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

- NDK may not be appropriate for Android programmers who need to use only Java code and framework APIs

- NDK can be useful for
    - Squeeze extra performance out of a device
        - Achieve low latency
        - Run computationally intensive applications
    - Reuse C/C++ libraries
    - Port apps between platforms
    - Perform platform-dependent operations which are not handled with Java
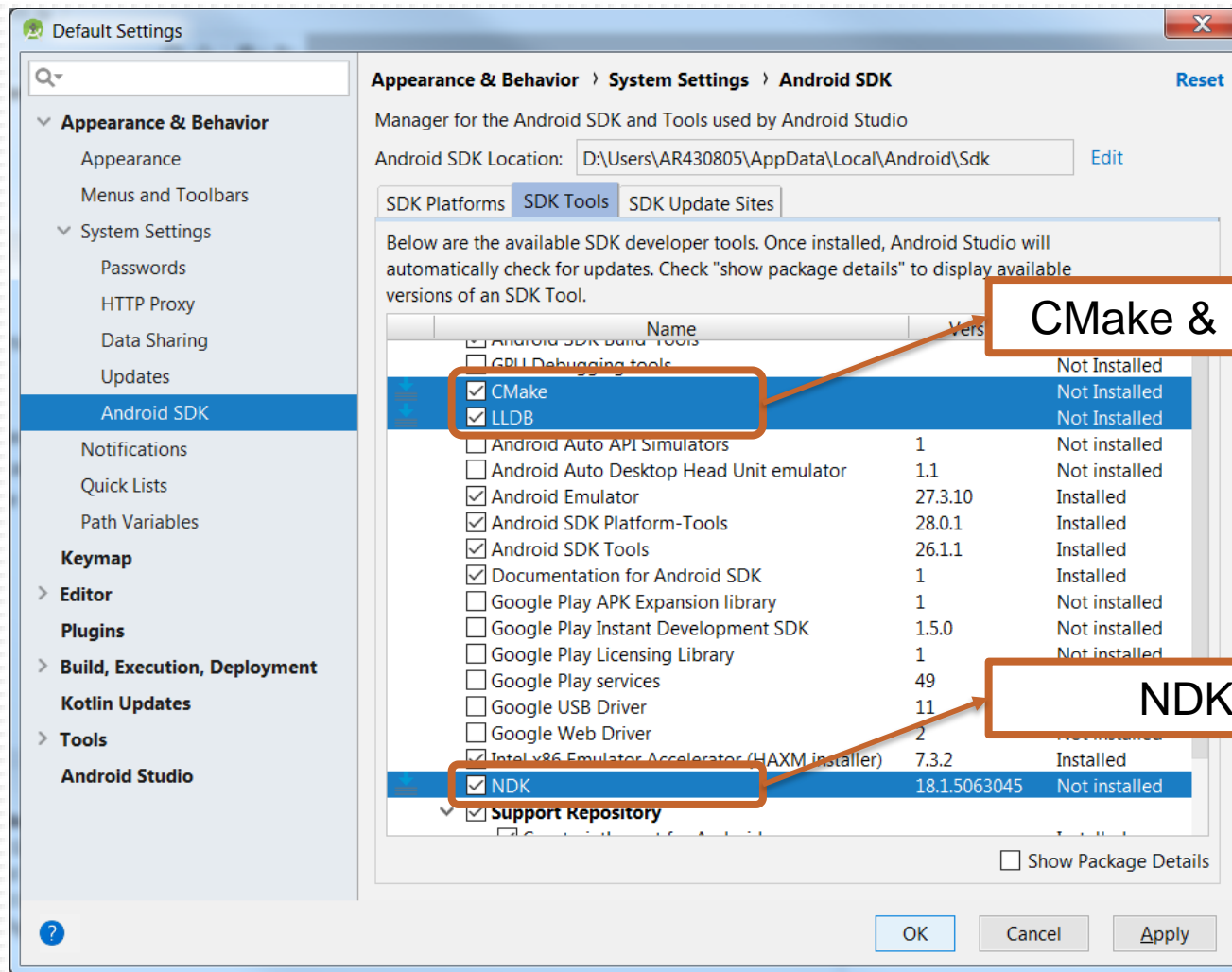
Java Native Interface (JNI)

- Java code can call functions in the native library through the Java Native Interface (JNI)

- JNI defines two key data structures, "JavaVM" and "JNIEnv"

- The JNIEnv provides most of the JNI functions

- JNI allows using native methods to

  - Create, inspect, and update Java objects

  - Call Java methods

  - Catch and throw exceptions

  - Perform runtime type checking

- Refer below link to learn how JNI works

  - https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html

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

2- Add C++ Support
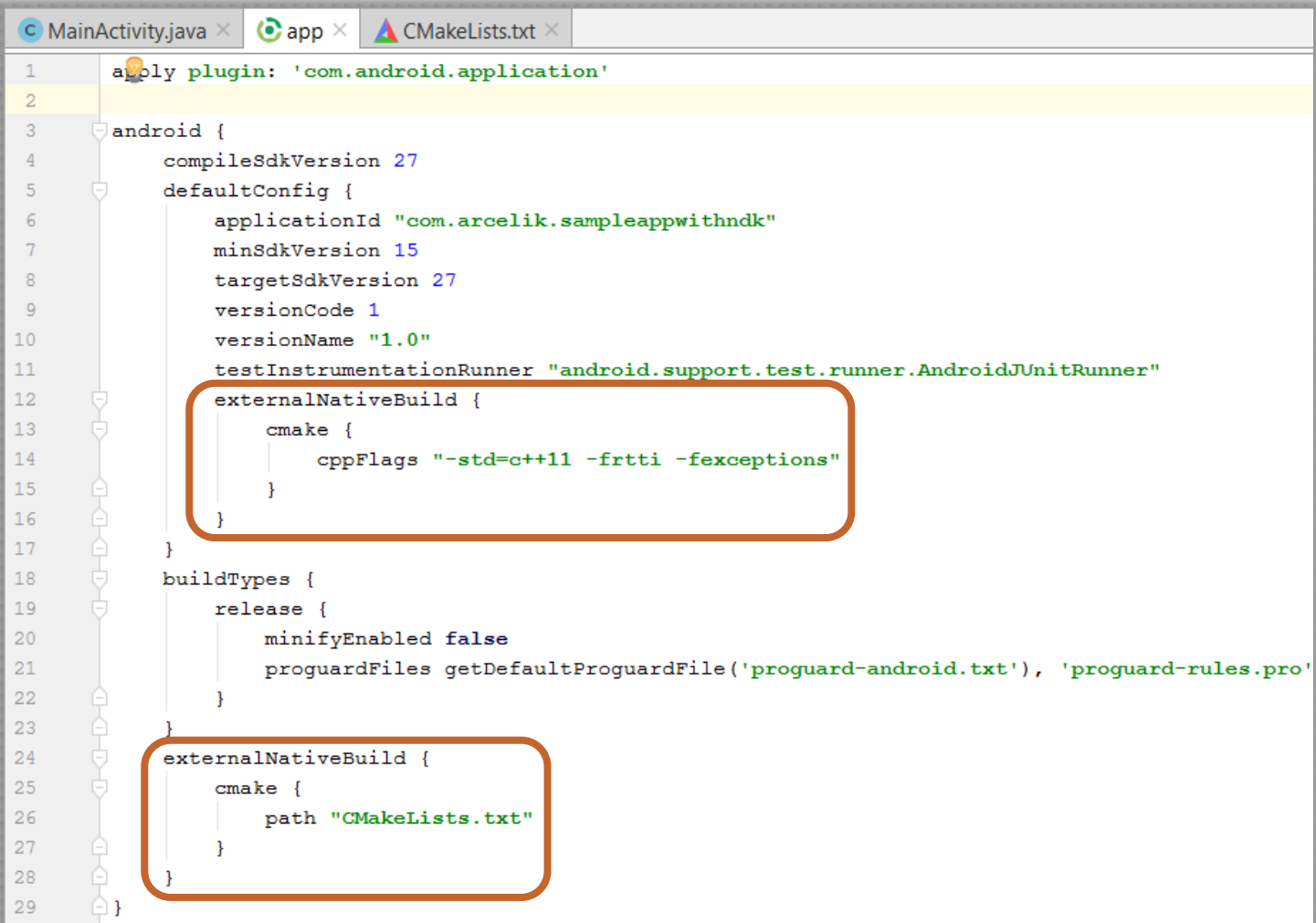
3- Customize C++ Support
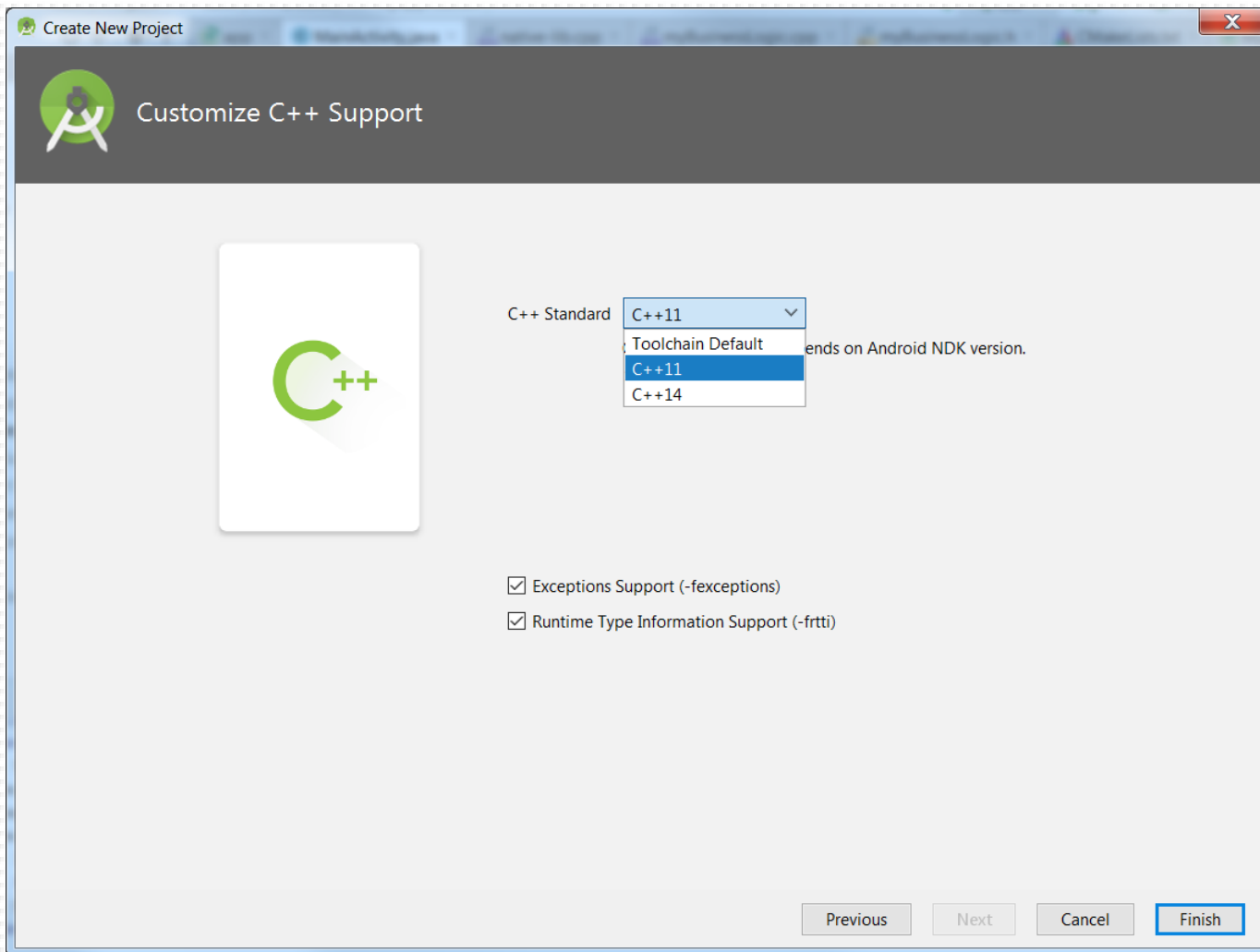
Automatically Generated Application

Gradle Build File
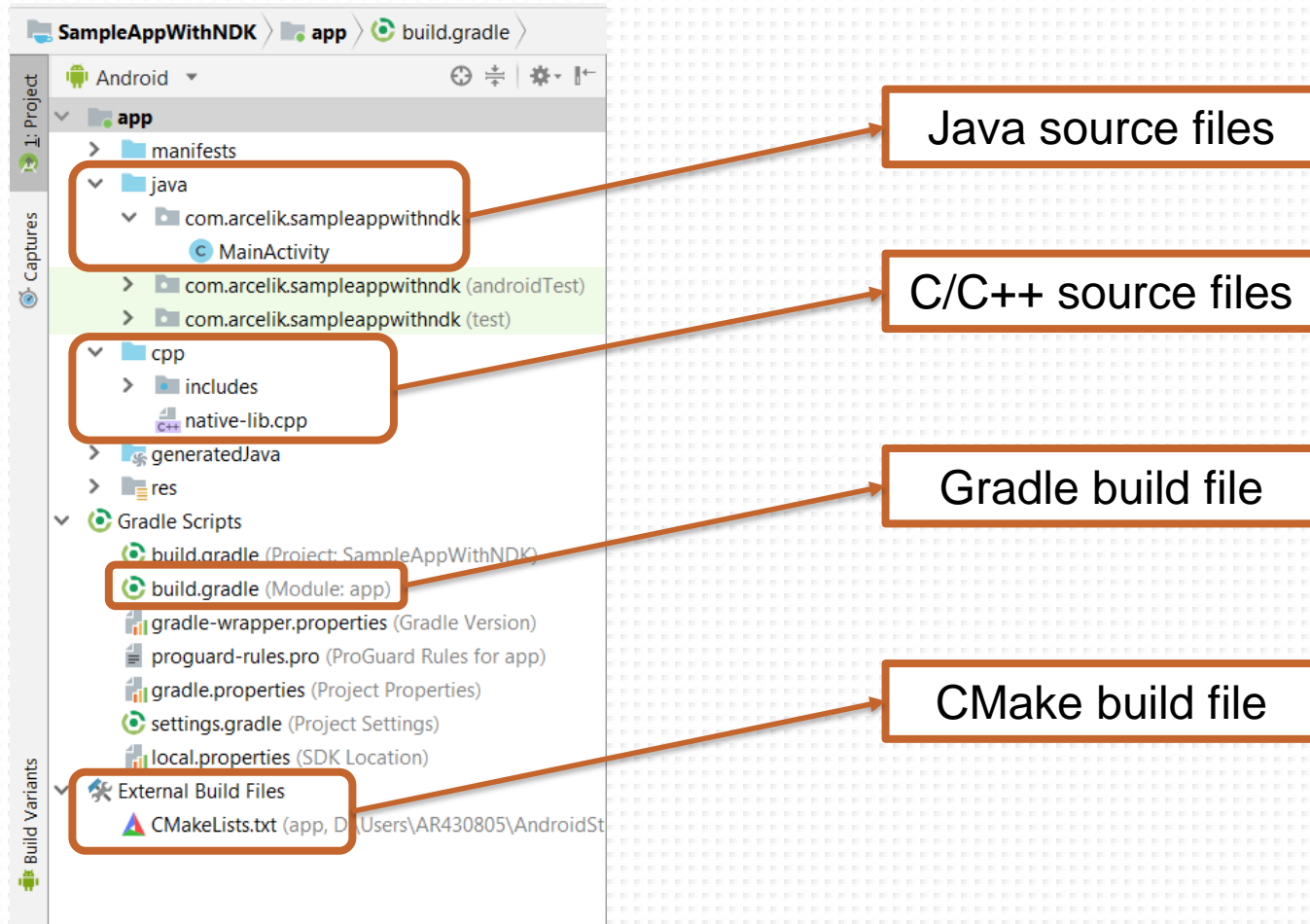
```
  MainActivity.java ×     app ×     CMakeLists.txt ×

1     apply plugin: 'com.android.application'
2
3     android {
4         compileSdkVersion 27
5         defaultConfig {
6             applicationId "com.arcelik.sampleappwithndk"
7             minSdkVersion 15
8             targetSdkVersion 27
9             versionCode 1
10            versionName "1.0"
11            testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
12            externalNativeBuild {
13                cmake {
14                    cppFlags "-std=c++11 -frtti -fexceptions"
15                }
16            }
17        }
18        buildTypes {
19            release {
20                minifyEnabled false
21                proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
22            }
23        }
24        externalNativeBuild {
25            cmake {
26                path "CMakeLists.txt"
27            }
28        }
29    }
```

CMake Build File

```
C MainActivity.java ×    app ×    ⚠ CMakeLists.txt ×

 1    # For more information about using CMake with Android Studio, read the
 2    # documentation: https://d.android.com/studio/projects/add-native-code.html
 3
 4    # Sets the minimum version of CMake required to build the native library.
 5    cmake_minimum_required(VERSION 3.4.1)
 6
 7    # Creates and names a library, sets it as either STATIC
 8    # or SHARED, and provides the relative paths to its source code.
 9    # You can define multiple libraries, and CMake builds them for you.
10    # Gradle automatically packages shared libraries with your APK.
11    add_library( # Sets the name of the library.
12            native-lib
13
14            # Sets the library as a shared library.
15            SHARED
16
17            # Provides a relative path to your source file(s).
18            src/main/cpp/native-lib.cpp)
19
20    # Searches for a specified prebuilt library and stores the path as a
21    # variable. Because CMake includes system libraries in the search path by
22    # default, you only need to specify the name of the public NDK library
23    # you want to add. CMake verifies that the library exists before
24    # completing its build.
25    find_library( # Sets the name of the path variable.
26            log-lib
27
28            # Specifies the name of the NDK library that
29            # you want CMake to locate.
30            log)
31
32    # Specifies libraries CMake should link to your target library. You
33    # can link multiple libraries, such as libraries you define in this
34    # build script, prebuilt third-party libraries, or system libraries.
35    target_link_libraries( # Specifies the target library.
36            native-lib
37
38            # Links the target library to the log library
39            # included in the NDK.
40            ${log-lib})
41
```
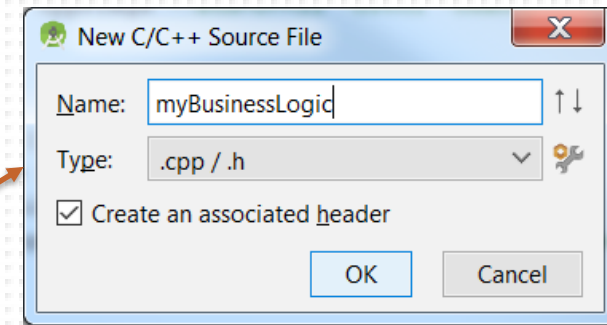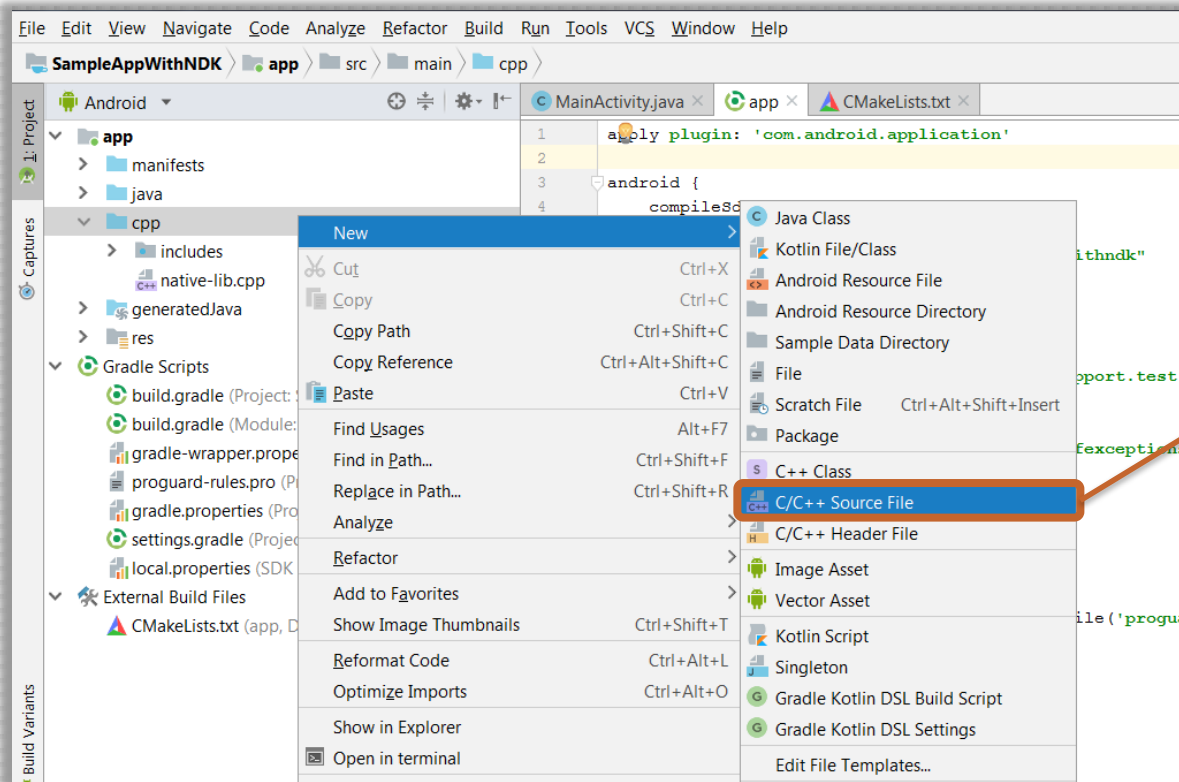
Adding C/C++ Support to Existing Project

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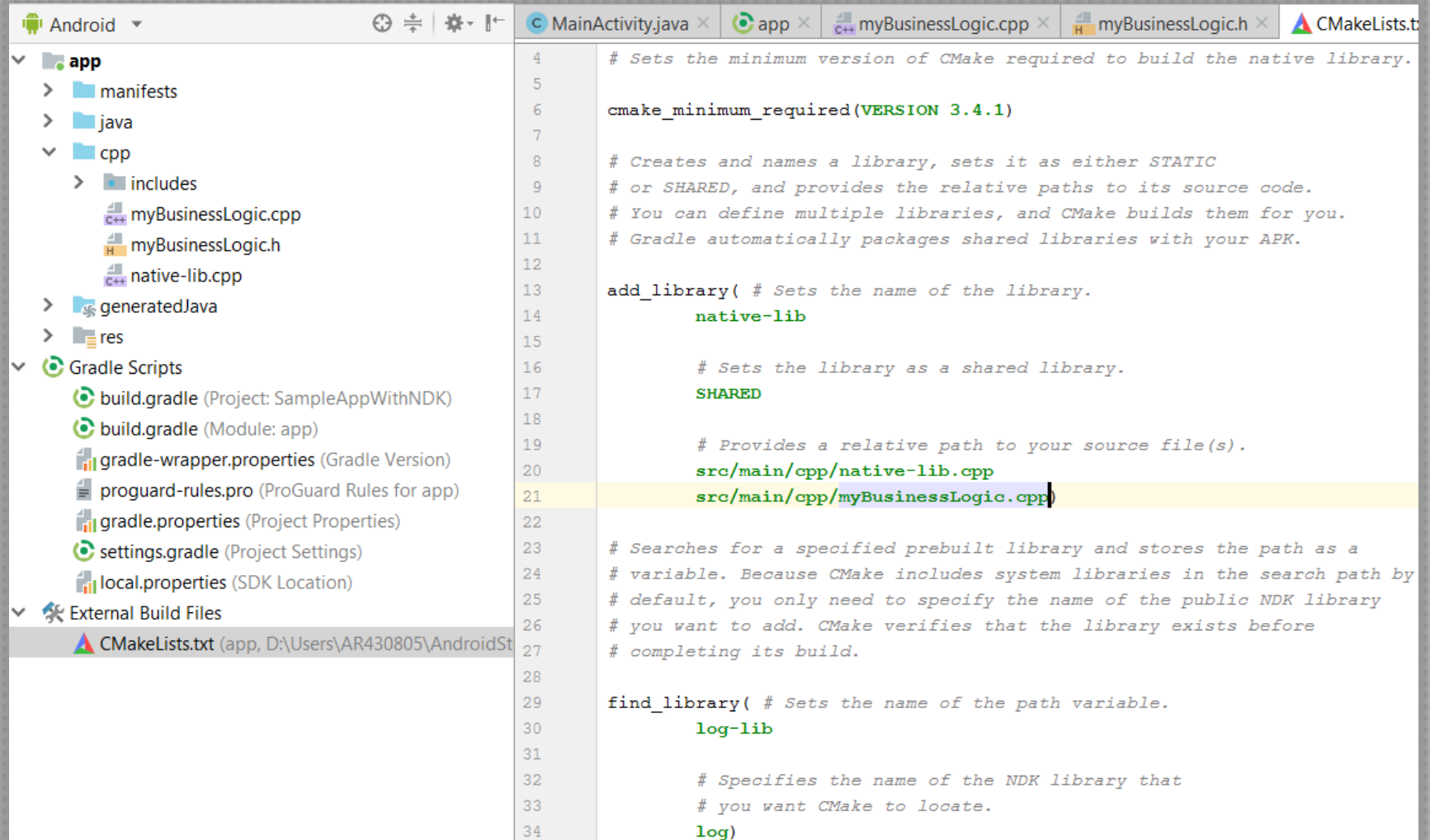
1. Create new C/C++ source files
   - https://developer.android.com/studio/projects/add-native-code#create-sources
2. Configure CMake
   - https://developer.android.com/studio/projects/configure-cmake
3. Link Gradle to your native library
   - https://developer.android.com/studio/projects/gradle-external-native-builds

Adding C/C++ Sources to Android Project

Adding New C/C++ Sources

Configure CMake


```
Android ▾                    ⊕ ÷ | ☼ ▾ |←       C MainActivity.java ✕   C app ✕   myBusinessLogic.cpp ✕   myBusinessLogic.h ✕   ▲ CMakeLists.t

∨  📁 app                                          4    # Sets the minimum version of CMake required to build the native library.
   >  📁 manifests                                 5
   >  📁 java                                       6    cmake_minimum_required(VERSION 3.4.1)
   ∨  📁 cpp                                         7
      >  📁 includes                                8    # Creates and names a library, sets it as either STATIC
         myBusinessLogic.cpp                        9    # or SHARED, and provides the relative paths to its source code.
         myBusinessLogic.h                         10    # You can define multiple libraries, and CMake builds them for you.
         native-lib.cpp                            11    # Gradle automatically packages shared libraries with your APK.
   >  📁 generatedJava                             12
   >  📁 res                                        13    add_library( # Sets the name of the library.
∨  ⊙ Gradle Scripts                                14          native-lib
      build.gradle (Project: SampleAppWithNDK)     15
      build.gradle (Module: app)                   16          # Sets the library as a shared library.
      gradle-wrapper.properties (Gradle Version)   17          SHARED
      proguard-rules.pro (ProGuard Rules for app)  18
      gradle.properties (Project Properties)       19          # Provides a relative path to your source file(s).
      settings.gradle (Project Settings)           20          src/main/cpp/native-lib.cpp
      local.properties (SDK Location)              21          src/main/cpp/myBusinessLogic.cpp
∨  ⚒ External Build Files                          22
      ▲ CMakeLists.txt (app, D:\Users\AR430805\AndroidSt  23    # Searches for a specified prebuilt library and stores the path as a
                                                   24    # variable. Because CMake includes system libraries in the search path by
                                                   25    # default, you only need to specify the name of the public NDK library
                                                   26    # you want to add. CMake verifies that the library exists before
                                                   27    # completing its build.
                                                   28
                                                   29    find_library( # Sets the name of the path variable.
                                                   30          log-lib
                                                   31
                                                   32          # Specifies the name of the NDK library that
                                                   33          # you want CMake to locate.
                                                   34          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                                          Library name

        # Sets the library as a shared library.
        SHARED                                              STATIC | SHARED

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

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

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

```cpp
#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                              ──────►  Target libraries

        # Links the target library to the log library
        # included in the NDK.
        ${log-lib}                              ──────►  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	armeabi-v7a	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
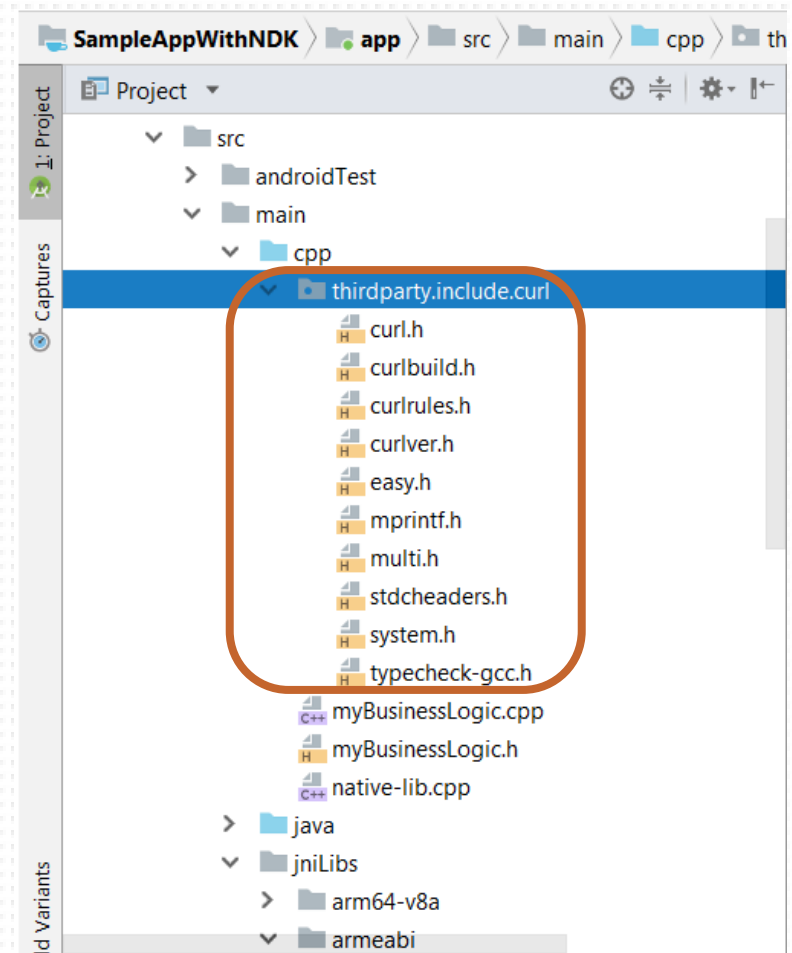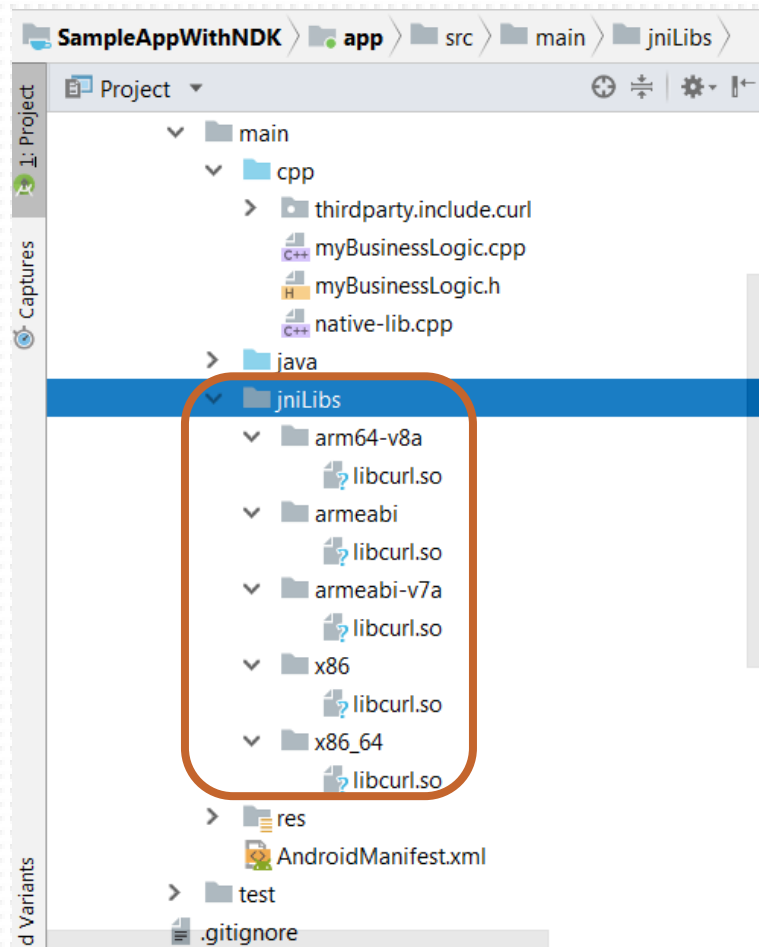
3- Configure CMake

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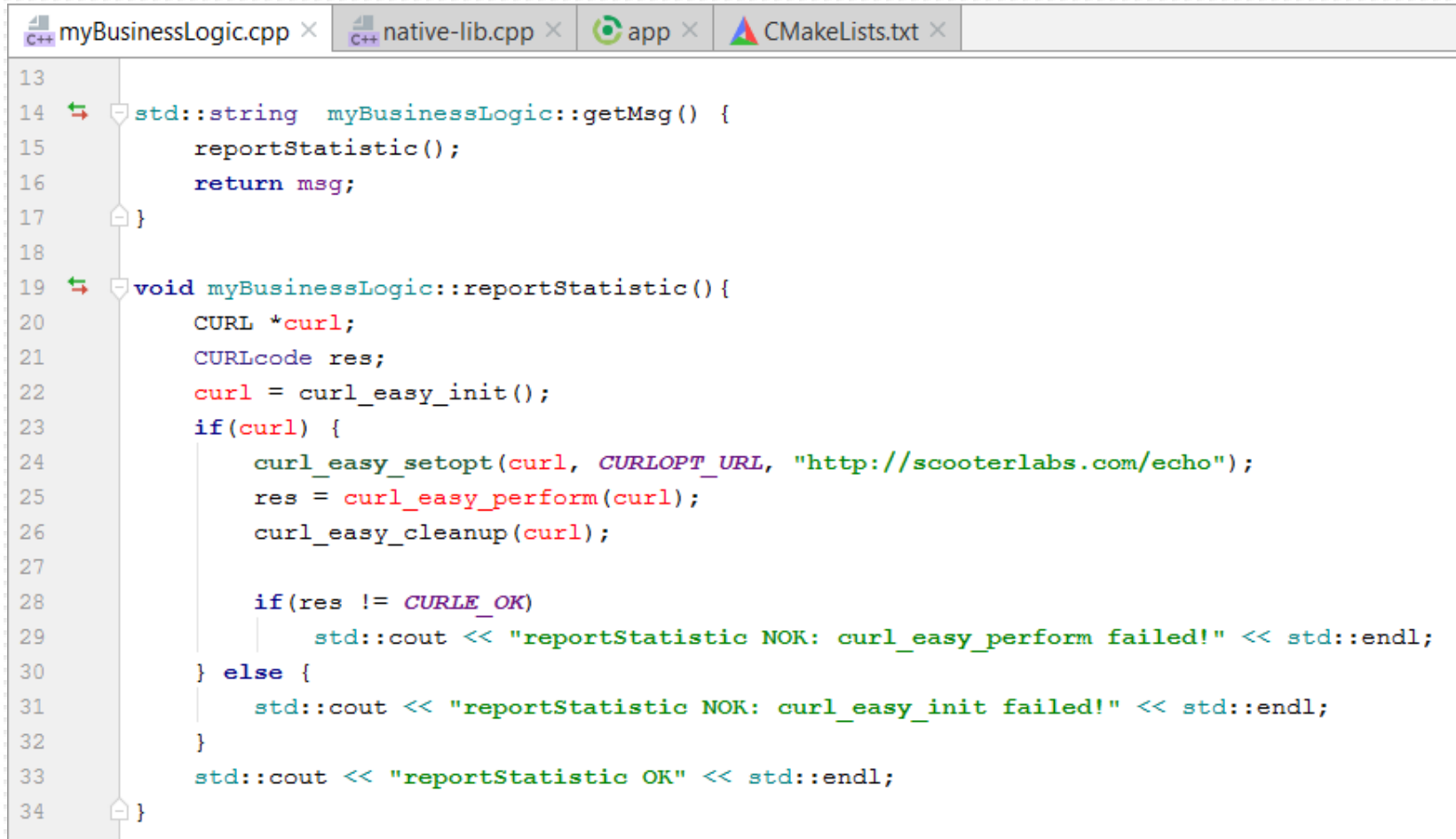
- Link imported library to your target library (native-lib.so)

```
# 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

```
myBusinessLogic.cpp ×    native-lib.cpp ×    app ×    CMakeLists.txt ×

13
14  ⇄  std::string  myBusinessLogic::getMsg() {
15          reportStatistic();
16          return msg;
17      }
18
19  ⇄  void myBusinessLogic::reportStatistic(){
20          CURL *curl;
21          CURLcode res;
22          curl = curl_easy_init();
23          if(curl) {
24              curl_easy_setopt(curl, CURLOPT_URL, "http://scooterlabs.com/echo");
25              res = curl_easy_perform(curl);
26              curl_easy_cleanup(curl);
27
28              if(res != CURLE_OK)
29                  std::cout << "reportStatistic NOK: curl_easy_perform failed!" << std::endl;
30          } else {
31              std::cout << "reportStatistic NOK: curl_easy_init failed!" << std::endl;
32          }
33          std::cout << "reportStatistic OK" << std::endl;
34      }
```

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!

Return type (pointer to a Java string)

```
#include <jni.h>

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

pointer to the VM

pointer to the implicit this object passed from the Java side

Additional arguments added to the Java side function (optional)

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

JNI Functions

- Each function is accessible at a fixed offset through the JNIEnv argument

- The JNIEnv type 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

Using Native Method Arguments

```cpp
#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

```cpp
#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


```
  C MainActivity.java ×    app ×   myBusinessLogic.cpp ×   myBusinessLogic.h ×   native-lib.cpp ×   CMakeLists.txt ×
 1      #include <jni.h>
 2      #include <string>
 3
 4      #include "myBusinessLogic.h"
 5
 6      extern "C" JNIEXPORT jstring JNICALL
 7      Java_com_arcelik_sampleappwithndk_MainActivity_getMessageFromJNI(
 8              JNIEnv *env, jobject thisObj, jboolean appendExclamationMark) {
 9
10          jstring nameMemberOfJavaClass = jstring("unknown");
11          std::string argForCppFunction = "unassigned";
12
13          // Get a reference to this object's class
14          jclass thisClass = env->GetObjectClass(thisObj);
15
16          // Get the Field ID of the instance variables "message"
17          jfieldID findName = env->GetFieldID(thisClass, "name", "Ljava/lang/String;");
18          if (findName != NULL){
19              // Get the object given the Field ID
20              nameMemberOfJavaClass = (jstring)env->GetObjectField(thisObj, findName);
21
22              //get the value of Field ID
23              const char *cStrName = env->GetStringUTFChars(nameMemberOfJavaClass, NULL);
24              if(cStrName != NULL)
25                  argForCppFunction = cStrName;
26          }
27
28          if(appendExclamationMark)
29              argForCppFunction.append("!");
30
31          //we got name parameter from java class. Now ready to run our business logic
32          myBusinessLogic* bl = new myBusinessLogic(argForCppFunction);
33          const char* message = bl->getMsg().c_str();
34          delete bl;
35
36          return env->NewStringUTF(message);
37      }
```

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

1- Load Native Library

- Native libraries are loaded with the System.loadLibrary method

```java
// 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

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

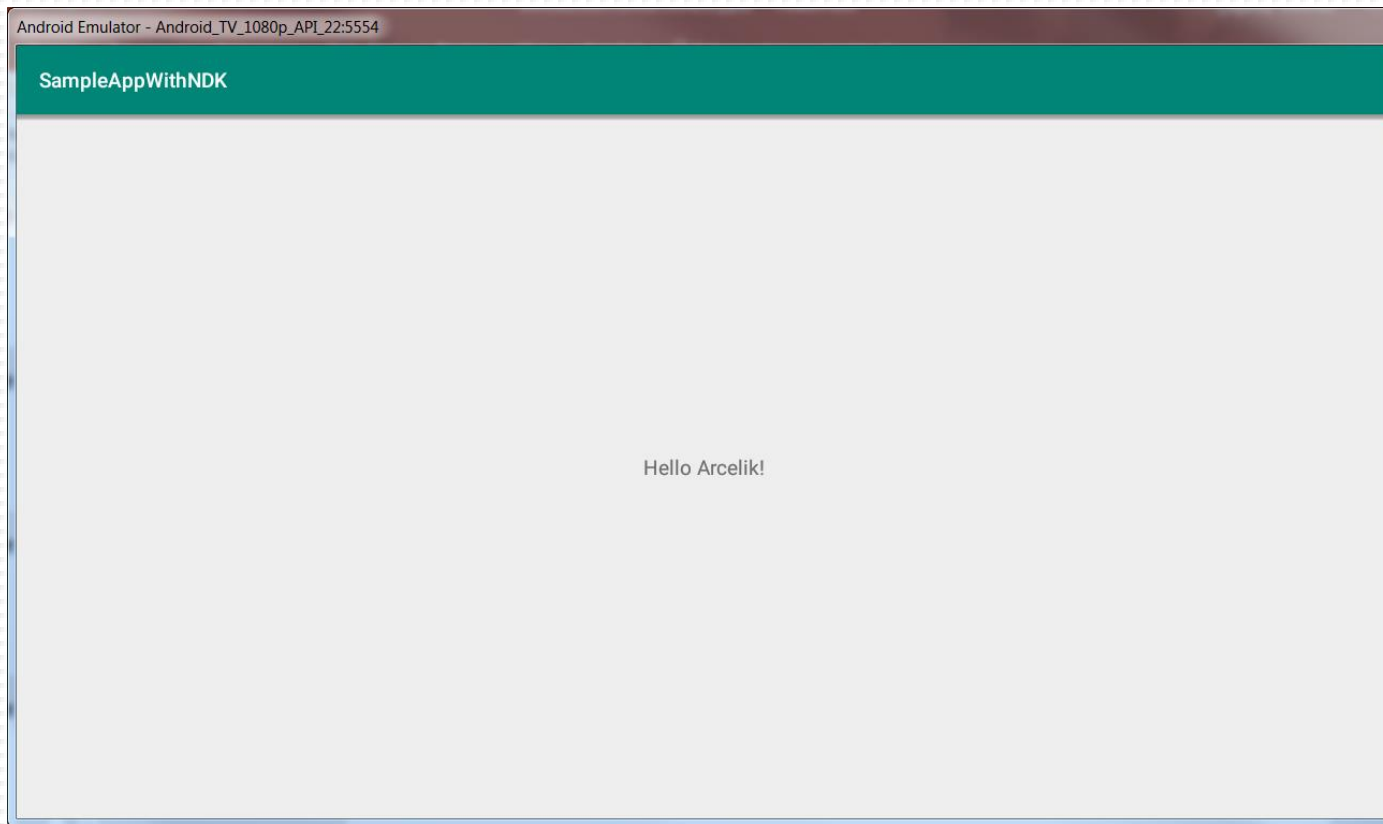# Checking Native Library III

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
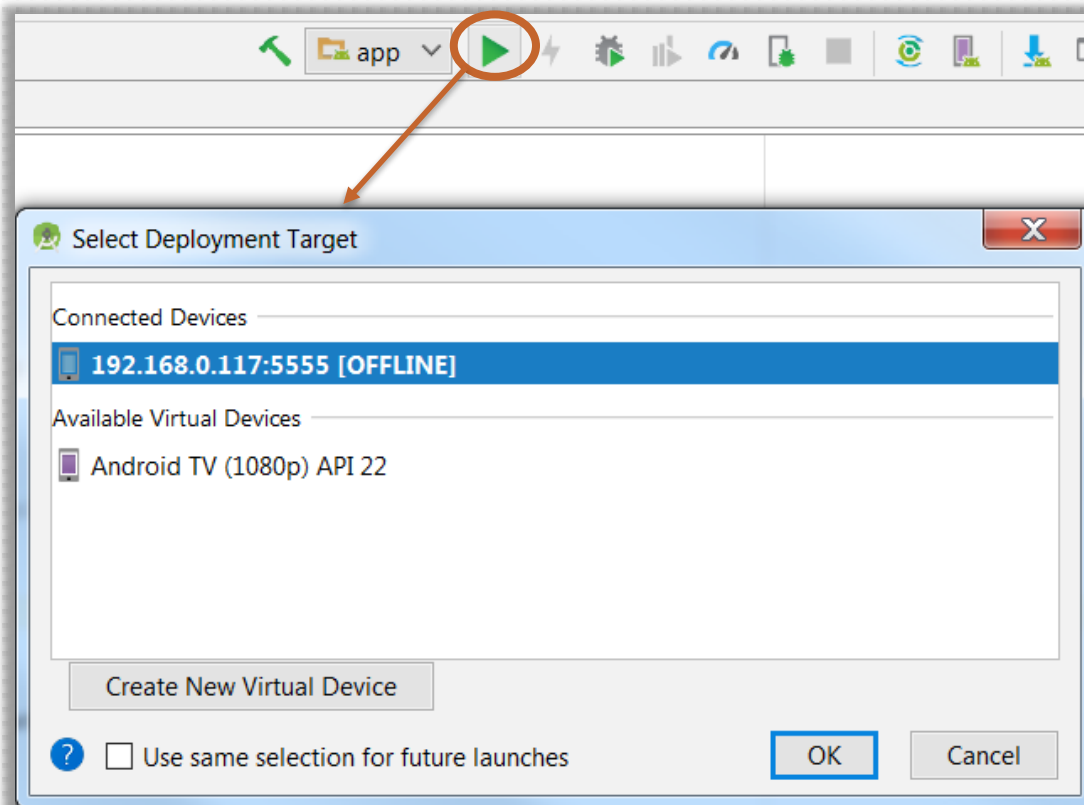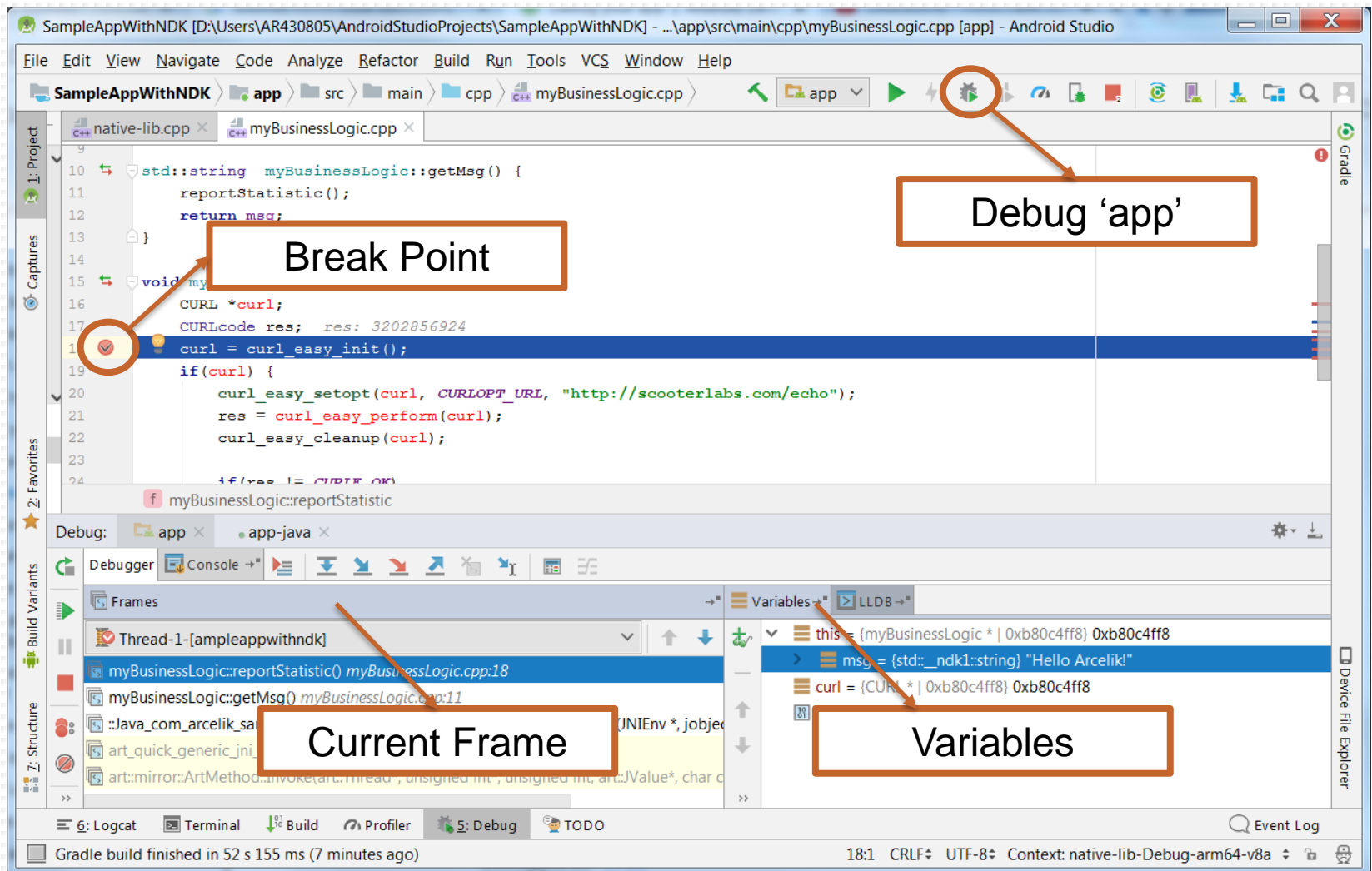
Running Sample App on Fire TV Stick II

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

QUESTIONS?