Introduction to Android Native Development Kit (NDK)

Çağatay Sönmez

14.12.2018

Arcelik Electronics Plant Division, R&D Team - Software Design

Agenda

- Android Native Development Kit
- Java Native Interface
- Creating New Project with C/C++ Support
- Adding C/C++ Sources to Android Project
- Adding Prebuilt Libraries to Android Project
- Declaring Native Methods
- Calling Native Function from Java
- Analyzing apk File to Check Native Libraries
- Running the Sample Application
- Debugging the Sample Application

Meterials

- Application source code can be found on GitHub
  - https://github.com/CagataySonmez/Android-for-Beginners/tree/master/4-IntroductionToAndroid-NDK
- Basic guide for Android Native Development Kit (NDK)
  - https://developer.android.com/ndk/guides/
- Basic guide for Java Native Interface (JNI)
  - https://docs.oracle.com/javase/8/docs/technotes/guides/jni/

Android Native Development Kit (NDK)

- **NDK** allows using C and C++ code with Android
- Java code can then call functions in your native library through the Java Native Interface (**JNI**) framework
- **CMake** is Android Studio's default build tool to compile native libraries
- **LLDB** is Android Studio's default debugger tool to debug native code
- NDK may not be appropriate for Android programmers who need to use only Java code and framework APIs

Why NDK?

- Squeeze extra performance out of a device to
  - achieve low latency
  - run computationally intensive applications
- Perform platform-dependent operations which are not handled with Java
- Reuse your own or other developers' C or C++ libraries
- Port apps between platforms (e.g. iOS, Android)

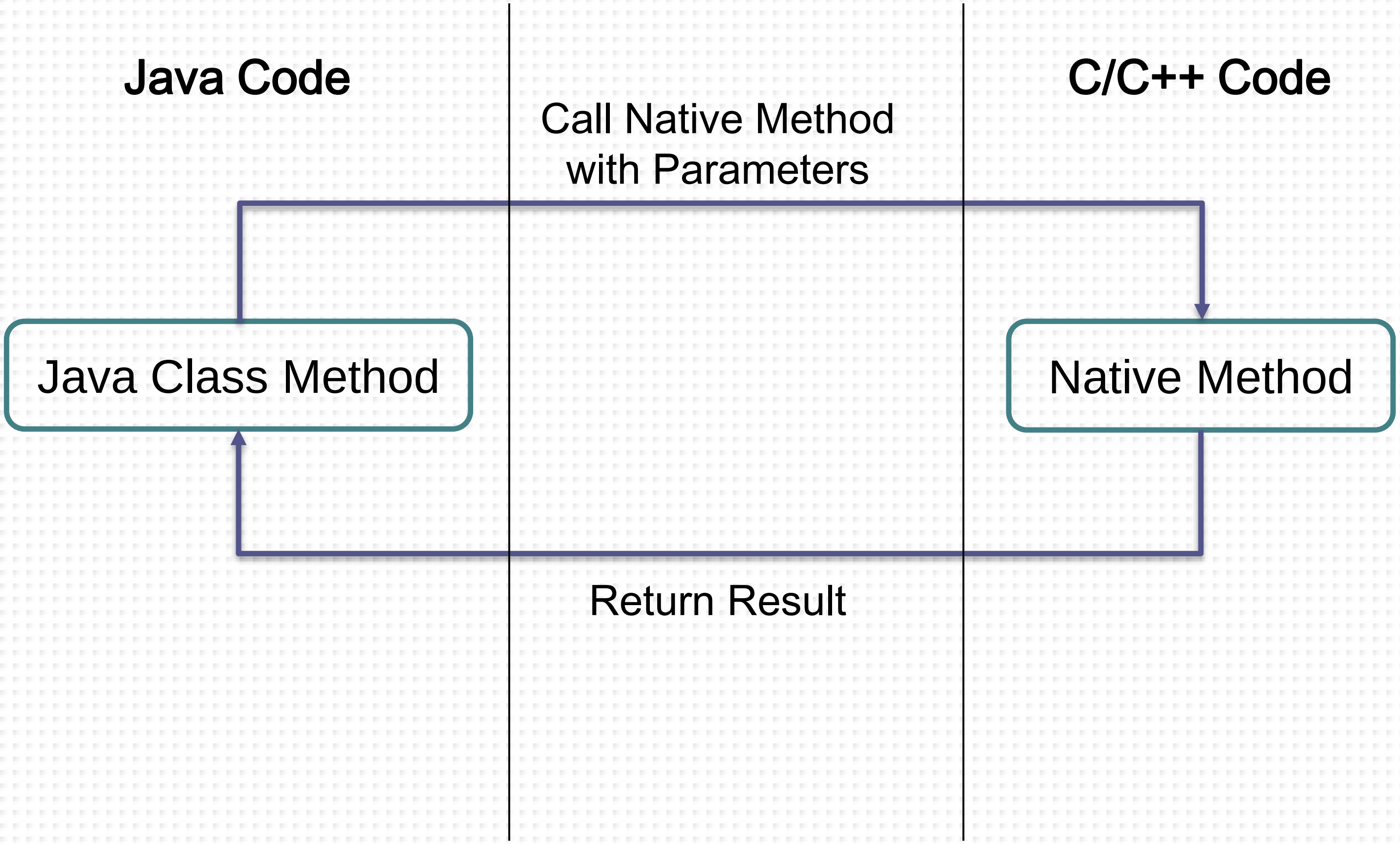
Java Native Interface (JNI)

- JNI is a native programming interface
- JNI enables Java code running in a JVM to interoperate with apps and libraries written in other languages such as C, C++
- JNI allows using native methods to
  - Create, inspect, and update Java objects
  - Call Java methods
  - Catch and throw exceptions
  - Perform runtime type checking

Example use cases of JNI

- The standard Java class library does not support the platform-dependent features needed by the application
- You already have a library written in another language, and wish to make it accessible to Java code through the JNI
- You want to implement a small portion of time-critical code in a lower-level language such as assembly

How JNI Works?



Java Code

C/C++ Code

Call Native Method with Parameters

Java Class Method

Native Method

Return Result

How JNI Works? Cont.

Java Code

```java
public class HelloWorld {

  native void MethodName(String arg);

  public static void main(String[] args)
  {
    MethodName("Hello World!");
  }
}
```

C/C++ Code

```c
#include <jni.h>

extern "C" JNIEXPORT void JNICALL
Java_ClassName_MethodName
(JNIEnv *env, jobject obj, jstring arg)
{
   //Do your work
}
```

JNI Types

- Primitive Types
  - jboolean, jbyte, jchar, jshort, jint, jlong, jfloat, jdouble, void
- Reference Types
  - The top of the hierarchy is jobject
  - Subclasses of jobject: jclass, jstring, jarray and jthrowable
  - Subclasses of jarray: jobjectArray, jbooleanArray, jbyteArray, jcharArray, jshortArray, jintArray, jlongArray, jfloatArray, jdoubleArray
- Check below link for more
  - https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/types.html

JNIEnv

- JNIEnv is one of the key data structures
- JNIEnv is a pointer to a structure storing all JNI function pointers
    - FindClass
    - IsInstanceOf
    - GetMethodID
    - CallObjectMethod
    - …
- Check below list for whole functions
    - https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html

Referencing Java Objects

- Primitive types are copied between Java and native code
- Arbitrary Java objects are passed by reference
- The VM must keep track of all objects that have been passed to the native code, so that these objects are not freed by the garbage collector
- The JNI divides object references used by the native code into two categories
  - local references
  - global references

Local References

- Every argument passed to a native method, and almost every object returned by a JNI function is a "local reference"
- Local references are valid for the duration of a native method call, and are automatically freed after the native method returns
- Even if the object itself continues to live on after the native method returns, the reference is not valid
- Local references are only valid in the thread in which they are created; the native code **must not pass** local references from one thread to another

Global References

- If you want to hold on to a reference for a longer period, you must use a "global" reference
- Global references remain valid until they are explicitly freed
- The **NewGlobalRef** function takes the local reference as an argument and returns a global one
- The global reference is guaranteed to be valid until you call **DeleteGlobalRef**

General JNI Tips

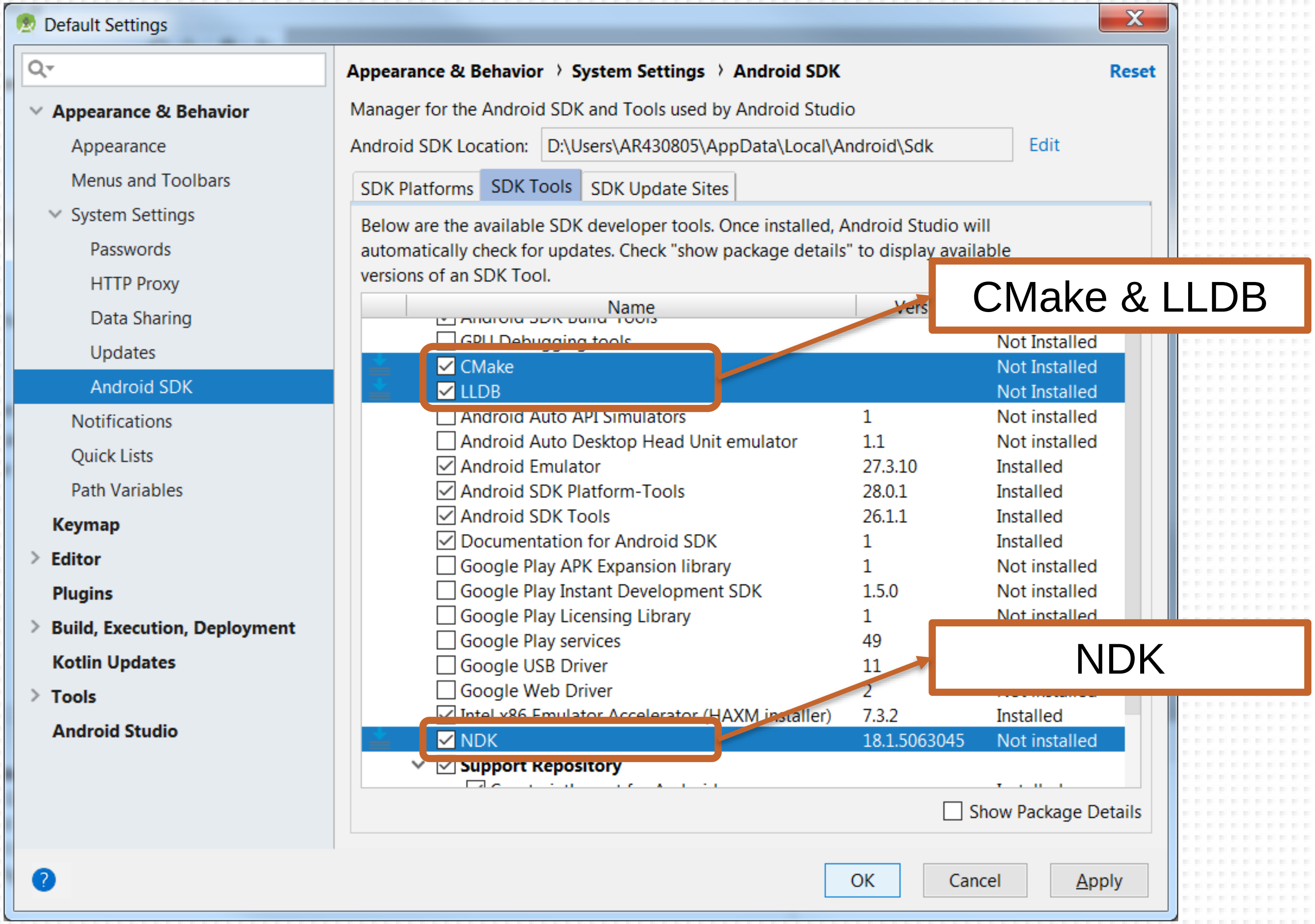
- Minimize marshalling of resources across the JNI layer
  - Marshalling across the JNI layer has non-trivial costs
- Avoid asynchronous communication between Java code and code native code
  - This will keep your JNI interface easier to maintain
- Minimize the number of threads that need to touch or be touched by JNI
- Keep your interface code in a low number of easily identified C++ and Java source locations to facilitate future refactors

Creating New Project with C/C++ Support

Create New Project with C/C++ Support

1. Download required SDK tools
2. Check ‘Include C++ Support’ checkbox in the new Project wizard
3. Customize project in ‘Customize C++ Support’ section
   - C++ Standard
   - Exceptions Support
   - Runtime Type Information (RTTI) Support

1- Dowload required SDK tools



Default Settings

Appearance & Behavior › System Settings › Android SDK

Manager for the Android SDK and Tools used by Android Studio

Android SDK Location: D:\Users\AR430805\AppData\Local\Android\Sdk

Below are the available SDK developer tools. Once installed, Android Studio will automatically check for updates. Check "show package details" to display available versions of an SDK Tool.

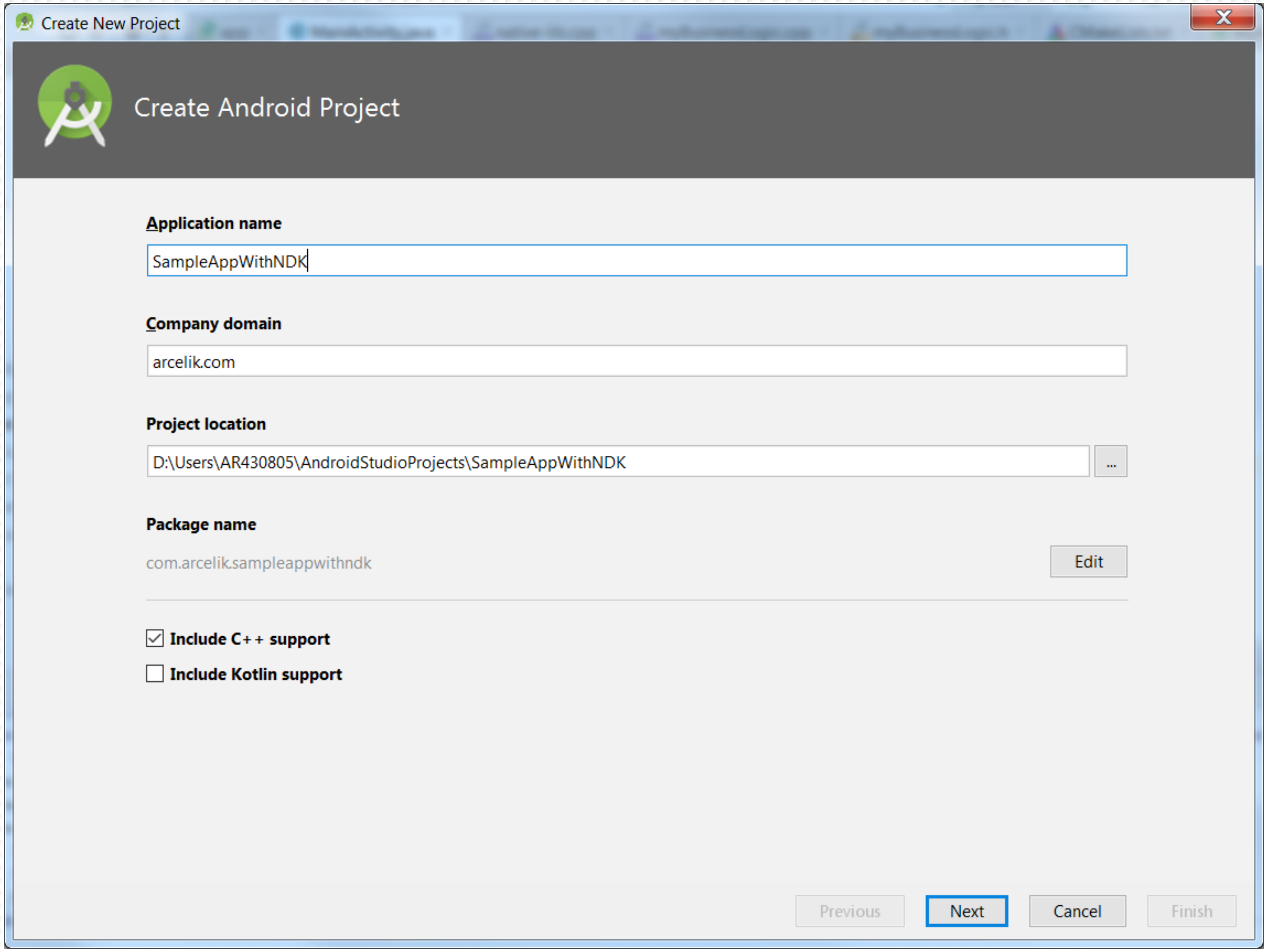
Name	Vers	
☑ Android SDK Build-Tools		
□ GPU Debugging tools		Not Installed
☑ CMake		Not Installed
☑ LLDB		Not Installed
□ Android Auto API Simulators	1	Not installed
□ Android Auto Desktop Head Unit emulator	1.1	Not installed
☑ Android Emulator	27.3.10	Installed
☑ Android SDK Platform-Tools	28.0.1	Installed
☑ Android SDK Tools	26.1.1	Installed
☑ Documentation for Android SDK	1	Installed
□ Google Play APK Expansion library	1	Not installed
□ Google Play Instant Development SDK	1.5.0	Not installed
□ Google Play Licensing Library	1	Not installed
□ Google Play services	49	
□ Google USB Driver	11	
□ Google Web Driver	2	Not installed
☑ Intel x86 Emulator Accelerator (HAXM installer)	7.3.2	Installed
☑ NDK	18.1.5063045	Not installed
☑ **Support Repository**		
☑ ConstraintLayout for Android		Installed

□ Show Package Details

CMake & LLDB

NDK

2- Add C++ Support



Create New Project

Create Android Project

Application name

SampleAppWithNDK

Company domain

arcelik.com

Project location

D:\Users\AR430805\AndroidStudioProjects\SampleAppWithNDK

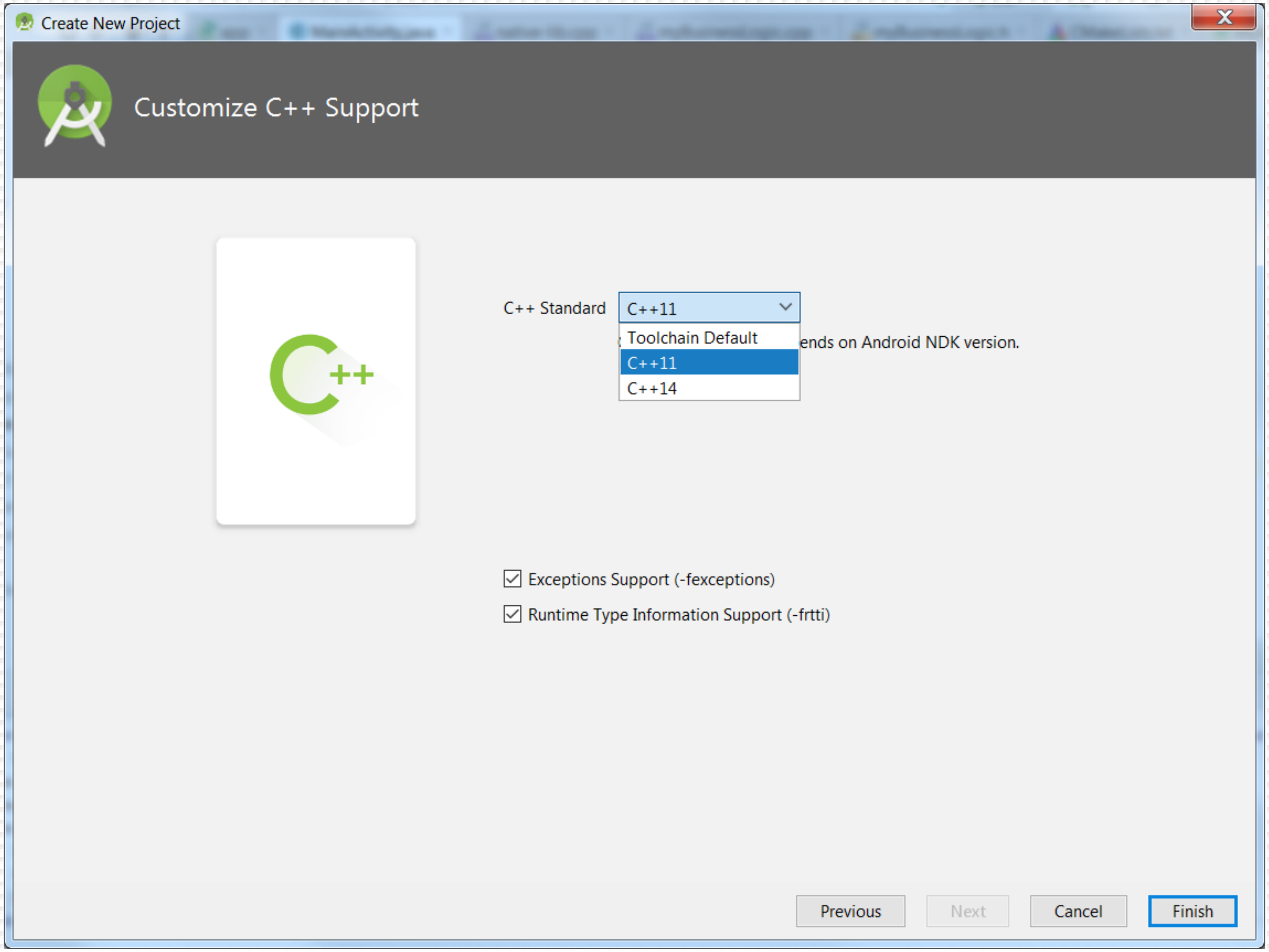
Package name

com.arcelik.sampleappwithndk

☑ Include C++ support

☐ Include Kotlin support

3- Customize C++ Support



Create New Project

Customize C++ Support

C++ Standard: C++11

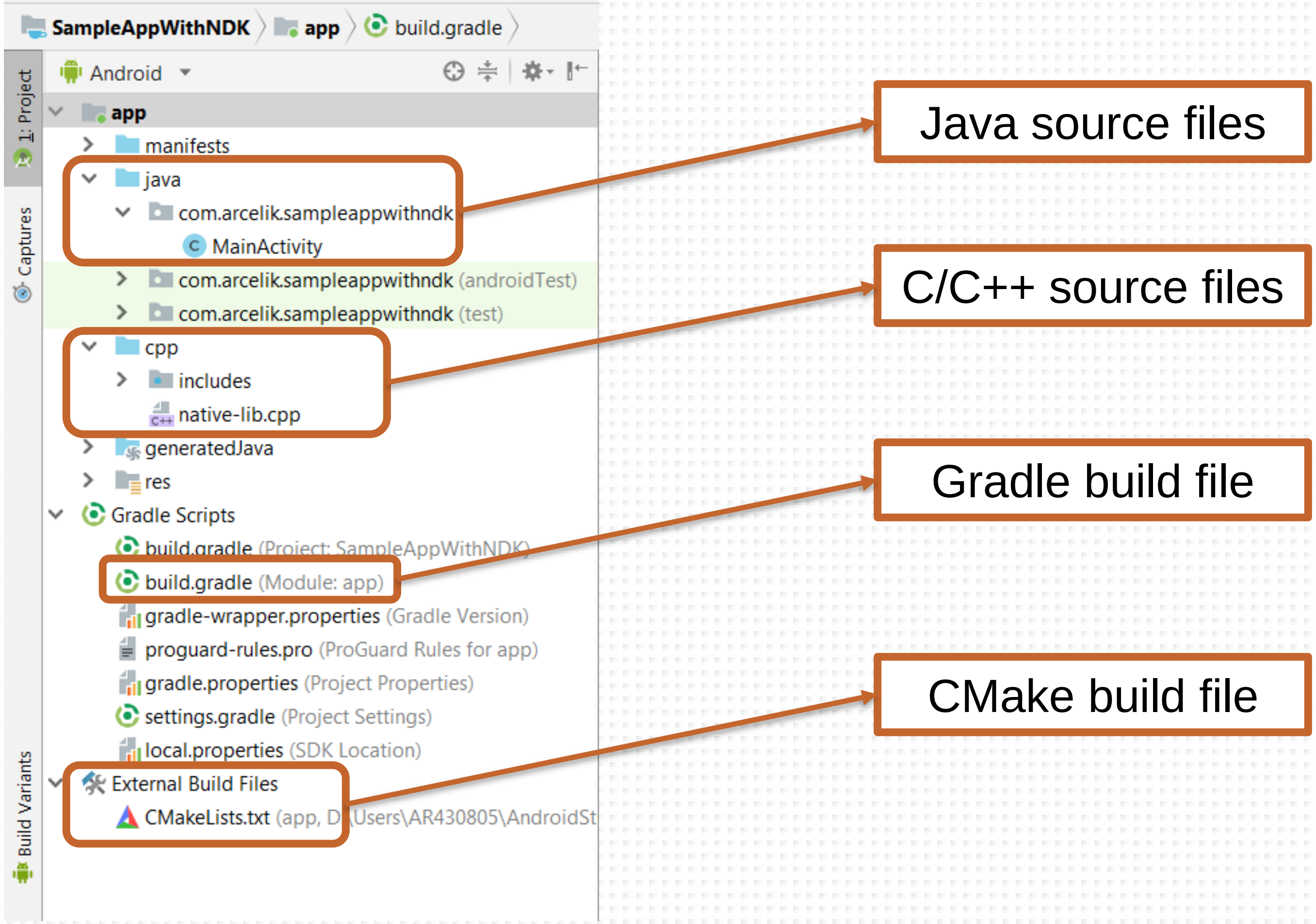
- Toolchain Default
- C++11
- C++14

ends on Android NDK version.

☑ Exceptions Support (-fexceptions)

☑ Runtime Type Information Support (-frtti)

Automatically Generated Application



```
SampleAppWithNDK 〉 app 〉 build.gradle 〉
Android ▾
app
  > manifests
  ∨ java
    ∨ com.arcelik.sampleappwithndk
        MainActivity
    > com.arcelik.sampleappwithndk (androidTest)
    > com.arcelik.sampleappwithndk (test)
  ∨ cpp
    > includes
      native-lib.cpp
  > generatedJava
  > res
∨ Gradle Scripts
    build.gradle (Project: SampleAppWithNDK)
    build.gradle (Module: app)
    gradle-wrapper.properties (Gradle Version)
    proguard-rules.pro (ProGuard Rules for app)
    gradle.properties (Project Properties)
    settings.gradle (Project Settings)
    local.properties (SDK Location)
∨ External Build Files
    CMakeLists.txt (app, D:\Users\AR430805\AndroidSt
```

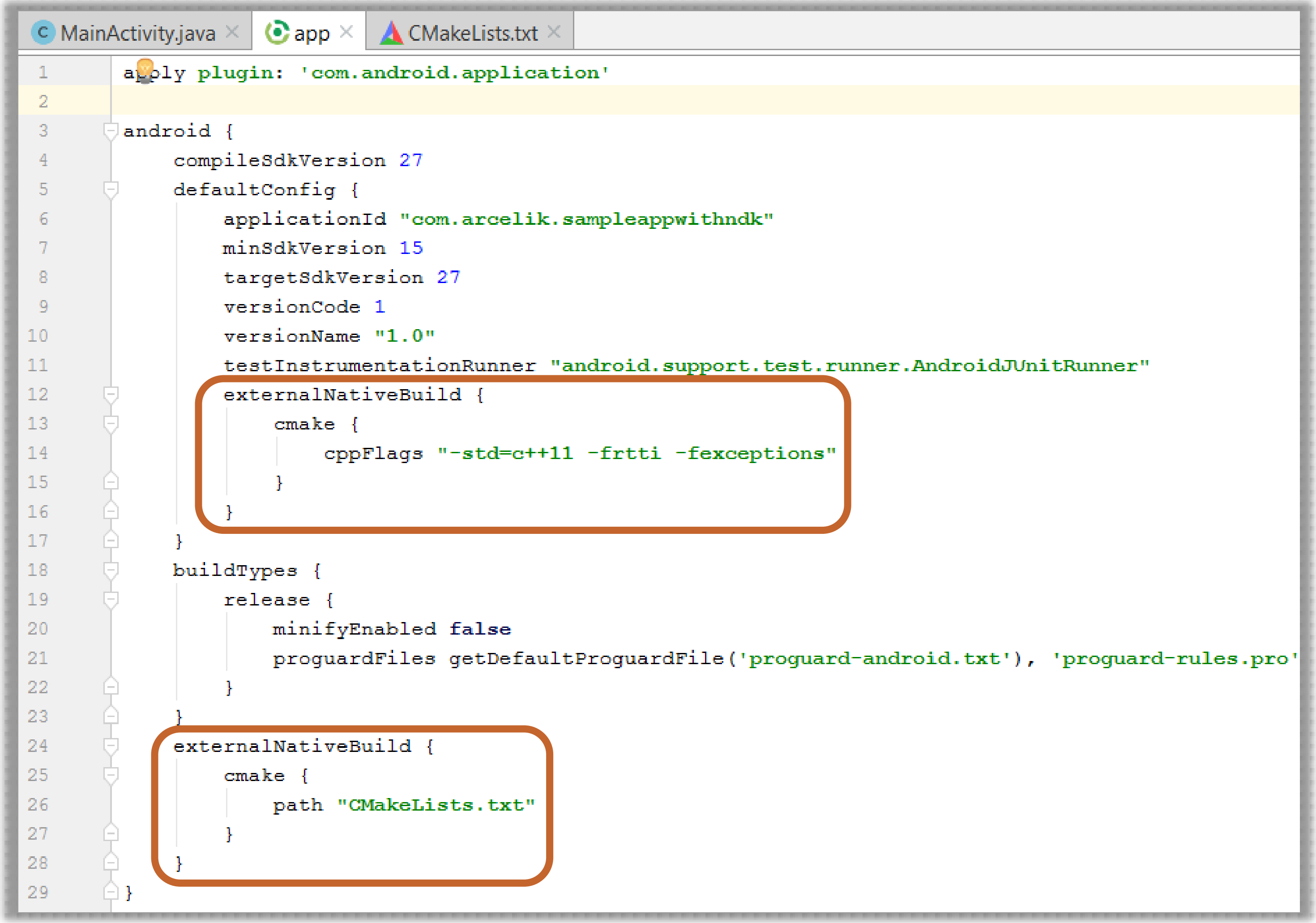
Java source files

C/C++ source files

Gradle build file

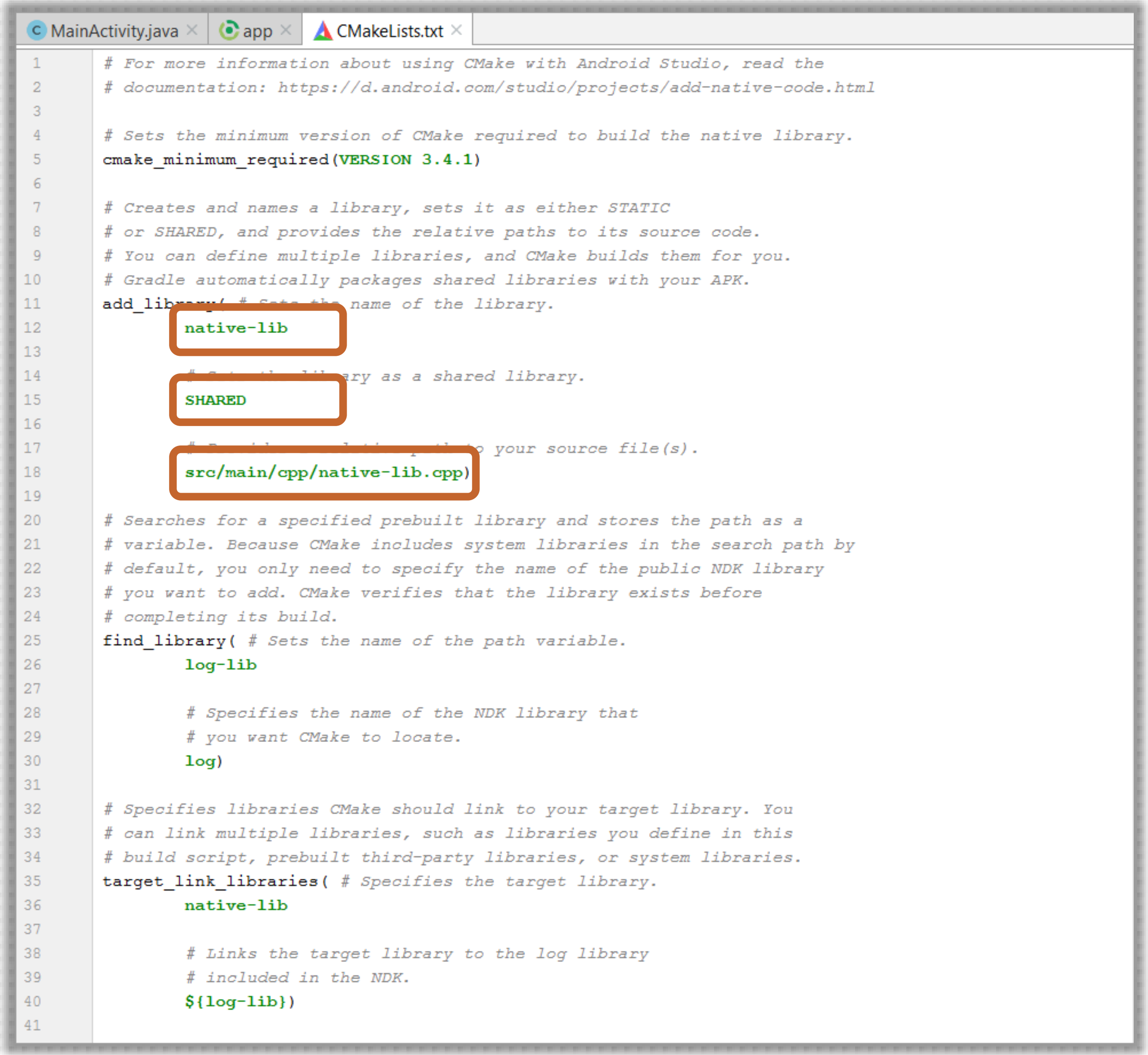
CMake build file

Gradle Build File



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

android {
    compileSdkVersion 27
    defaultConfig {
        applicationId "com.arcelik.sampleappwithndk"
        minSdkVersion 15
        targetSdkVersion 27
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
        externalNativeBuild {
            cmake {
                cppFlags "-std=c++11 -frtti -fexceptions"
            }
        }
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
    externalNativeBuild {
        cmake {
            path "CMakeLists.txt"
        }
    }
}
```

CMake Build File



```
# For more information about using CMake with Android Studio, read the
# documentation: https://d.android.com/studio/projects/add-native-code.html

# Sets the minimum version of CMake required to build the native library.
cmake_minimum_required(VERSION 3.4.1)

# Creates and names a library, sets it as either STATIC
# or SHARED, and provides the relative paths to its source code.
# You can define multiple libraries, and CMake builds them for you.
# Gradle automatically packages shared libraries with your APK.
add_library( # Sets the name of the library.
        native-lib

        # Sets the library as a shared library.
        SHARED

        # Provides a relative path to your source file(s).
        src/main/cpp/native-lib.cpp)

# Searches for a specified prebuilt library and stores the path as a
# variable. Because CMake includes system libraries in the search path by
# default, you only need to specify the name of the public NDK library
# you want to add. CMake verifies that the library exists before
# completing its build.
find_library( # Sets the name of the path variable.
        log-lib

        # Specifies the name of the NDK library that
        # you want CMake to locate.
        log)

# Specifies libraries CMake should link to your target library. You
# can link multiple libraries, such as libraries you define in this
# build script, prebuilt third-party libraries, or system libraries.
target_link_libraries( # Specifies the target library.
        native-lib

        # Links the target library to the log library
        # included in the NDK.
        ${log-lib})
```

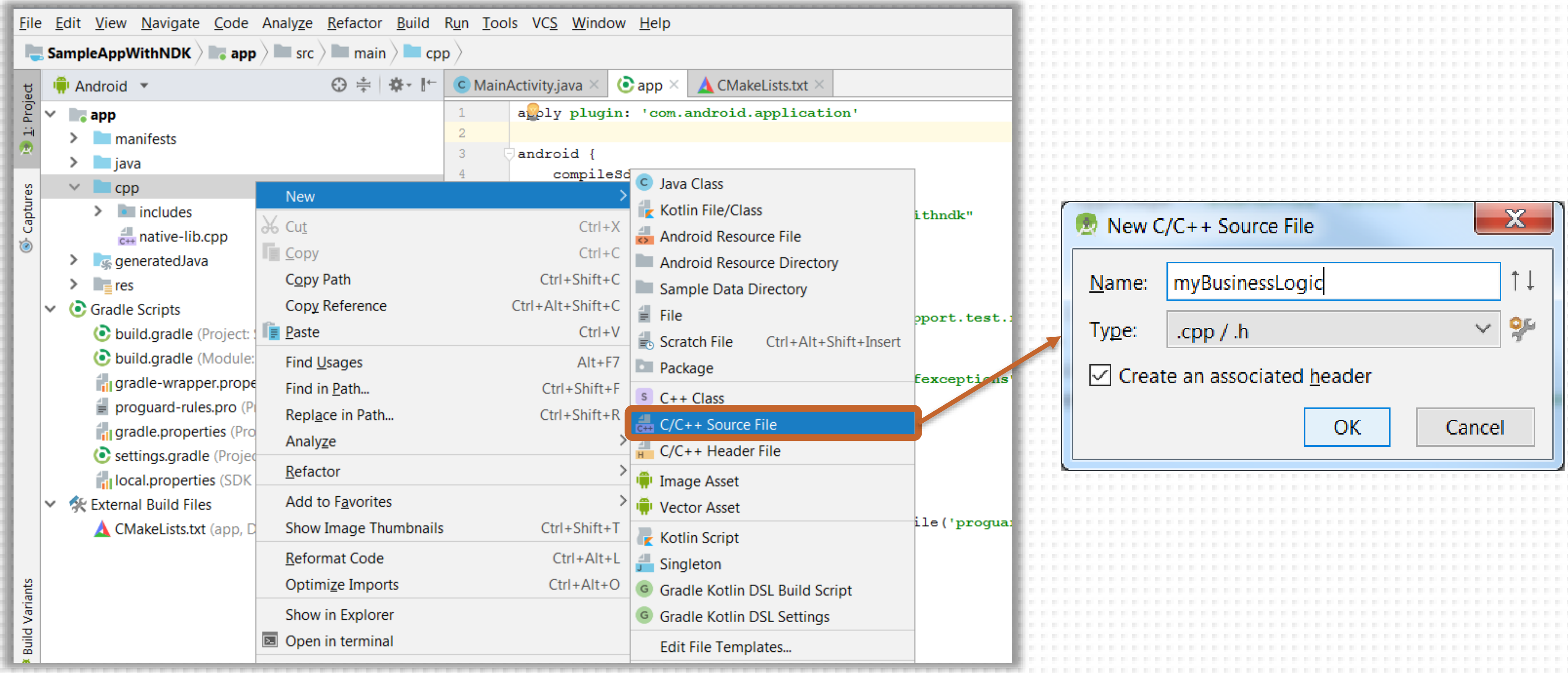
Adding C/C++ Support to Existing Project

Add C/C++ Support to Existing Project

1. Create new C/C++ source files
   - https://developer.android.com/studio/projects/add-native-code#create-sources
2. Configure CMake
   - https://developer.android.com/studio/projects/configure-cmake
3. Link Gradle to your native library
   - https://developer.android.com/studio/projects/gradle-external-native-builds

Adding C/C++ Sources to Android Project

Adding New C/C++ Sources



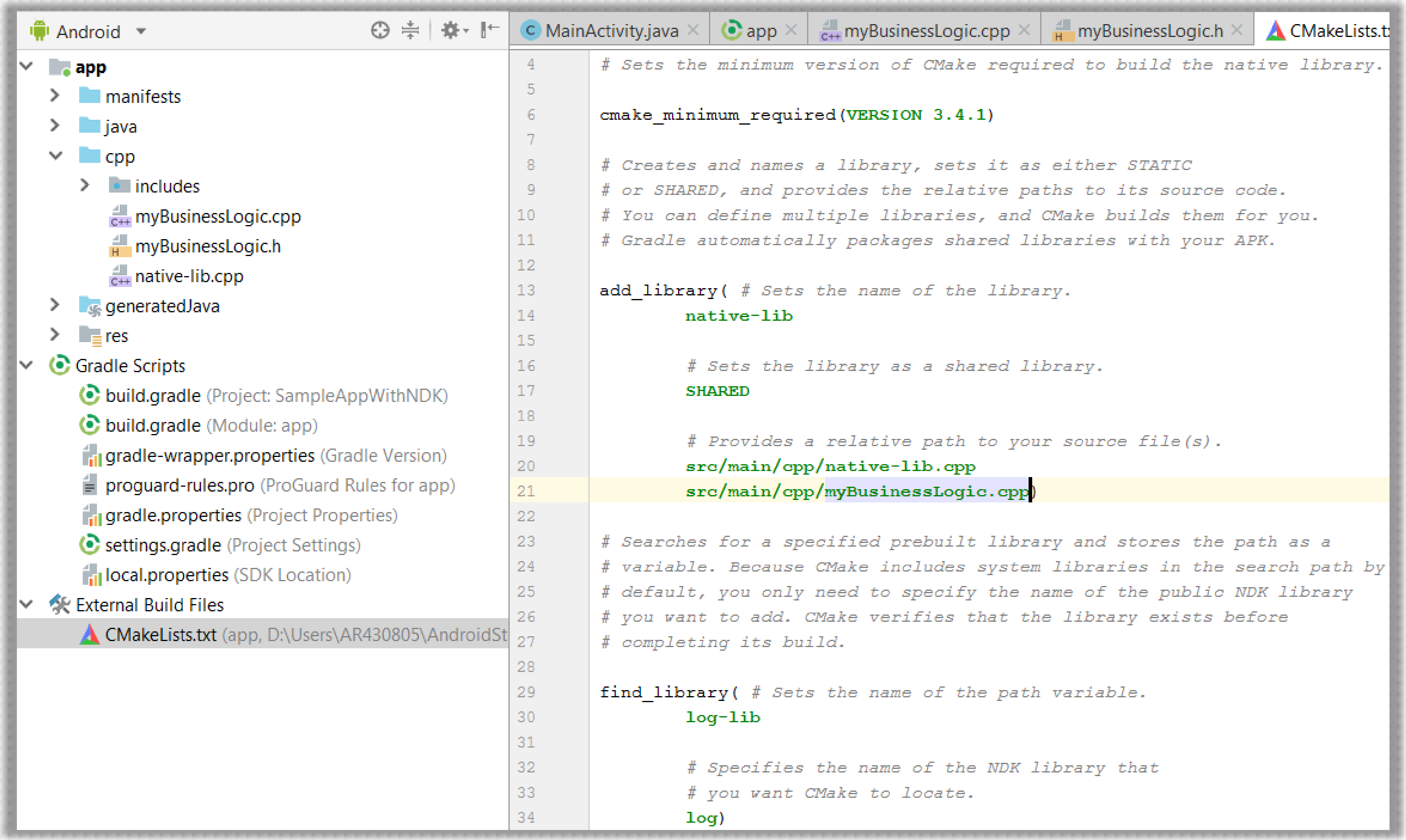
```
apply plugin: 'com.android.application'

android {
    compileSd
ithndk"
pport.test.
fexceptions'
ile('progua
```

Name: myBusinessLogic

Type: .cpp / .h

☑ Create an associated header

Configure CMake

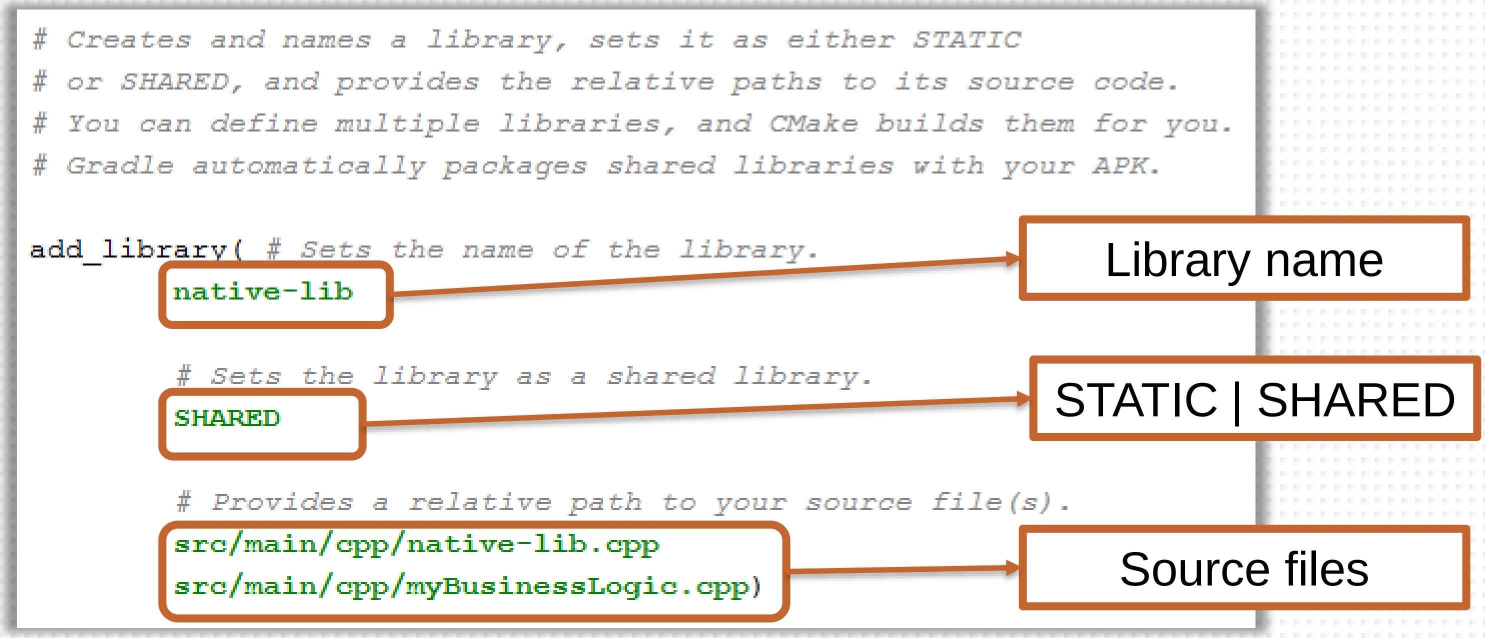


```
app
    manifests
    java
    cpp
        includes
        myBusinessLogic.cpp
        myBusinessLogic.h
        native-lib.cpp
    generatedJava
    res
Gradle Scripts
    build.gradle (Project: SampleAppWithNDK)
    build.gradle (Module: app)
    gradle-wrapper.properties (Gradle Version)
    proguard-rules.pro (ProGuard Rules for app)
    gradle.properties (Project Properties)
    settings.gradle (Project Settings)
    local.properties (SDK Location)
External Build Files
    CMakeLists.txt (app, D:\Users\AR430805\AndroidSt
```

```
# Sets the minimum version of CMake required to build the native library.

cmake_minimum_required(VERSION 3.4.1)

# Creates and names a library, sets it as either STATIC
# or SHARED, and provides the relative paths to its source code.
# You can define multiple libraries, and CMake builds them for you.
# Gradle automatically packages shared libraries with your APK.

add_library( # Sets the name of the library.
        native-lib

        # Sets the library as a shared library.
        SHARED

        # Provides a relative path to your source file(s).
        src/main/cpp/native-lib.cpp
        src/main/cpp/myBusinessLogic.cpp)

# Searches for a specified prebuilt library and stores the path as a
# variable. Because CMake includes system libraries in the search path by
# default, you only need to specify the name of the public NDK library
# you want to add. CMake verifies that the library exists before
# completing its build.

find_library( # Sets the name of the path variable.
        log-lib

        # Specifies the name of the NDK library that
        # you want CMake to locate.
        log)
```

CMake - add_library()

- Adds a library to the project using the specified source files
  - https://cmake.org/cmake/help/latest/command/add_library.html



```
# Creates and names a library, sets it as either STATIC
# or SHARED, and provides the relative paths to its source code.
# You can define multiple libraries, and CMake builds them for you.
# Gradle automatically packages shared libraries with your APK.

add_library( # Sets the name of the library.
             native-lib

             # Sets the library as a shared library.
             SHARED

             # Provides a relative path to your source file(s).
             src/main/cpp/native-lib.cpp
             src/main/cpp/myBusinessLogic.cpp)
```

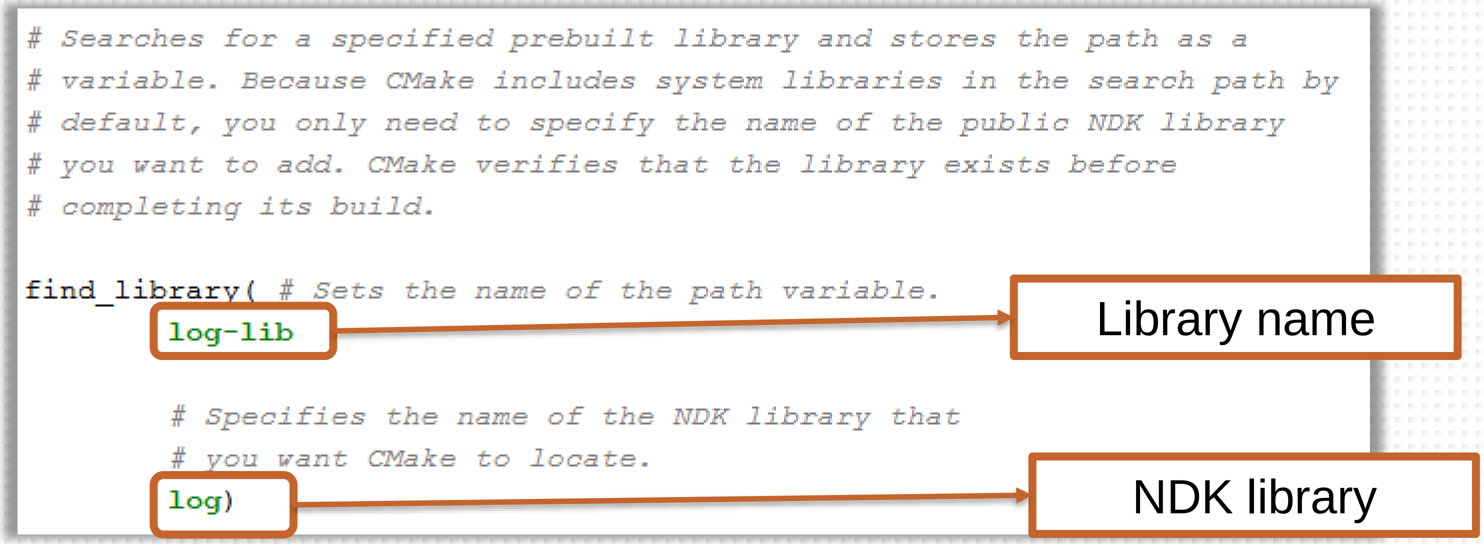
Library name

STATIC | SHARED

Source files

CMake - find_library()

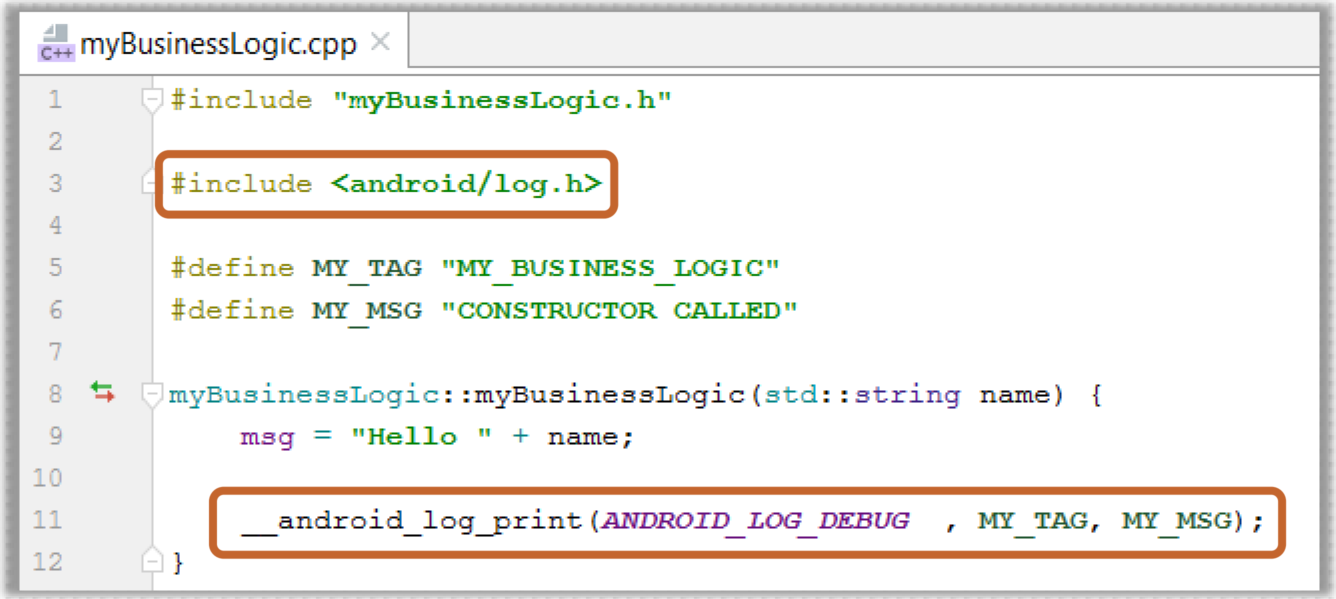
- Finds the public NDK library
  - https://cmake.org/cmake/help/latest/command/find_library.html
- Check below link to see all Android NDK Native APIs
  - https://developer.android.com/ndk/guides/stable_apis



```
# Searches for a specified prebuilt library and stores the path as a
# variable. Because CMake includes system libraries in the search path by
# default, you only need to specify the name of the public NDK library
# you want to add. CMake verifies that the library exists before
# completing its build.

find_library( # Sets the name of the path variable.
        log-lib

        # Specifies the name of the NDK library that
        # you want CMake to locate.
        log)
```

Example public NDK Library - liblog

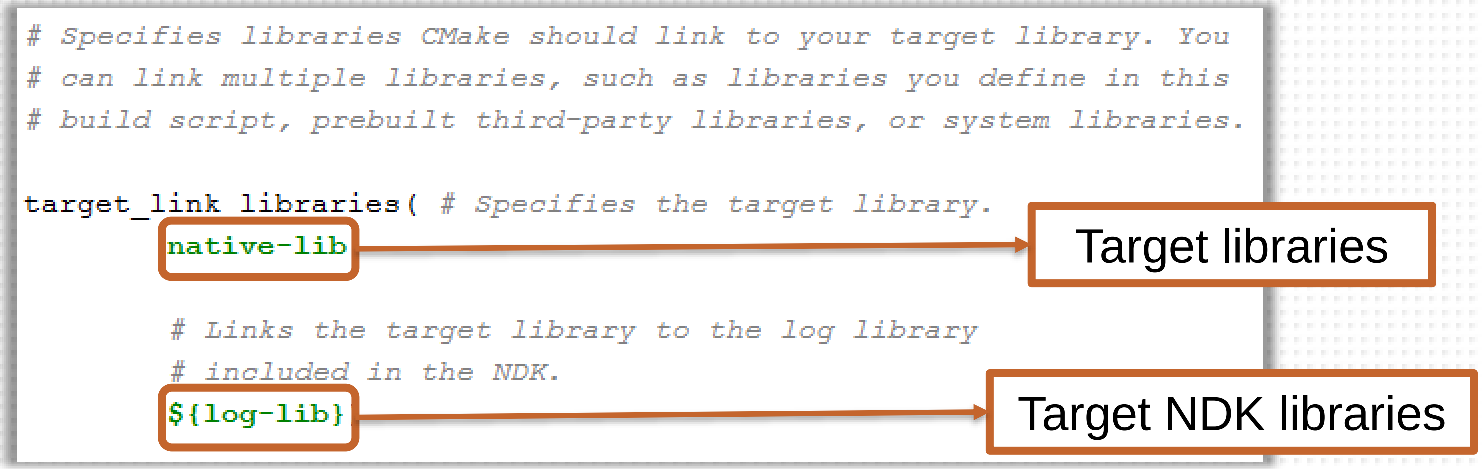
- <android/log.h> contains various functions that an app can use to send log messages to logcat from native code



```
#include "myBusinessLogic.h"

#include <android/log.h>

#define MY_TAG "MY_BUSINESS_LOGIC"
#define MY_MSG "CONSTRUCTOR CALLED"

myBusinessLogic::myBusinessLogic(std::string name) {
    msg = "Hello " + name;

    __android_log_print(ANDROID_LOG_DEBUG  , MY_TAG, MY_MSG);
}
```

CMake - target_link_libraries()

- Specifies libraries or flags to use when linking a given target
  - https://cmake.org/cmake/help/latest/command/target_link_libraries.html



```
# Specifies libraries CMake should link to your target library. You
# can link multiple libraries, such as libraries you define in this
# build script, prebuilt third-party libraries, or system libraries.

target_link libraries( # Specifies the target library.
        native-lib

        # Links the target library to the log library
        # included in the NDK.
        ${log-lib}
```

Target libraries

Target NDK libraries

Adding Prebuilt Libraries to Android Project

Adding Prebuilt Libraries

- Follow below principles to use prebuilt libraries
  1. Cross compile the source code of the library
  2. Copy prebuilt libraries and header files to Android Project
  3. Configure CMake
  4. Write your code using prebuilt library

1- Cross Compile for Android

- Different Android devices use different CPU architectures
- Each architecture has its own Application Binary Interface (ABI)
- Find toolchains in *Android/sdk/ndk-bundle/toolchains*
- Compile source code for the following architectures

Architecture	ABI	Toolchain binary
arm	armeabi-v7a	arm-linux-androideabi
arm64	arm64-v8a	aarch64-linux-android
x86	x86	i686-linux-android
x86_64	x86_64	x86_64-linux-android

Cross Compile autoconf-based Projects

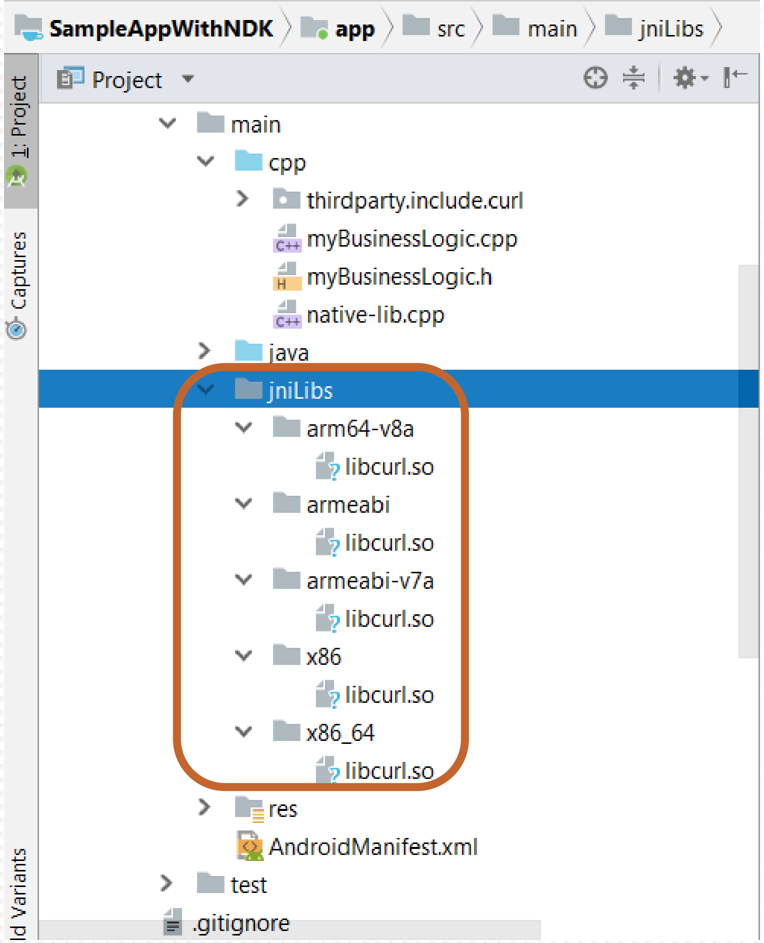
- A autoconf-based project would look more like this:

```
# Add the standalone toolchain to the search path.
export PATH=toolchain_path

# Tell configure what tools to use.
target_host=aarch64-linux-android
export CC=$target_host-gcc
export CXX=$target_host-g++
export LD=$target_host-ld
export AR=$target_host-ar
export STRIP=$target_host-strip

# Tell configure what flags Android requires.
export CFLAGS="-fPIE -fPIC"
export LDFLAGS="-pie"

tar zxvf xyz.tar.gz
cd xyz
./configure --host=$target_host
make
```

ABI Management

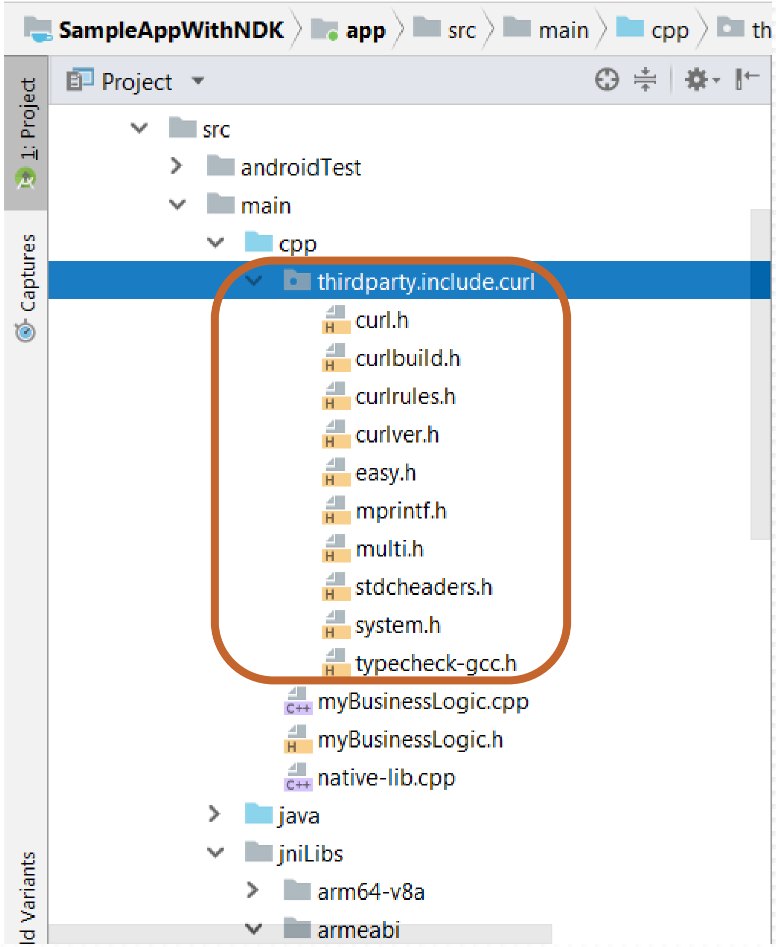
- Supported ABIs
  - armeabi (deprecated in r16, removed in NDK r17)
  - armeabi-v7a (extends armeabi)
  - arm64-v8a
  - x86
  - x86_64
- Libraries should be located inside the APK matching the following pattern:
  - /lib/**<abi>**/lib**<name>**.so
  - e.g. /lib/armeabi-v7a/libfoo.so

2- Copy prebuilt libraries & header files



SampleAppWithNDK 〉 app 〉 src 〉 main 〉 jniLibs 〉

```
main
  cpp
    thirdparty.include.curl
    myBusinessLogic.cpp
    myBusinessLogic.h
    native-lib.cpp
  java
  jniLibs
    arm64-v8a
      libcurl.so
    armeabi
      libcurl.so
    armeabi-v7a
      libcurl.so
    x86
      libcurl.so
    x86_64
      libcurl.so
  res
  AndroidManifest.xml
test
.gitignore
```

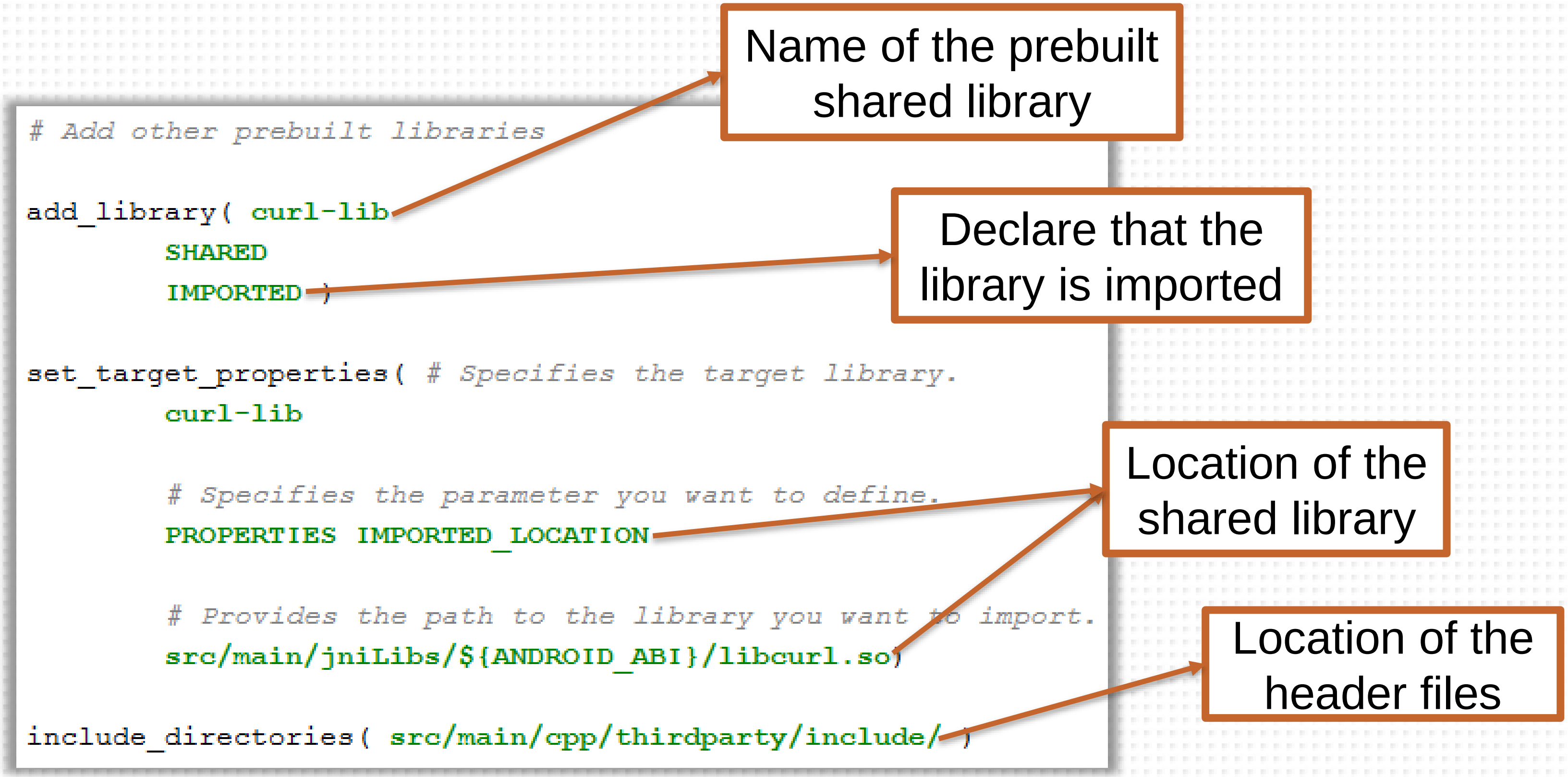


SampleAppWithNDK 〉 app 〉 src 〉 main 〉 cpp 〉 th

```
src
  androidTest
  main
    cpp
      thirdparty.include.curl
        curl.h
        curlbuild.h
        curlrules.h
        curlver.h
        easy.h
        mprintf.h
        multi.h
        stdcheaders.h
        system.h
        typecheck-gcc.h
      myBusinessLogic.cpp
      myBusinessLogic.h
      native-lib.cpp
    java
    jniLibs
      arm64-v8a
      armeabi
```

3- Configure CMake

- Tell CMake that you want to import the library into the project



```
# Add other prebuilt libraries

add_library( curl-lib
        SHARED
        IMPORTED )

set_target_properties( # Specifies the target library.
        curl-lib

        # Specifies the parameter you want to define.
        PROPERTIES IMPORTED_LOCATION

        # Provides the path to the library you want to import.
        src/main/jniLibs/${ANDROID_ABI}/libcurl.so)

include_directories( src/main/cpp/thirdparty/include/ )
```

Name of the prebuilt shared library

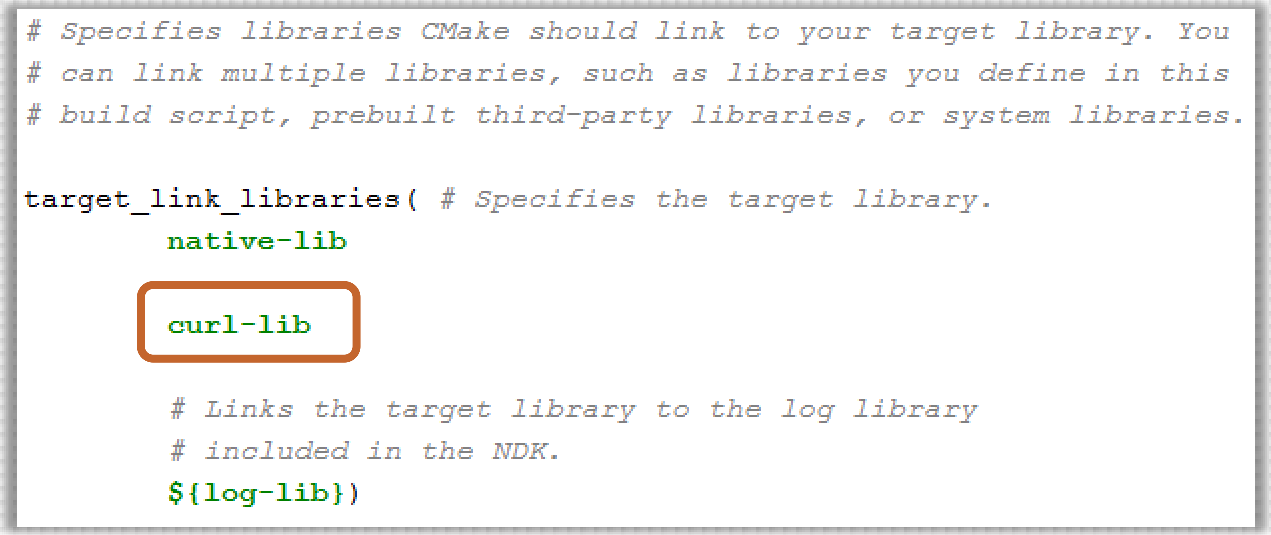
Declare that the library is imported

Location of the shared library

Location of the header files

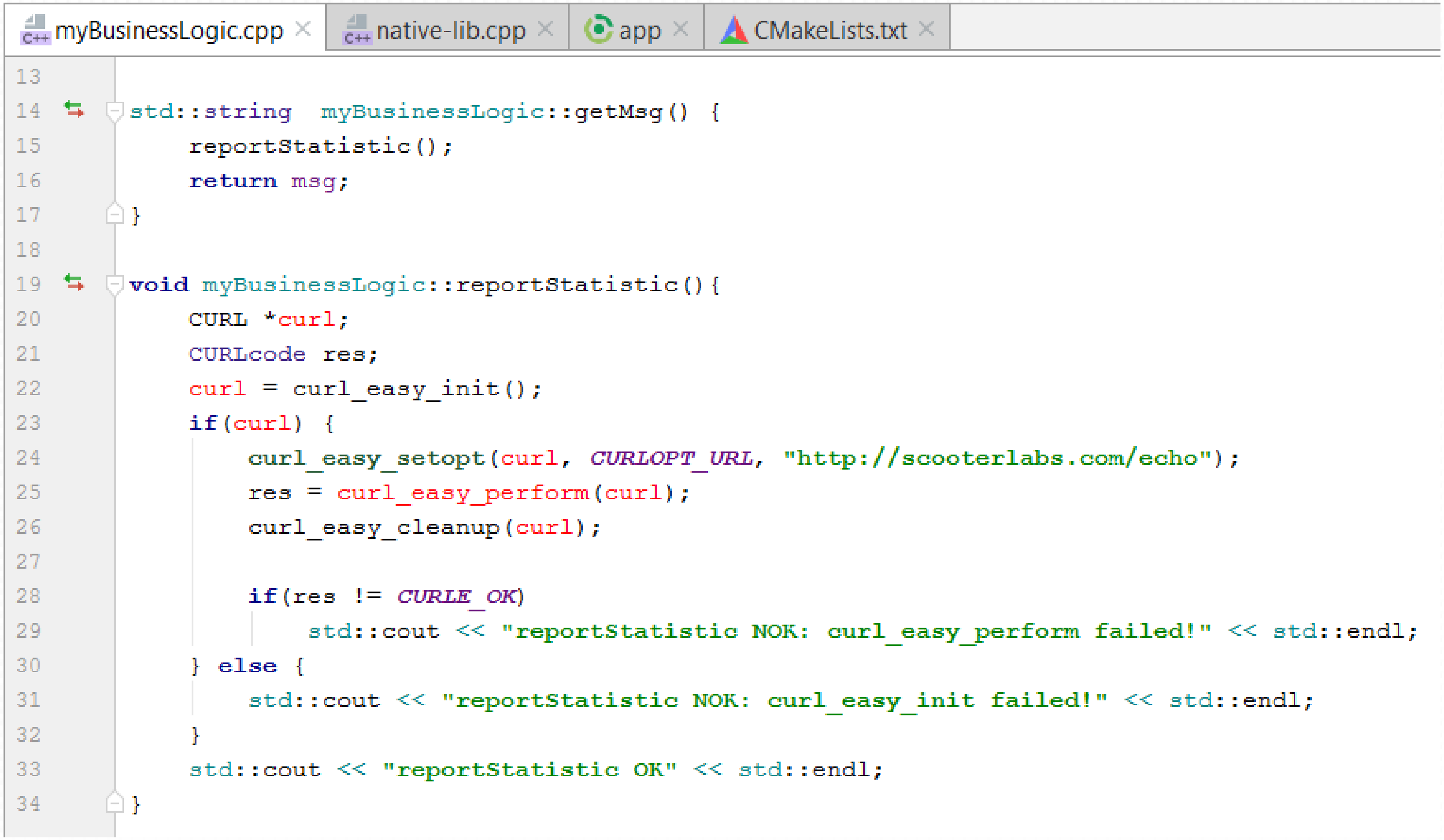
3- Configure Cmake cont.

- Link imported library to your target library (native-lib.so)



```
# Specifies libraries CMake should link to your target library. You
# can link multiple libraries, such as libraries you define in this
# build script, prebuilt third-party libraries, or system libraries.

target_link_libraries( # Specifies the target library.
        native-lib

        curl-lib

        # Links the target library to the log library
        # included in the NDK.
        ${log-lib})
```

Write Your Code Using Prebuilt Library

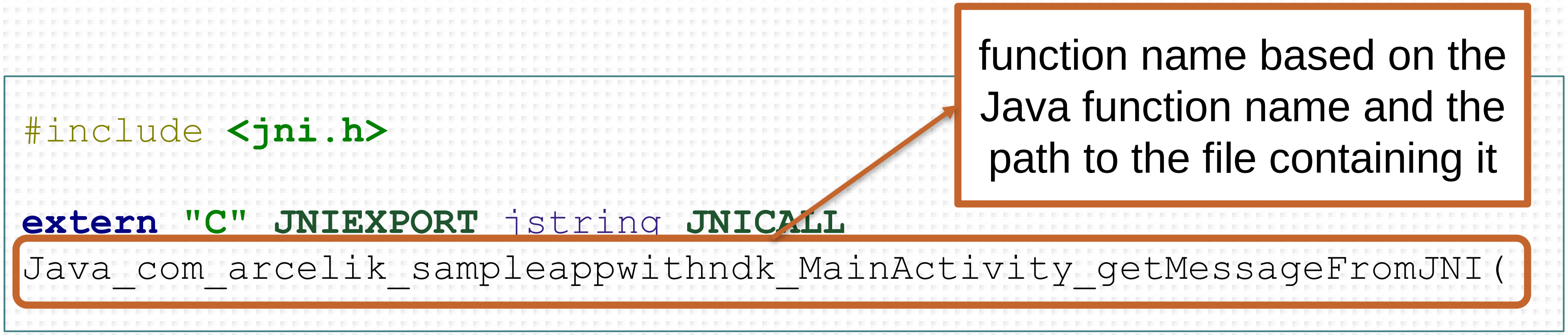


```

std::string  myBusinessLogic::getMsg() {
    reportStatistic();
    return msg;
}

void myBusinessLogic::reportStatistic(){
    CURL *curl;
    CURLcode res;
    curl = curl_easy_init();
    if(curl) {
        curl_easy_setopt(curl, CURLOPT_URL, "http://scooterlabs.com/echo");
        res = curl_easy_perform(curl);
        curl_easy_cleanup(curl);

        if(res != CURLE_OK)
            std::cout << "reportStatistic NOK: curl_easy_perform failed!" << std::endl;
    } else {
        std::cout << "reportStatistic NOK: curl_easy_init failed!" << std::endl;
    }
    std::cout << "reportStatistic OK" << std::endl;
}
```

Declaring Native Methods

Native Method Naming Convention

- Construct function name according to the following rules:
  1. Prepend Java_ to it.
  2. Describe the filepath relative to the top-level source directory.
  3. Use underscores in place of forward slashes.
  4. Omit the .java file extension.
  5. After the last underscore, append the function name.



```
#include <jni.h>

extern "C" JNIEXPORT jstring JNICALL
Java_com_arcelik_sampleappwithndk_MainActivity_getMessageFromJNI(
```

function name based on the Java function name and the path to the file containing it

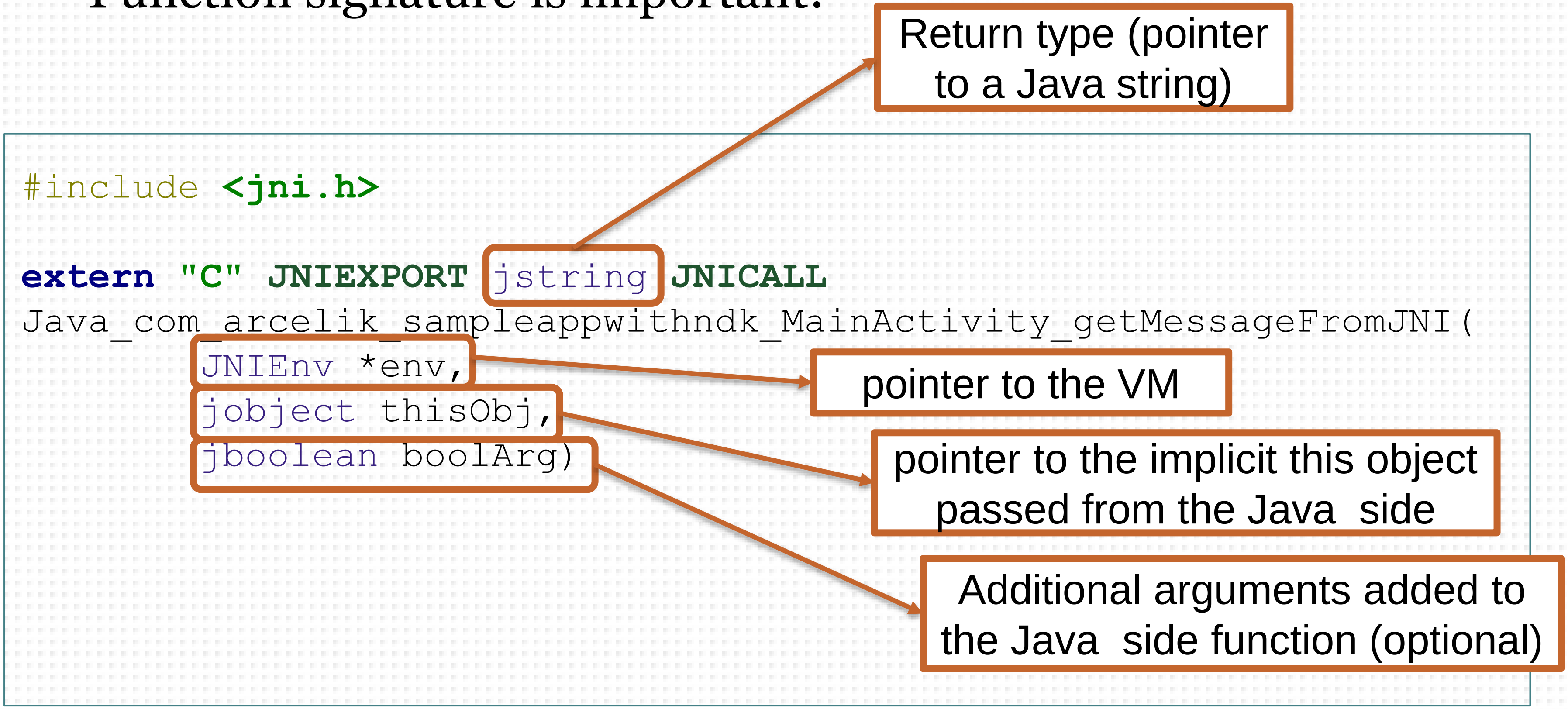
Native Method Naming Convention II

```
#include <jni.h>

extern "C" JNIEXPORT jstring JNICALL
Java_com_arcelik_sampleappwithndk_MainActivity_getMessageFromJNI(
```

refers to a Java function called **getMessageFromJNI** resides in app/src/main/java**/com/arcelik/sampleappwithndk/MainActivity.java**

Native Method Signature

- Function signature is important!



```
#include <jni.h>

extern "C" JNIEXPORT jstring JNICALL
Java_com_arcelik_sampleappwithndk_MainActivity_getMessageFromJNI(
        JNIEnv *env,
        jobject thisObj,
        jboolean boolArg)
```

Return type (pointer to a Java string)

pointer to the VM

pointer to the implicit this object passed from the Java side

Additional arguments added to the Java side function (optional)

Using Native Method Arguments

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

extern "C" JNIEXPORT jstring JNICALL
Java_com_arcelik_sampleappwithndk_MainActivity_getMessageFromJNI(
        JNIEnv *env,
        jobject thisObj,
        jboolean boolArg) {

    ...

    if(boolArg) {
        //do something here
    }

    ...

}
```

Access Object Field from Native Code I

- To access an object's field from native code, do the following:
    - Get the class object reference for the class with **GetObjectClass**
    - Get the field ID for the field with **GetFieldID**
    - Get the contents of the field with something appropriate, such as **GetObjectField**

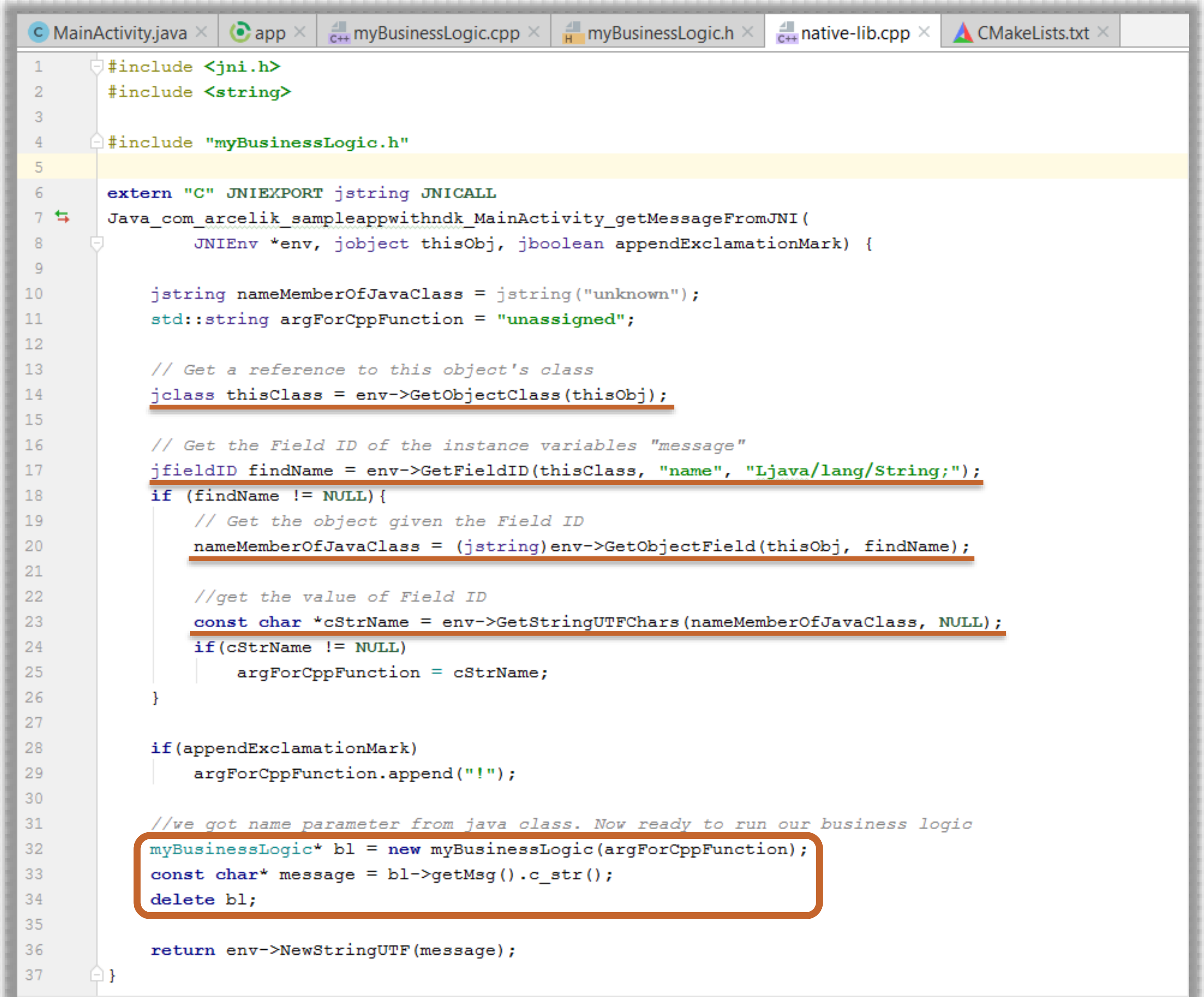
Access Object Field from Native Code II

```
Java_com_arcelik_sampleappwithndk_MainActivity_getMessageFromJNI(
        JNIEnv *env,
        jobject thisObj) {

    jstring nameMemberOfJavaClass = jstring("unknown");

    // Get a reference to this object's class
    jclass thisClass = env->GetObjectClass(thisObj);

    // Get the Field ID of the instance variables "message"
    jfieldID findName = env->GetFieldID(thisClass, "name", "Ljava/lang/String;");
    if (findName != NULL){
        // Get the object given the Field ID
        nameMemberOfJavaClass = (jstring)env->GetObjectField(thisObj, findName);
    }

    ...
}
```

Wrap Your Business Logic

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

extern "C" JNIEXPORT jstring JNICALL
Java_com_arcelik_sampleappwithndk_MainActivity_getMessageFromJNI(
        JNIEnv *env,
        jobject thisObj,
        jboolean boolArg) {

    ...

    myBusinessLogic* bl = new myBusinessLogic(argForCppFunction);
    const char* message = bl->getMsg().c_str();
    delete bl;

    return env->NewStringUTF(message);

}
```

JNI Native Function at a Glance



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

#include "myBusinessLogic.h"

extern "C" JNIEXPORT jstring JNICALL
Java_com_arcelik_sampleappwithndk_MainActivity_getMessageFromJNI(
        JNIEnv *env, jobject thisObj, jboolean appendExclamationMark) {

    jstring nameMemberOfJavaClass = jstring("unknown");
    std::string argForCppFunction = "unassigned";

    // Get a reference to this object's class
    jclass thisClass = env->GetObjectClass(thisObj);

    // Get the Field ID of the instance variables "message"
    jfieldID findName = env->GetFieldID(thisClass, "name", "Ljava/lang/String;");
    if (findName != NULL){
        // Get the object given the Field ID
        nameMemberOfJavaClass = (jstring)env->GetObjectField(thisObj, findName);

        //get the value of Field ID
        const char *cStrName = env->GetStringUTFChars(nameMemberOfJavaClass, NULL);
        if(cStrName != NULL)
            argForCppFunction = cStrName;
    }

    if(appendExclamationMark)
        argForCppFunction.append("!");

    //we got name parameter from java class. Now ready to run our business logic
    myBusinessLogic* bl = new myBusinessLogic(argForCppFunction);
    const char* message = bl->getMsg().c_str();
    delete bl;

    return env->NewStringUTF(message);
}
```

Calling Native Functions

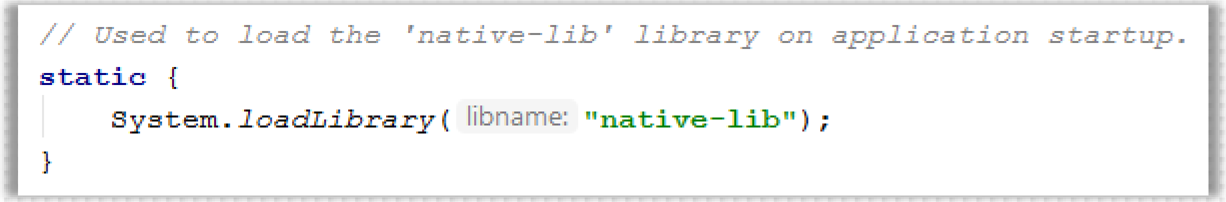
Calling Native Function from Java Side

- To call native function from Java source
  1. Load platform-specific native library
  2. Declare related method with native keyword
  3. Call native function via the method declared in step 2

Native methods are loaded with the method

1- Load Native Library

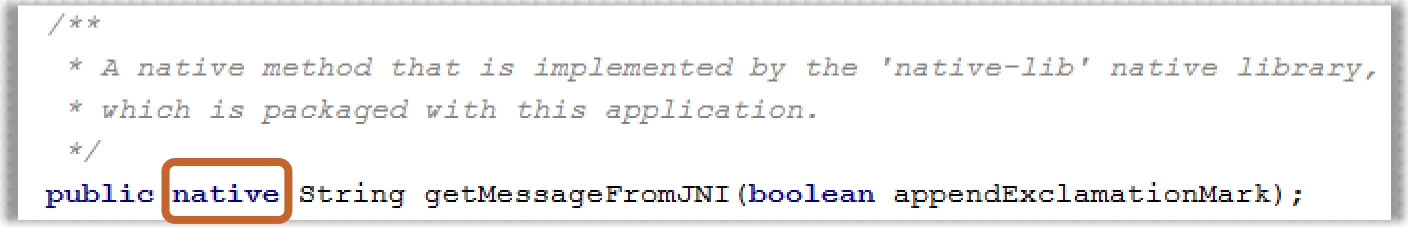
- Native libraries are loaded with the System.loadLibrary method



```
// Used to load the 'native-lib' library on application startup.
static {
    System.loadLibrary( libname: "native-lib");
}
```

2- Declare Java Native Method

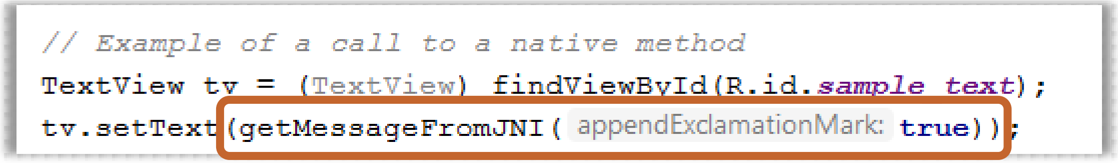
- Declare **native** Java method corresponding to the native method
- The **native** keyword tells the virtual machine that the function is implemented on the native side (in the shared library)



```
/**
 * A native method that is implemented by the 'native-lib' native library,
 * which is packaged with this application.
 */
public native String getMessageFromJNI(boolean appendExclamationMark);
```

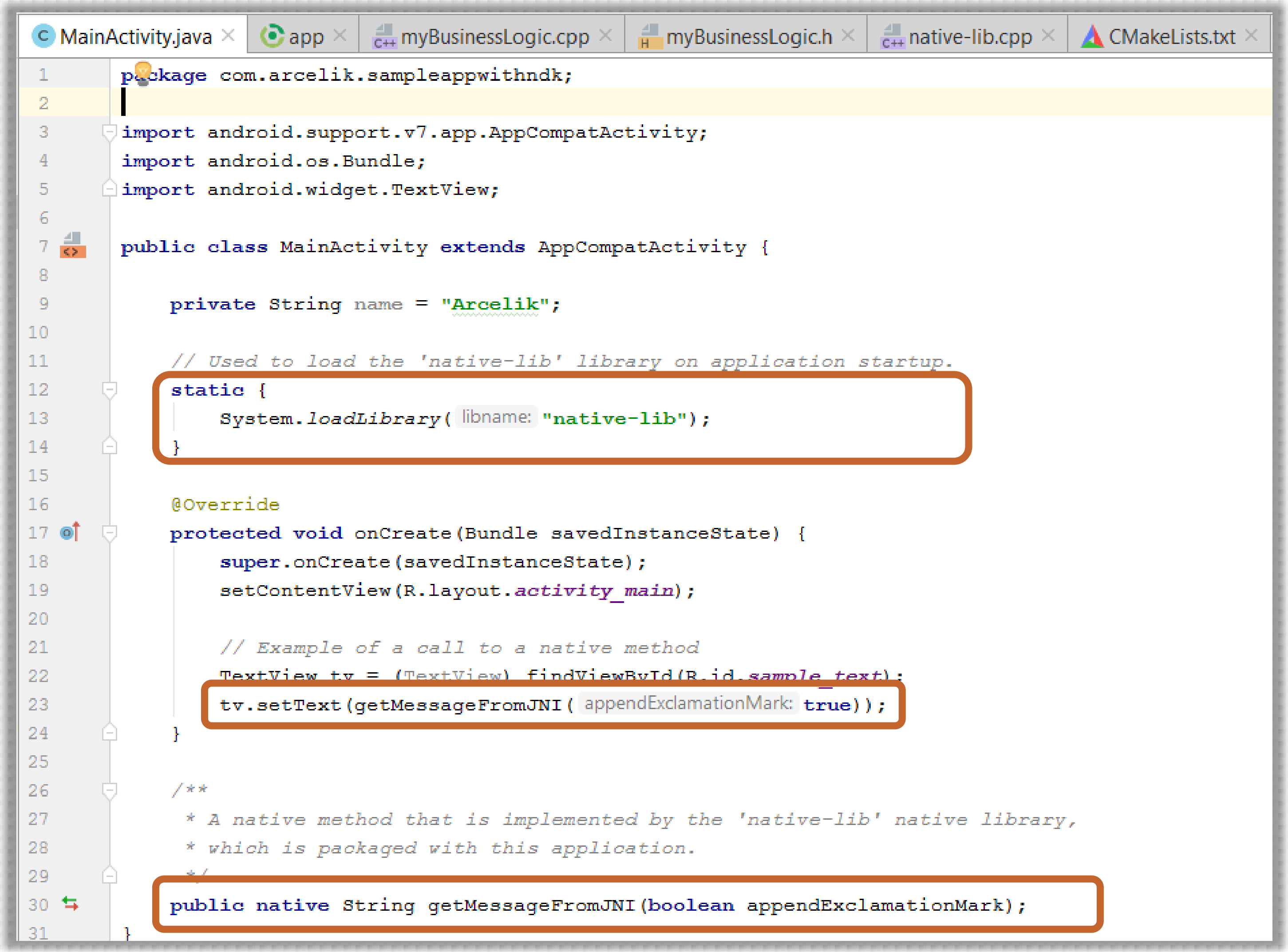
3- Call Native Method

- The JNI interface pointer (JNIEnv*) is the first argument to native methods
- The second argument to a nonstatic native method is a reference to the object.
- The second argument to a static native method is a reference to its Java class.



```
// Example of a call to a native method
TextView tv = (TextView) findViewById(R.id.sample_text);
tv.setText(getMessageFromJNI( appendExclamationMark: true));
```

Java Side Implementation at a Glance



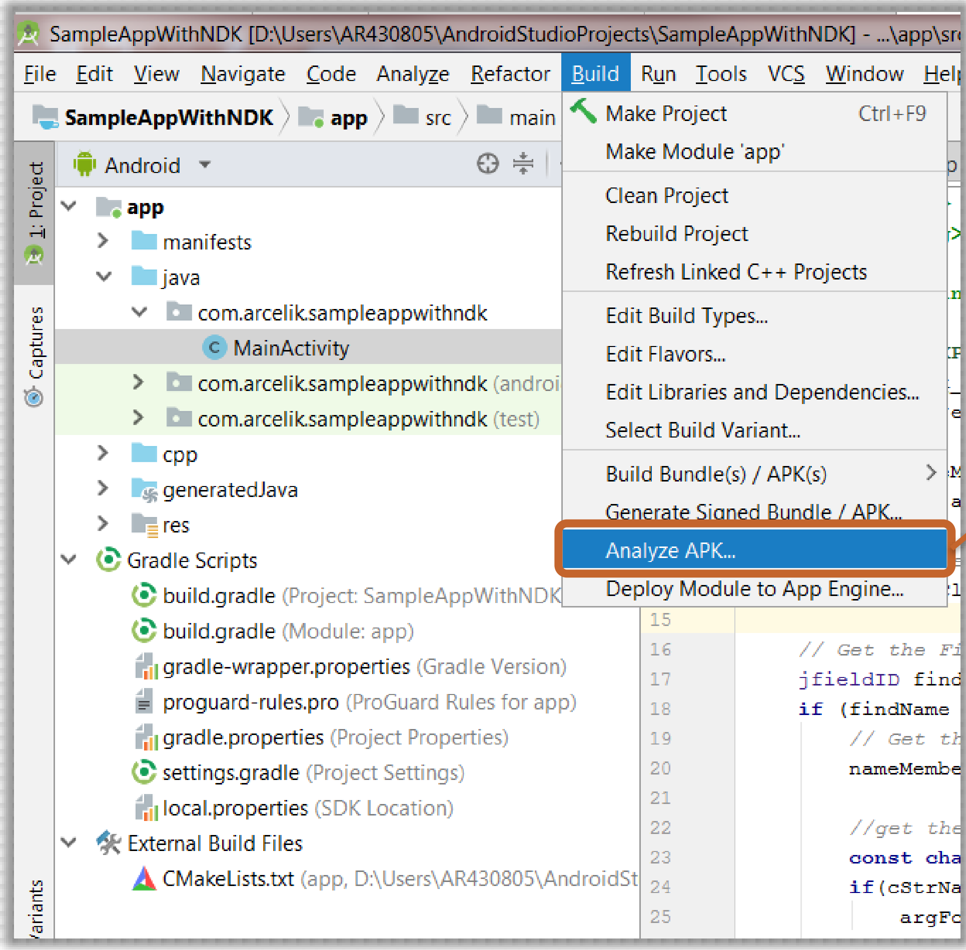
```java
package com.arcelik.sampleappwithndk;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {

    private String name = "Arcelik";

    // Used to load the 'native-lib' library on application startup.
    static {
        System.loadLibrary( libname: "native-lib");
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // Example of a call to a native method
        TextView tv = (TextView) findViewById(R.id.sample_text);
        tv.setText(getMessageFromJNI( appendExclamationMark: true));
    }

    /**
     * A native method that is implemented by the 'native-lib' native library,
     * which is packaged with this application.
     */
    public native String getMessageFromJNI(boolean appendExclamationMark);
}
```

Analyzing apk to Check Native Libraries

Checking Native Library I

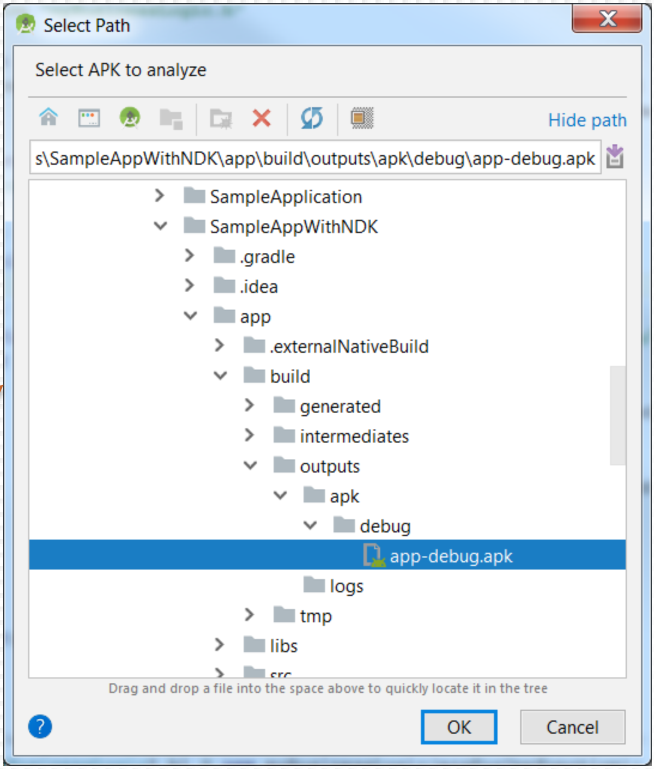
- Generating native library can be done incorrectly
- Be sure the native library is embedded to APK
- Use APK Analyzer
  - Select Build > Analyze APK
  - Select the APK from the app/build/outputs/apk/ directory and click OK.

Checking Native Library II



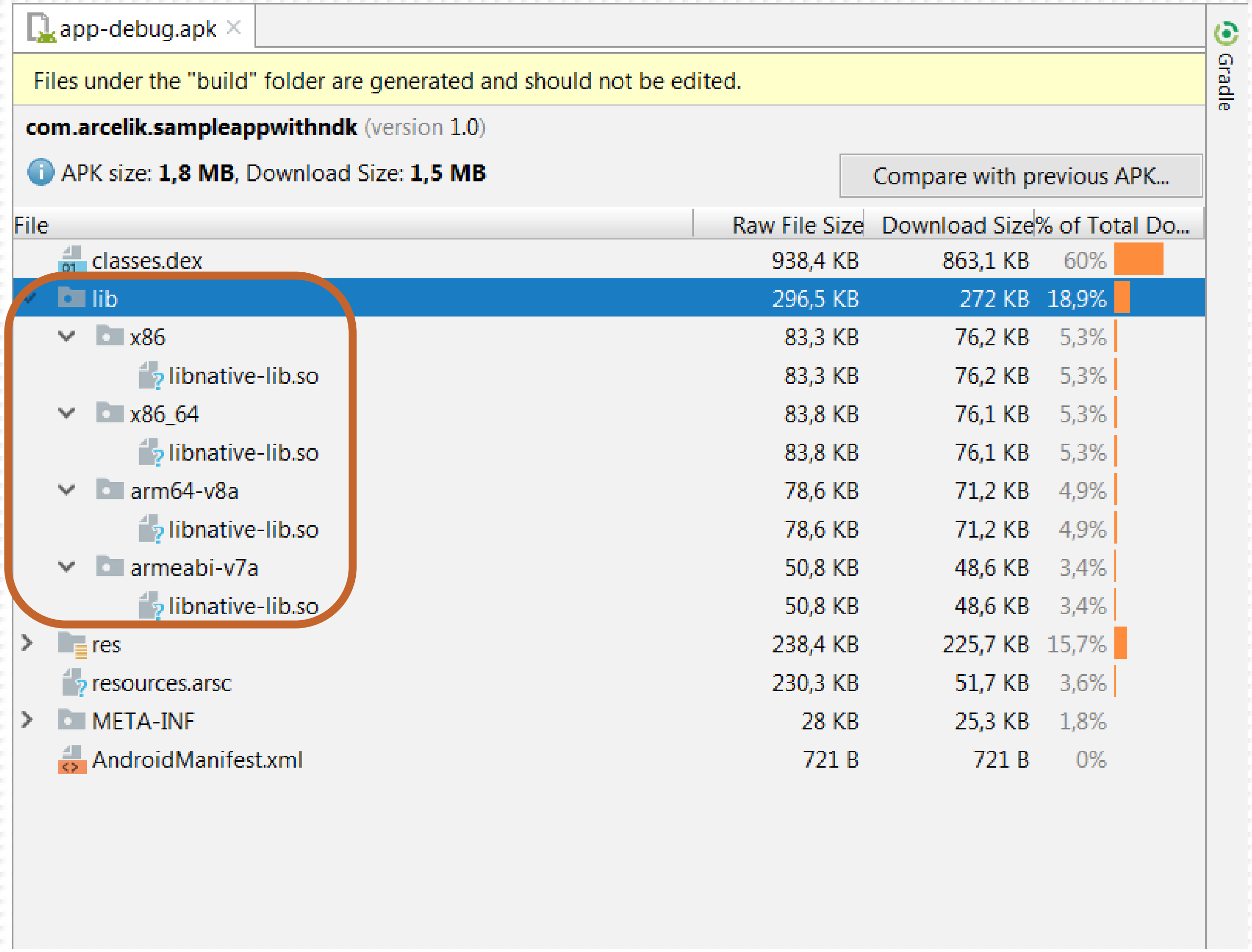
SampleAppWithNDK [D:\Users\AR430805\AndroidStudioProjects\SampleAppWithNDK] - ...\app\src
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
SampleAppWithNDK app src main
Android
app
manifests
java
com.arcelik.sampleappwithndk
MainActivity
com.arcelik.sampleappwithndk (androi
com.arcelik.sampleappwithndk (test)
cpp
generatedJava
res
Gradle Scripts
build.gradle (Project: SampleAppWithNDK
build.gradle (Module: app)
gradle-wrapper.properties (Gradle Version)
proguard-rules.pro (ProGuard Rules for app)
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)
External Build Files
CMakeLists.txt (app, D:\Users\AR430805\AndroidSt
Make Project Ctrl+F9
Make Module 'app'
Clean Project
Rebuild Project
Refresh Linked C++ Projects
Edit Build Types...
Edit Flavors...
Edit Libraries and Dependencies...
Select Build Variant...
Build Bundle(s) / APK(s)
Generate Signed Bundle / APK...
Analyze APK...
Deploy Module to App Engine...
1: Project
Captures
Variants

```
// Get the Fi
jfieldID find
if (findName
    // Get th
    nameMembe

    //get the
    const cha
    if(cStrNa
        argFc
```



s\SampleAppWithNDK\app\build\outputs\apk\debug\app-debug.apk

Checking Native Library III



Files under the "build" folder are generated and should not be edited.

com.arcelik.sampleappwithndk (version 1.0)

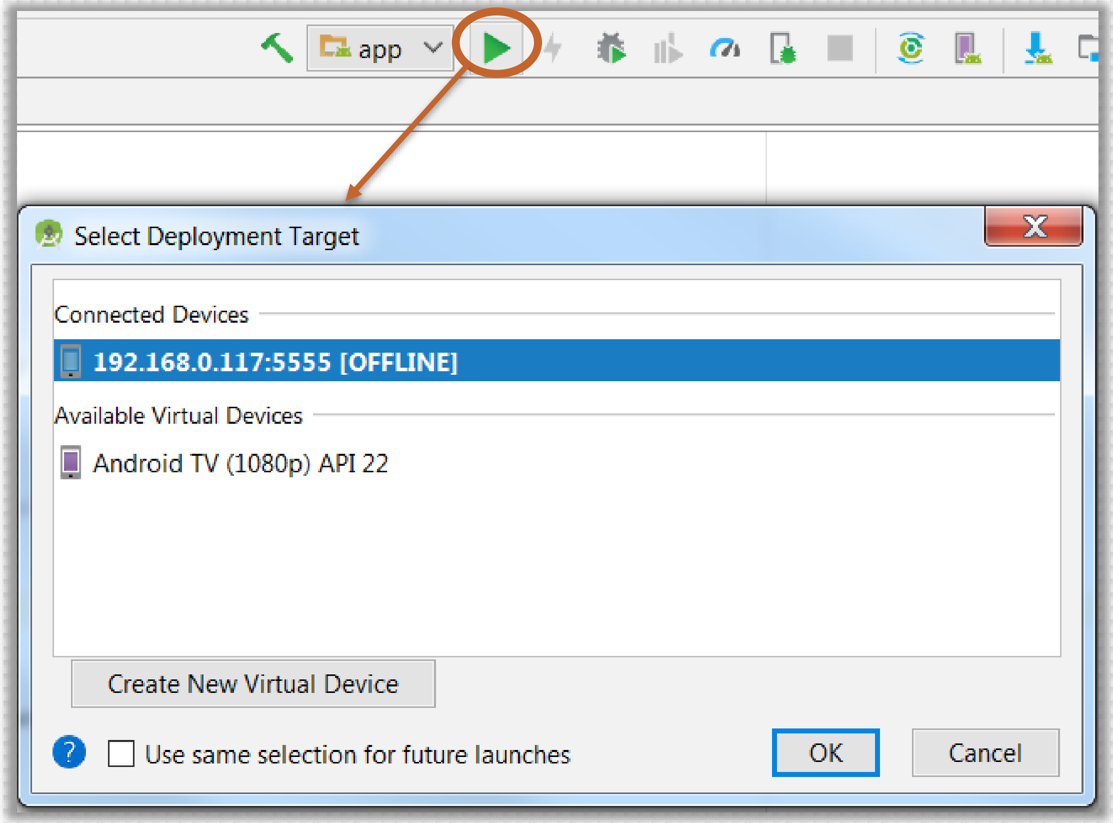
APK size: **1,8 MB**, Download Size: **1,5 MB**

File	Raw File Size	Download Size	% of Total Do...
classes.dex	938,4 KB	863,1 KB	60%
lib	296,5 KB	272 KB	18,9%
x86	83,3 KB	76,2 KB	5,3%
libnative-lib.so	83,3 KB	76,2 KB	5,3%
x86_64	83,8 KB	76,1 KB	5,3%
libnative-lib.so	83,8 KB	76,1 KB	5,3%
arm64-v8a	78,6 KB	71,2 KB	4,9%
libnative-lib.so	78,6 KB	71,2 KB	4,9%
armeabi-v7a	50,8 KB	48,6 KB	3,4%
libnative-lib.so	50,8 KB	48,6 KB	3,4%
res	238,4 KB	225,7 KB	15,7%
resources.arsc	230,3 KB	51,7 KB	3,6%
META-INF	28 KB	25,3 KB	1,8%
AndroidManifest.xml	721 B	721 B	0%

Running the Sample Application

Running Sample App on Fire TV Stick I

- Running the Android apps with NDK support is same as the other Android apps explained in the previous sessions



Select Deployment Target

Connected Devices

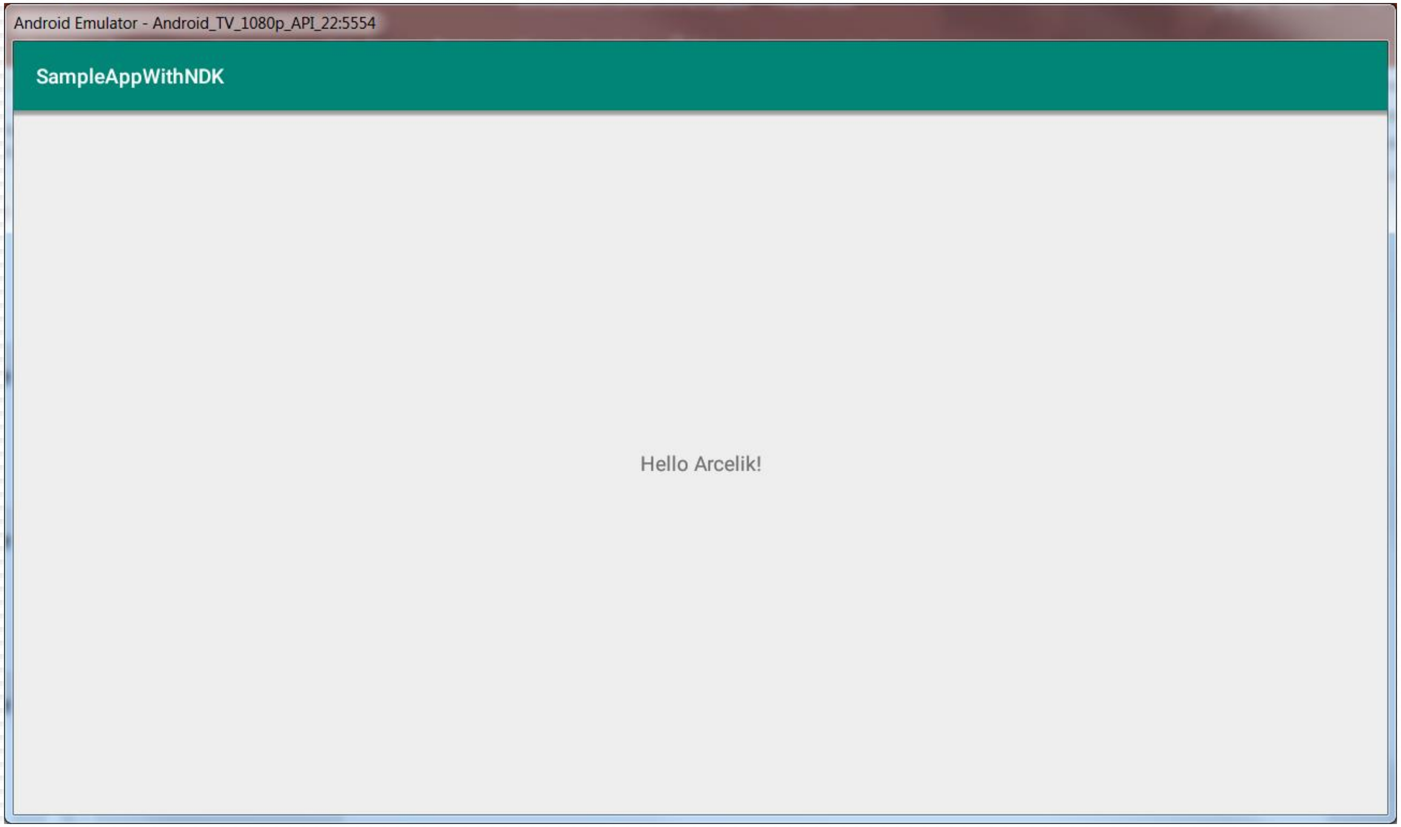
192.168.0.117:5555 [OFFLINE]

Available Virtual Devices

Android TV (1080p) API 22

□ Use same selection for future launches

Running Sample App on Fire TV Stick II



Android Emulator - Android_TV_1080p_API_22:5554

SampleAppWithNDK

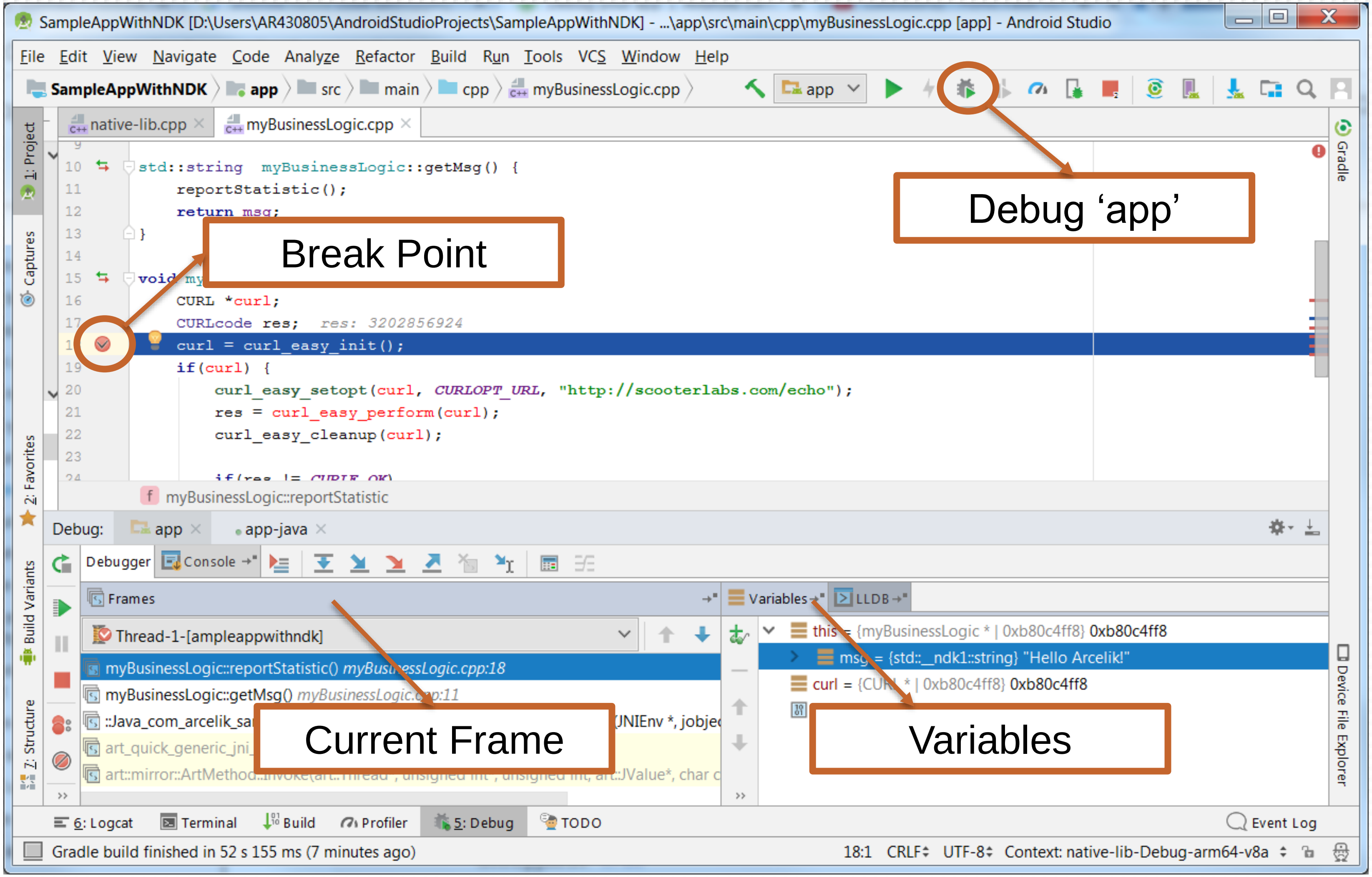
Hello Arcelik!

Debugging the Sample Application

Debugging Sample App on Fire TV Stick I

- If your project includes C/C++ code, you need to install **LLDB** from the SDK Manager
- You need to enable debugging in the device developer options
- Debugging the Android apps with NDK support is same as the other Android apps explained in the previous sessions
  - Set some breakpoints in the app code
  - Click *Debug* button in the toolbar and select a deployment target to start debugging
  - You can also attach a running process by clicking *Attach Debugger* in the toolbar and selecting a process

Debugging Sample App on Fire TV Stick II



SampleAppWithNDK [D:\Users\AR430805\AndroidStudioProjects\SampleAppWithNDK] - ...\app\src\main\cpp\myBusinessLogic.cpp [app] - Android Studio

Debug 'app'

Break Point

```
std::string  myBusinessLogic::getMsg() {
    reportStatistic();
    return msg;
}

void my
    CURL *curl;
    CURLcode res;  res: 3202856924
    curl = curl_easy_init();
    if(curl) {
        curl_easy_setopt(curl, CURLOPT_URL, "http://scooterlabs.com/echo");
        res = curl_easy_perform(curl);
        curl_easy_cleanup(curl);

        if(res != CURLE_OK)
```

myBusinessLogic::reportStatistic

Thread-1-[ampleappwithndk]

```
myBusinessLogic::reportStatistic() myBusinessLogic.cpp:18
myBusinessLogic::getMsg() myBusinessLogic.cpp:11
::Java_com_arcelik_sa                              JNIEnv *, jobjec
art_quick_generic_jni_
art::mirror::ArtMethod::Invoke(art::Thread*, unsigned int*, unsigned int, art::JValue*, char c
```

Current Frame

```
this = {myBusinessLogic * | 0xb80c4ff8} 0xb80c4ff8
  msg = {std::__ndk1::string} "Hello Arcelik!"
curl = {CURL * | 0xb80c4ff8} 0xb80c4ff8
```

Variables

Gradle build finished in 52 s 155 ms (7 minutes ago)   18:1 CRLF÷ UTF-8÷ Context: native-lib-Debug-arm64-v8a

QUESTIONS?