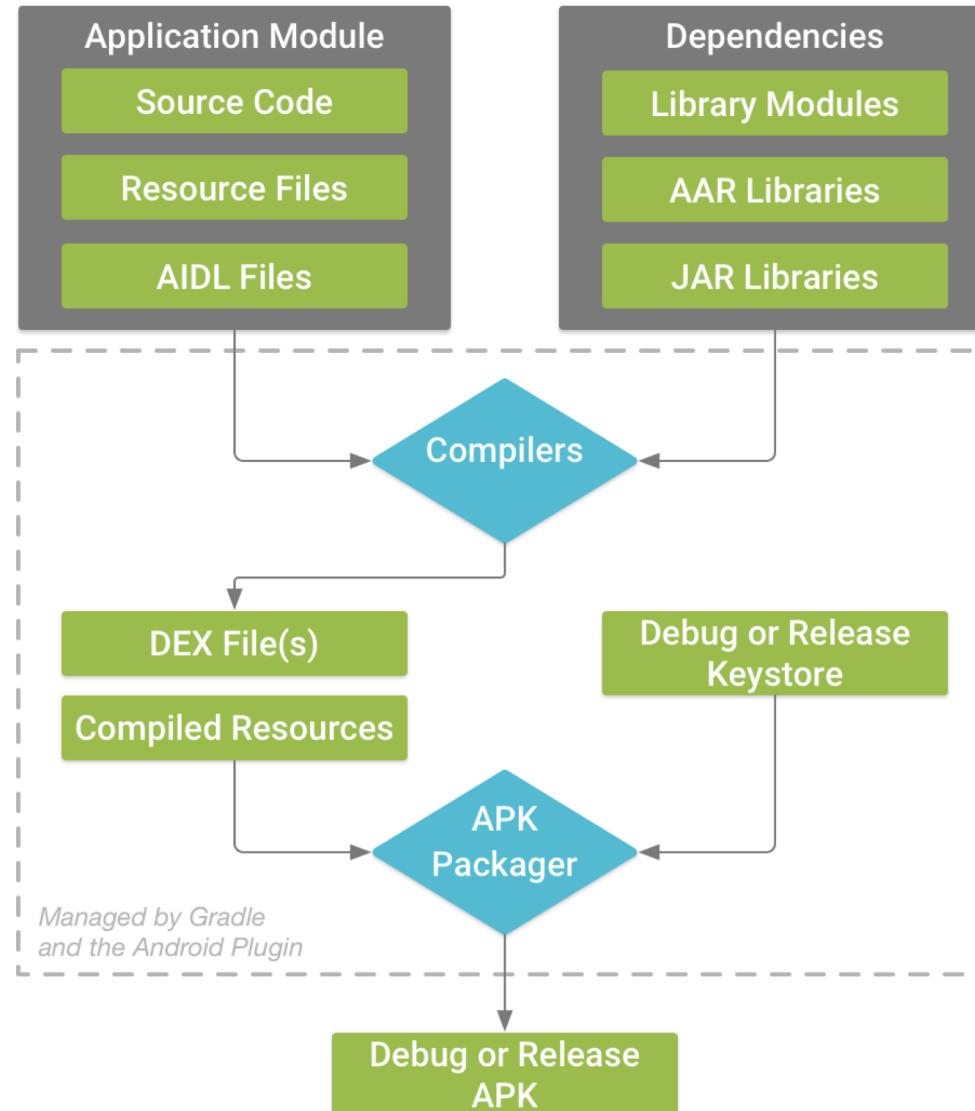


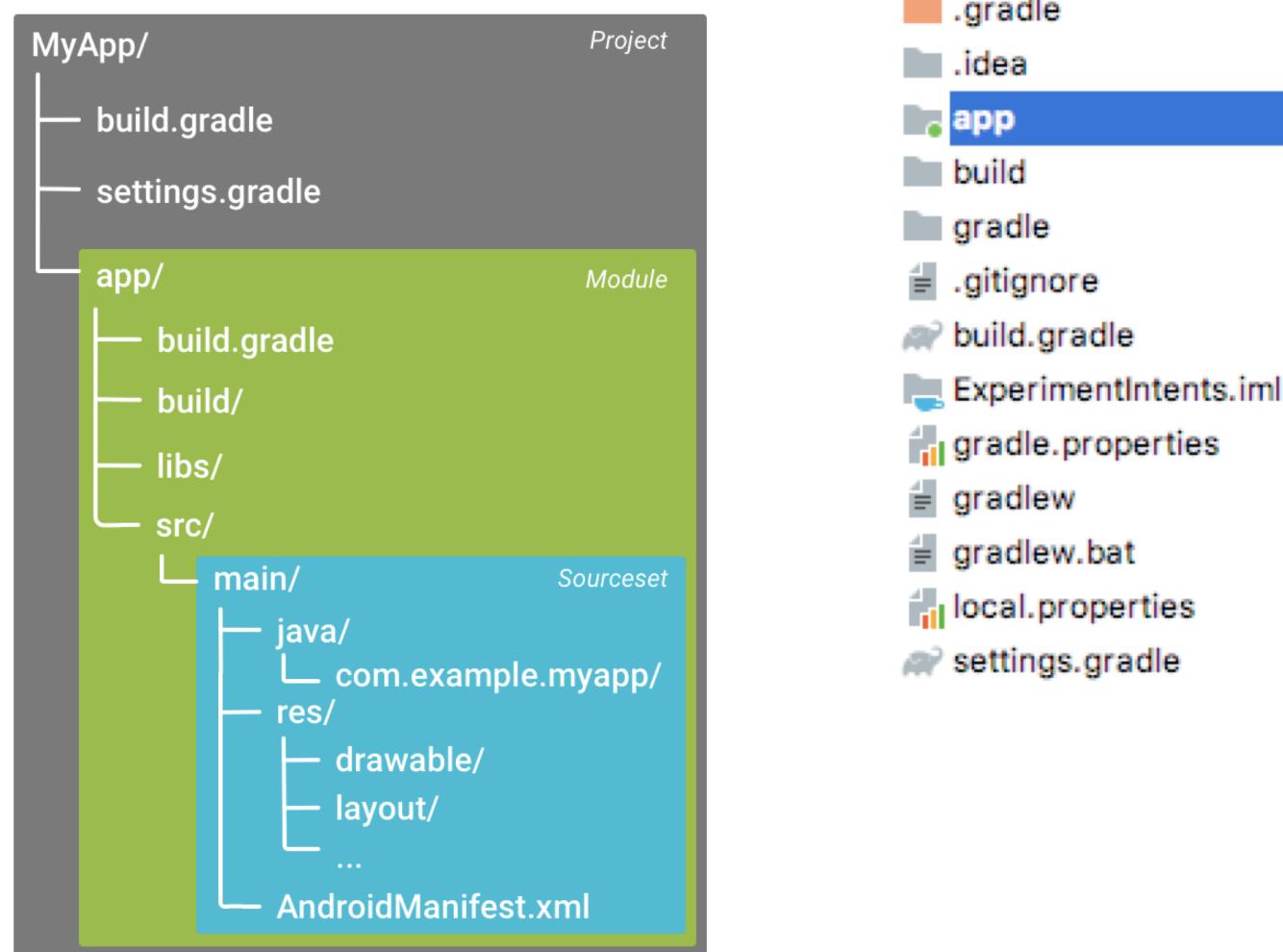
# NATIVE LIBRARIES AND NDK

---

# The build process

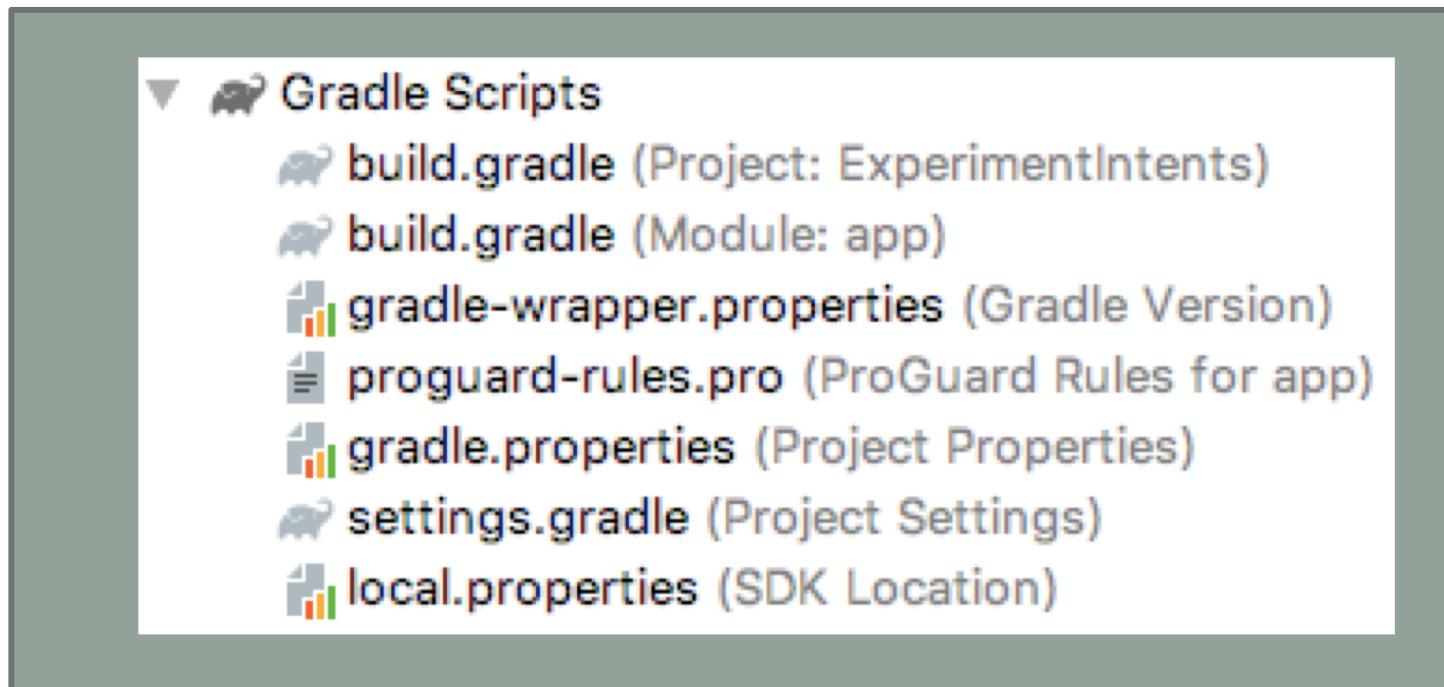


# Default Project structure



# Build environment and scripts

- Based on the **gradle** toolkit
- The **.properties** file defines several configuration variables of the build environment (i.e.  **sdk.dir=**)
- The build process is controlled by several **scripts** using a **DSL** and having project or module scope



# Project setting [setting.gradle]

- include ':app'

```
rootProject.name='MyApp'
```

- Lists the modules to include in the build

# Project script [gradle.build]

- defines build configurations that apply to **all modules** in your project
- The top-level build file uses the `buildscript` block to define the Gradle *repositories* and *dependencies* that are common to all modules in the project
- Dependencies are transitive and processed once

# Modules and dependencies

- **Dependency on a local library module**
- **implementation project(":mylibrary")**
  - the build system **compiles** the library module and packages the resulting compiled contents in the APK.
- **Dependency on a local binary**
- **implementation fileTree(dir: 'libs', include: ['\*.jar'])**
- **implementation files('libs/foo.jar', 'libs/bar.jar')**
  - the JAR files already exist
- **Remote binary dependency**
- **implementation 'com.example.android:app-magic:12.3'**

# Adding a native (C/C++) library

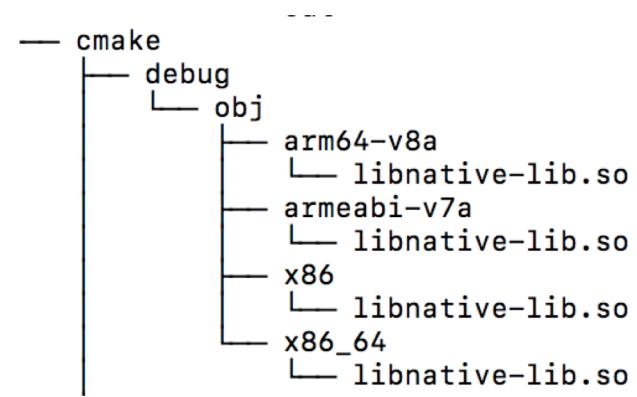
- A module can be a shared or static library (.so, .a)
- It can be generated during the build process from the source code via **NDK**
- or be prebuilt (only the .so is available)

# Build process of native code (CMake)

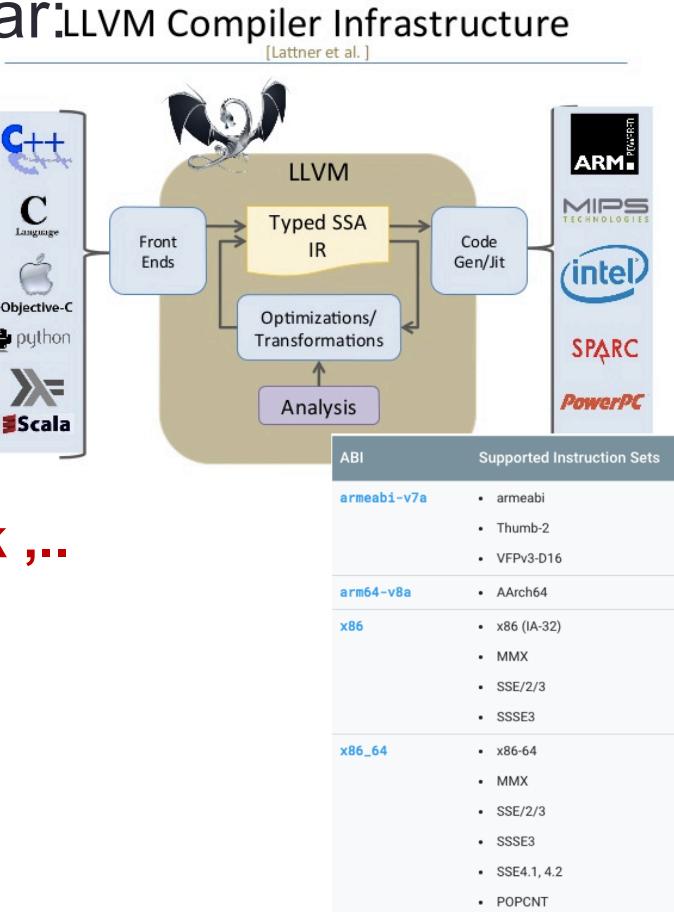
- The recommended way to control and configure how to build from C/C++ is by **CMake**
- The CMake script **CMakeLists.txt**, describes how importing and linking against prebuilt or platform libraries
- It is referred from the **build.gradle** file
- different builds addressing the different possible ABI of smartphones (cross compilation)

DSL:

```
externalNativeBuild {  
    cmake {  
        path "src/main/cpp/CMakeLists.txt"  
        version "3.10.2"  
    }  
}
```



# What is NDK?

- Native Development Kit (NDK) is a set of tools that allow to embed native code into Android applications
- It exploits the **LLVM** project, and in particular:  


The diagram illustrates the LLVM Compiler Infrastructure. On the left, a vertical stack of icons represents supported languages: C++, C, Objective-C, Python, and Scala. Arrows point from these languages to 'Front Ends'. These front ends feed into the central 'LLVM' block, which contains 'Typed SSA IR' and 'Optimizations/Transformations'. Below the LLVM block is an 'Analysis' module. An arrow points from 'Optimizations/Transformations' to 'Code Gen/Jit'. From 'Code Gen/Jit', arrows point to various back end blocks: 'ARM', 'MIPS TECHNOLOGIES', 'intel', 'SPARC', and 'PowerPC'. A legend at the bottom right lists ABIs and supported instruction sets for each architecture.

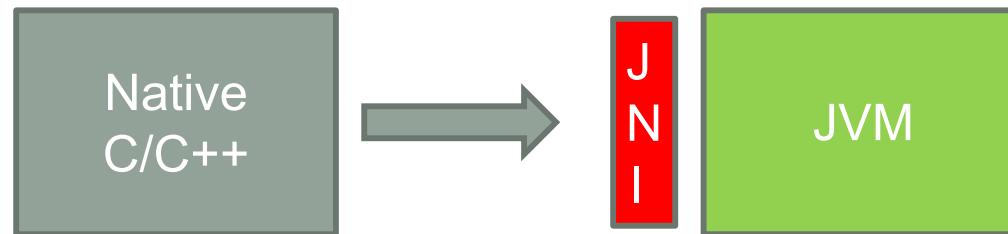
ABI	Supported Instruction Sets
armeabi-v7a	<ul style="list-style-type: none"><li>armeabi</li><li>Thumb-2</li><li>VFPv3-D16</li></ul>
arm64-v8a	<ul style="list-style-type: none"><li>AArch64</li></ul>
x86	<ul style="list-style-type: none"><li>x86 (IA-32)</li><li>MMX</li><li>SSE/2/3</li><li>SSSE3</li></ul>
x86_64	<ul style="list-style-type: none"><li>x86-64</li><li>MMX</li><li>SSE/2/3</li><li>SSSE3</li><li>SSE4.1, 4.2</li><li>POPCNT</li></ul>
- **CLANG** C/C++ cross compiler
  - 32-bit ARM, AArch64, x86, and x86-64.
  - C library: **bionic**
  - C++ library: **LLVM libc++**
  - NDK libraries: **openGL, Vulkan, Neural Network ,...**
- **CMake** as build tool

# Two approaches

- Application written in Java/Kotlin with (few) methods written in C/C++ are accessed via the **Java Native Interface**
- **Native Activity**: Entire activities are implemented in C/C++

# JNI

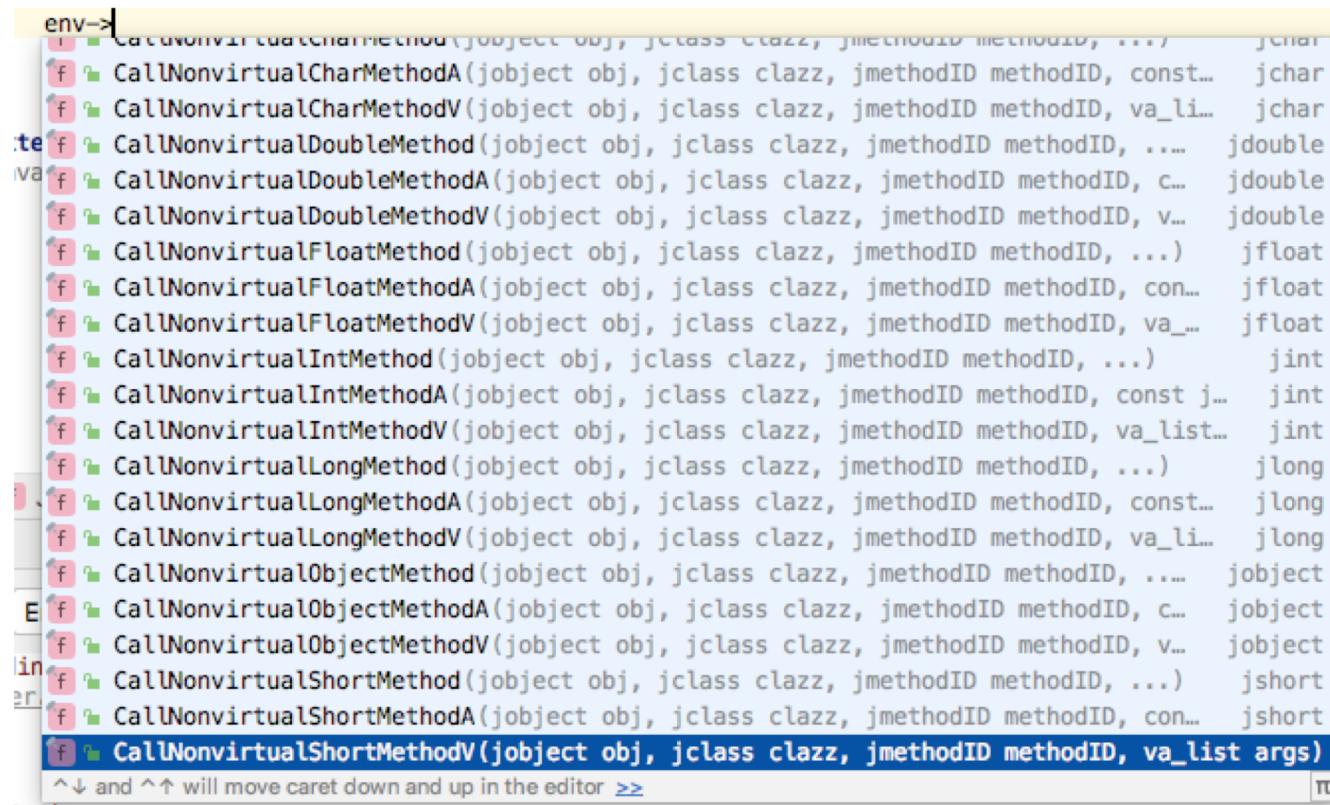
- A set of functions that allow to access the JVM from **native** code to:
- **Create, inspect, and update** Java objects (including arrays and strings)
- **Call Java methods**
- Catch and throw exceptions
- Load classes and obtain class information
- Perform runtime type checking



**Native code accesses Java VM features by calling JNI functions.**  
In C/C++ they are defined in the <jni.h> header file

# JNI

- A native method always receives two arguments
- a pointer to the JNI interface, **JNIEnv \*env**
- a pointer to a java object bounded to the function



```
env->
  CallNonvirtualCharMethod(jobject obj, jclass clazz, jmethodID methodID, ... ) jchar
  f  CallNonvirtualCharMethodA(jobject obj, jclass clazz, jmethodID methodID, const... jchar
  f  CallNonvirtualCharMethodV(jobject obj, jclass clazz, jmethodID methodID, va_li... jchar
  te f  CallNonvirtualDoubleMethod(jobject obj, jclass clazz, jmethodID methodID, ...) jdouble
  va f  CallNonvirtualDoubleMethodA(jobject obj, jclass clazz, jmethodID methodID, c... jdouble
  f  CallNonvirtualDoubleMethodV(jobject obj, jclass clazz, jmethodID methodID, v... jdouble
  f  CallNonvirtualFloatMethod(jobject obj, jclass clazz, jmethodID methodID, ...) jfloat
  f  CallNonvirtualFloatMethodA(jobject obj, jclass clazz, jmethodID methodID, con... jfloat
  f  CallNonvirtualFloatMethodV(jobject obj, jclass clazz, jmethodID methodID, va... jfloat
  f  CallNonvirtualIntMethod(jobject obj, jclass clazz, jmethodID methodID, ...) jint
  f  CallNonvirtualIntMethodA(jobject obj, jclass clazz, jmethodID methodID, const j... jint
  f  CallNonvirtualIntMethodV(jobject obj, jclass clazz, jmethodID methodID, va_li... jint
  f  CallNonvirtualLongMethod(jobject obj, jclass clazz, jmethodID methodID, ...) jlong
  f  CallNonvirtualLongMethodA(jobject obj, jclass clazz, jmethodID methodID, const... jlong
  f  CallNonvirtualLongMethodV(jobject obj, jclass clazz, jmethodID methodID, va_li... jlong
  f  CallNonvirtualObjectMethod(jobject obj, jclass clazz, jmethodID methodID, ...) jobject
  E  f  CallNonvirtualObjectMethodA(jobject obj, jclass clazz, jmethodID methodID, c... jobject
  f  CallNonvirtualObjectMethodV(jobject obj, jclass clazz, jmethodID methodID, v... jobject
  f  CallNonvirtualShortMethod(jobject obj, jclass clazz, jmethodID methodID, ...) jshort
  f  CallNonvirtualShortMethodA(jobject obj, jclass clazz, jmethodID methodID, con... jshort
  f  CallNonvirtualShortMethodV(jobject obj, jclass clazz, jmethodID methodID, va_li... jshort
```

^↓ and ^↑ will move caret down and up in the editor >>

# JNI [data manipulation]

- A key part of JNI is a support to translate **primitive** native data types to java primitive data types

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

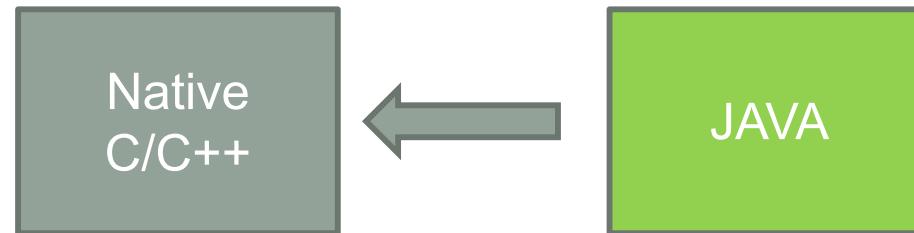
# JNI [data manipulation example]

- env->NewStringUTF(«hello»);

```
env->New_
f  NewBooleanArray(jsize length) jbooleanArray
f  NewByteArray(jsize length) jbyteArray
f  NewCharArray(jsize length) jcharArray
f  NewDirectByteBuffer(void *address, jlong capacity) jobject
f  NewDoubleArray(jsize length) jdoubleArray
f  NewFloatArray(jsize length) jfloatArray
f  NewGlobalRef(jobject obj) jobject
f  NewIntArray(jsize length) jintArray
f  NewLocalRef(jobject ref) jobject
f  NewLongArray(jsize length) jlongArray
f  NewObject(jclass clazz, jmethodID methodID, ...) jobject
f  NewObjectA(jclass clazz, jmethodID methodID, const jvalue *args) jobject
f  NewObjectArray(jsize length, jclass elementClass, jobject initialElemen... jobjectArray
f  NewObjectV(jclass clazz, jmethodID methodID, va_list args) jobject
f  NewShortArray(jsize length) jshortArray
f  NewString(const jchar *unicodeChars, jsize len) jstring
f  NewStringUTF(const char *bytes) jstring
f  NewWeakGlobalRef(jobject obj) jweak
f  ThrowNew(jclass clazz, const char *message) jint
^↓ and ^↑ will move caret down and up in the editor >>
```

# JNI

- JNI also allows to call native method from java code (or Kotlin)
- Java code sees **external** symbols (functions) defined in a shared library (result of the native code compilation)
  - Recall that a .so is an *ELF* file with code + meta-info that makes code in the file accessible



# JNI

- Java/Kotlin loads the **shared library** at **run-time**:
  - `System.loadLibrary("native-lib")`
- Extern functions are retrieved from the library

```
external fun stringFromJNI(): String
companion object {
    // Used to load the 'native-lib' library on application startup.
    init {
        System.loadLibrary( libname: "native-lib" )
    }
}
```

# Name convention

- The name of a native function is the concatenation of
  - the prefix `Java_`
  - the mangled fully-qualified class name
  - the **method name**

```
Java_com_example_macc_nativefirst_MainActivity_stringFromJNI(JNIEnv*  
env,jobject /* this */){  
  
    return env->NewStringUTF("hello");  
  
}
```

# Example 1: understanding the template

#include <jni.h>      This function is exported  
#include <string>      returned type      body of the function

extern "C" JNIEXPORT jstring JNICALL

Java\_com\_example\_macc\_nativefirst\_MainActivity\_stringFromJNI(  
JNIEnv\* env,  
jobject /\* this \*/ {

std::string hello = "Hello from C++";  
return env->NewStringUTF(hello.c\_str());

}

## Example 2: adding hidden native code

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

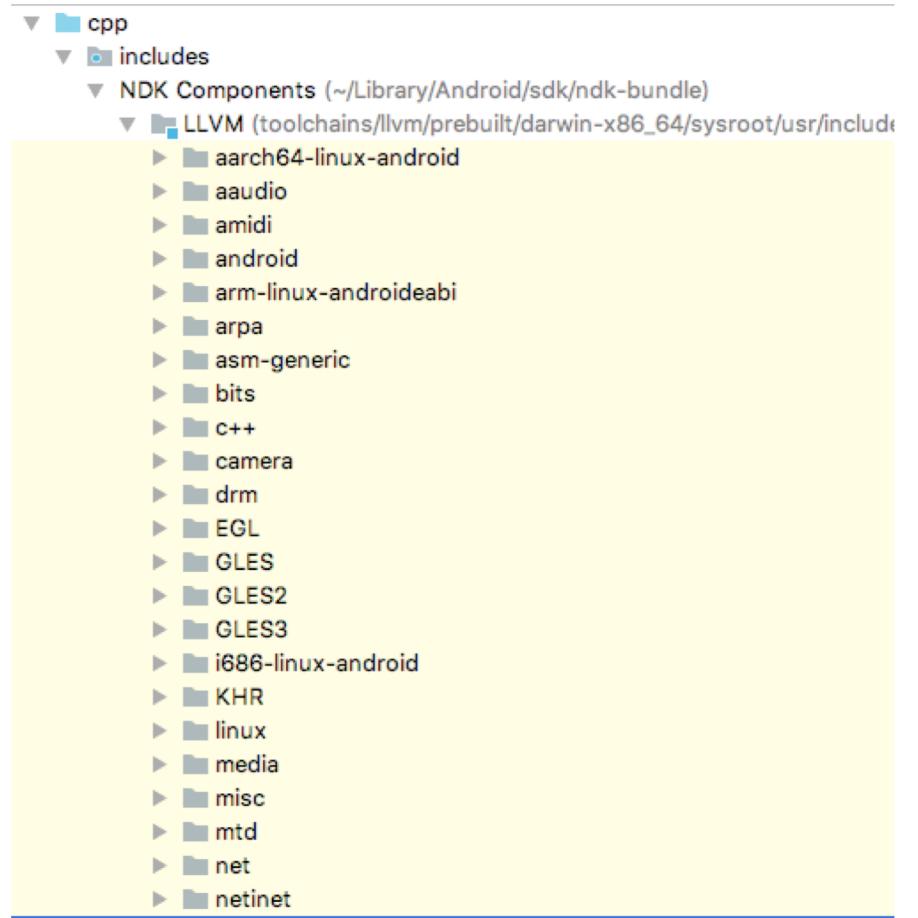
int hidden_worker( ) {
    return 2;
}

extern "C" JNIEXPORT jstring JNICALL
Java_com_example_macc_nativefirst_MainActivity_stringFromJNI(
    JNIEnv* env,
    jobject /* this */ {

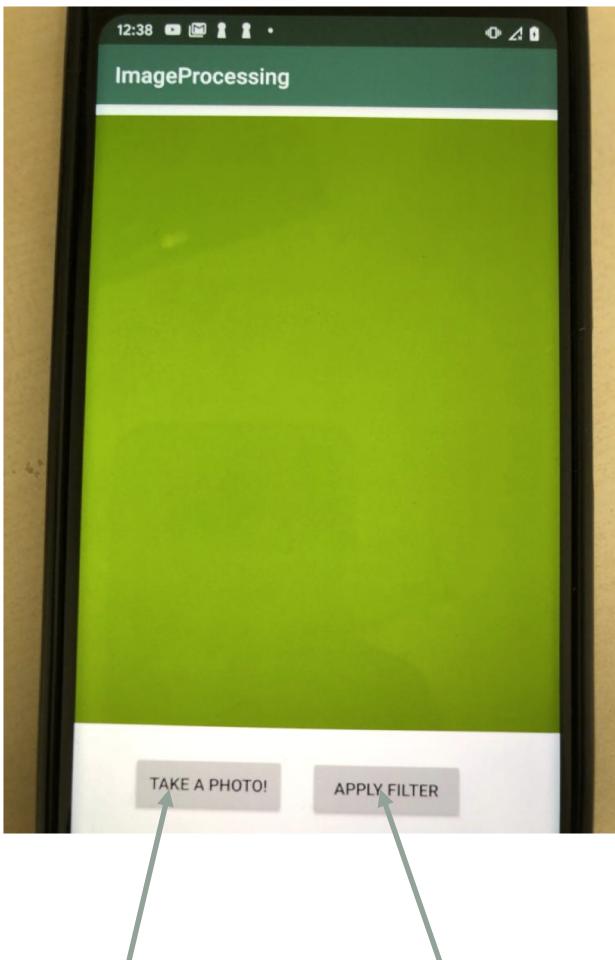
    int a = hidden_worker();
    std::string res = std::to_string(a);
    return env->NewStringUTF(res.c_str());
}
```

# Example 3: Android NDK Native APIs

- Use a prebuilt shared library without the source code
- NDK has many useful libraries



# An example: openCV



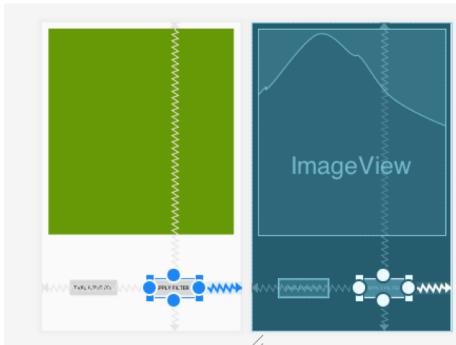
Take of a photo

Apply a filter (use OpenCV)

- Step 1:
  - Use cam to take a photo
  - Set the taken photo as ImageView
- Permission to use the camera
- Intent for the camera
- Getting back the result
- Step 2:
  - Install OpenCV
  - Apply a blur filter to the image

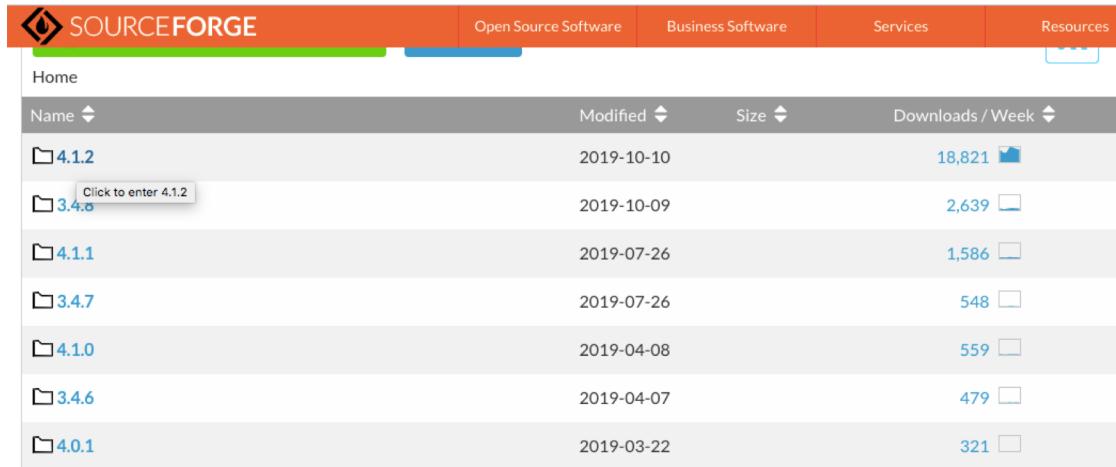
# Step 1

```
class MainActivity : AppCompatActivity() {  
  
    val MY_CAMERA_REQUEST_CODE = 1  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
  
        if (checkSelfPermission(Manifest.permission.CAMERA) != PackageManager.PERMISSION_GRANTED)  
            ActivityCompat.requestPermissions(this, arrayOf(Manifest.permission.CAMERA), MY_CAMERA_REQUEST_CODE)  
    }  
  
    fun takePhoto(v : View) {  
        val takePicture = Intent(MediaStore.ACTION_IMAGE_CAPTURE)  
        startActivityForResult(takePicture, MY_CAMERA_REQUEST_CODE)  
    }  
  
    fun applyFilter(v : View) {  
        if ((v as Button).text=="Apply Filter") {  
            v.setBackgroundColor(Color.BLUE)  
            (v as Button).text="Remove Filter"  
            return  
        }  
        (v as Button).text="Apply Filter"  
        v.setBackgroundColor(Color.GRAY)  
    }  
  
    override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {  
        super.onActivityResult(requestCode, resultCode, data)  
        if (requestCode == MY_CAMERA_REQUEST_CODE && resultCode == RESULT_OK) {  
            val extra = data?.extras  
            val bitmap :Bitmap = extra?.get("data") as Bitmap  
            imageView.setImageBitmap(bitmap)  
        }  
    }  
}
```



# Step 2: 1/4 [download OpenCV sdk]

- Download and unzip OpenCV
- <https://sourceforge.net/projectsopencvlibrary/files/4.1.2/>



A screenshot of the SourceForge website showing a list of OpenCV versions. The page has a header with tabs for Home, Open Source Software, Business Software, Services, and Resources. The main content area shows a table with columns for Name, Modified, Size, and Downloads / Week. The table lists the following versions:

Name	Modified	Size	Downloads / Week
4.1.2	2019-10-10		18,821
3.4.5	2019-10-09		2,639
4.1.1	2019-07-26		1,586
3.4.7	2019-07-26		548
4.1.0	2019-04-08		559
3.4.6	2019-04-07		479
4.0.1	2019-03-22		321

opencv-4.1.2-android-sdk.zip

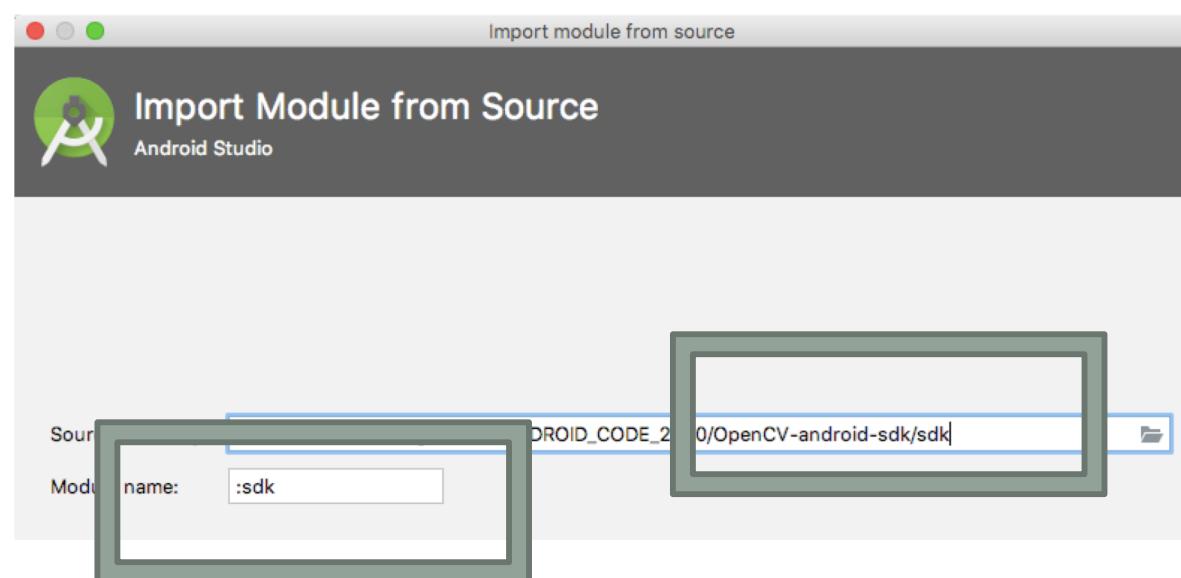
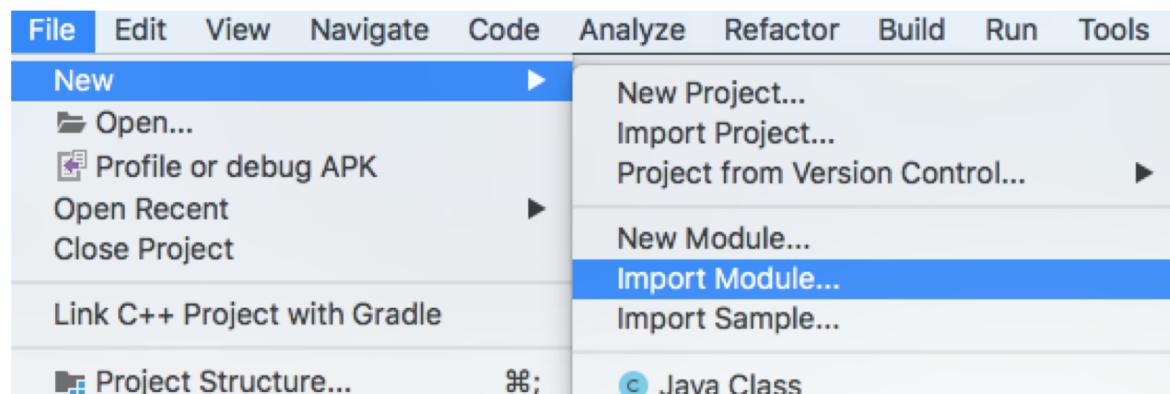
2019-10-10

228.5 MB

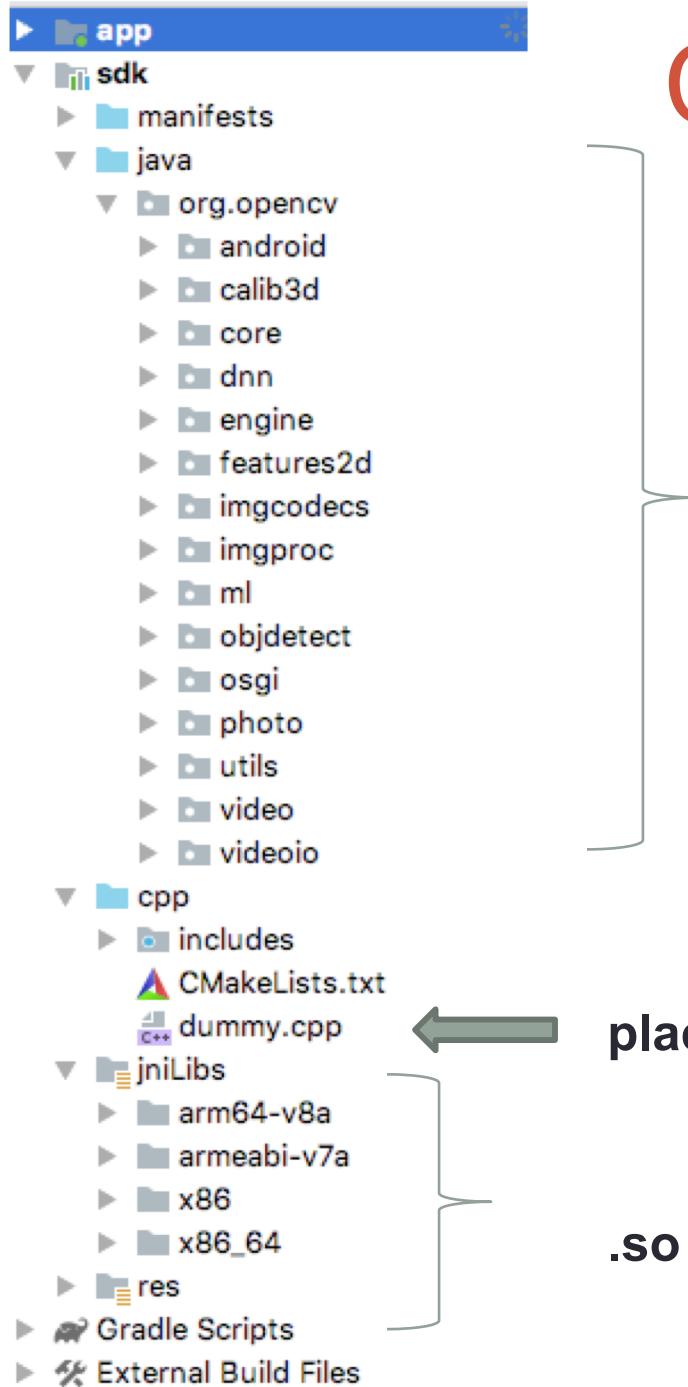
1,936



# Step 2: 2/4 [adding OpenCV as a module]



# Check result of the build



## java packages

- Java to native method mapping
- Additional methods (e.g. load library, etc)

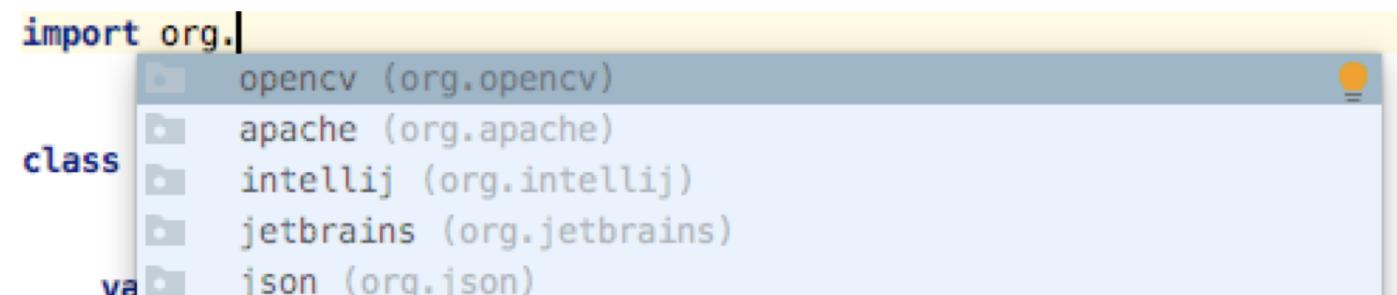
placeholder for additional native code

.so library compiled for 4 ABI

## Step 2: 3/4 [change build.gradle of app]

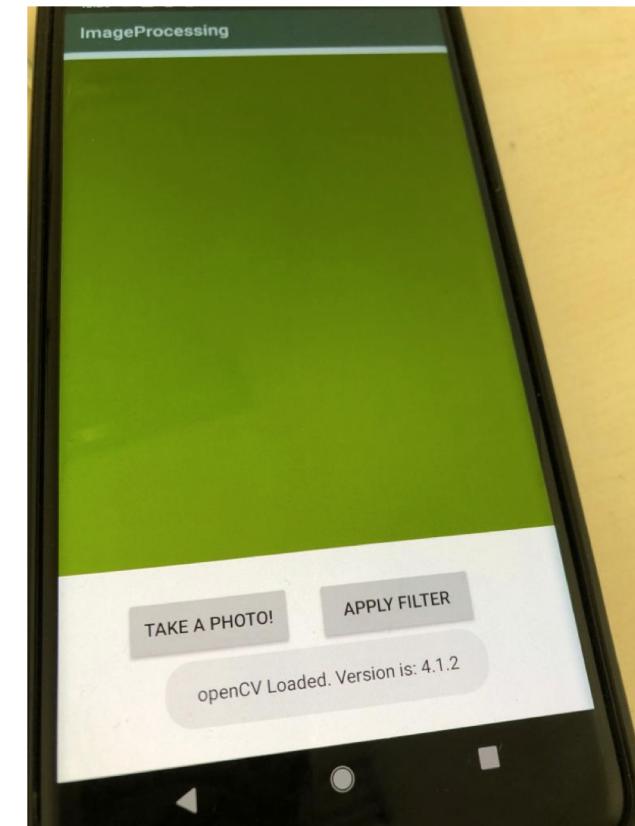


```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation project(':sdk')
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    implementation 'androidx.appcompat:appcompat:1.1.0'
```



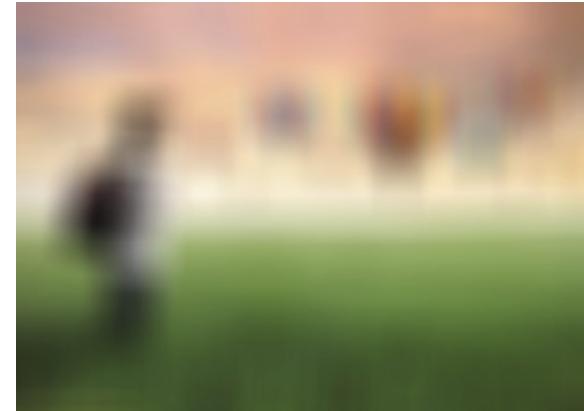
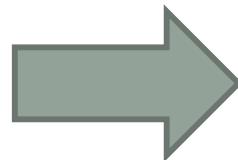
## Step 2: 4/4 [load library]

```
if (OpenCVLoader.initDebug()){\n//OK... print OpenCVLoader.OPENCV_VERSION\n}
```



# Step 2: applying the blur filter

```
fun applyFilter(v : View) {  
    var src :Mat = Mat()  
    var dst :Mat = Mat()  
    Utils.bitmapToMat(photo,src)  
    val size = Size( width: 45.0, height: 45.0)  
    val point = Point( x: 20.0, y: 30.0)  
    Imgproc.blur(src, dst, size, point, Core.BORDER_DEFAULT)  
    Utils.matToBitmap(dst,photo)  
}
```



# Get the image to process

- Directly from the camera using an Intent, however this returns only a *thumbnail*
- In order to get a full image, one need to store the photo into a temporary file (using a file depends on the API level, keep changing... ☺ )
- Get the photo from the gallery is much simpler
- Take the photo than the easiest way is to use the **Picasso** library