HINDARIUM

Developing Objective-C apps for Android using Mac OS X

Hindarium Author: Ivan Vučica ivan@vucica.net http://ivan.vucica.net

Table of Contents

Ivan Vučica

Understanding, setting up and testing the Java environment

This text is produced out of dissatisfaction with current Android documentation on native development, and even on Java development. It is difficult to understand how to get even the samples to run, and much more difficult to combine documentation of NDK and SDK. Objective-C support is provided by a third-party package, and using this compiler is even less documented.

Comments, criticism and suggestions are welcome.

Android design

Android calls apps “activities”. These are pieces of software written in Java. There are various “API levels”. These API levels determine compatibility with an OS. For example, Android 2.1 supports activities of API level 7 and below.

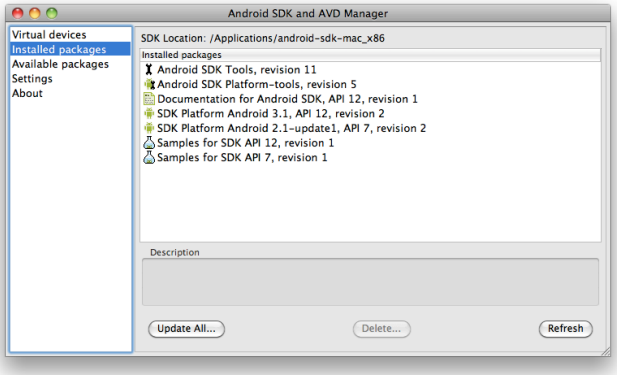
There are two pieces of puzzle: the Android SDK, which builds Java applications into .apk packages, and the Android NDK[1] which build C and C++ code into .so files, the shared objects. These shared objects cannot by themselves be packaged, nor can they be run on Android devices or the emulator. For that, Java code needs to be written.

First chapter of this document concentrates on setting up the environment under the Mac OS X platform. Certainly, vast majority of the text will apply to the Linux platform as well, and probably significantly less to the Windows platform. Some familiarity with C development under UNIX operating systems, as well as with the command line, is expected.

Text is written with Mac OS X 10.6 Snow Leopard in mind, which still ships with Java. This is about to change with Mac OS X 10.7 Lion, and separate installation of Java will almost certainly be necessary prior to any further steps. “ant” should be shipping with OS X 10.6; if not, it probably ships with the freely available Xcode 3 IDE.

[1] NDK - Native Development Kit

Understanding and setting up the SDK



Android SDK allows you to compile and test Android activities - Android apps. To install it, one downloads base environment for one's particular platform. Currently, Windows, Mac and Linux are supported as development platforms. Base environment can be downloaded from Android's web site: http://developer.android.com/sdk/index.html

SDK's file for Mac OS X is currently called "android-sdk_r11-mac_x86.zip". Download it and extract it where desired. A good place to do so is the /Applications folder, and rest of this text will presume this is where the SDK was installed. Resulting folder, /Applications/android-sdk-mac_x86, will also be referred to by the name sdk.

Installing a platform package

System you have just unpacked is just the base environment. To actually develop, you will need to install a "platform package". This package actually defines an operating system version. Packages bear the name of the OS and also specify the API level for the operating system.

To install a platform package, you will use a GUI tool launchable from command line. (Under Snow Leopard, you can also run the executable file directly from Finder.) Executable is called android and is located in the sdk/tools/ folder.

```
$ cd /Applications/android-sdk-mac_x86
$ cd tools
$ ./android
```

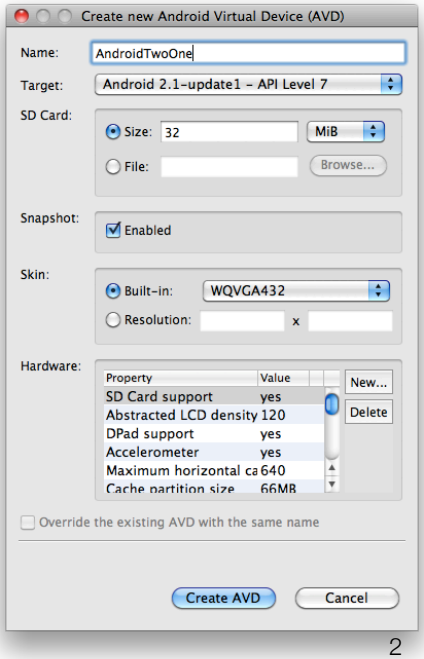
On the menu on the left, go to "Available packages" and under "Android Repository", choose packages to installed. You are required to install "Android SDK Platform-tools" and at least one "SDK Platform" package.

On using Android 2.1 as minimum platform

In documentation, Google states that on May 2nd 2011, operating system 2.1 accounted for 24.5% of devices, while Android 2.2 accounted for 65.9% of devices accessing the Android Market in two weeks preceding the date[1]. These numbers demonstrate that these two operating systems account for more than 90% of Android install base. Adding Android 2.3 (1.0%), 2.3.3 (3.0%) and 3.0 (0.3%) means that the supported install base accounts for almost 95% of the install base.

Hence, using 2.1 as a minimum operating system for one's apps and avoiding any features that will require 2.2 and 2.3 means one will be unable to support only a small part of the install base. One example of a useful feature that would be unavailable to the vast majority of the install base is "native activity" introduced in Android 2.3, adding the ability to write almost purely native applications without manually writing almost any Java code.

Recommended SDK Platform to install is for Android 2.1, API 7. Software built for 2.1 will run on vast majority of available devices. You can optionally install later platforms to allow use of features available devices, as long as you test that your software does not break on 2.1. Testing on Android 2.1 in emulator requires installation of SDK Platform for Android 2.1.



Installing additional packages

You are also strongly recommended to install "Documentation for Android SDK" and "Samples for SDK API 7", as well as samples for any other operating system you have installed. Samples package is required to follow the next section of the document, explaining how to build the samples.

Configuring the PATH environment variable

To more easily call various Android utilities, you may want to add this to your ~/.bash_profile:

```
export PATH="/Applications/android-sdk-mac_x86/tools:$PATH"
export PATH="/Applications/android-sdk-mac_x86/platform-tools:$PATH"
```

Alternatively, create a shell script with these two lines. Then ‘source’ the script from your shell using the dot command.

```
$ . ~/android_environment.sh
```

In either case, you will no longer need to type the entire path to utilities such as “android” or “adb”.

Creating an emulated device

Emulated devices are created through the “android” utility. It is easiest to simply use the same GUI that was used for downloading the SDK components. Presuming you have used the preceding advice and added tools to the path, just launch the “Android SDK and AVD Manager” from the shell

```
$ android
```

On the Virtual devices tab, create a new AVD[2] by clicking on the New... button. Choose the Android 2.1 target, and add various hardware components to the device. It is highly recommended you at least add the SD Card support, but you would probably do well by simply adding all the components offered.

You are highly recommended to turn on the “Snapshot” option. Enabling this option means that the emulator will save the device state on exit, and load it on startup. This is important since the full boot process takes well over a minute. This is even more important with later versions of the operating system, in which the boot process takes even longer and hence lengthens the development process even further.

Additionally, the offered default WQVGA800 resolution is very large and it may not fit on your screen. WQVGA400 is a nice setting, but you will definitely be adjusting the setting during development as you develop for various devices.

[2] AVD - Android Virtual Device. An emulated device.

Getting Java samples to run

Android documentation quite clearly suggest using Eclipse IDE for beginner developers. However, since we are attempting to use NDK and to write as little Java code as possible, we will definitely need to use command line. We will attempt to avoid using any sort of IDE to more easily automate the build process.

Java developers are familiar with the build tool "ant". This build tool serves a purpose similar to that of "make" for C developers. Input for "ant" is a file called build.xml.

Examining the provided examples in sdk/samples/android-7, one can notice that the samples do not ship with build.xml. This is because with Android SDK, one describes the entire software package using AndroidManifest.xml. This file can be directly used by the ADT Plugin for Eclipse to build the software. If one wishes to use "ant" to build the package, one needs to use the "android" command line utility - the same one that previously displayed GUI for downloading the packages for the SDK.

Generating the build.xml

To demonstrate generation of build.xml, let's use the "Lunar Lander" sample game as a demonstration. We will list the targets (primarily consisting of emulated devices), see the reference number for the target, and specify it for the "android" utility when generating the build.xml file.

```
$ cd /Applications/android-sdk-mac_x86/samples/android-7/LunarLander
$ android list targets
a list of numbered device targets is printed out. pick one and specify it in the line below.
$ android update project -p . -s --target 1
Updated default.properties
Updated local.properties
Added file ./build.xml
Added file ./proguard.cfg
Updated default.properties
Updated local.properties
Added file ./tests/build.xml
Added file ./tests/proguard.cfg
```

[exec] error: device not found

Getting the above error? Try restarting the emulator without snapshots in place. adb can apparently get confused when you use snapshots.

This will generate build.xml and various other support files needed to actually build the .apk which can be installed on the device.

Building the project

To build the project, one needs to simply run "ant" with the argument "debug". This will sign the built binary with a debug key, which cannot be used in production. To sign for public release, use the "release" argument.

```
$ ant debug
Buildfile: /Applications/android-sdk-mac_x86/samples/android-7/LunarLander/build.xml
    [setup] Android SDK Tools Revision 11
    [setup] Project Target: Android 2.1-update1
   … and so on….
debug:
     [echo] Running zip align on final apk…
     [echo] Debug Package: /Applications/android-sdk-mac_x86/samples/android-7/LunarLander/bin/LunarLander-
debug.apk
BUILD SUCCESSFUL
Total time: 5 seconds
```

This has produced the .apk.

Installing the built project

Project needs to be installed on a running emulator or a device. Run the emulator by launching “android” utility, choosing the device you created previously, and clicking on Start. If this is the first time you are launching the device (or if you have Snapshots disabled), boot will take some time. Wait for the OS to launch completely and become responsive.

Now, go back to the “LunarLander” folder in Terminal and run “ant install”.

```
$ cd /Applications/android-sdk-mac_x86/samples/android-7/LunarLander
$ ant install
… skipping a lot of output …
install:
    [echo] Installing /Applications/android-sdk-mac_x86/samples/android-7/LunarLander/bin/LunarLander-debug.apk onto default emulator or device…
    [exec] 1010 KB/s (122528 bytes in 0.118s)
    [exec]     pkg: /data/local/tmp/LunarLander-debug.apk
    [exec] Success

BUILD SUCCESSFUL
Total time: 9 seconds
```

Installation process uses a utility called “adb”[3]. You can manually install .apk files by using “adb install file”. You can connect to device’s shell by using “adb shell”. “adb” is installed in the sdk/platform-tools folder.

If you have multiple devices connected, you will be interested in adb’s option “-s”. To see the list of specified devices:

```
$ adb devices
List of devices attached
emulator-5554   device
```

You can see that currently, only one device is attached, and that is the emulator. To direct adb to install an .apk on that device, use:

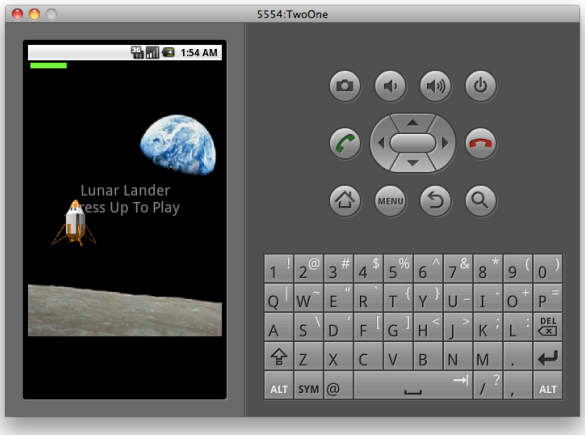
```
$ adb install -s emulator-5554 file.apk
```

Manually installing as just described is useful when multiple devices are connected, or when the .apk is already built. Otherwise, using “ant install” is faster and easier.

Running the built project

Built project will be available in the application list on the emulator. Click on the up arrow symbol  on the bottom of the device screen to bring up the list of applications. Then find the “Lunar Lander” icon and launch it.

[3] ADB: Android Debugging Bridge



Congratulations. A Java project has been built and installed on the Android emulator. You can now close the emulator by clicking on the close button. If you have activated Snapshots, as recommended, quitting will take some time to complete. This is normal, since device status needs to be written to disk.

Understanding, setting up and testing the Android Native Development Kit

SDK is all about building Java applications. Since we are focusing on avoiding writing Java code, we will skip further discussion on writing Java. Since we are trying to support 2.1 and later devices, writing Java code cannot be avoided; in fact, it may be quite necessary for numerous operations, including accessing resources shippping with the application. (Some of these operations can be avoided in Android 2.3 with NativeActivity support.)

Android NDK is officially provided toolchain for compiling C and C++ code to extend Java code. It is presented to developers as an option to write high-performance modules for their Java applications, instead of being presented as a portability layer that makes it possible to easily transfer code written for other mobile and desktop platforms. While not significantly easing porting of applications from other platforms, NDK simplifies porting of games.

NDK is a toolchain for compiling C and C++ code. It provides a compiler, a set of makefiles and a set of tools that is supposed to make compiling code easier, such as “ndk-build”. Currently the provided compiler is GCC. Output from the NDK are .so files - that is, shared objects. Shared objects are a concept similar to Windows’ .dll[4] files and Mac OS X’s .dylib. They are commonly found on Linux and some other systems. These files contain a set of C functions that can be dynamically loaded into an existing process after it has already started, or even during its startup. Loading these functions allows them to be used by the executable process.

NDK produces shared objects that implement JNI[5]. This means Java code can call C functions from these files, and C functions from these files can call Java code.

Once again: NDK does not allow producing purely native applications. Android is not designed to support this scenario, especially not the version targeted by this document (2.1).

---

[4] DLL - dynamically linked library

[5] JNI - Java Native Interface

Installing the NDK

To get started with the NDK, download it from Android's site. http://developer.android.com/sdk/ndk/index.html

Pick the version for your operating system. For Mac OS X, current filename is "android-ndk-r5c-darwin-x86.tar.bz2". Unpack this file anywhere on the disk; there is, currently, nothing in the NDK that depends on exact disk location. It may, however, be a good idea to keep the NDK in the same location as the SDK. For this reason, you may want to install the unpacked NDK inside the sdk folder; for example, put the android-ndk-r5c/ folder inside /Applications/android-sdk-mac_x86.

That means the full path will be: /Applications/android-sdk-mac_x86/android-ndk-r5c/

Above path will also be referenced in the text as ndk.

Updating the PATH variable for NDK tools

To make calling various NDK utilities easier, consider modifying your PATH variable. Previously, you were advised to modify either your ~/.bash_profile or create a "sourcable" shell script called, for example, ~/android_environment.sh.

Add the following line to either .bash_profile or android_environment.sh:

```
export PATH="/Applications/android-sdk-mac_x86/android-ndk-r5c:$PATH"
```

Adapt it for your install path for the NDK. By adding the above line, we can call "ndk-build" without prepending the ndk path.

Building an application using native code

For demonstration purposes, we will build the “hello-jni” project, located in ndk/samples/hello-jni.

```
$ cd /Applications/android-sdk-mac_x86/android-ndk-r5c/samples/hello-jni
$ ndk-build
Gdbserver      : [arm-linux-androideabi-4.4.3] libs/armeabi/gdbserver
Gdbsetup       : libs/armeabi/gdb.setup
Compile thumb  : hello-jni = hello-jni.c
SharedLibrary  : libhello-jni.so
Install        : libhello-jni.so = libs/armeabi/libhello-jni.so
```

ndk-build uses “make” to process various .mk files in the project. To see commands called by ndk-build, pass it the V=1 argument:

```
$ ndk-build V=1
```

Now that we have built the native code (libhello-jni.so file), we need to build the Java part of the project. We will proceed just as before:

```
$ android update project -p . -s --target 1
$ ant install # note that ‘ant debug’ will be automatically called. for install to succeed, make sure the emulator is running
… skipping parts of output ...
install:
    [echo] Installing /Applications/android-sdk-mac_x86/android-ndk-r5c/samples/hello-jni/bin/HelloJni-debug.apk onto default emulator or device…
    [exec] 1566 KB/s (80901 bytes in 0.050s)
    [exec]     pkg: /data/local/tmp/HelloJni-debug.apk
    [exec] Success

BUILD SUCCESSFUL
Total time: 8 seconds
```

Now, in the emulator, launch the 

application. You should see text “Hello from JNI!” - and this text was actually generated in the C code.

A bit about what just happened

Here’s the C code (an excerpt from ndk/samples/hello-jni/jni/hello-jni.c)

```
jstring
Java_com_example_hellojni_HelloJni_stringFromJNI( JNIEnv* env,
                                                  jobject thiz )
{
    return (*env)->NewStringUTF(env, "Hello from JNI !");
}
```

Although, like every other Hello World example, this example is not very useful, is clearly demonstrates that C code can easily be called from Java.

Note the name of the function. This is because this function was actually declared to exist in Java part of the code. More specifically, it was declared to exist in com.example.hellojni, in class HelloJni, as function named stringFromJNI. Take a look at ndk/samples/hello-jni/src/com/example/hellojni/HelloJni.java. Here’s an excerpt:

```
package com.example.hellojni;
```

```
/* ... code skipped ... */

public class HelloJni extends Activity
{
    /* ... code skipped ... */

    /* A native method that is implemented by the
     * 'hello-jni' native library, which is packaged
     * with this application.
     */
    public native String  stringFromJNI();

    /* ... code skipped ... */

    /* this is used to load the 'hello-jni' library on application
     * startup. The library has already been unpacked into
     * /data/data/com.example.HelloJni/lib/libhello-jni.so at
     * installation time by the package manager.
     */
    static {
        System.loadLibrary("hello-jni");
    }
}
```

Function is actually resolved inside the loaded .so when it is first called. So, one can declare functions that are never implemented in C, and unless they are called, there will be no problem.

It is important to note that Java can also be called from C. This can be rather slow, so it might be wise to avoid it at all costs in high performance applications such as games.

Setting up the Objective-C 2.0 GCC for NDK

To get Objective-C to work, you need a different compiler than the one NDK provides. There is a project on Google Code that produced exactly what you need. http://code.google.com/p/android-gcc-objc2-0/

Only binary download they provide is for Darwin; that is, for Mac OS X. Compile process is lengthy, so this document will only cover the use of a January 2010 release called “ndk-arm-eabi-i386-apple-darwin.tar.bz2”.

Installation

Unpacking this file will create a single folder “usr/”. Inside “usr/”, you will find the structure “/usr/local/android” inside which the standard GCC system root can be found. Create a new folder “ndk-objc/” and move the “usr/” folder into it. Place it anywhere on the disk; recommended location, however, is “<sdk>/ndk-objc” - for example, full path to “usr” is recommended to be: /Applications/android-sdk-mac_x86/ndk-objc/usr

Writing an Objective-C project

Now we will assemble a small project. We are treading the unknown territory, since the author of android-gcc-objc2-0 did not document how to produce a working .so from Objective-C sources.

We will not be using ndk-build, since we do not need the Android makefiles. Usual NDK build process was not designed to support non-NDK compilers nor was it designed to support Objective-C.

First we need to construct a test project. We’ll start with the Objective-C file.

Objective-C file: test.m

We cannot use Foundation, AppKit or UIKit since these frameworks are not ported to this platform. What we can use in place of NSObject is a piece of GCC runtime, a class called Object. This class used to be the base class before Objective-C 2.0; today, it is deprecated or even nonexistent on Mac OS X.

Create a directory called “android-obj-c”. In it, create a single subdirectory “objc-jni”, and inside create “test.m” with the following contents:

```
#import <objc/Object.h>
#import <stdio.h>
@interface Something : Object
{
	int x;
}
-init;
-(void)incrementByNumber:(int)num ifSmallerThan:(int)small;
-(char*)provideString;
// @property (assign) int x;
@end

@implementation Something
-init
{
	if(self=[super init])
```

```
	{
		printf("Yay!\n");
		x = 6;
		// [self incrementByNumber:1 ifSmallerThan:30];
	}
	return self;
}
-(void)incrementByNumber:(int)num ifSmallerThan:(int)small
{
	if(x < small)
		x += 6;
}
-(char*)provideString
{
	return "Hello from ObjC JNI!";
}
// @synthesize x;
@end

char* objc_main(){
	Something *s = [[Something alloc] init];
	[s incrementByNumber:5 ifSmallerThan:30];
	//printf("X is %d\n", s.x);
	return [s provideString];
}

#include <string.h>
#include <jni.h>

jstring
Java_com_example_helloobjcjni_HelloObjcJni_stringFromJNI( JNIEnv* env,
                                                  jobject thiz )
{
    char* aString = objc_main(); /* just test calling ObjC code */
    return (*env)->NewStringUTF(env, aString);
}
```

Note that for now, we have commented out the property “x”. This is because we want to first see if Objective-C 1.0 works.

Makefile for our Objective-C code

Now create “objc-jni/Makefile”. This is almost certainly not the best nor the proper way to produce the required .so, however it will produce a shared object.

```
SDK=/Applications/android-sdk-mac_x86
NDK=$(SDK)/android-ndk-r5c
NDKOBJC=$(SDK)/ndk-objc

# specify full path to arm-eabi-gcc from ObjC-2.0 NDK
OBJC=$(NDKOBJC)/usr/local/android/bin/arm-eabi-gcc

# specify the assembler to use. for some reason, ObjC GCC likes
# to use the system-wide assembler (which, being an x86 assembler,
# does not understand what ARM is all about).
AS=$(NDKOBJC)/usr/local/android/bin/arm-eabi-as

# specify sysroot for 2.1 NDK
SYSROOT=$(NDK)/platforms/android-5/arch-arm/

# specify full path to NDK's g++
# - we're using arm-eabi-4.4.0
# - we're not using arm-linux-androideabi-4.4.3
# arm-eabi-4.4.0 is the compiler for 2.1 target. this is also the compiler
# that more closely matches the objc compiler.
#LD=$(NDK)/toolchains/arm-eabi-4.4.0/prebuilt/darwin-x86/bin/arm-eabi-g++
LD=$(NDKOBJC)/usr/local/android/bin/arm-eabi-ld

# object files that are included in our output .so
# have the same names as .m files, but with extension
# .o.
OBJECTS=test.o

# what is our primary target?
all: libtest-objc.so
```

```
# how to clean?
clean:
	-rm libtest-objc.so
	-rm $(OBJECTS)
	-rm *.s

# precisely define how to compile .m files into
# .o files. don't depend on gnu make doing the
# right thing

# two phase process:
# - compile to assembler
# - use assembler to produce .o
# this is because objc ndk's gcc uses system-wide
# 'as', which does not speak ARM assembly.
test.s: test.m
	$(OBJC) --sysroot $(SYSROOT) -isysroot $(SYSROOT) -I$(SYSROOT)/usr/include $< -o $@ -S
test.o: test.s
	$(AS) -I$(SYSROOT)/usr/include $< -o $@

# define how our output, the .so, should be
# produced.

libtest-objc.so: $(OBJECTS)
	$(LD) $(OBJECTS) -L$(SYSROOT)/usr/lib -lobjc -lc -E -shared -o libtest-objc.so
```

Supporting files for Android Java code

We will now adapt the hello-jni example to make use of our code. Let's see the remaining files.

AndroidManifest.xml: project description

AndroidManifest.xml, placed in project root, describes our project:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.example.helloobjcjni"
      android:versionCode="1"
      android:versionName="1.0">
    <uses-sdk android:minSdkVersion="3" />
    <application android:label="@string/app_name"
                 android:debuggable="true">
        <activity android:name=".HelloObjcJni"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

We have just changed "HelloJni" and "hellojni" to "HelloObjcJni" and "helloobjcjni", respectively.

strings.xml: string resources

Manifest refers to a string resource, "@string/app_name". Hence we need to have a file "res/values/strings.xml":

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">HelloObjcJni</string>
</resources>
```

HelloObjcJni.java: Java code making use of Objective-C code

We need a piece of Java code that will actually make use of our code.

$ mkdir -p src/com/example/helloobjcjni/

Now, enter the following code in src/com/example/helloobjcjni/HelloObjcJni.java (this is nearly a copypaste of HelloJni.java):

```
/*
 * Copyright (C) 2009 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package com.example.helloobjcjni;

import android.app.Activity;
import android.widget.TextView;
import android.os.Bundle;


public class HelloObjcJni extends Activity
{
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);

        /* Create a TextView and set its content.
         * the text is retrieved by calling a native
         * function.
         */
        TextView  tv = new TextView(this);
        tv.setText( stringFromJNI() );
        setContentView(tv);
    }

    /* A native method that is implemented by the
     * 'hello-jni' native library, which is packaged
     * with this application.
     */
    public native String  stringFromJNI();

    /* This is another native method declaration that is *not*
     * implemented by 'hello-jni'. This is simply to show that
     * you can declare as many native methods in your Java code
     * as you want, their implementation is searched in the
     * currently loaded native libraries only the first time
     * you call them.
     *
     * Trying to call this function will result in a
     * java.lang.UnsatisfiedLinkError exception !
     */
    public native String  unimplementedStringFromJNI();

    /* this is used to load the 'hello-jni' library on application
     * startup. The library has already been unpacked into
     * /data/data/com.example.HelloJni/lib/libhello-jni.so at
     * installation time by the package manager.
     */
    static {
        System.loadLibrary("test-objc");
    }
}
```

Helper Makefile for building both ObjC and Java code

To make building the code and cleaning the code easier, here's a Makefile that can be put in the project's root (next to the AndroidManifest.xml file):

```
all: objcjni java-code

install: all
      ant install

objcjni:
```

```
	cd objc-jni ; make ; cd ..
	mkdir -p obj/local/armeabi
	cp objc-jni/libtest-objc.so obj/local/armeabi/
	mkdir -p libs/armeabi
	cp objc-jni/libtest-objc.so libs/armeabi/

java-code:
	android update project -p . --target 1
	ant

clean: clean-java-code clean-objcjni

clean-java-code:
	-rm -rf obj/
	-rm -rf bin/
	-rm -rf gen/
	-rm build.xml
	-rm default.properties
	-rm local.properties
	-rm proguard.cfg
	rm -rf libs
clean-objcjni:
	cd objc-jni ; make clean ; cd ..
```

This defines targets “all”, “install” and “clean”.

Testing the code

$ make install

If you are seeing a crash, you can inspect the device log with adb:

$ adb logcat

Optionally tell the program to exit immediately and pipe it to UNIX tail command to see last 50 lines:

$ adb logcat -d | tail -n 50

If everything has gone well - congratulations! You have a piece of Objective-C code running on the Android.

Final set of files for HelloObjcJni project

Here is the final list of files, presenting the directory structure you need to have:

$ find . -type f
./AndroidManifest.xml
./Makefile
./objc-jni/Makefile
./objc-jni/test.m
./res/values/strings.xml
./src/com/example/helloobjcjni/HelloObjcJni.java

You can download the finished project from http://ivan.vucica.net/public/android-objc/android-objc-test.tar.gz

Appendix: What next?

Missing frameworks

Objective-C support coming with GCC ships only with Objective-C runtime. It does not ship with any frameworks such as Foundation, AppKit or UIKit. Without Foundation, you don't get NSString or even NSObject.

There are several implementations of Foundation one can use: best known ones are GNUstep and Cocotron. It would be an interesting accomplishment to compile one of these on Android.

User interface

There could be several approaches here.

One could write a reimplementation of UIKit or AppKit by painting into either using Java or into an OpenGL ES context.

One could attempt to port GNUstep's or Cocotron's implementation of AppKit that would paint using Java or into an OpenGL ES context.

One could write a partial reimplementation of UIKit or AppKit that would actually create Android user interface elements.

One could write a GUI specific to Android, incompatible with existing frameworks such as UIKit or Android.

One could wrap Android user interface classes using Objective-C, or even try using a Java-to-Objective-C bridge.

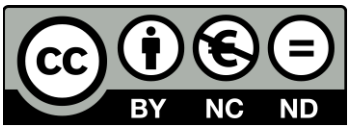
Authors of "android-gcc-objc2-0"

Fork of GCC used here was adapted from Apple's fork of GCC. Excellent work was done by the folks from Versv. Project description on Google Code mentioned that ports of some of Apple's libraries are being developed. So if you decide you need libraries with commercial support, you may want to poke them:

http://www.versv.com/

Appendix: Copyright

Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported



Copyright

Contact

Comments can be directed to the author:

- email: ivan@vucica.net, ivucica@hindarium.com
- web site: http://ivan.vucica.net/, http://blog.vucica.net/
- twitter: @ivucica

Hindarium is a small indie development team from Croatia.

- email: contact@hindarium.com
- web site: http://www.hindarium.com/

To support the work, purchase our games such as Zombie Ball for iPhone. Feel free to drop us a note why you made the purchase!