Using the Android* Native Development Kit (NDK)

Xavier Hallade, Developer Evangelist, Intel Corporation
@ph0b - ph0b.com

Agenda

- The Android* Native Development Kit (NDK)
- How to use the NDK
- Supporting several CPU architectures
- Debug and Optimization
- Q&A

The Android Native Development Kit (NDK)

Android* Native Development Kit (NDK)

What is it?

Build scripts/toolkit to incorporate native code in Android* apps via the Java Native Interface (JNI)

Why use it?

Performance

e.g., complex algorithms, multimedia applications, games

Differentiation

app that takes advantage of direct CPU/HW access

e.g., using SSE4.2 for optimization

Software code reuse

Why not use it?

Performance improvement isn't always guaranteed, the complexity is…

Installing the Android* NDK

NDK is a platform dependent archive (self-extracting since r10c).

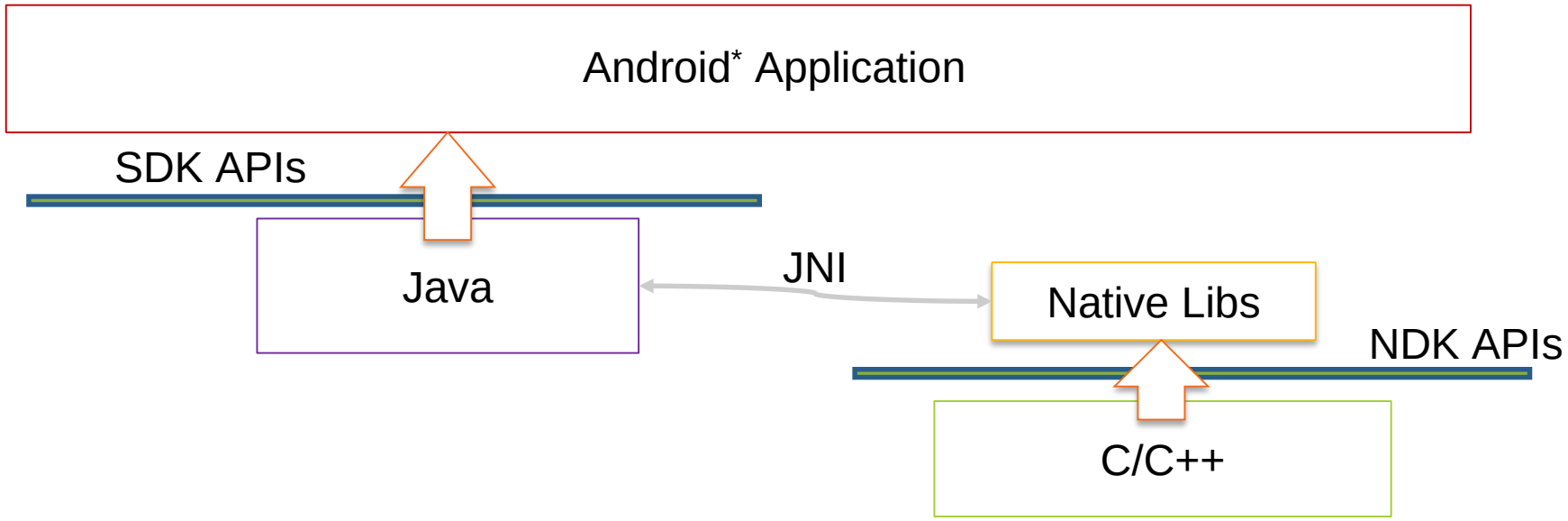
It provides:

- Build script **ndk-build(.cmd)**
- Other scripts for toolchains generation, debug, etc
- Android* headers and libraries
- gcc/clang crosscompilers
- Documentation and samples (useful and not easy to find online)

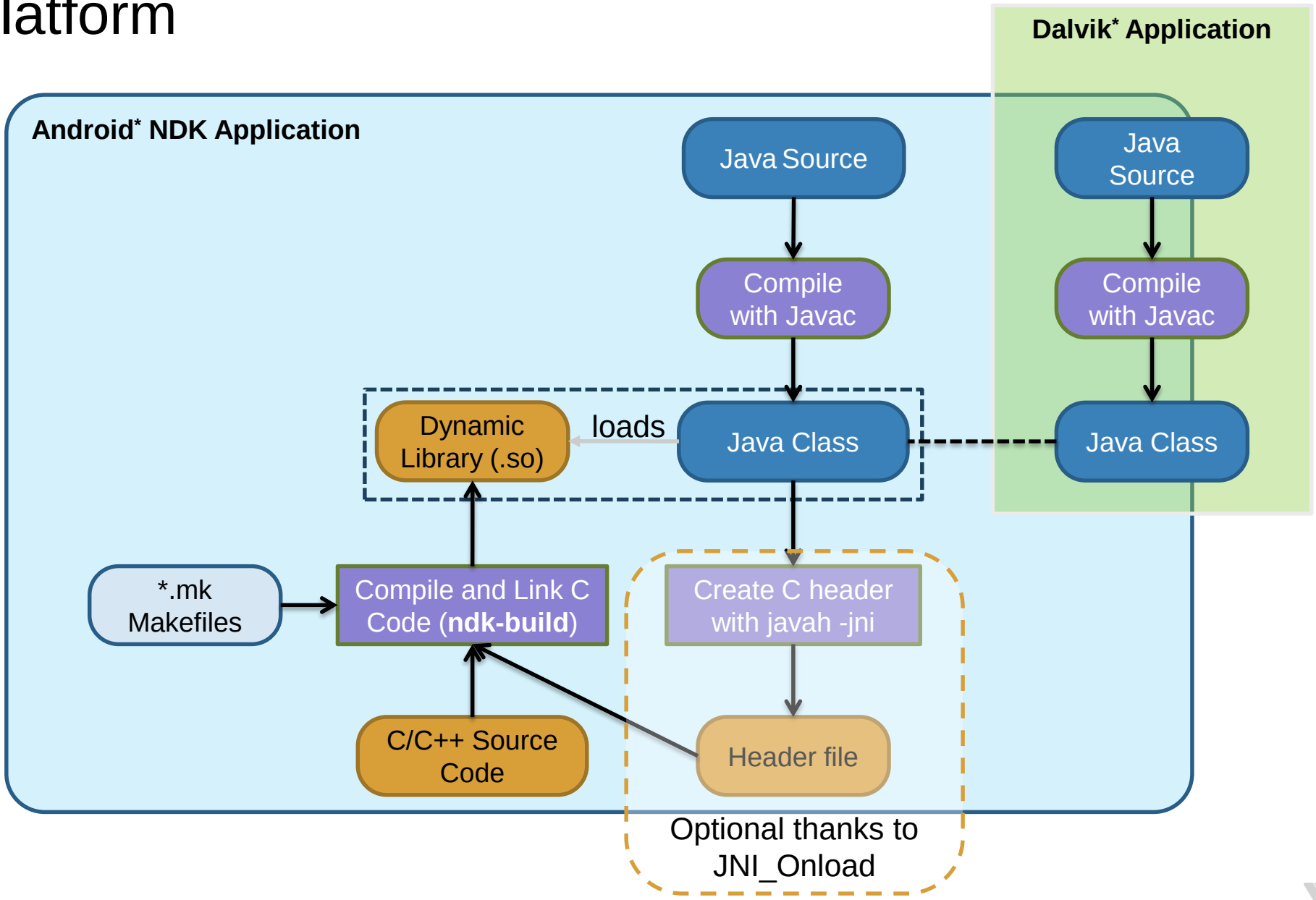
Downloads

Platform (32-bit target)	Package	Size (Bytes)	MD5 Checksum
Windows 32-bit	android-ndk32-r10b-windows-x86.zip	502720425	9fa4f19bca7edd6eefa63fe788737987
Windows 64-bit	android-ndk32-r10b-windows-x86_64.zip	531912027	e4dd2e0c6f38e3ad936c366bdf6b1d4e
Mac OS X 32-bit	android-ndk32-r10b-darwin-x86.tar.bz2	406998070	db3626b2c5f3245d90e2724f7bcf4c3e
Mac OS X 64-bit	android-ndk32-r10b-darwin-x86_64.tar.bz2	413652124	7ca4a84e9c56c38acdafb007e7cd33c5
Linux 32-bit	android-ndk32-r10b-linux-	421052081	e8f55daa5c9de7ab79aaaf5d7d751b69

NDK Application Development



NDK Platform



Compatibility with Standard C/C++

Bionic C Library:

- Lighter than standard GNU C Library
- Not POSIX compliant
- pthread support included, but limited
- No System-V IPCs
- Access to Android* system properties

Bionic is not binary-compatible with the standard C library

It means you generally need to (re)compile everything using the Android NDK toolchain

Android* C++ Support

By default, system is used. It lacks:
Standard C++ Library support (except some headers)
C++ exceptions support
RTTI support
Fortunately, you have other libs available with the NDK:

Runtime	Exceptions	RTTI	STL
system	No	No	No
gabi++	Yes	Yes	No
stlport	Yes	Yes	Yes
gnustl	Yes	Yes	Yes
libc++	Yes	Yes	Yes

Choose which library to compile against in your Makefile (Application.mk file):

```
APP_STL := gnustl_shared
```

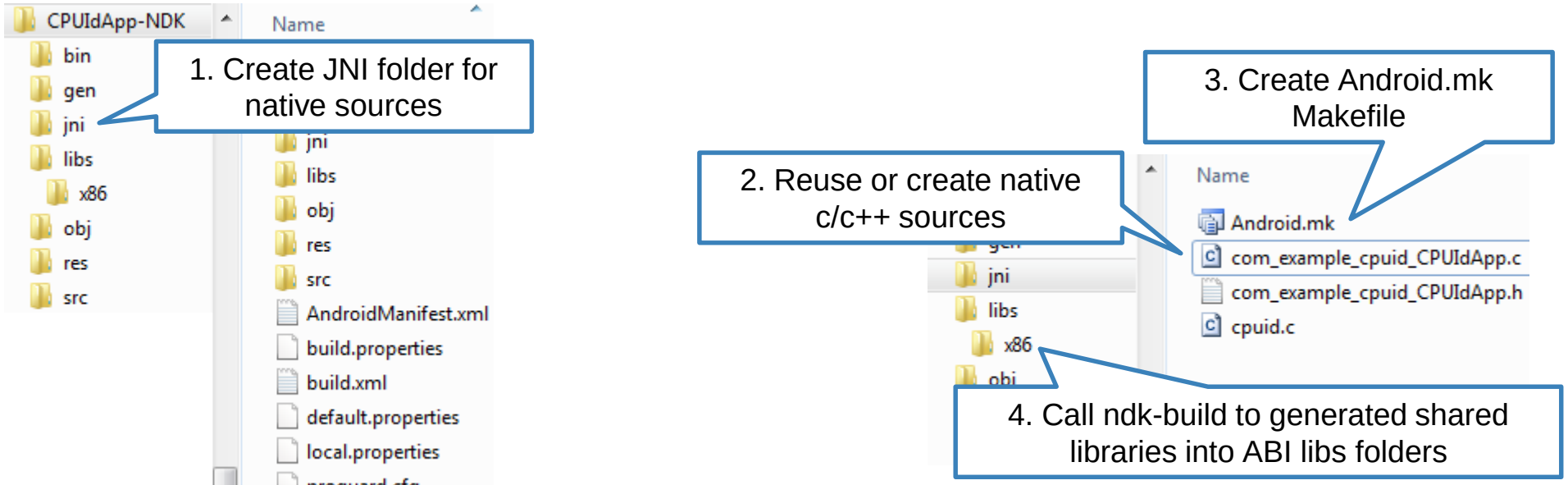
Postfix the runtime with _static or _shared

For using C++ features, you also need to enable these in your Makefile:

```
LOCAL_CPP_FEATURES += exceptions rtti
```

How to use the NDK

Manually Adding Native Code to an Android* Project



Integrating Native Functions with Java*

Declare native methods in your Android* application (Java*) using the ‘native’ keyword:

```
public native String stringFromJNI();
```

Provide a native shared library built with the NDK that contains the methods used by your application:

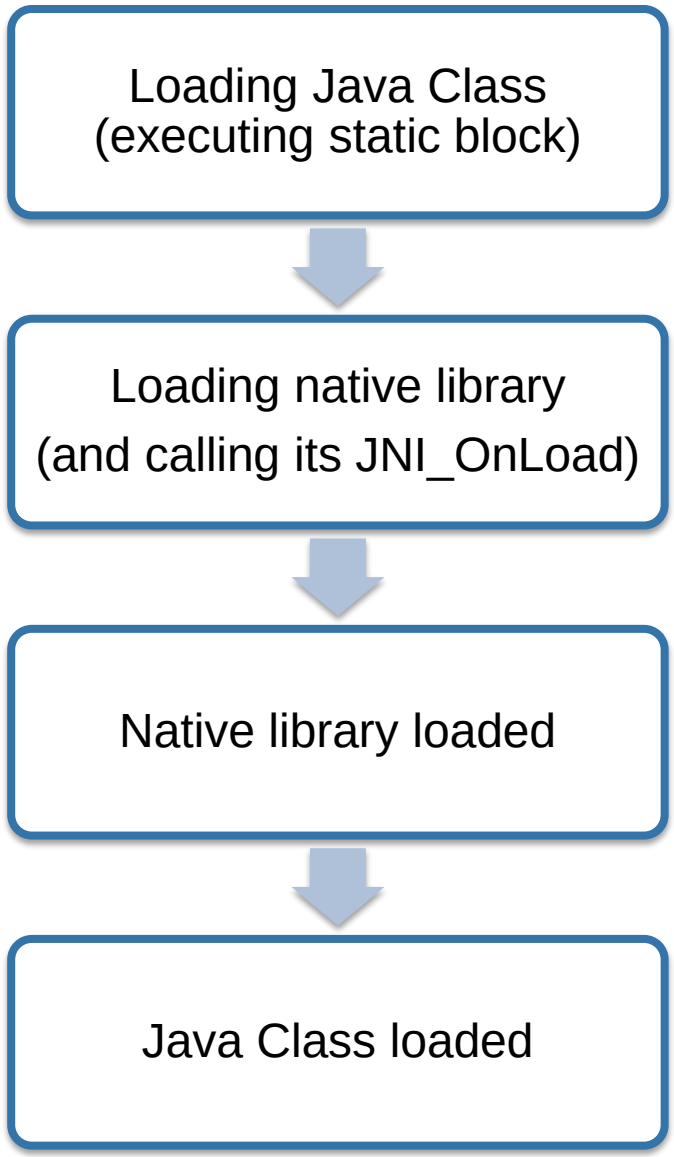
```
libMyLib.so
```

Your application must load the shared library (before use… during class load for example):

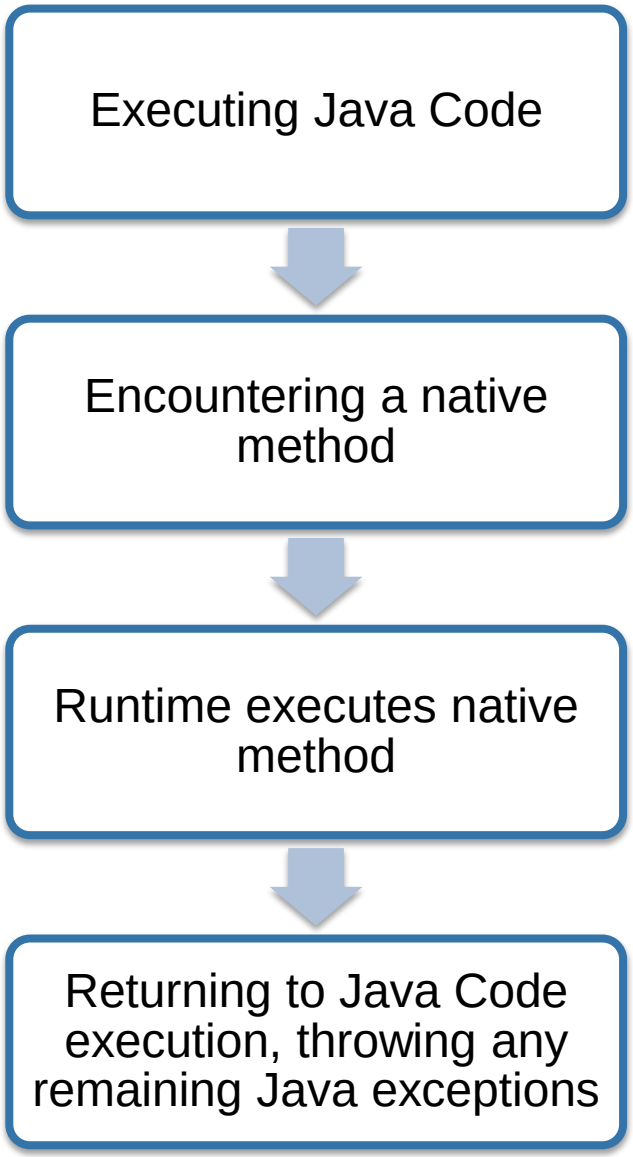
```
static {
    System.loadLibrary("MyLib");
}
```

There is two ways to associate your native code to the Java methods: javah and JNI_OnLoad

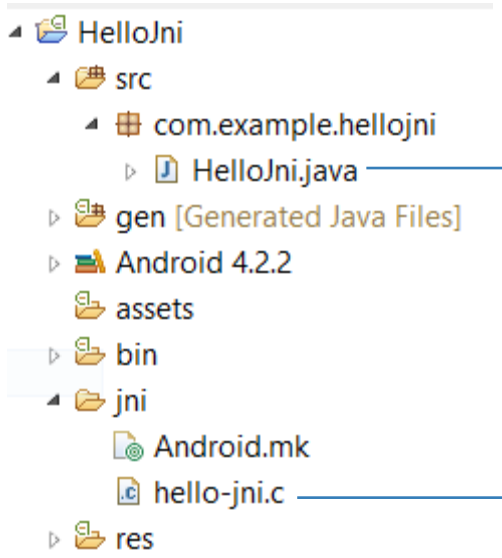
Classic Execution flow



And later, from any thread:



Javah Method




```
...
    {
        ...
        tv.setText( stringFromJNI() );
        ...
    }

    public native String  stringFromJNI();

    static {
        System.loadLibrary("hello-jni");
    }
```

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

Javah Method

“javah” generates the appropriate JNI header stubs from the compiled Java classes files.

Example:

```
> javah -d jni -classpath bin/classes com.example.hellojni.HelloJni
```

Generates `com_example_hellojni_HelloJni.h` file with this definition:

```
JNIEXPORT jstring JNICALL
Java_com_example_hellojni_HelloJni_stringFromJNI(JNIEnv *, jobject);
```

Android Makefiles (*.mk)

Android.mk

module settings and declarations

```
include $(CLEAR_VARS)
LOCAL_MODULE    := hello-jni
LOCAL_SRC_FILES := hello-jni.c
include $(BUILD_SHARED_LIBRARY)
```

Predefined macro can be:
BUILD_SHARED_LIBRARY,
BUILD_STATIC_LIBRARY,
PREBUILT_SHARED_LIBRARY, PREBUILT_STATIC_LIBRARY

Other useful variables:

```
LOCAL_C_INCLUDES := ./headers/
LOCAL_EXPORT_C_INCLUDES := ./headers/
LOCAL_SHARED_LIBRARIES := module_shared
LOCAL_STATIC_LIBRARIES := module_static
```

Application.mk

Application-wide settings

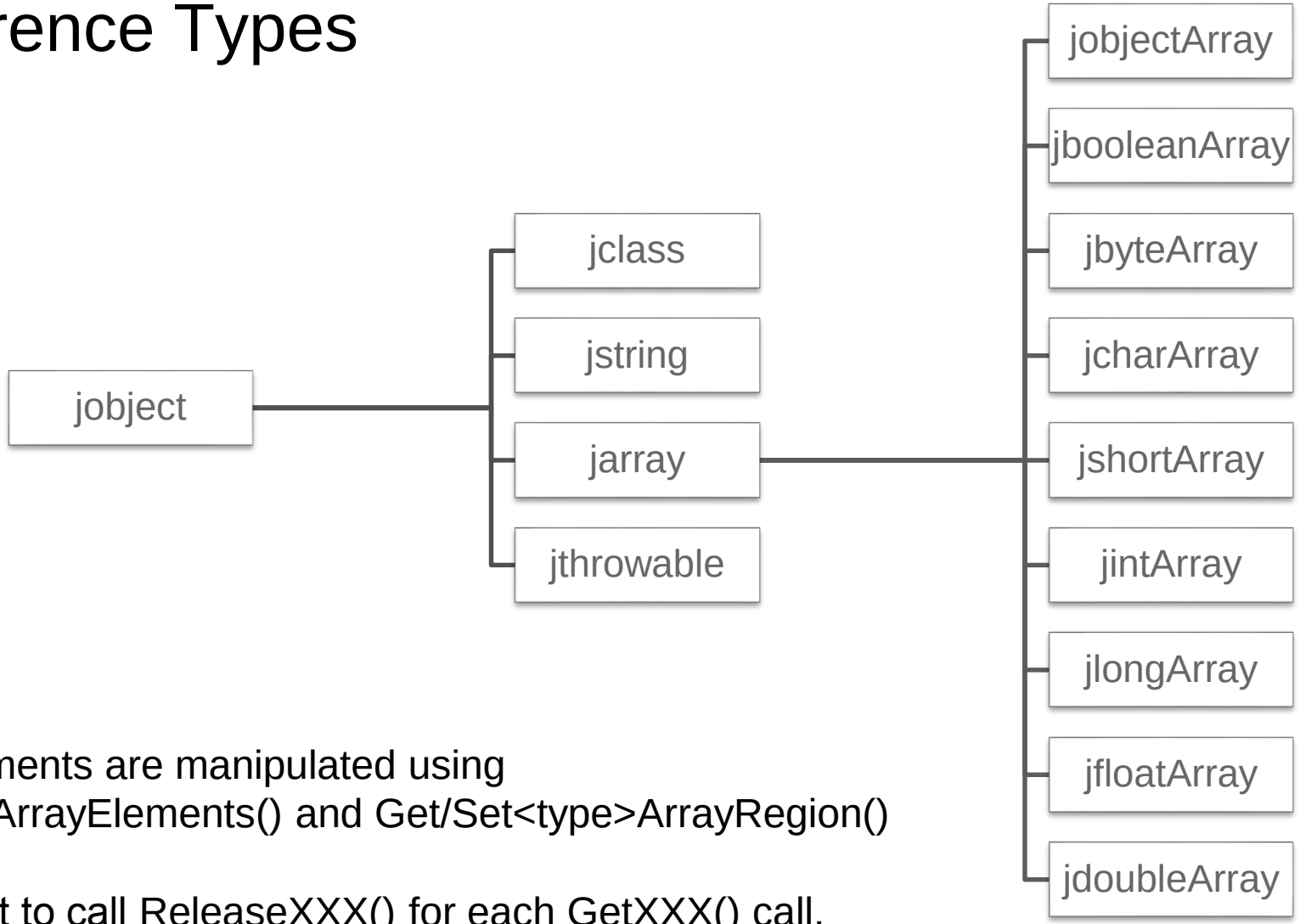
```
APP_PLATFORM := android-15
#~=minSDKVersion
APP_CFLAGS := -O3
APP_STL := gnustl_shared #or other STL if you need extended C++ support
APP_ABI := all #or all32, all64…
APP_OPTIM := release #default
NDK_TOOCLHAIN_VERSION := 4.8 #4.6 is default, 4.8 brings perfs, 4.9 also but less stable
```

Working with the Java Native Interface

JNI Primitive Types

Java* Type	Native Type	Description
boolean	jboolean	unsigned 8 bits
byte	jbyte	signed 8 bits
char	jchar	unsigned 16 bits
short	jshort	signed 16 bits
int	jint	signed 32 bits
long	jlong	signed 64 bits
float	jfloat	32 bits
double	jdouble	64 bits
void	void	N/A

JNI Reference Types



Arrays elements are manipulated using
Get<type>ArrayElements() and Get/Set<type>ArrayRegion()

Don’t forget to call ReleaseXXX() for each GetXXX() call.

Creating a Java* String

C:

```
jstring string =
        (*env)->NewStringUTF(env, "new Java String");
```

C++:

```
jstring string = env->NewStringUTF("new Java String");
```

- Memory is handled by the JVM, jstring is always a reference.
- You can call DeleteLocalRef() on it once you finished with it.

Main difference with using JNI from C or in C++ is the nature of “env” as you can see it here.

Memory Handling of Java* Objects

Memory handling of Java* objects is done by the JVM:

- You only deal with references to these objects
- Each time you get a reference, you must not forget to delete it after use
- local references are still automatically deleted when the native call returns to Java
- References are local by default
- Global references can be created using NewGlobalRef()

Getting a C string from Java* String

```
const char *nativeString = (*env)-
>GetStringUTFChars(javaString, null);
…
(*env)->ReleaseStringUTFChars(env, javaString, nativeString);

//more secure, more efficient if you want a copy anyway
int tmpjstrlen = env->GetStringUTFLength(tmpjstr);
char* fname = new char[tmpjstrlen + 1];
env->GetStringUTFRegion(tmpjstr, 0, tmpjstrlen, fname);
fname[tmpjstrlen] = 0;
…
delete fname;
```

Calling Java* Methods

On an object instance:

```
jclass clazz = (*env)->GetObjectClass(env, obj);
jmethodID mid = (*env)->GetMethodID(env, clazz, "methodName",
"(…)…");
if (mid != NULL)
        (*env)->Call<Type>Method(env, obj, mid, parameters…);
```

Static call:

```
jclass clazz = (*env)->FindClass(env, "java/lang/String");
jmethodID mid = (*env)->GetStaticMethodID(env, clazz, "methodName",
"(…)…");
if (mid != NULL)
        (*env)->CallStatic<Type>Method(env, clazz, mid, parameters…);
```

- (…)…: method signature
- parameters: list of parameters expected by the Java* method
- <Type>: Java method return type

Handling Java* Exceptions

```
// call to java methods may throw Java exceptions

jthrowable ex = (*env)->ExceptionOccurred(env);
if (ex!=NULL) {
        (*env)->ExceptionClear(env);
        // deal with exception
}

(*env)->DeleteLocalRef(env, ex);
```

Throwing Java* Exceptions

```
jclass clazz =
(*env->FindClass(env, "java/lang/Exception");

if (clazz!=NULL)
    (*env)->ThrowNew(env, clazz, "Message");
```

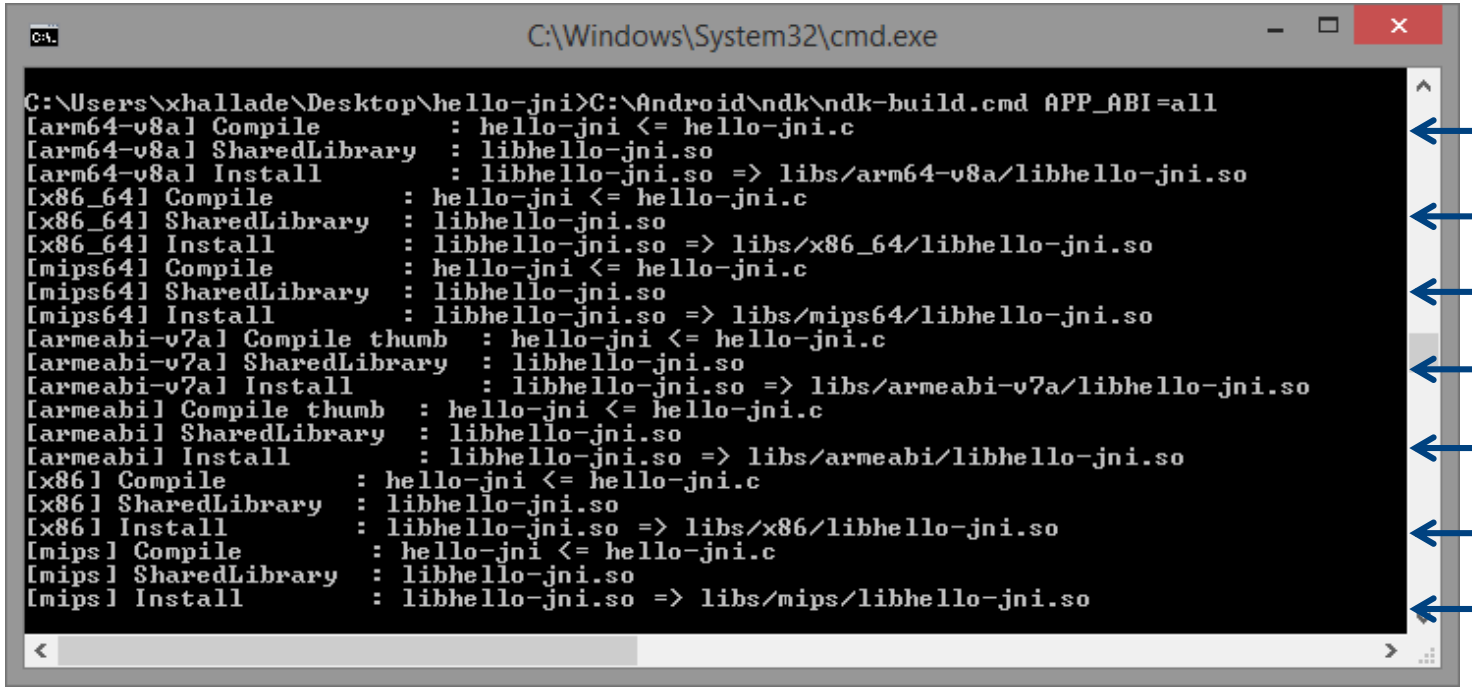
The exception will be thrown only when the JNI call returns to Java*, it will not break the current native code execution.

Handling ABIs

Configuring NDK Target ABIs

Include all ABIs by setting APP_ABI to all in jni/**Application.mk**:

```
APP_ABI=all
```



The NDK will generate optimized code for all target ABIs

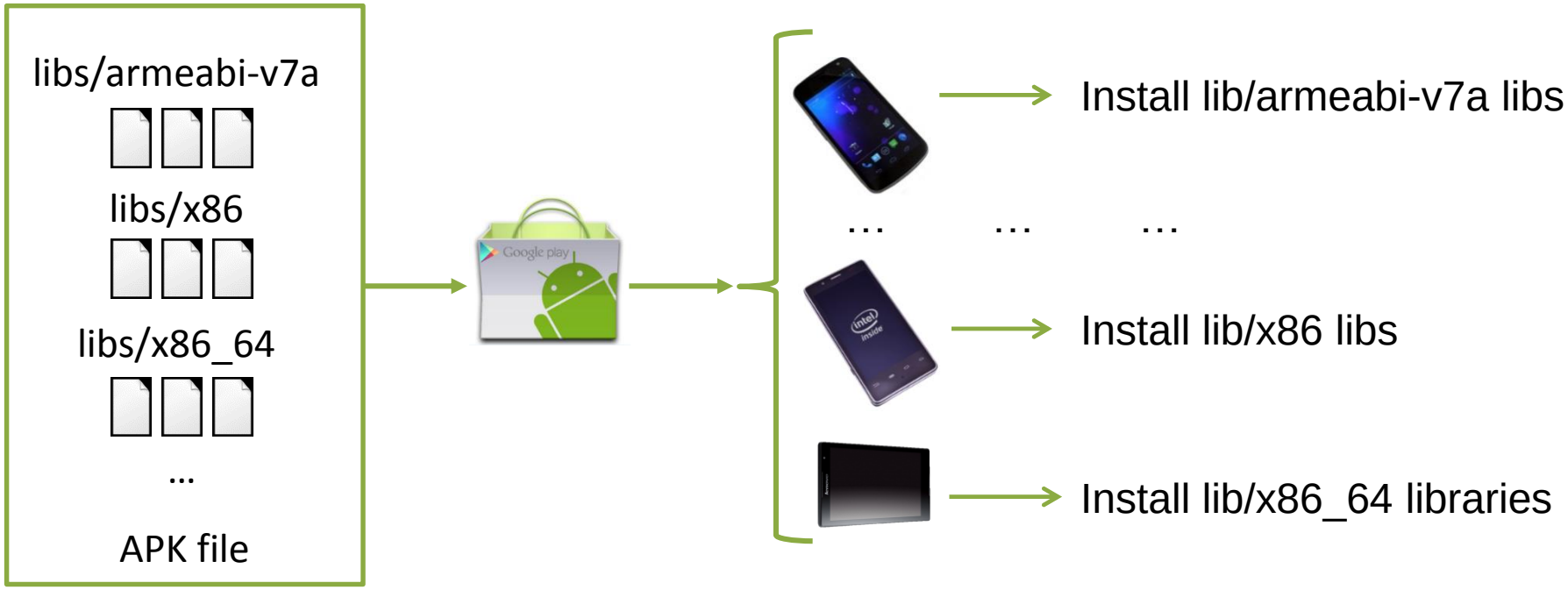
You can also pass APP_ABI variable to ndk-build, and specify each ABI:

```
ndk-build APP_ABI=x86
```

`all32` and `all64` are also possible values.

“Fat” APKs

By default, an APK contains libraries for every supported ABIs.



Libs for the selected ABI are installed, the others remain inside the downloaded APK

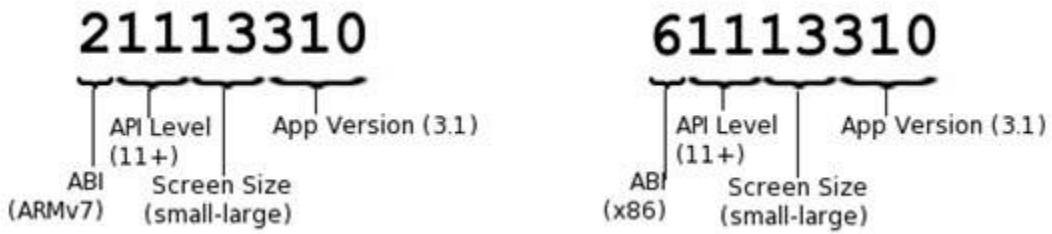
Multiple APKs

Google Play* supports multiple APKs for the same application.
What compatible APK will be chosen for a device entirely depends on the android:versionCode

If you have multiple APKs for multiple ABIs, best is to simply prefix your current version code with a digit representing the ABI:

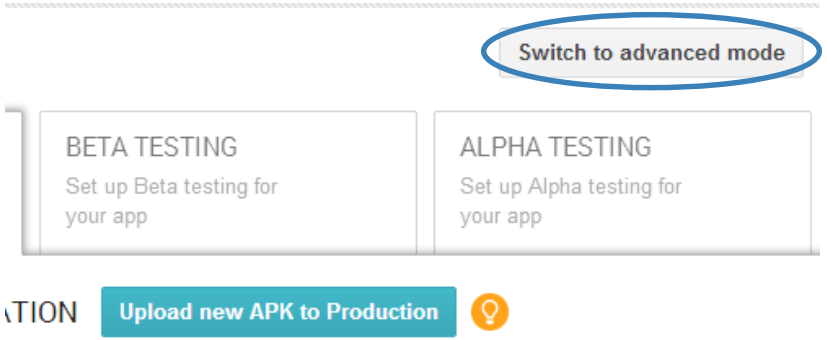
2310 → ARMv7
3310 → ARM64
6310 → x86
7310 → X86_64

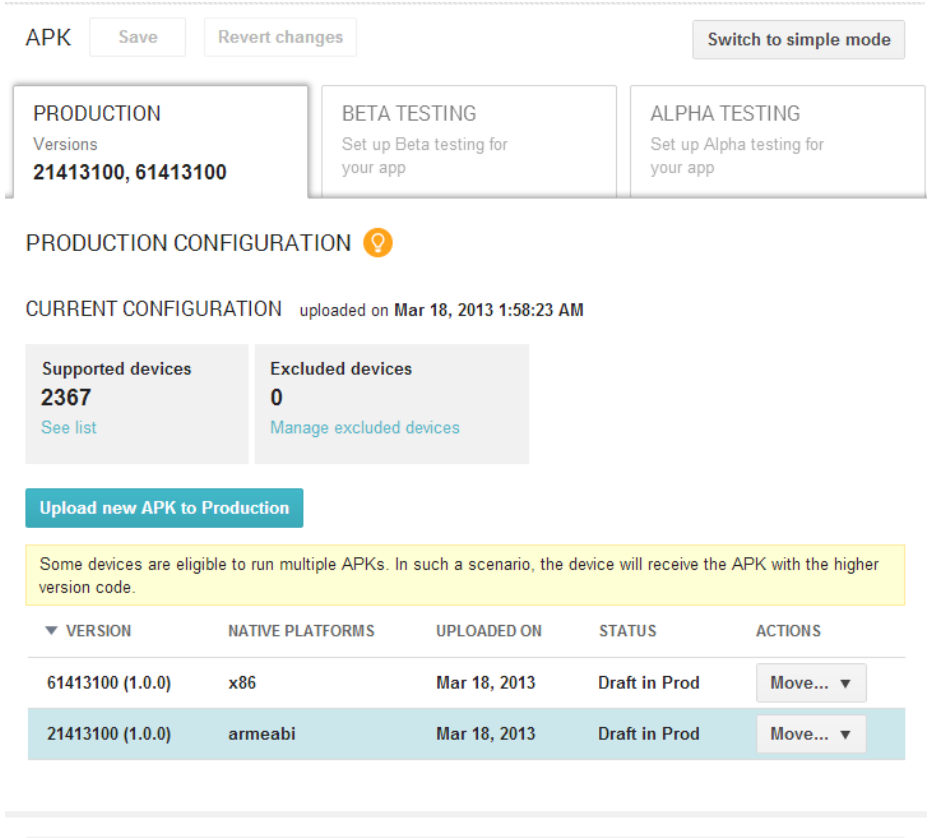
You can have more options for multiple APKs, here is a convention that will work if you're using



Uploading Multiple APKs to the store

Switch to Advanced mode before uploading the second APK.





3rd party libraries x86 support

Game engines/libraries with x86 support:

- Havok Anarchy SDK: android x86 target available
- Unreal Engine 3: android x86 target available
- Marmalade: android x86 target available
- Cocos2Dx: set APP_ABI in Application.mk
- FMOD: x86 lib already included, set ABIs in Application.mk
- AppGameKit: x86 lib included, set ABIs in Application.mk
- libgdx: x86 supported by default in latest releases
- AppPortable: x86 support now available
- Adobe Air: x86 support now available
- Unity: in beta, will be released soon.
- …

Debugging native code

Debugging with logcat

NDK provides log API in <android/log.h>:

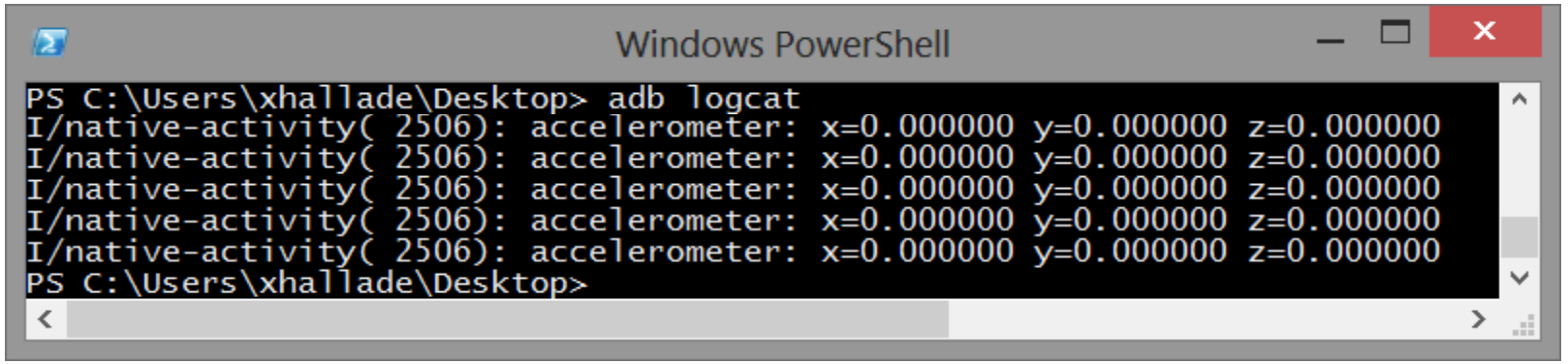
```
int __android_log_print(int prio, const char *tag,
    const char *fmt, ...)
```

Usually used through this sort of macro:

```
#define LOGI(...) ((void)__android_log_print(ANDROID_LOG_INFO, "APPTAG", __VA_ARGS__))
```

Usage Example:

LOGI("accelerometer: x=%f y=%f z=%f", x, y, z);



Debugging with logcat

To get more information on native code execution:

```
adb shell setprop debug.checkjni 1
```

(already enabled by default in the emulator)

And to get memory debug information (root only):

```
adb shell setprop libc.debug.malloc 1
```

-> leak detection

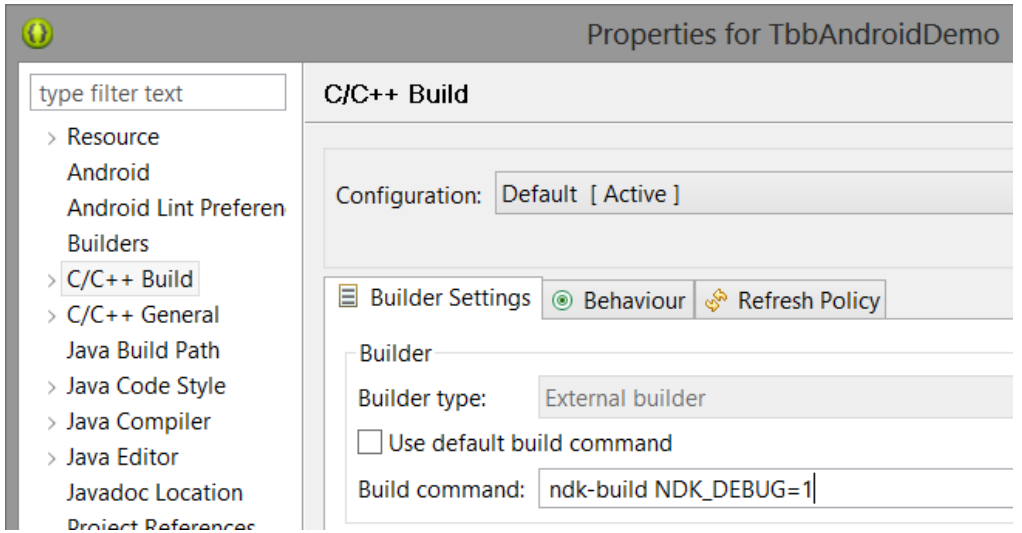
```
adb shell setprop libc.debug.malloc 10
```

-> overruns detection

`adb shell start/stop` -> reload environment

Debugging with GDB and Eclipse

Native support must be added to your project
Pass NDK_DEBUG=1 APP_OPTIM=debug to the ndk-build command, from the project properties:



NDK_DEBUG flag is supposed to be automatically set for a debug build, but this is not currently the case.

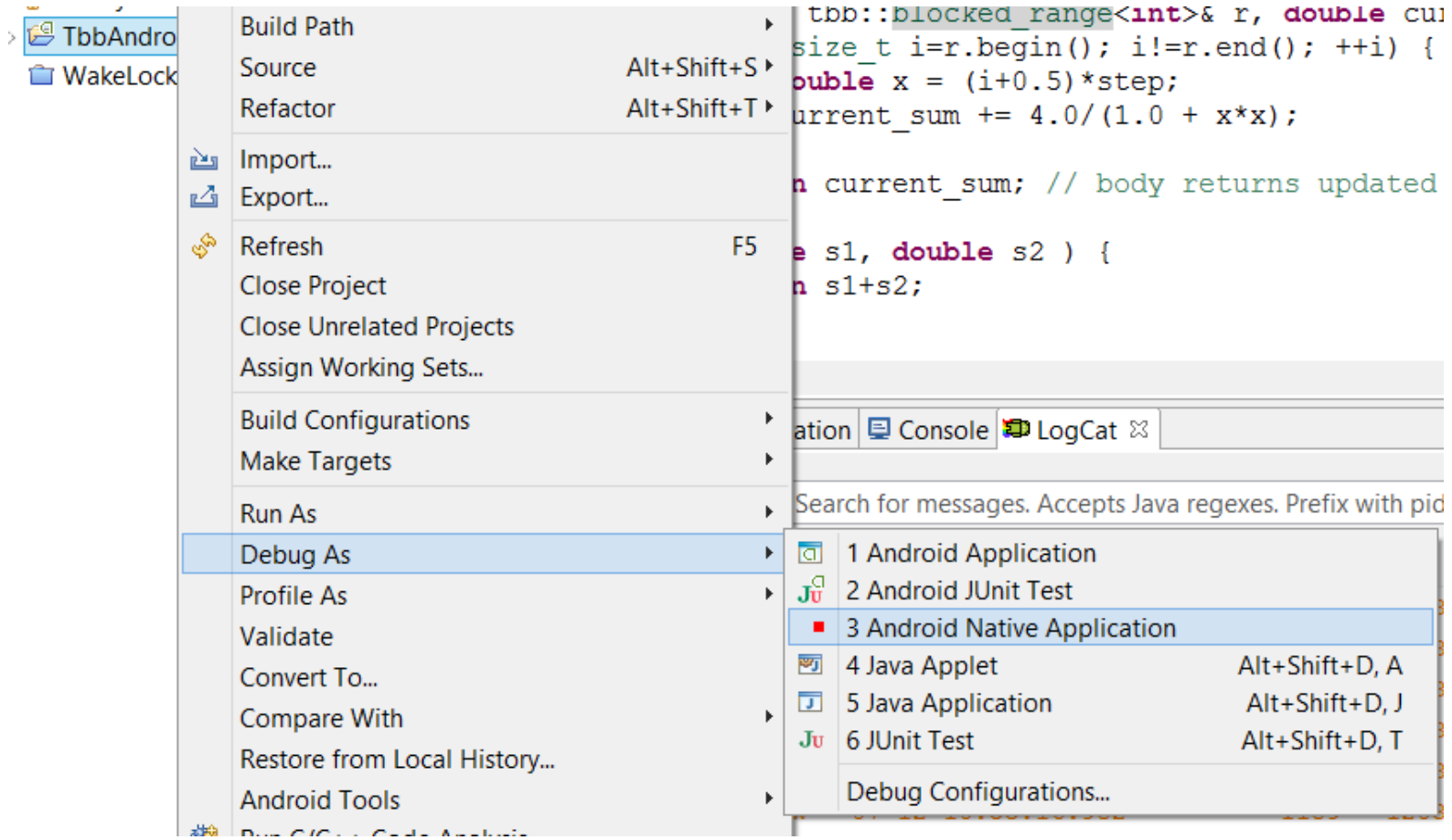
Debugging with GDB and Eclipse*

When NDK_DEBUG=1 is specified, a “gdbserver” file is added to your libraries

```
C:\Android\ndk\ndk-build.cmd NDK_DEBUG=1 all
Gdbserver      : [x86-4.6] libs/x86/gdbserver
Gdbsetup       : libs/x86/gdb.setup
"Compile++ x86  : TbbAndroidDemo <= TbbAndroidDemo.cpp
SharedLibrary  : libTbbAndroidDemo.so
Install        : libTbbAndroidDemo.so => libs/x86/libTbbAndroidDemo.so
Install        : libtbb.so => libs/x86/libtbb.so
Install        : libgnustl_shared.so => libs/x86/libgnustl_shared.so

**** Build Finished ****
```

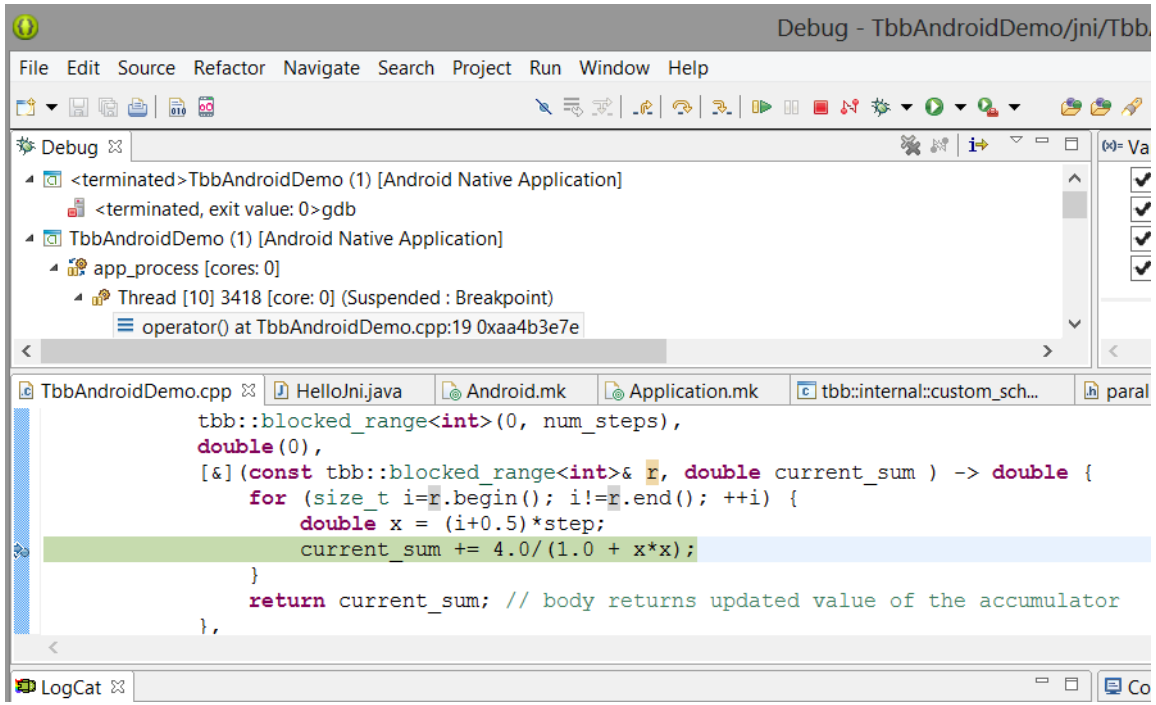
Debugging with GDB and Eclipse*

Debug your project as a native Android* application:



Debugging with GDB and Eclipse

From Eclipse “Debug” perspective, you can manipulate breakpoints and debug your project



Your application will run before the debugger is attached, hence breakpoints you set near application launch will be ignored

Going further with the NDK

Android* NDK Samples

Sample App	Type
hello-jni	Call a native function written in C from Java*.
bitmap-plasma	Access an Android* Bitmap object from C.
san-angeles	EGL and OpenGL* ES code in C.
hello-gl2	EGL setup in Java and OpenGL ES code in C.
native-activity	C only OpenGL sample (no Java, uses the NativeActivity class).
two-libs	Integrates more than one library
…	

JNI_OnLoad Method – Why ?

- Proven method
- No more surprises after methods registration
- Less error prone when refactoring
- Add/remove native functions easily
- No symbol table issue when mixing C/C++ code
- Best spot to cache Java* class object references

JNI_OnLoad Method

In your library name your functions as you wish and declare the mapping with JVM methods:

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

static JNINativeMethod exposedMethods[] = {
  {"stringFromJNI","()Ljava/lang/String;",(void*)stringFromJNI},
}
```

()Ljava/lang/String; is the JNI signature of the Java* method, you can retrieve it using the javap utility:

```
> javap -s -classpath bin\classes -p com.example.hellojni.HelloJni
Compiled from "HelloJni.java"
…
public native java.lang.String stringFromJNI();
  Signature: ()Ljava/lang/String;
…
```

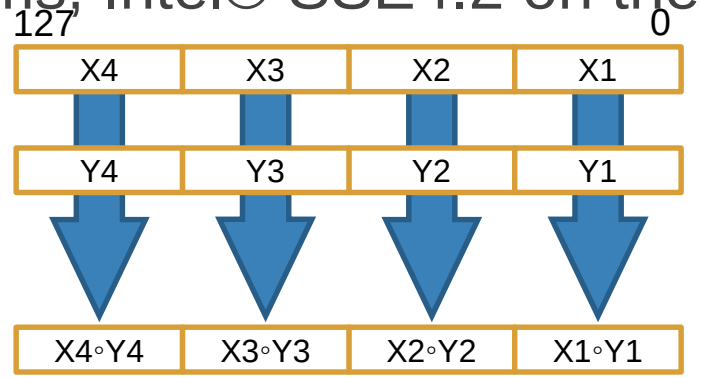
JNI_OnLoad Method

```
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;

    jclass clazz = env->FindClass("com/example/hellojni/HelloJni");
    if(clazz==NULL)
        return JNI_ERR;

    env->RegisterNatives(clazz, exposedMethods,
sizeof(exposedMethods)/sizeof(JNINativeMethod));
    env->DeleteLocalRef(clazz);

    return JNI_VERSION_1_6;
}
```

JNI_OnLoad is the library entry point called during load.
Here it applies the mapping defined on the previous slide.

Vectorization

SIMD instructions up to SSSE3 available on current Intel® Atom™ processor based platforms, Intel® SSE4.2 on the Intel Silvermont Microarchitecture



SSSE3, SSE4.2

Vector size: **128 bit**

Data types:

- 8, 16, 32, 64 bit integer
- 32 and 64 bit float

VL: 2, 4, 8, 16

On ARM*, you can get vectorization through the ARM NEON* instructions

Two classic ways to use these instructions:

- Compiler auto-vectorization
- Compiler intrinsics

GCC Flags

```
ifeq ($(TARGET_ARCH_ABI),x86)
  LOCAL_CFLAGS += -ffast-math -mtune=atom -mssse3 -mfpmath=sse
endif
ifeq ($(TARGET_ARCH_ABI),x86_64)
  LOCAL_CFLAGS += -ffast-math -mtune=slm -msse4.2
endif
```

ffast-math influence round-off of fp arithmetic and so breaks strict IEEE compliance

The other optimizations are totally safe

Add -ftree-vectorizer-verbose to get a vectorization report

`NDK_TOOLCHAIN_VERSION:=4.8` or `4.9` in Application.mk to use latest version

Android* Studio NDK support

- Having .c(pp) sources inside jni folder ?
  - ndk-build automatically called on a generated Android.mk, ignoring any existing .mk
  - All configuration done through build.gradle (moduleName, ldLibs, cFlags, stl)
  - You can change that to continue relying on your own Makefiles:
    http://ph0b.com/android-studio-gradle-and-ndk-integration/
- Having .so files to integrate ?
  - Copy them to jniLibs/ABI folders or integrate them from a .aar library
- Use APK splits to generate one APK per arch with a computed versionCode
  http://tools.android.com/tech-docs/new-build-system/user-guide/apk-splits

Q&A

xavier.hallade@intel.com
@ph0b – ph0b.com

Some last comments

- In Application.mk, ANDROID_PLATFORM must be the same as your minSdkLevel. This is especially important with Android-L.
- With Android L (ART), JNI is more strict than before:
  - pay attention to objects and JNIEnv references, threads and methods mapping