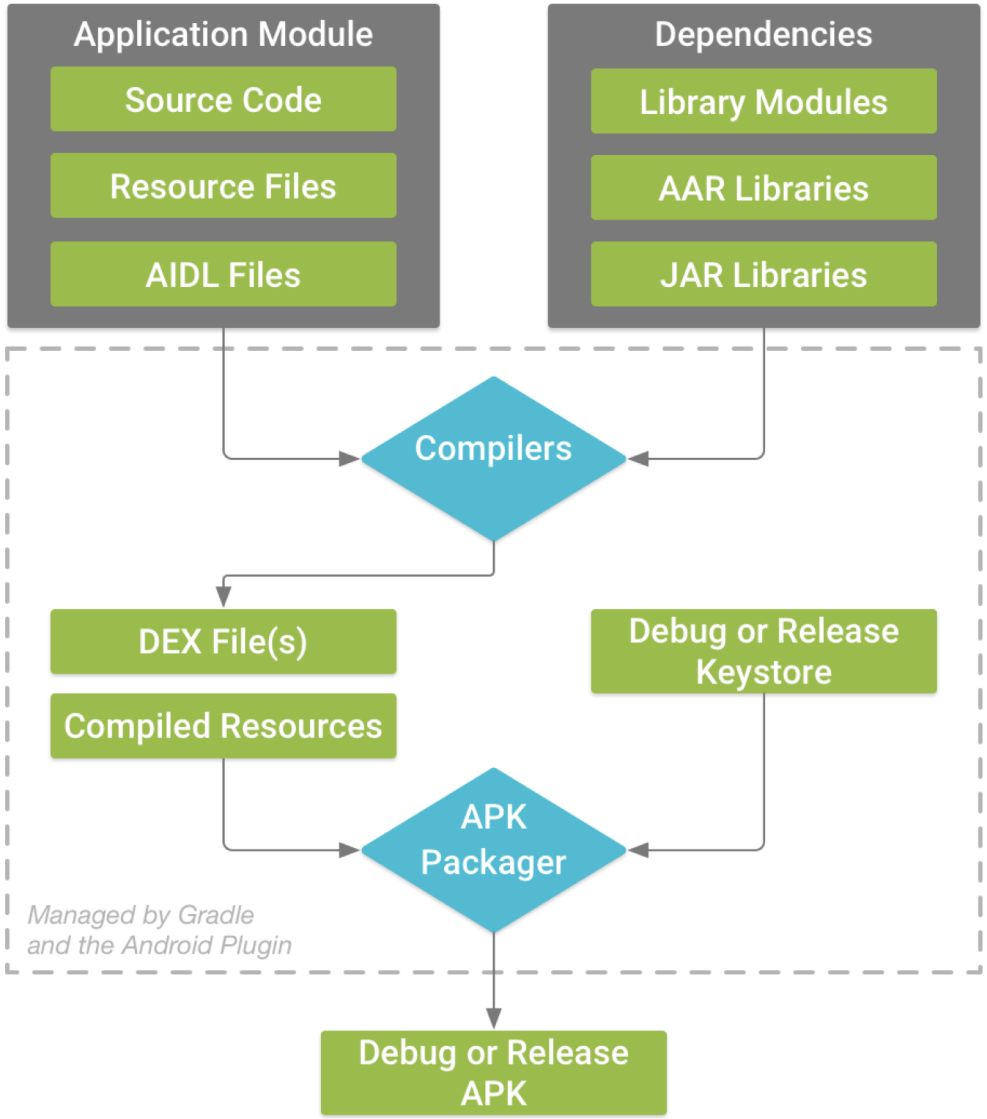
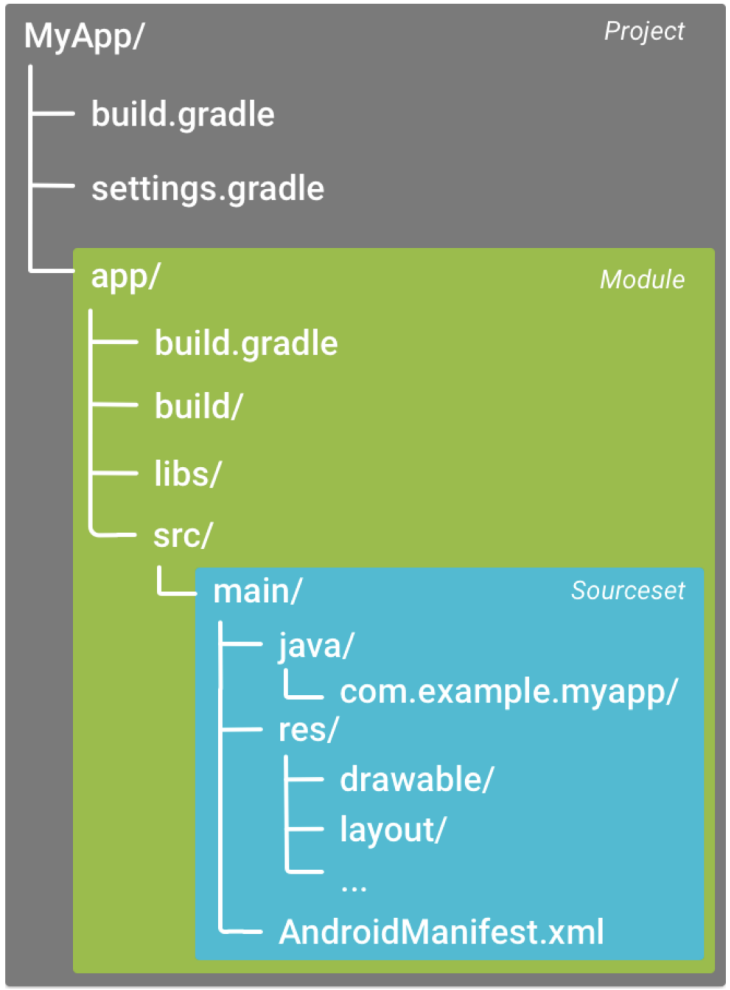
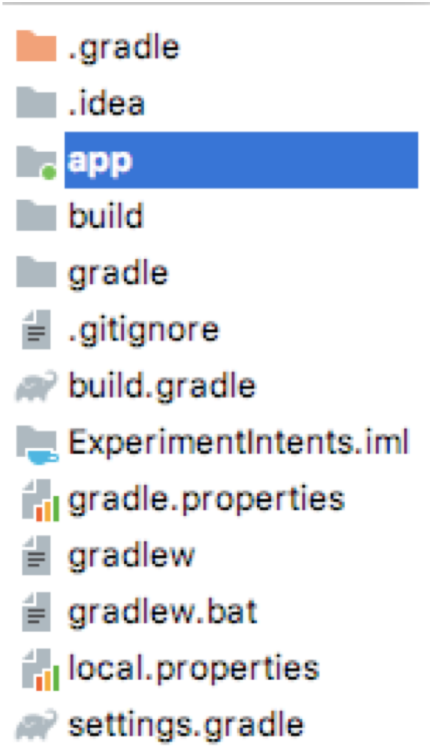
NATIVE LIBRARIES AND NDK

The build process



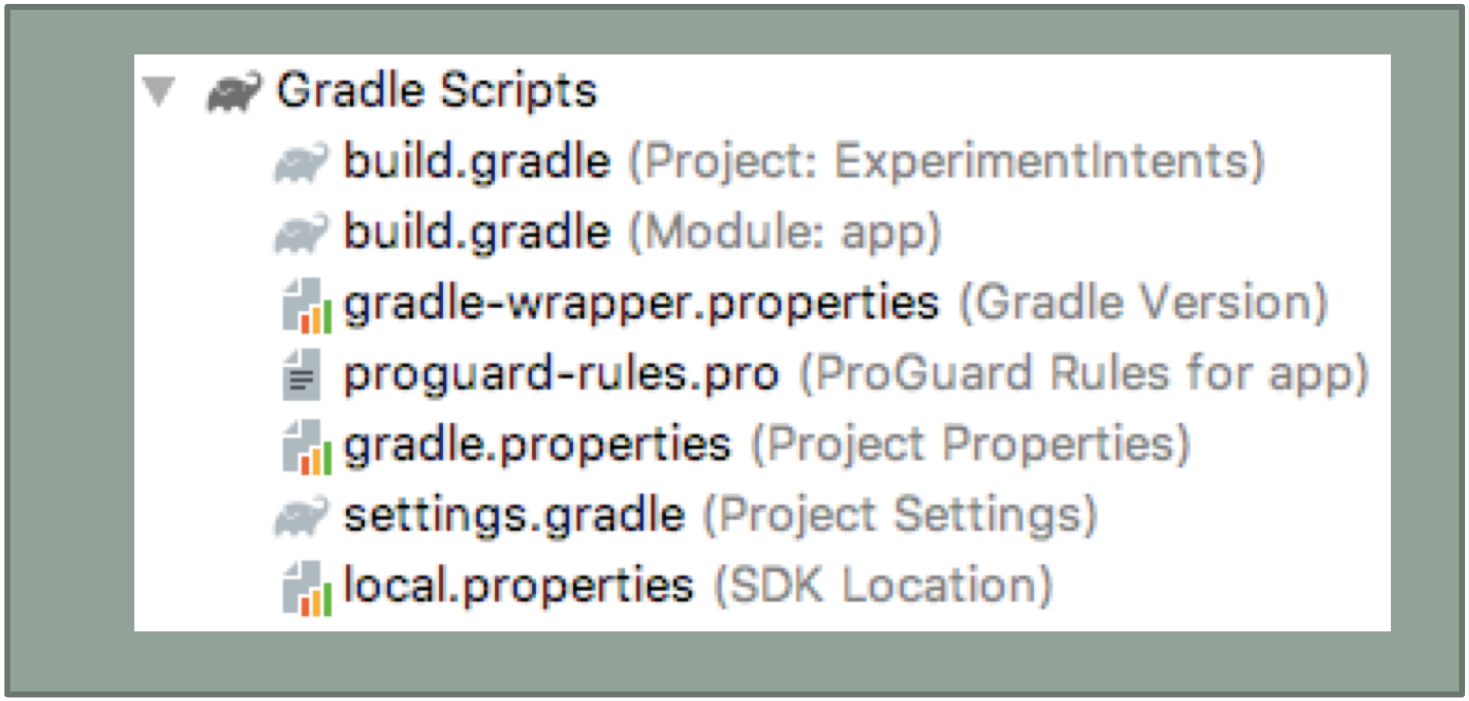
Default Project structure





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**

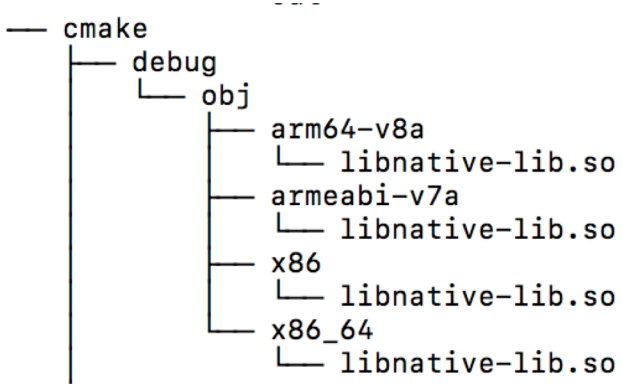
- or be prebuilt (only the .so is available)

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)

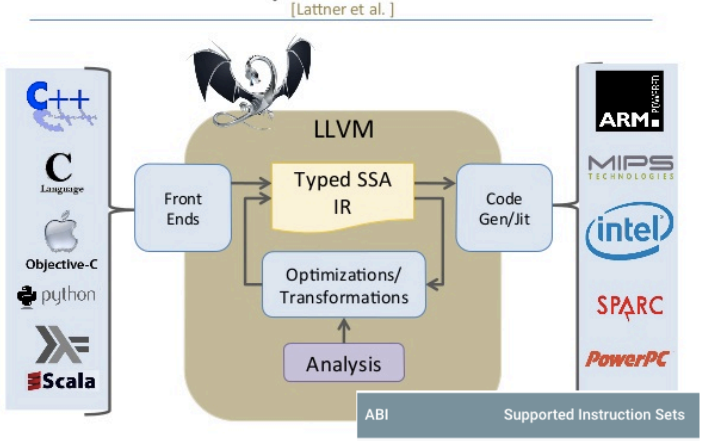
DSL:

```
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:



- **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 ,..**

ABI	Supported Instruction Sets
armeabi-v7a	armeabi Thumb-2 VFPv3-D16
arm64-v8a	AArch64
x86	x86 (IA-32) MMX SSE/2/3 SSSE3
x86_64	x86-64 MMX SSE/2/3 SSSE3 SSE4.1, 4.2 POPCNT

- **CMake** as build tool

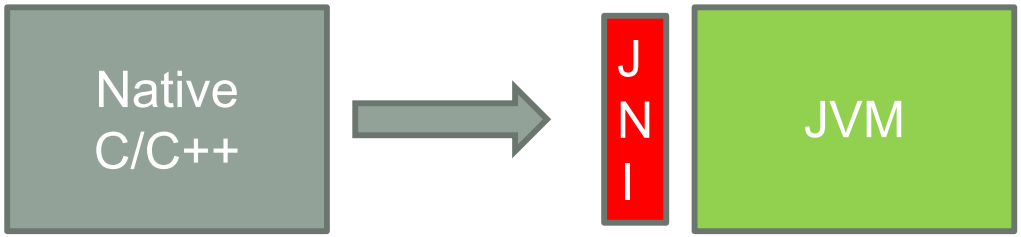
Two approaches

- 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

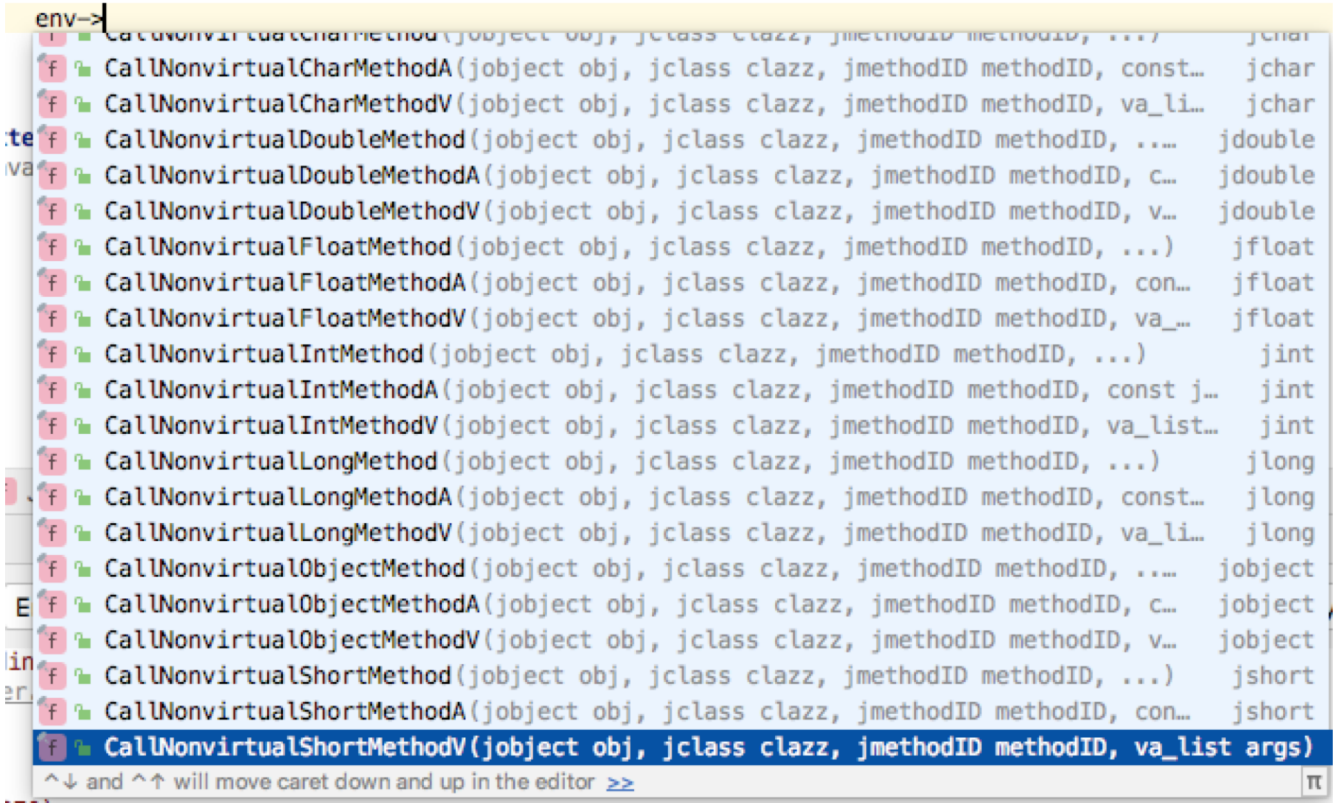
- A set of functions that allow to access the JVM from **native** code to:
- **Create**, **inspect**, and **update** Java objects (including arrays and strings)
- **Call Java methods**
- Catch and throw exceptions
- Load classes and obtain class information
- Perform runtime type checking



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



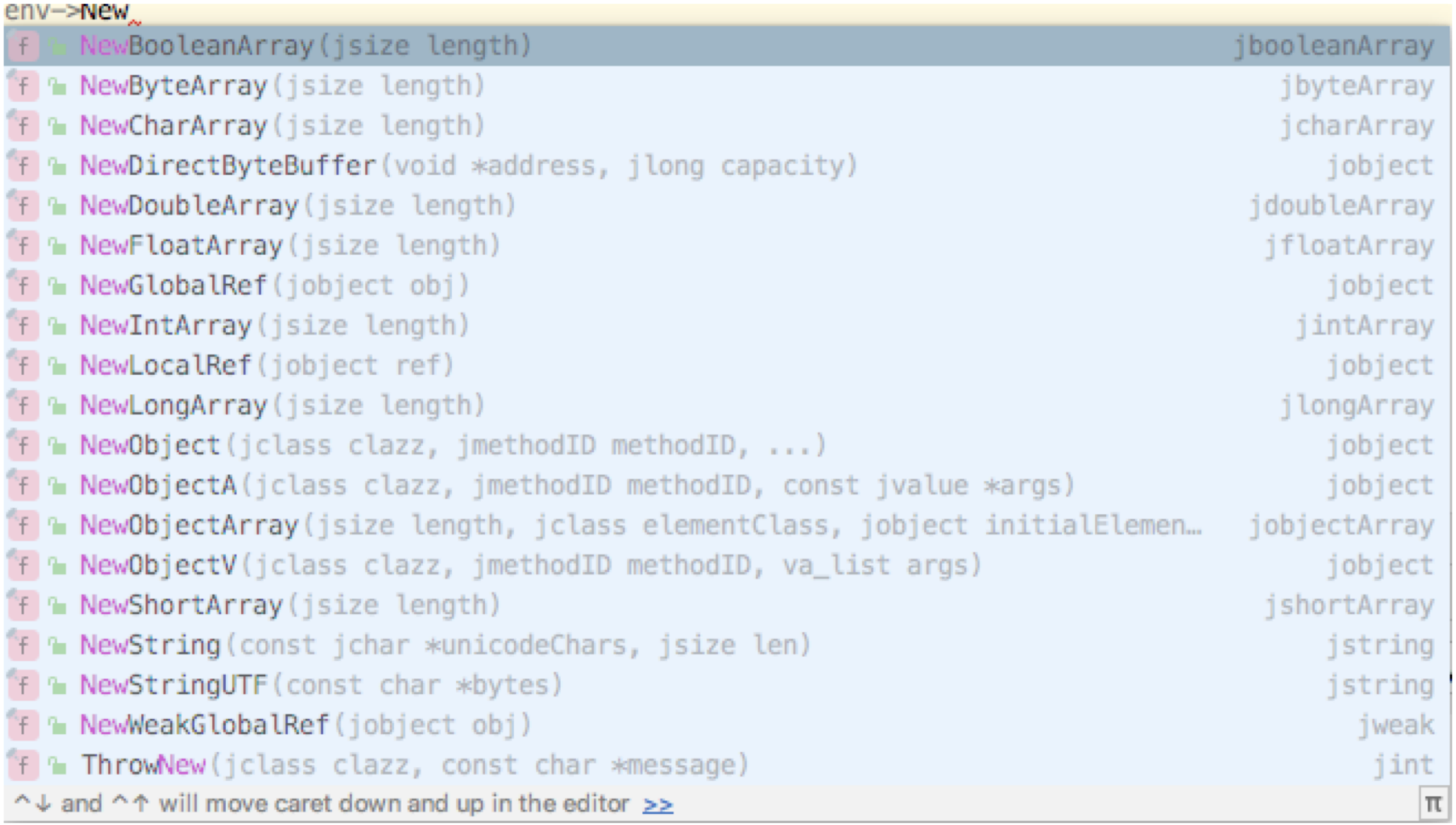
JNI [data manipulation]

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

Java Type	Native Type	Description
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

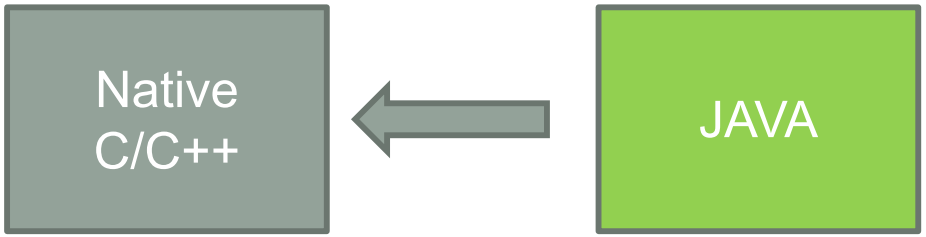
JNI [data manipulation example]

- env->NewStringUTF(«hello»);



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

This function is exported

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

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());

}
```

returned type

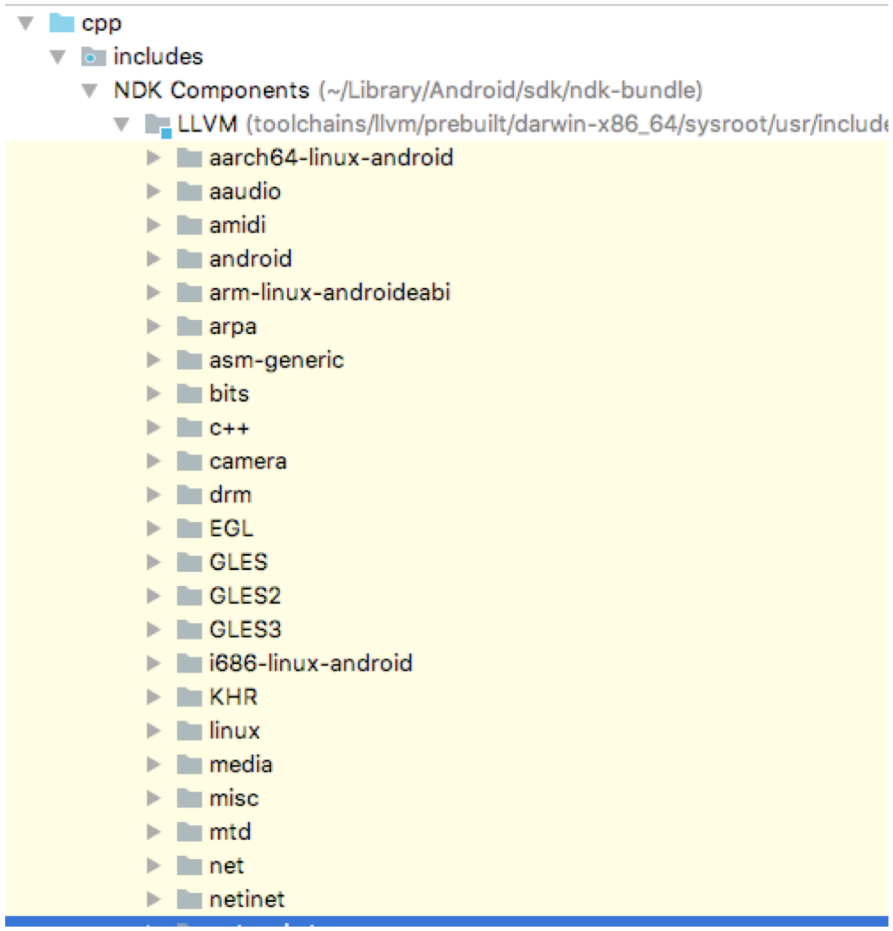
body of the function

Example 2: adding hidden native code

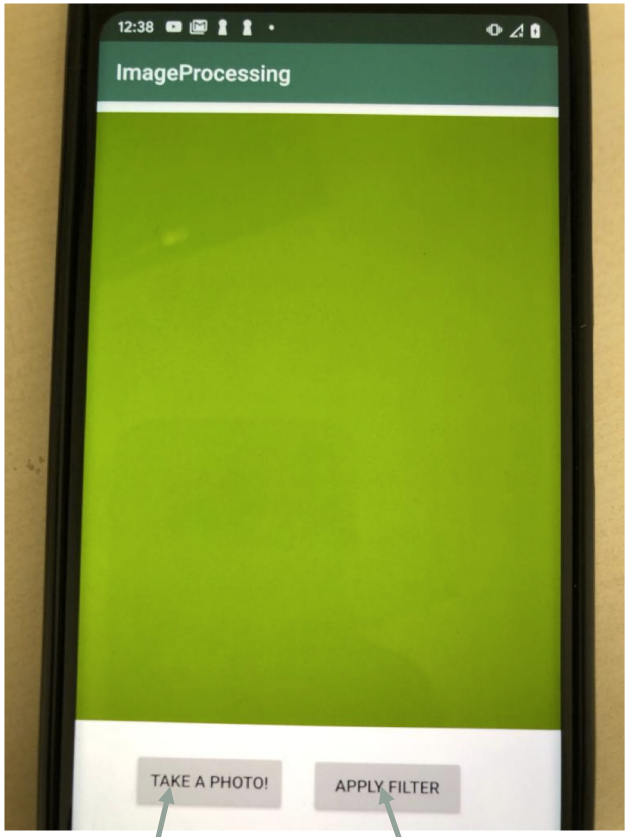
```
#include <jni.h>
#include <string>

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



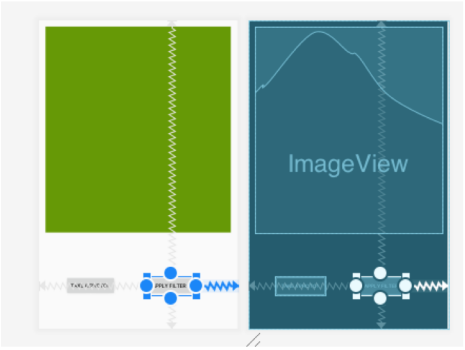
- Step 1:
- Use cam to take a photo
- Set the taken photo as ImageView

- Permission to use the camera
- Intent for the camera
- Getting back the result

- Step 2:
- Install OpenCV
- Apply a blur filter to the image

Take of a photo **Apply a filter (use OpenCV)**

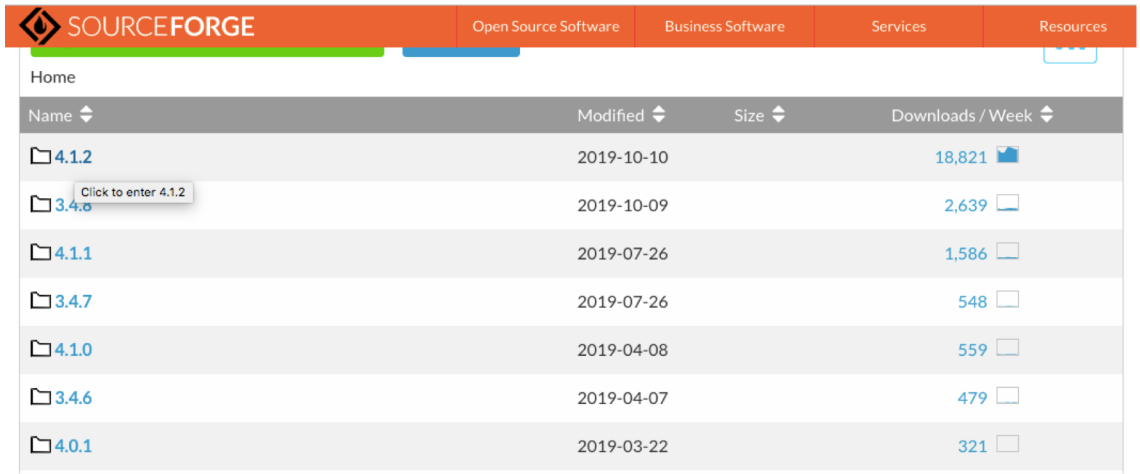
Step 1

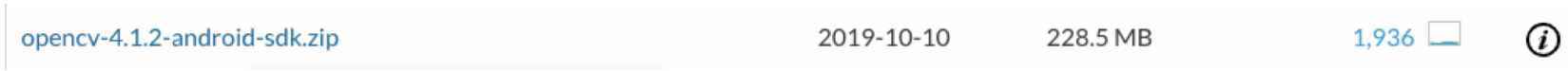



```
class MainActivity : AppCompatActivity() {


    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)
            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)
        }

}
```

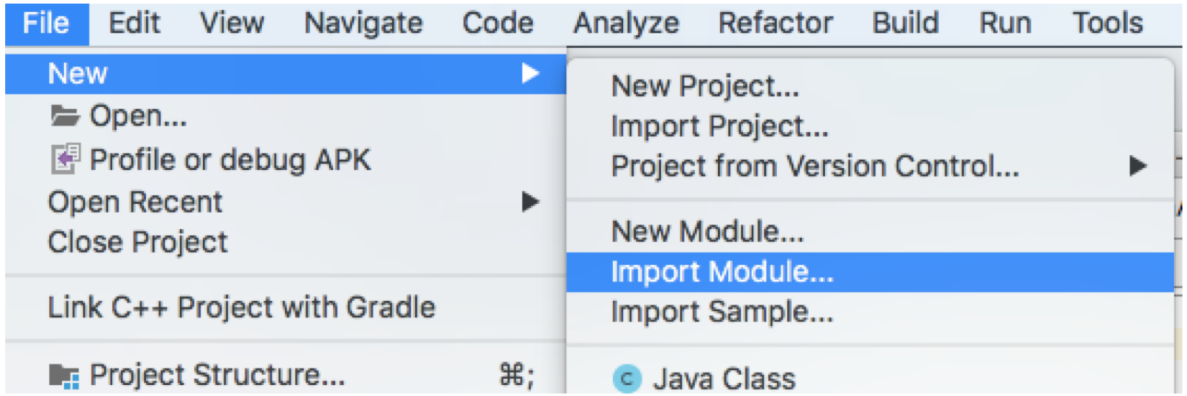
Step 2: 1/4 [download OpenCV sdk]

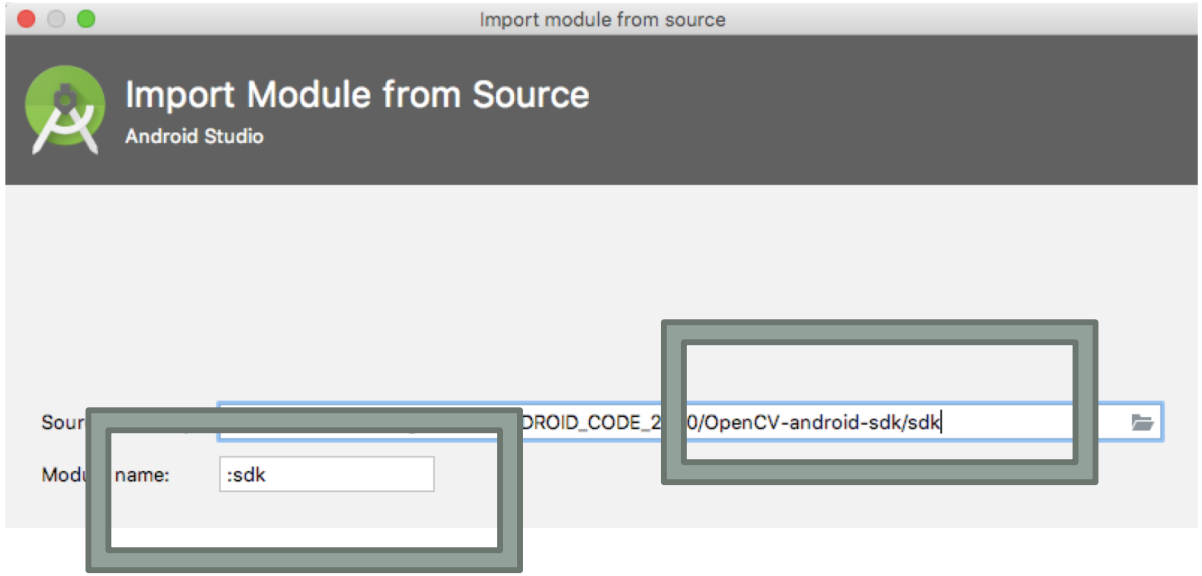
- Download and unzip OpenCV
- https://sourceforge.net/projects/opencvlibrary/files/4.1.2/



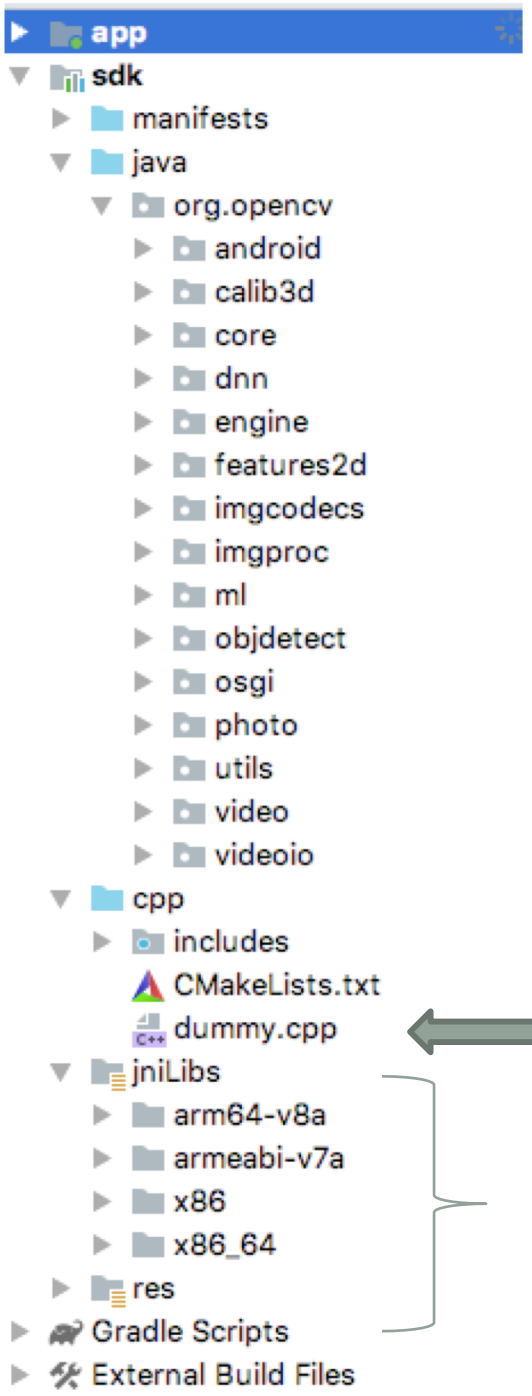


Step 2: 2/4 [adding OpenCV as a module]





Check result of the build



java packages

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

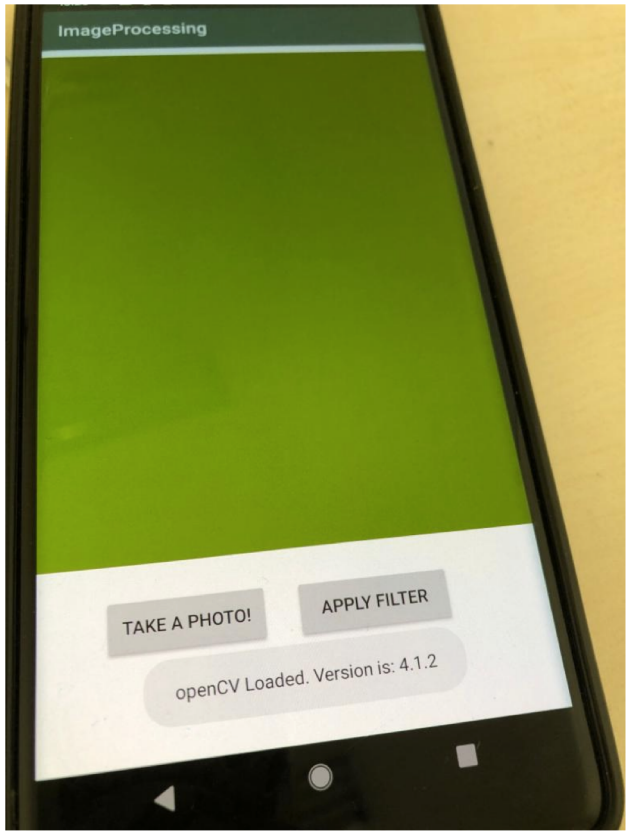
placeholder for additional native code

.so library compiled for 4 ABI

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

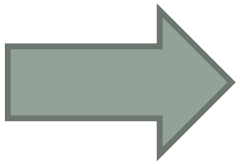
Step 2: 4/4 [load library]

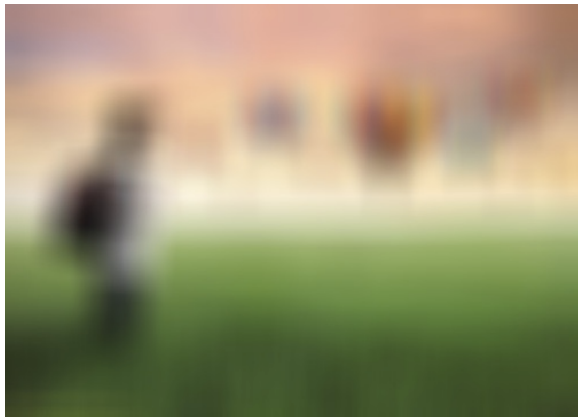
```
if (OpenCVLoader.initDebug()){
//OK… print OpenCVLoader.OPENCV_VERSION
 }
```



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