NDK**

Start a CMD shell, change directory to the project's root directory, and run "ndk-build" script provided by Android NDK (the Android NDK installed directory shall be in the PATH).

// Change directory to <project-root>

> **ndk-build**

Compile thumb : myjni <= HelloJNI.c

SharedLibrary : libmyjni.so

Install : libmyjni.so => libs/armeabi/libmyjni.so

NOTES:

* Use "ndk-build --help" to display the command-line options.
* Use "ndk-build V=1" to display the build messages.
* Use "ndk-build -B" to perform a force re-built.

**Step 5: Run the Android App**

Run the android app, via "Run As" ⇒ "Android Application". You shall see the message from the native program appears on the screen.

Check the "LogCat" panel to confirm that the shared library "libmyjni.so" is loaded.

...: Trying to load lib /data/data/com.example.androidhellojni/lib/libmyjni.so ...

...: Added shared lib /data/data/com.example.androidhellojni/lib/libmyjni.so ...