EMBEDDED SYSTEMS PROGRAMMING 2015-16

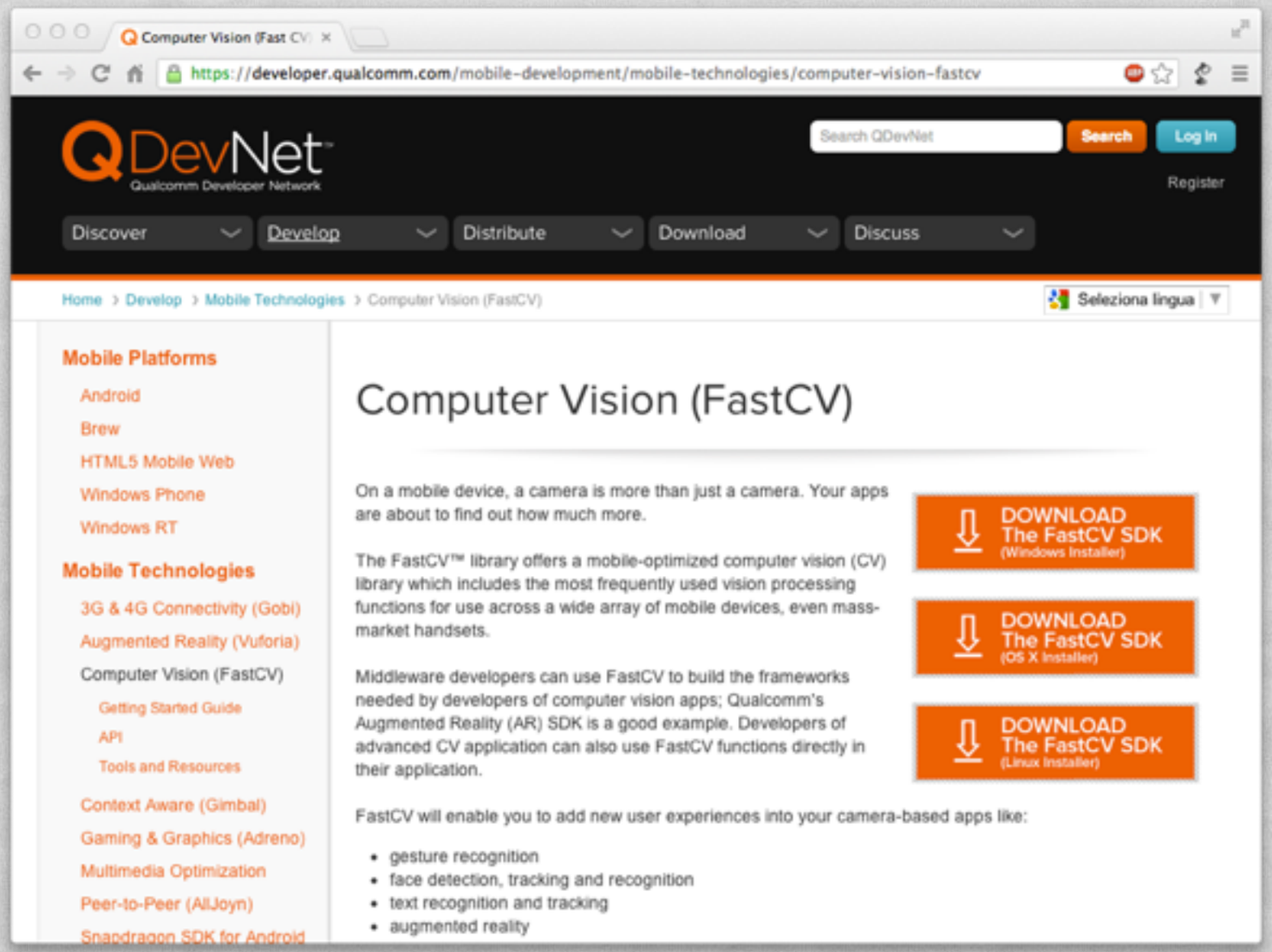
Android NDK

WHAT IS THE NDK?

- The Android NDK is a set of cross-compilers, scripts and libraries that allows to embed **native code** into Android applications
- Native code is compiled **from C/C++ sources**
- Requires Android 1.5 (API level 3) or later

NATIVE CODE: WHY? (1/2)

- Reason #1: you already have a native-code library, and wish to make it accessible to Java code without rewriting it



NATIVE CODE: WHY? (2/2)

- Reason#2: native code might be faster than bytecode

Bytecode recompiled by a fast JIT compiler;
overheads in passing parameters

- From Google's documentation:

> *...you should understand that **the NDK will not benefit most apps**. As a developer, you need to balance its benefits against its drawbacks. Notably, using native code on Android generally does not result in a noticeable performance improvement, but it always increases your app complexity. In general, you should only use the NDK if it is essential to your app—never because you simply prefer to program in C/C++*

NDK: WHAT IS INCLUDED

- **Cross-compilers** for ARM, x86 and MIPS architectures, both 32- and 64-bits
- Native **libraries** (with corresponding header files) that are “stable”, i.e., guaranteed to be supported in the future.
  Among the libraries: libc, libm, libz, OpenGL ES libs, ...
- A custom **build system** to ease the specification of how your C/C++ code should be compiled & linked
- Documentation and examples (of course)

APPROACHES TO NATIVE CODE

With the Android NDK there are currently two approaches to native code development

- **JNI**: the application is mostly written in Java. The (few) methods written in C/C++ are accessed via the Java Native Interface

- **Native activity**: entire activities are implemented in C/C++.
  Supported in Android 2.3 (API Level 9) or later

JNI (1/5)

- The **Java Native Interface** (JNI) is a standard Java programming interface that allows to
  - call native code from Java
  - invoke Java methods from code written in other languages (e.g., C/C++ or assembly)
  - map Java data types to/from native data types

- Android adds some small extra conventions to JNI

JNI (2/5)

Calling native code from Java

- Native methods are declared in Java by prepending the **`native`** keyword
- Libraries providing the bytecode are loaded with the **`System.loadLibrary`** method

```
class foo
{
    native double bar(int i, String s);

    static
    {
        System.loadLibrary("my_lib");
    }

    ...
}
```

JNI (3/5)

Assigning names to C/C++ methods

- The C/C++ name of a native method is concatenated by the following components:
  - the prefix `Java_`,
  - the mangled fully-qualified class name,
  - an underscore ("_") separator,
  - the mangled method name,
  - for overloaded native methods, two underscores ("__") followed by the mangled argument signature

JNI (4/5)

Parameters of C/C++ methods

- C/C++ **parameters** to a native method are different from the parameters declared in Java
  - The first C/C++ parameter is a pointer to the JNI interface
  - The second parameter is a reference to the object for nonstatic methods, and a reference to the Java class for static methods
  - The remaining parameters correspond to regular Java parameters
- The **return value** in C/C++ is the same as in Java, modulo the mapping of C/C++ data types to Java data types

JNI (5/5)

Mapping of data types

Java type	C/C++ Type	Description
`boolean`	`jboolean`	8 bit, unsigned
`char`	`jchar`	16 bit, unsigned
`int`	`jint`	32 bit, signed
`String`	`jstring`	Different encodings
`...`	`...`	...

- JNI provides a rich set of functions, accessible via the JNI interface, to manipulate strings and arrays

JNI: EXAMPLE

```
package pkg;

class foo
{
    native double bar(int i, String s);

    static
    {
        System.loadLibrary("my_lib");
    }

    ...
}
```

Java: declaration

```
jdouble Java_pkg_foo_bar(JNIEnv *env, // ptr to JNI interface
                         jobject obj, // "this" pointer
                         jint i,      // first "real" parameter
                         jstring s)   // second "real" parameter
{
    ... /* Method implementation */
}
```

C: implementation

NDK: ANDROID.MK

- Purpose: making native sources known to the NDK build system
- Syntax derived from GNU Make
- Easier to use than GNU Make: for instance, it is not necessary to list header files since such dependencies are resolved automatically
- Sources can be grouped into modules (i.e., libraries)

ANDROID.MK: EXAMPLE

- `Android.mk` from the `hello-jni` sample project

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := hello-jni
LOCAL_SRC_FILES := hello-jni.c

include $(BUILD_SHARED_LIBRARY)
```

- **`LOCAL_PATH`**: where source files are located
- **`LOCAL_MODULE`**: name of the module
- **`LOCAL_SRC_FILES`**: source files needed to build the module

NDK: APPLICATION.MK

- Purpose #1: list the modules which are needed by an application
- Purpose #2: describe how the application should be compiled, e.g. by specifying the target hardware architecture, options for the compiler and linker, etc.

- Optional

APPLICATION.MK: EXAMPLE

- `Application.mk` **from the** `bitmap-plasma` **sample project**

```
# The ARMv7 is significantly faster
# due to the use of the hardware FPU
APP_ABI := armeabi armeabi-v7a

APP_PLATFORM := android-8
```

- **`APP_ABI`**: specifies one or more architectures to compile for. The default is `armeabi` (ARMv5TE)
- **`APP_PLATFORM`**: target API level

NDK: NDK-BUILD

- The **`ndk-build`** shell script parses `.mk` files and manages required modules automatically

- **`<ndk>/ndk-build`**
  Build required native-code modules. The generated modules are automatically copied to the proper location in the application's project directory

- **`<ndk>/ndk-build NDK_DEBUG=1`**
  Build modules and include debug symbols

- **`<ndk>/ndk-build clean`**
  Clean all generated modules

HOW TO USE THE TOOLS

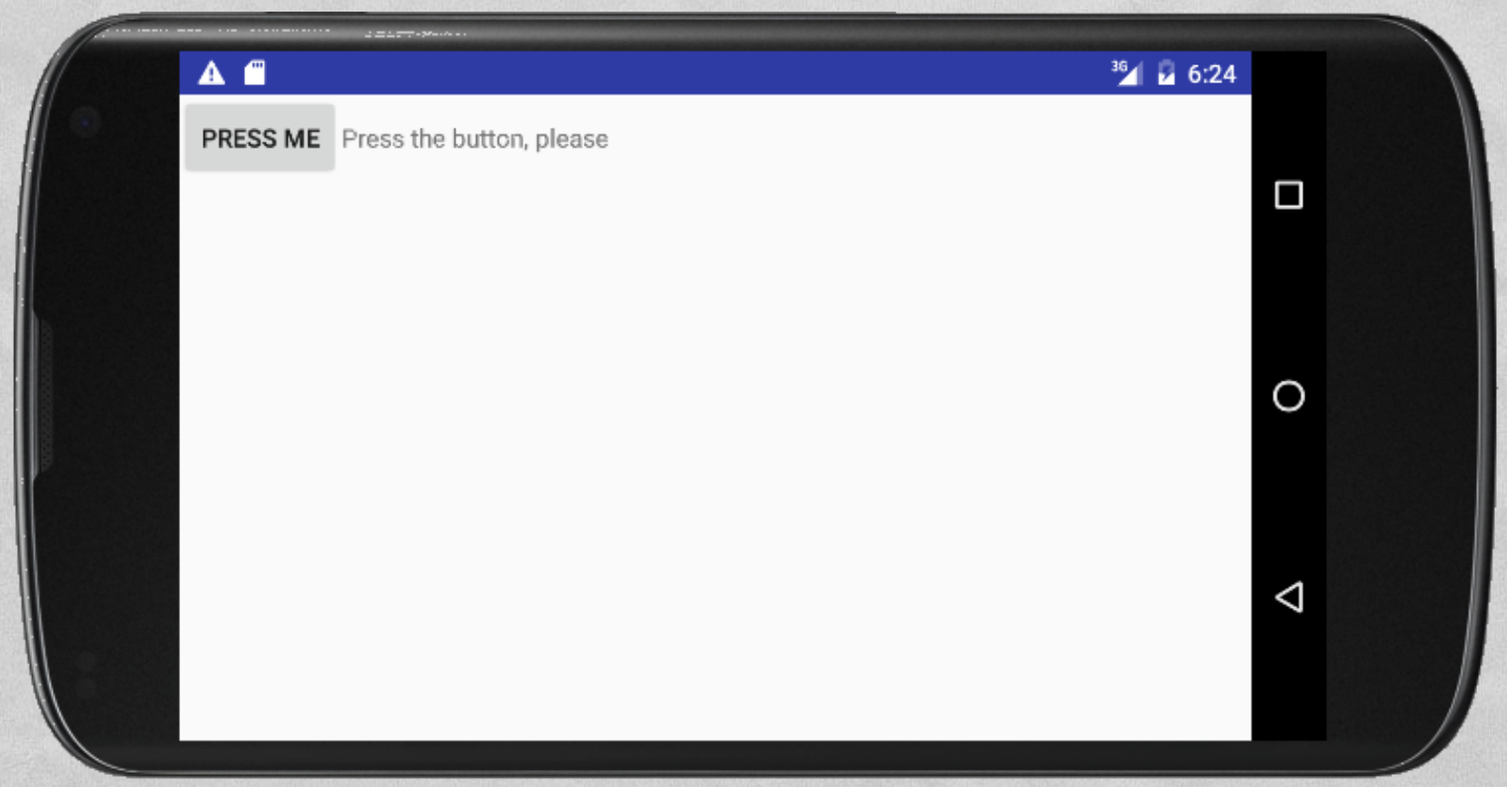
1. Place native sources under `<mod>/jni/...`
2. Create `<mod>/jni/Android.mk`
3. Optional: create `<mod>/jni/Application.mk`
4. Build native code by running the `ndk-build` script

```
pcte: ~ cd <prj>
pcte: ~ <ndk>/ndk-build
```

You can also use Android Studio instead

NDK: A FULL EXAMPLE

- Modify the “Hello World! (With Button) app
- When the button is pressed, the text still changes, but the new text is provided by a C++ function



APPLICATION FILES

- **`HelloWithButton.java`**
  Main activity, contains the Java code of the application

- **`jni/HelloWB_JNI.cpp`**
  Contains the C++ code of the application.
  The “native function” returns a string that embodies a random number

- **`jni/Android.mk`**

- **`AndroidManifest.xml`**, **`build.gradle`**

HELLOWITHBUTTON.JAVA (1/2)

```
package it.unipd.dei.esp1516.hellowithbuttonjni;

import android.os.Bundle;
import android.app.Activity;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import android.widget.LinearLayout;

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

        // Create the TextView
        final TextView tv = new TextView(this);
        tv.setText("Press the button, please");
        // Create the Button
        Button bu = new Button(this);
        bu.setText("Press me");
        // Set the action to be performed when the button is pressed
        bu.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                // Perform action on click
                tv.setText(stringFromJNI());
            }
        });
        // Create the layout
        LinearLayout mylayout = new LinearLayout(this);
        // Add the UI elements to the layout
        mylayout.addView(bu);
        mylayout.addView(tv);
        // Display the layout
        setContentView(mylayout);
    }
…
```

HELLOWITHBUTTON.JAVA (2/2)

```
…
    // Declaration of the native stringFromJNI() method.
    // The method is implemented by the 'hello-jni' native library,
    // which is packaged with this application
    public native String stringFromJNI();

    // Declaration of another native method that is not implemented
    // anywhere; trying to call it will result in a
    // java.lang.UnsatisfiedLinkError exception.
    // 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
    public native String unimplementedStringFromJNI();

    // Loads 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("HelloWB_JNI");
    }

}
```

HELLOWB_JNI.CPP

```cpp
#include <jni.h>
#include <stdlib.h>     // required for rand()
#include <stdio.h>      // required for snprintf()

// For JNI to locate your native functions automatically, they have to match
// the expected function signatures. C++ function names get mangled by the
// compiler (to support overloading and other things) unless extern "C" is specified
extern "C" {

/* This is a trivial native method that returns a new VM string
 * containing a pseudorandom double.
 */
jstring
Java_it_unipd_dei_esp1516_hellowithbuttonjni_HelloWithButton_stringFromJNI(
        JNIEnv* env,
        jobject thiz )
{
    char buf[64];   // local buffer
    double r;

    // Produce a pseudorandom double and place it into a C++ string
    r = (double)rand() / (double)RAND_MAX;
    snprintf(buf, 64, "Good: %f", r);

    // Convert the C++ string into something that can be shared with Java
    // This is C++: notice we use "env->..." instead of "(*env)->..."
    return env->NewStringUTF(buf);
}

} // end extern
```

ANDROID.MK

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := HelloWB_JNI
LOCAL_SRC_FILES := HelloWB_JNI.cpp

include $(BUILD_SHARED_LIBRARY)
```

- The invocation of `ndk-build` produces a library called (on *nix systems) **`HelloWB_JNI.so`**

ANDROIDMANIFEST.XML

- Automatically generated from properties that the programmer specifies via Android Studio

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="it.unipd.dei.esp1516.hellowithbuttonjni">

    <application
        android:allowBackup="false"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".HelloWithButton">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

PROJECT'S BUILD.GRADLE

```
// Top-level build file where you can add config options common to all sub-projects/modules.

buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        // New experimental plugin for NDK integration. See
        // http://tools.android.com/tech-docs/new-build-system/gradle-experimental/0-4-0
        // http://tools.android.com/tech-docs/new-build-system/gradle-experimental
        classpath 'com.android.tools.build:gradle-experimental:0.4.0'

        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

APP'S BUILD.GRADLE (1/2)

```
apply plugin: 'com.android.model.application'

model {
    android {
        compileSdkVersion = 23
        buildToolsVersion = "23.0.2"

        defaultConfig.with {
            applicationId = "it.unipd.dei.esp1516.hellowithbuttonjni"
            minSdkVersion.apiLevel = 15
            targetSdkVersion.apiLevel = 23
            versionCode = 1
            versionName = "1.0"

            buildConfigFields.with {
                create() {
                    type = "int"
                    name = "VALUE"
                    value = "1"
                }
            }
        }
    }
...
```

APP'S BUILD.GRADLE (2/2)

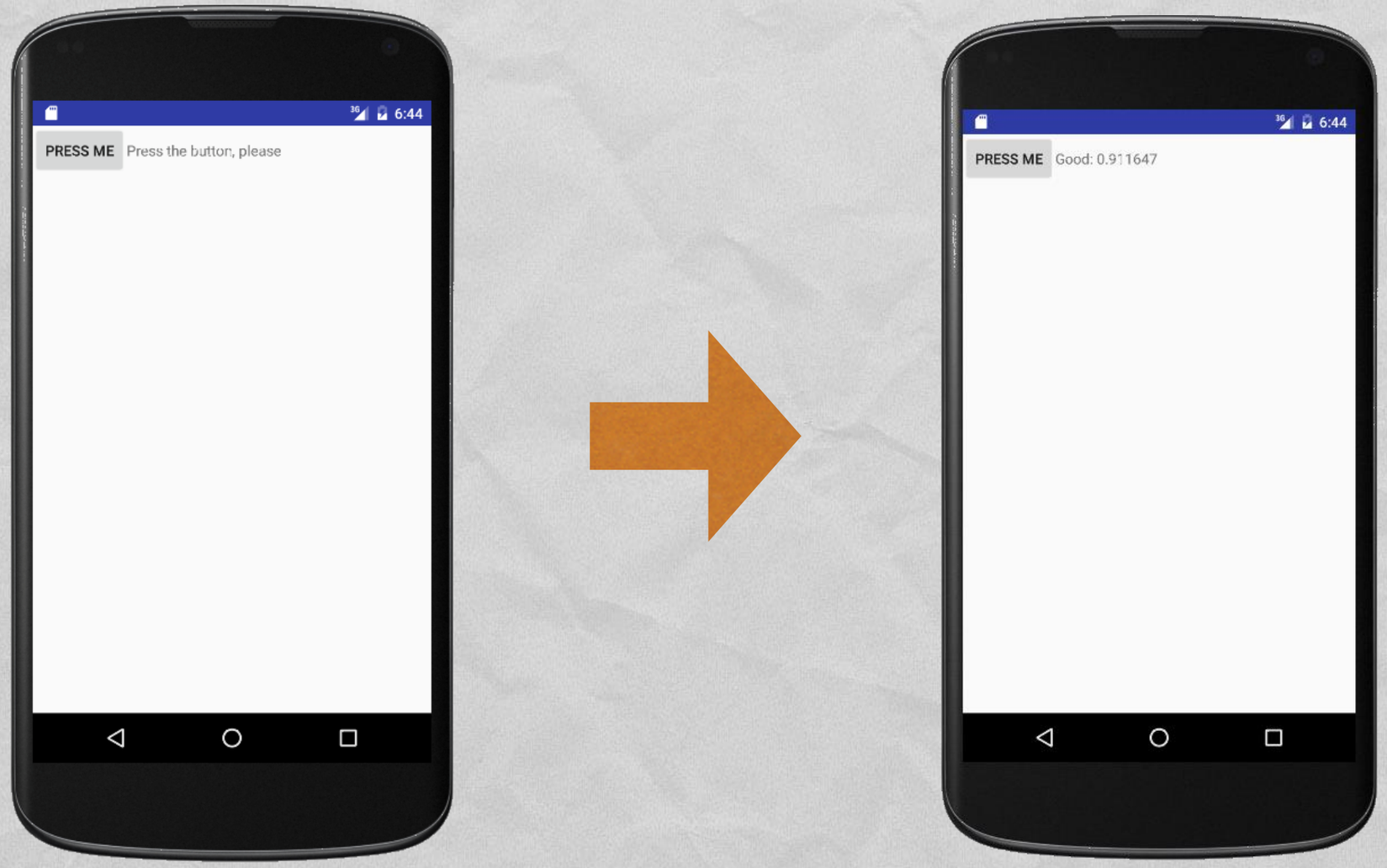
```
…
    android.buildTypes {
        release {
            minifyEnabled = false
            proguardFiles.add(file("proguard-rules.pro"))
        }
    }

    android.ndk {
        moduleName = "HelloWB_JNI"
    }
}


dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    testCompile 'junit:junit:4.12'
    compile 'com.android.support:appcompat-v7:23.1.1'
}
```

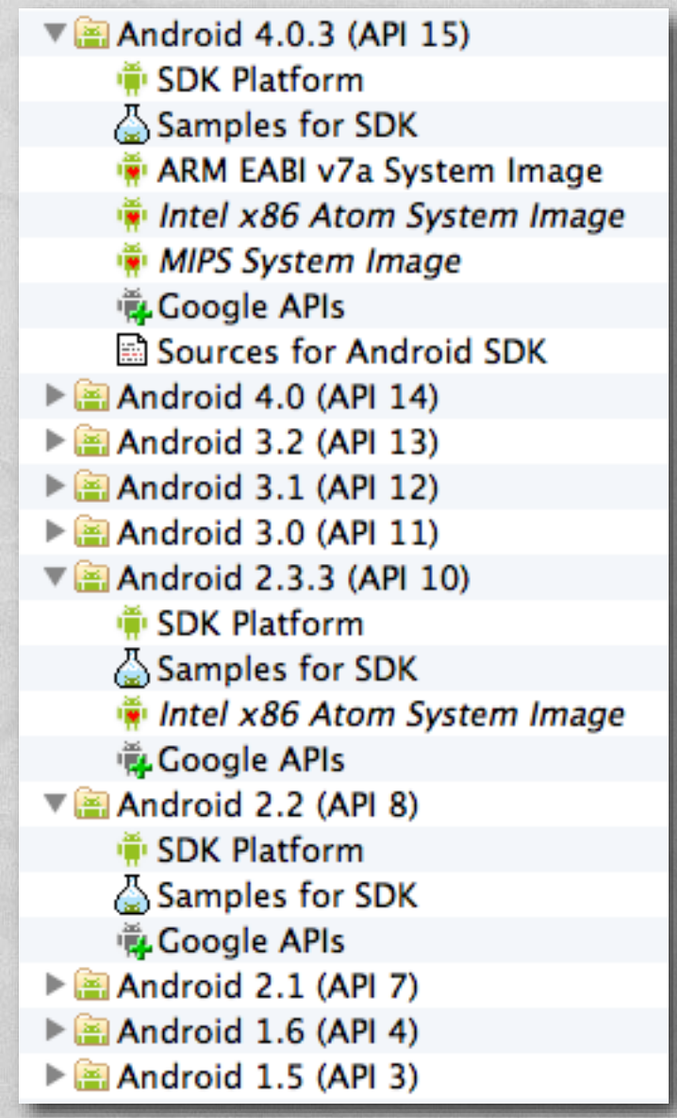
FINAL RESULT

- When the button is pressed, a random number appears



EMULATOR

- The SDK includes emulators for non-ARM architectures
- Some architectures are supported only in some API levels



NATIVE ACTIVITIES

- In Android 2.3 (API level 9) and later it is possible to **write entire activities in C/C++**
- Lifecycle callbacks (`onCreate()`, `onPause()`, ...) are implemented in C/C++ as well
- Most of the features included in Android libraries still need to be accessed through JNI
- For more info: read `docs/NATIVE-ACTIVITY.html` included in the NDK documentation

CAVEATS

- The JNI does not check for programming errors such as passing `NULL` pointers or illegal argument types
- Memory resources allocated by native code are not managed by a garbage collector and should be explicitly released
- The NDK only provides system headers for a very limited subset set of native Android APIs and libraries

REFERENCES

- NDK page on developer.android.com
- JNI specification
- Android Tools Project site
- developerWorks tutorial: "Reuse existing C code with the Android NDK"

LAST MODIFIED: MARCH 14, 2016