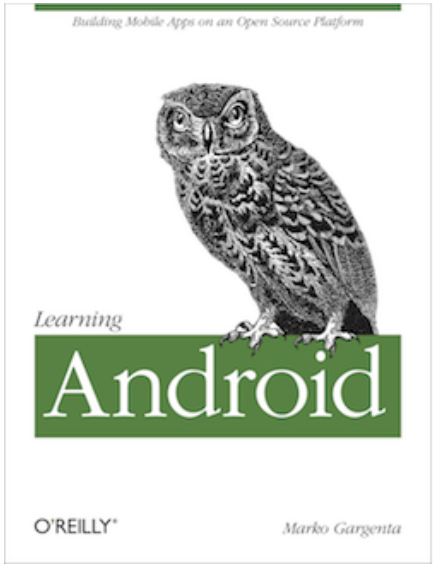
# Getting Native with NDK

# About Marko Gargenta





*Marko Gargenta*

## Entrepreneur, Author, Speaker

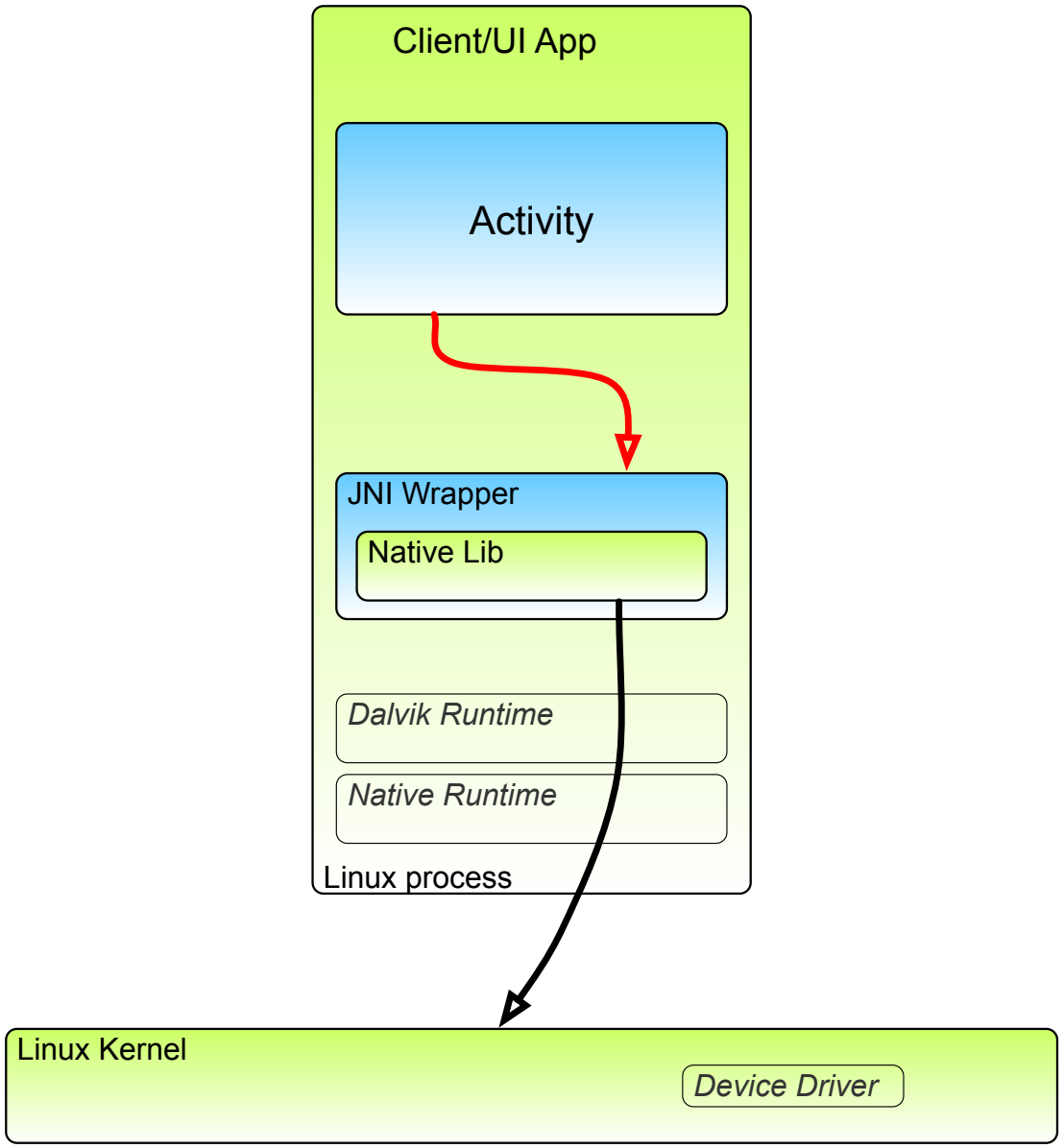
- Developer of Android Bootcamp for Marakana.
- Instructor for 1,000s of developers on Android at Qualcomm, Cisco, Motorola, Intel, DoD and other great orgs.
- Author of Learning Android published by O'Reilly.
- Speaker at OSCON (4x), ACM, IEEE(2x), SDC(2x), AnDevCon(3x), DroidCon.
- Co-Founder of SFAndroid.org
- Co-Chair of Android Open conference: Android Open

# Objectives of NDK Module

Android is put together of about equal part Java and C. So, no wonder that we need an easy way to bridge between these two totally different worlds. Java offers Java Native Interface (JNI) as a framework connecting the world of Java to the native code. Android goes a step further by packaging other useful tools and libraries into a Native Development Kit, or NDK. NDK makes developing C/C++ code that works with an Android app much simpler than if one was to do it by hand. Topics covered include:

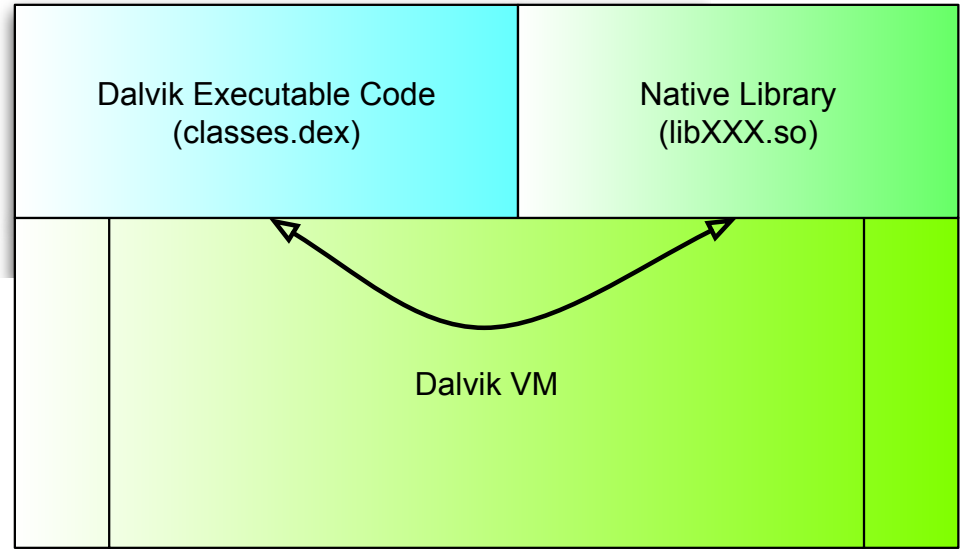
- What is in NDK?
- Why NDK?
- Java Native Interface (JNI)
- Using NDK
- NDK and JNI by Example
- NDK’s Stable APIs
- Lab: NDK

# NDK in Action




*Using NDK to connect Activity to native code*

# Dalvik Runs Native



# What is in NDK?

## NDK is a toolchain

Cross-compiler, linker, what you need to build for ARM, x86, MIPS, etc.

## NDK provides a way to bundle lib.so into your APK

The native library needs to be loadable in a secure way.

## NDK "standardizes" various native platforms

It provides headers for `libc`, `libm`, `libz`, `liblog`, `libjnigrahics`, OpenGL/OpenSL ES, JNI headers, minimal C++ support headers, and Android native app APIs.

## NDK comes with docs and samples

Helps you get up to speed with native development.

# Why NDK?

## For performance

Sometimes, native code still runs faster.

## For legacy support

You may have that C/C++ code you'd like to use in your app.

## For access to low-level libraries

In a rare case when there is no Java API to do something.

## For cross-platform development

C is the new portable language.

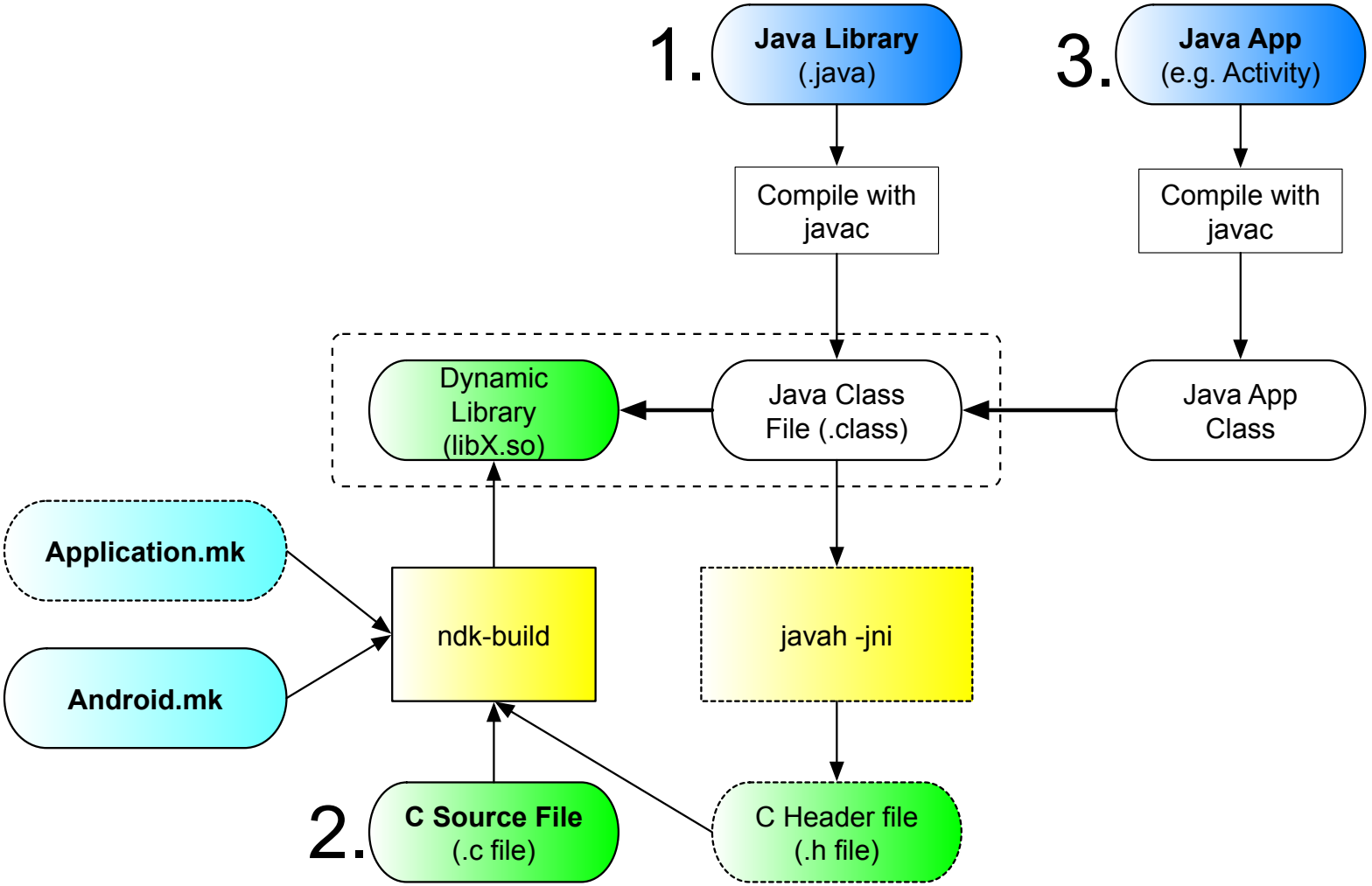
## But

Adding JNI to your app **will** make it more complex.

## Plus

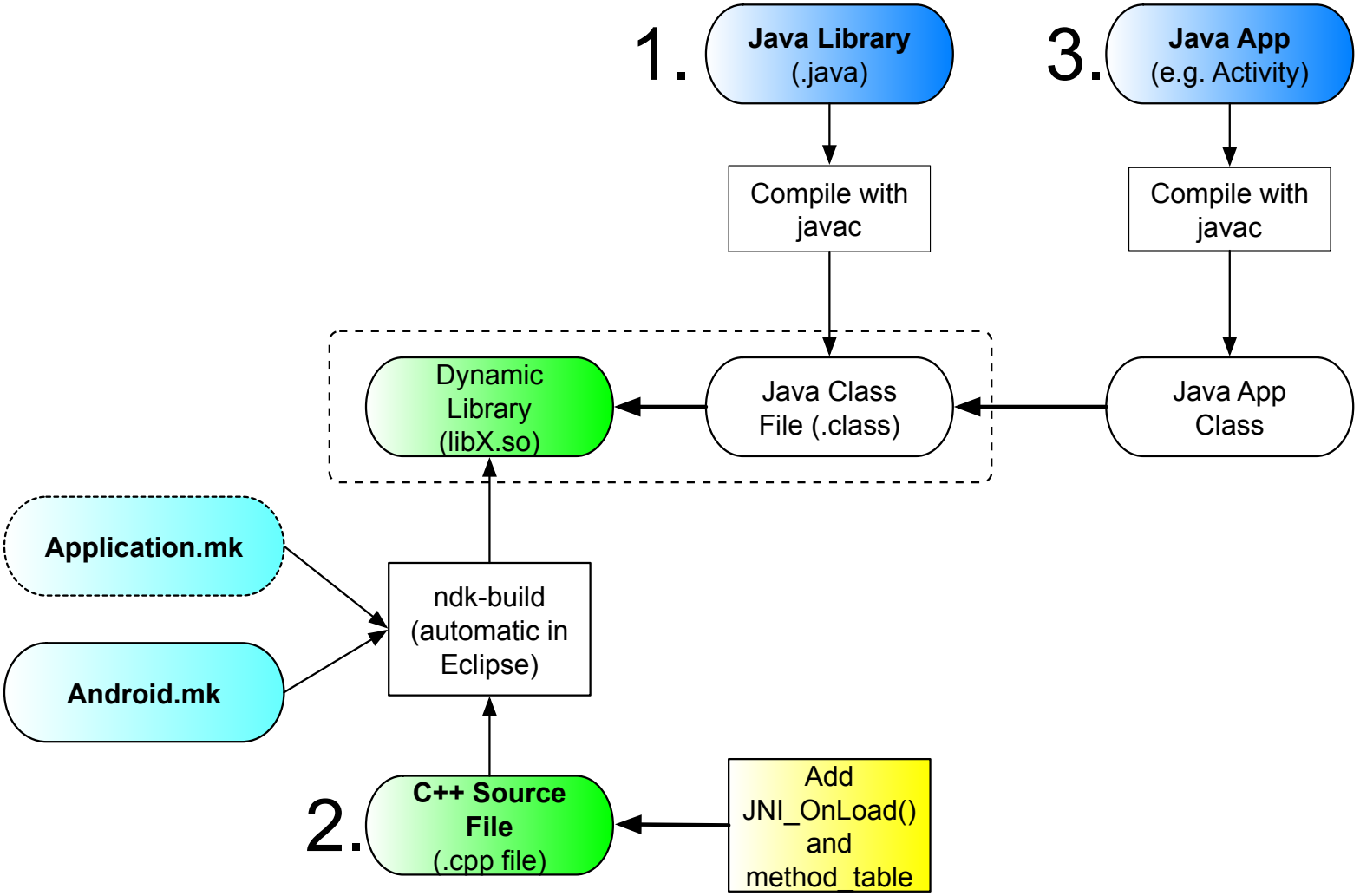
Java often offers much richer APIs, memory protection, OOP, productivity.

# Using NDK with C




*NDK Process Using C*

# Using NDK with C++




*NDK Process Using C++*

# NDK and JNI by Example

- NDK is best explained via an example (based on Fibonacci)
- The code is available
  - As a ZIP archive: https://github.com/marakana/FibonacciNative/zipball/master
  - By Git: `git clone https://github.com/marakana/FibonacciNative.git`

- Start by creating a new Android Project
  - Project Name: FibonacciNative
  - Build Target: Android 2.2 (API 8) or later
  - Application Name: Fibonacci Native
  - Package: `com.marakana.android.fibonaccinative`
  - Create Activity: `FibonacciActivity`

# Fibonacci - Java Native Function Prototypes

We start off by defining C function prototypes as `native` Java methods (wrapped in some class):

*FibonacciNative/src/com/marakana/android/fibonaccinative/FibLib.java*

```
package com.marakana.android.fibonaccinative;

import android.util.Log;

public class FibLib {
    private static final String TAG = "FibLib";

    private static long fib(long n) {
        return n <= 0 ? 0 : n == 1 ? 1 : fib(n - 1) + fib(n - 2);
    }

    // Recursive Java implementation of the Fibonacci algorithm
    // (included for comparison only)
    public static long fibJR(long n) {
        Log.d(TAG, "fibJR(" + n + ")");
        return fib(n);
    }

    // Function prototype for future native recursive implementation
    // of the Fibonacci algorithm
    public native static long fibNR(long n);


    // Iterative Java implementation of the Fibonacci algorithm
```

```
    // (included for comparison only)
    public static long fibJI(long n) {
        Log.d(TAG, "fibJI(" + n + ")");
        long previous = -1;
        long result = 1;
        for (long i = 0; i <= n; i++) {
            long sum = result + previous;
            previous = result;
            result = sum;
        }
        return result;
    }

    // Function prototype for future iterative recursive implementation
    // of the Fibonacci algorithm
    public native static long fibNI(long n);

    static {
        // as defined by LOCAL_MODULE in Android.mk
        System.loadLibrary("com_marakana_android_fibonaccinative_FibLib");
    }
}
```

# Fibonacci - Function Prototypes in a C Header File

We then extract our C header file with our function prototypes:

1. On the command line, change to your project's root directory

```
$ cd /path/to/workspace/FibonacciNative
```

2. Create `jni` sub-directory

```
$ mkdir jni
```

3. Extract the C header file from `com.marakana.android.fibonaccinative.FibLib` class:

```
$ javah -jni -classpath bin/classes -d jni com.marakana.android.fibonaccinative.FibLib
```

Prior to ADT r14, compiled class files were kept directly in the `bin/` directory, so in our `javah` command we would've used `-classpath bin` instead.

4. Check out the resulting file:

*FibonacciNative/jni/com_marakana_android_fibonaccinative_FibLib.h*

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_marakana_android_fibonaccinative_FibLib */

#ifndef _Included_com_marakana_android_fibonaccinative_FibLib
#define _Included_com_marakana_android_fibonaccinative_FibLib
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     com_marakana_android_fibonaccinative_FibLib
 * Method:    fibNR
 * Signature: (J)J
 */
JNIEXPORT jlong JNICALL Java_com_marakana_android_fibonaccinative_FibLib_fibNR
  (JNIEnv *, jclass, jlong);

/*
 * Class:     com_marakana_android_fibonaccinative_FibLib
 * Method:    fibNI
 * Signature: (J)J
 */
JNIEXPORT jlong JNICALL Java_com_marakana_android_fibonaccinative_FibLib_fibNI
  (JNIEnv *, jclass, jlong);

#ifdef __cplusplus
}
#endif
#endif
```

The function prototype names are name-spaced to the classname they are found in.


note

# Fibonacci - Provide C Implementation

We provide the C implementation of `com_marakana_android_fibonacci_FibLib.h` header file:

*FibonacciNative/jni/com_marakana_android_fibonaccinative_FibLib.c*

```c
/* Include the header file that was created via "javah -jni" command */
#include "com_marakana_android_fibonaccinative_FibLib.h"
#include <android/log.h>

/* Recursive implementation of the fibonacci algorithm (in a helper function) */
static jlong fib(jlong n) {
    return n <= 0 ? 0 : n == 1 ? 1 : fib(n - 1) + fib(n - 2);
}

/* Actual implementation of JNI-defined `fibNR` (recursive) function */
JNIEXPORT jlong JNICALL Java_com_marakana_android_fibonaccinative_FibLib_fibNR
  (JNIEnv *env, jclass clazz, jlong n) {
        __android_log_print(ANDROID_LOG_DEBUG, "FibLib.c", "fibNR(%lld)", n);
        return fib(n);
}

/* Actual implementation of JNI-defined `fibNI` (iterative) function */
JNIEXPORT jlong JNICALL Java_com_marakana_android_fibonaccinative_FibLib_fibNI
  (JNIEnv *env, jclass clazz, jlong n) {
        jlong previous = -1;
        jlong result = 1;
        jlong i;
        __android_log_print(ANDROID_LOG_DEBUG, "FibLib.c", "fibNI(%lld)", n);
        for (i = 0; i <= n; i++) {
                jlong sum = result + previous;
                previous = result;
                result = sum;
        }
        return result;
}
```

# Fibonacci - An Alternative Implementation (C++)

We could also use an alternative mechanism of linking native-code to managed code by pre-registering our functions. This leads to earlier detection of method-function mismatch issues, a slight performance improvement, and spares us the redundancy of the header file and the use of the `javah` command.

- See Registering Native Methods and JNI_OnLoad

*FibonacciNative/jni/com_marakana_android_fibonaccinative_FibLib.cpp*

```
#include <jni.h>
#include <android/log.h>

namespace com_marakana_android_fibonaccinative {
        static jlong fib(jlong n) {
                return n <= 0 ? 0 : n == 1 ? 1 : fib(n - 1) + fib(n - 2);
        }

        static jlong fibNR(JNIEnv *env, jclass clazz, jlong n) {
                __android_log_print(ANDROID_LOG_DEBUG, "FibLib.c", "fibNR(%lld)", n);
                return fib(n);
        }

        static jlong fibNI(JNIEnv *env, jclass clazz, jlong n) {
                jlong previous = -1;
                jlong result = 1;
                jlong i;
                __android_log_print(ANDROID_LOG_DEBUG, "FibLib.c", "fibNI(%lld)", n);
                for (i = 0; i <= n; i++) {
```

```
                        jlong sum = result + previous;
                        previous = result;
                        result = sum;
                }
                return result;
        }

        static JNINativeMethod method_table[] = {
                        { "fibNR", "(J)J", (void *) fibNR },
                        { "fibNI", "(J)J", (void *) fibNI }
        };
}

using namespace com_marakana_android_fibonaccinative;

extern "C" jint JNI_OnLoad(JavaVM* vm, void* reserved) {
    JNIEnv* env;
    if (vm->GetEnv(reinterpret_cast<void**>(&env), JNI_VERSION_1_6) != JNI_OK) {
        return JNI_ERR;
    } else {
        jclass clazz = env->FindClass("com/marakana/android/fibonaccinative/FibLib");
        if (clazz) {
                jint ret = env->RegisterNatives(clazz, method_table, sizeof(method_table) / sizeof(method_table[0]));
                env->DeleteLocalRef(clazz);
                return ret == 0 ? JNI_VERSION_1_6 : JNI_ERR;
        } else {
                return JNI_ERR;
        }
    }
}
```

Most of the Android's JNI-based shared libraries are built using this, "alternative", approach where the functions are pre-

registered.

# Fibonacci - Makefile

We need a `Android.mk` makefile, which will be used by NDK to compile our JNI code into a shared library:

*FibonacciNative/jni/Android.mk*

```
# Defines the root to all other relative paths
# The macro function my-dir, provided by the build system,
# specifies the path of the current directory (i.e. the
# directory containing the Android.mk file itself)
LOCAL_PATH := $(call my-dir)

# Clear all LOCAL_XXX variables with the exception of
# LOCAL_PATH (this is needed because all variables are global)
include $(CLEAR_VARS)

# List all of our C files to be compiled (header file
# dependencies are automatically computed)
LOCAL_SRC_FILES := com_marakana_android_fibonaccinative_FibLib.c

# The name of our shared module (this name will be prepended
# by lib and postfixed by .so)
LOCAL_MODULE := com_marakana_android_fibonaccinative_FibLib

# We need to tell the linker about our use of the liblog.so
LOCAL_LDLIBS += -llog

# Collects all LOCAL_XXX variables since "include $(CLEAR_VARS)"
#  and determines what to build (in this case a shared library)
include $(BUILD_SHARED_LIBRARY)
```


It's easiest to copy the `Android.mk` file from another (sample) project and adjust `LOCAL_SRC_FILES` and `LOCAL_MODULE` as necessary


See `/path/to/ndk-installation-dir/docs/ANDROID-MK.html` for the complete reference of Android make files (build system)

# Fibonacci - Compile Our Shared Module

Finally, from the root of our project (i.e. `FibonacciNative/`), we run `ndk-build` to build our code into a shared library (`FibonacciNative/libs/armeabi/libcom_marakana_android_fibonacci_FibLib.so`):

```
$ ndk-build
Compile thumb  : com_marakana_android_fibonaccinative_FibLib <= com_marakana_android_fibonaccinative_FibLib.c
SharedLibrary  : libcom_marakana_android_fibonaccinative_FibLib.so
Install        : libcom_marakana_android_fibonaccinative_FibLib.so =>
libs/armeabi/libcom_marakana_android_fibonaccinative_FibLib.so
```


The command `ndk-build` comes from the NDK's installation directory (e.g. `/path/to/android-ndk-r5b`), so it's easiest if we add this directory to our `PATH`.


On Windows, older version of NDK required Cygwin (a Unix-like environment and command-line interface for Microsoft Windows) to provide "shell" (bash) and "make" (gmake) to `ndk-build`.

To remove all generated binaries, run:

```
$ ndk-build clean
Clean: com_marakana_android_fibonaccinative_FibLib [armeabi]
Clean: stdc++ [armeabi]
```

# Fibonacci - Client

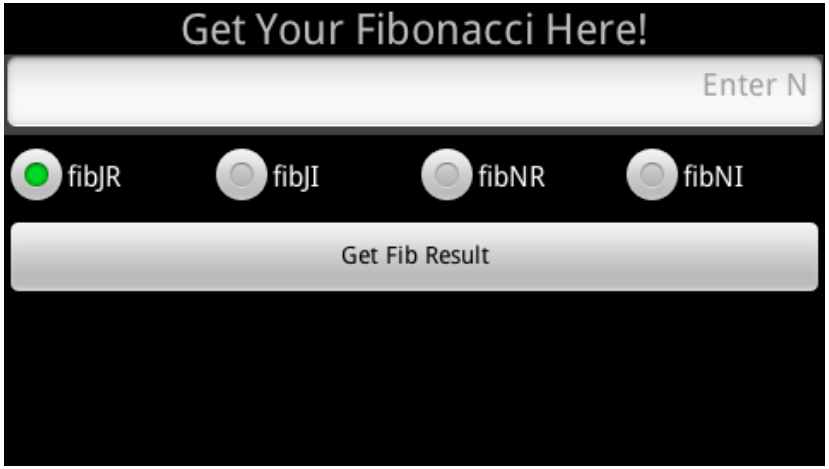
We can now build the "client" of our library (in this case a simple activity) to use our `FibLib` library.

## Fibonacci - String Resources

*FibonacciNative/res/values/strings.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <string name="hello">Get Your Fibonacci Numbers Here!</string>
    <string name="fibJR">fibJR</string>
    <string name="fibJI">fibJI</string>
    <string name="fibNR">fibNR</string>
    <string name="fibNI">fibNI</string>
    <string name="app_name">FibonacciNative</string>
    <string name="button">Get Fibonacci Result</string>

</resources>
```

## Fibonacci - User Interface (Layout)




*Fibonacci Native Main Layout*
*FibonacciNative/res/layout/main.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <!-- This is just a simple title ("Get Your Fibonacci Here!")  -->
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:text="@string/hello"
        android:textSize="25sp" />

    <!-- This is the entry box for our number "n"  -->
    <EditText
        android:id="@+id/input"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
```

```
    android:ems="10"
    android:gravity="right"
    android:inputType="number" >
    <requestFocus />
</EditText>

<!-- This radio group allows the user to select the fibonacci implementation type  -->
<RadioGroup
    android:id="@+id/type"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal" >

    <RadioButton
        android:id="@+id/type_fib_jr"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="@string/fibJR" />

    <RadioButton
        android:id="@+id/type_fib_ji"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="@string/fibJI" />

    <RadioButton
        android:id="@+id/type_fib_nr"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
```

```
            android:text="@string/fibNR" />

        <RadioButton
            android:id="@+id/type_fib_ni"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="@string/fibNI" />
    </RadioGroup>

    <!-- This button allows the user to trigger fibonacci calculation  -->
    <Button
        android:id="@+id/button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/button" />

    <!-- This is the output area for the fibonacci result  -->
    <TextView
        android:id="@+id/output"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center"
        android:textSize="20sp" />

</LinearLayout>
```

# Fibonacci - `FibonacciActivity`

*FibonacciNative/src/com/marakana/android/fibonaccinative/FibonacciActivity.java*

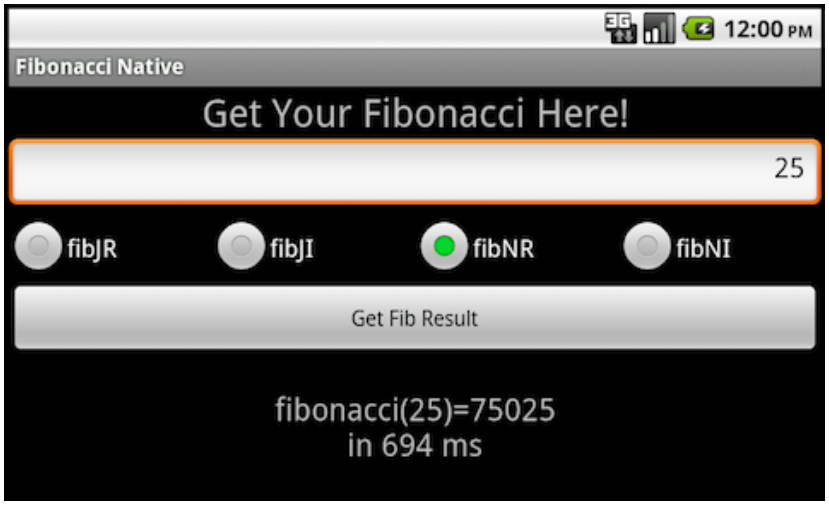
```
package com.marakana.android.fibonaccinative;
```

```
import android.app.Activity;
import android.app.ProgressDialog;
import android.os.AsyncTask;
import android.os.Bundle;
import android.os.SystemClock;
import android.text.TextUtils;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
import android.widget.RadioGroup;
import android.widget.TextView;

public class FibonacciActivity extends Activity implements OnClickListener {

        private EditText input;

        private RadioGroup type;

        private TextView output;

        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.main);
                this.input = (EditText) super.findViewById(R.id.input);
                this.type = (RadioGroup) super.findViewById(R.id.type);
                this.output = (TextView) super.findViewById(R.id.output);
                Button button = (Button) super.findViewById(R.id.button);
                button.setOnClickListener(this);
        }
```

```
public void onClick(View view) {
        String s = this.input.getText().toString();
        if (TextUtils.isEmpty(s)) {
                return;
        }

        final ProgressDialog dialog = ProgressDialog.show(this, "",
                        "Calculating...", true);
        final long n = Long.parseLong(s);
        new AsyncTask<Void, Void, String>() {

                @Override
                protected String doInBackground(Void... params) {
                        long result = 0;
                        long t = SystemClock.uptimeMillis();
                        switch (FibonacciActivity.this.type.getCheckedRadioButtonId()) {
                        case R.id.type_fib_jr:
                                result = FibLib.fibJR(n);
                                break;
                        case R.id.type_fib_ji:
                                result = FibLib.fibJI(n);
                                break;
                        case R.id.type_fib_nr:
                                result = FibLib.fibNR(n);
                                break;
                        case R.id.type_fib_ni:
                                result = FibLib.fibNI(n);
                                break;
                        }
                        t = SystemClock.uptimeMillis() - t;
                        return String.format("fib(%d)=%d in %d ms", n, result, t);
```

```
                        }

                        @Override
                        protected void onPostExecute(String result) {
                                dialog.dismiss();
                                FibonacciActivity.this.output.setText(result);
                        }
                }.execute();
        }
}
```

# Fibonacci - Result




*Fibonacci Native Result*

# NDK’s Stable APIs

The header files for NDK stable APIs are available at `/path/to/ndk/platforms/<android-platform>/<arch-name>/usr/include`.

# Android-specific Log Support

- Include `<android/log.h>` to access various functionality that can be used to send log messages to the kernel (i.e. logcat buffers) from our native code
- Requires that our code be linked to `/system/lib/liblog.so` with `LOCAL_LDLIBS += -llog` in our `Android.mk` file

# ZLib Compression Library

- Include `<zlib.h>` and `<zconf.h>` to access ZLib compression library
  - See `http://www.zlib.net/manual.html` for more info on ZLib

- Requires that our code be linked to `/system/lib/libz.so` with `LOCAL_LDLIBS += -lz` in our `Android.mk` file

# The OpenGL ES 1.x Library

- Include `<GLES/gl.h>` and `<GLES/glext.h>` to access OpenGL ES 1.x rendering calls from native code
  - The "1.x" here refers to both versions 1.0 and 1.1
    - Using 1.1 requires OpenGL-capable GPU
    - Using 1.0 is universally supported since Android includes software renderer for GPU-less devices
    - Requires that we include `<uses-feature>` tag in our manifest file to indicate the actual OpenGL version that we expect

- Requires that our code be linked to `/system/lib/libGLESv1_CM.so` with `LOCAL_LDLIBS += -lGLESv1_CM.so` in our `Android.mk` file
- Since API 4 (Android 1.6)

# The OpenGL ES 2.0 Library

- Include `<GLES2/gl2.h>` and `<GLES2/gl2ext.h>` to access OpenGL ES 2.0 rendering calls from native code
  - Enables the use of vertex and fragment shaders via the GLSL language
  - Since not all devices support OpenGL 2.0, we should include `<uses-feature>` tag in our manifest file to indicate this requirement

- Requires that our code be linked to `/system/lib/libGLESv2.so` with `LOCAL_LDLIBS += -lGLESv2.so` in our `Android.mk` file
- Since API 4 (Android 2.0)

# The `jnigraphics` Library

- Include `<android/bitmap.h>` to reliably access the pixel buffers of Java bitmap objects from native code
- Requires that our code be linked to `/system/lib/libjnigraphics.so` with `LOCAL_LDLIBS += -ljnigraphics` in our `Android.mk` file
- Since API 8 (Android 2.2)

# The OpenSL ES native audio Library

- Include `<SLES/OpenSLES.h>` and `<SLES/OpenSLES_Platform.h>` to perform audio input and output from native code
  - Based on Khronos Group OpenSL ES™ 1.0.1

- Requires that our code be linked to `/system/lib/libOpenSLES.so` with `LOCAL_LDLIBS += -lOpenSLES` in our `Android.mk` file
- Since API 9 (Android 2.3)

# The Android native application APIs

- Makes it possible to write our entire application in native code
  - Mainly added for gaming
  - Our code still depends on the Dalvik VM since most of the platform features are managed in the VM and accessed via JNI (Native → Java)

- Include `<android/native_activity.h>` to write an Android activity (with its life-cycle callbacks) in native code
  - A native activity would serve as the main entry point into our native application

- Include `<android/looper.h>`, `<android/input.h>`, `<android/keycodes.h>`, and `<android/sensor.h>` to listen to input events and sensors directly from native code
- Include `<android/rect.h>`, `<android/window.h>`, `<android/native_window.h>`, and `<android/native_window_jni.h>` for window management from native code
  - Includes ability to lock/unlock the pixel buffer to draw directly into it

- Include `<android/configuration.h>`, `<android/asset_manager.h>`, `<android/storage_manager.h>`, and `<android/obb.h>` for direct access to the assets embedded in our `.apk` files Opaque Binary Blob (OBB) files
  - All access is read-only

- Requires that our code be linked to `libandroid.so` with `LOCAL_LDLIBS += -landroid` in our `Android.mk` file
- Since API 9 (Android 2.3)

With the exception of the libraries listed above, the native system libraries in the Android platform are not considered "stable" and may change in future platform versions. Unless our library is being built for a specific Android ROM, we should only make use of the stable libraries provided by the NDK.

All the header files are available under: `/path/to/ndk-installation-dir/platforms/android-9/arch-arm/usr/include/`

See `/path/to/ndk-installation-dir/docs/STABLE-APIS.html` for the complete reference of NDK's stable APIs.

# Lab: NDK

The objective of this lab is to test your understanding of JNI and NDK. We will do so by adding JNI code to an existing application.

1. Start by importing LogNative application into Eclipse
    1. Menu Bar → *File* → *Import…* → *Git* → *Projects from Git* → *Next* >
    2. Under *Select Repository Source* select *URI* → *Next* >
    3. Under *Source Git Repository* → *Location* → *URI:* enter `https://github.com/marakana/LogNative.git` → *Next* >
    4. Under *Branch Selection*, leave all branches selected (checked) → *Next* >
    5. Under *Local Destination* → *Destination* specify directory of your choice (e.g. `~/android/workspace/LogNative`) → *Next* >
    6. Under *Select a wizard to use for importing projects*, leave *Wizard for project import* as *Import existing projects* → *Next* >
    7. Under *Import Projects* → *Projects*, leave *LogNative* as selected (checked) → *Finish*

        This project can also be downloaded as a ZIP file

2. Examine and test your project in Eclipse
    1. Run the application on a device/emulator
    2. Enter some tag and message to log, click on the *Log* button and observe via `adb logcat` that your message get logged (assuming *Java* was selected)

3. Implement `com.marakana.android.lognative.LogLib.logN(int priority, String tag, String msg)` in C
    1. Mark the method as `native`
    2. Remove its body
    3. Extract its function prototype into a C header file (hint: `javah`)
    4. Implement the function by taking advantage of `<android/log.h>` (i.e. `/system/lib/liblog.so`)
    5. Provide the makefile(s)

4. Build (via `ndk-build`)
5. Run your application
6. Test by selecting `Native` in the UI and checking that the log tag/message shows up in `adb logcat`
7. As a bonus:
    1. Throw `java.lang.NullPointerException` if `tag` or `msg` are null
    2. Throw `java.lang.IllegalArgumentException` if `priority` is not one of the allowed types or if `tag` or `msg` are empty

Don't forget to convert `tag` and `msg` strings from the Java format (`jstring`) to native format (`char *`) before trying to use them in `int __android_log_write(int prio, const char *tag, const char *text)`. Be sure to *free* the native strings before returning from the native method. Finally, don't forget to tell the linker about your use of the *log* library.

The solution is provided:

- As a ZIP archive: https://github.com/marakana/LogNative/zipball/solution
- By Git: `git clone https://github.com/marakana/LogNative.git -b solution`

# Summary of NDK Module

In this module, you learned how Android uses JNI to bridge between the world of Java and the native code. You also learned how NDK makes the process of working with JNI simpler by providing tools and framework for developing native libraries as well as packaging them with the app.

**Thank you!**

Marko Gargenta & Marakana Team

@MarkoGargenta

Special thanks to Aleksandar (Sasa) Gargenta as well as the rest of Marakana team for research related to NDK.

Slides & video of this presentation is available at Marakana.com

Yamba source code is available at https://github.com/marakana/