

Android NDK

JEON YONGTAE

<https://github.com/yongtaii/yongapps>



Android NDK 목차

01

Android SDK & NDK

Android SDK & NDK 설명 / 차이점

02

Android NDK 기능

OpenGL|ES, OpenSL|ES, 센서 조작, 리눅스 콜

03

Android Structure

Android의 구조

04

SDK/NDK 실행 과정

실행 환경, 실행 과정, 관계

05

Android NDK 장/단점

NDK가 가지는 장단점

06

JNI

Java Native Interface

07

Practice

Native Code를 활용한 문자열 표시



Android SDK & NDK

SDK & NDK 구성요소, 차이점



Android SDK & NDK



Android SDK

(Android Software Development Kit)

안드로이드를 개발하는데 필수 개발 kit.

애플리케이션 프레임워크 / 에뮬레이터 /
개발 툴(dx, adb 등)이 포함됨.

IDE(Android Studio)를 통해 개발하
는 방법이 일반적이다.



Android NDK

(Android Native Development Kit)

안드로이드 SDK와 함께 이용되는 개발
kit.

NDK를 이용해 앱 일부 또는 전부를
C/C++로 만들 수 있다.

안드로이드 NDK로 만들어진 실행 바이
너리는 네이티브 코드(cpu가 직접 이해
할 수 있는 코드, 기계어)가 된다.

헤더 라이브러리/도큐먼트 등이 포함.

Android SDK & NDK



Android SDK (Android Software Development Kit)

Applicaion Framework

Emulator

Development Kit
(dx, adb etc ..)

자바 개발환경, 에뮬레이터, 개발 툴 등 개발에 필요한 모든 요소가 구비되어 있다.



Android NDK (Android Native Development Kit)

개발 툴

Compiler

Linker

Header library

Document

툴체인, 라이브러리 등 C/C++에 필요한 파일만 있다.



Android NDK 의 기능



ANDROID NDK의 기능들



1. 3D그래픽스 OpenGL|ES

OpenGL|ES : 스마트폰이나 가정용 게임 등에 이용되는 임베디드용 3D 그래픽 라이브러리.

3D 그래픽 처리는 고속 연산을 필요 하므로, 자바코드 보다는 네이티브 코드가 빠르고 정교하다.



2. 사운드 OpenSL|ES

OpenSL|ES : 사운드의 녹음과 재생이 가능하고, 음원을 BGM 으로 재생하거나 효과음 재생이 가능하다.

음악 재생 시, 잔향 음과 같은 효과를 넣어서 재생이 가능하다. 예를 들어 동굴 등의 게임 장면에 잔향을 넣을 수 있다.



3. 센서, 터치패널

센서나 터치패널을 이용해 사용자로부터 입력을 받는다. 센서나 터치패널을 통한 입력은 애플리케이션을 조작하는데 반드시 필요한 요소다.



ANDROID NDK의 기능들



4. Assets 폴더로부터 데이터 입력

Assets 폴더로부터 데이터를 가져올 수 있다. Assets에 여러 종류의 데이터(캐릭터 데이터, 텍스처 데이터, 음악 데이터 등)을 넣어두고, 앱에 포함 할 수 있다.

리소스 데이터 처럼 파일을 추가할 때마다 빌드할 필요가 없고, 대량의 데이터를 관리할 때 매우 유용한 방법이다.



5. 리눅스의 시스템 콜

안드로이드 리눅스 커널 기반이고, 그 위에 프레임워크, Dalvik Vm이 있다. 그래서 Android는 하드웨어 제어할 때 거의 리눅스 커널을 이용한다. 안드로이드 SDK에서는 이러한 리눅스 커널의 API는 거의 Application 프레임 워크에 의해 은폐되어 있다. 하지만, NDK에서는 리눅스 커널의 시스템 콜을 직접 사용할 수 있다.





Android Structure

Application / Framework / Runtime / Library / HAL / kernal



Android Structure



Android Structure

Application

Application Framework

Library

Android Runtime

Hardware Abstraction Layer

Linux Kernal

Android Structure

Android Structure

Application

Application Framework

Library

Android Runtime

Hardware Abstraction Layer

Linux Kernal

Application

Application Framework

Button 이나 TextView 같은 안드로이드 애플리케이션을 동작시키는 데 필요한 라이브러리가 포함됨

Library

OpenGL|ES , SQLite 등 다수 미들웨어

Android Runtime

안드로이드 애플리케이션을 동작시키기 위한 실행환경.
Dalvik VM : 자바로 만든 안드로이드 애플리케이션이 CPU 종류에 상관없이 동작 가능하게 함.

Hardware Abstraction Layer

하드웨어를 추상화한 계층으로 HAL에 의해 정의된 API를 통해 정보를 가져와 안드로이드에 탑재한 하드웨어를 제어할 수 있다

Linux Kernal

일반적인 리눅스 커널에 안드로이드 환경에 맞게 커스터마이징 된 형태



SDK / NDK 실행 과정

실행 환경 / 파일 생성 과정



Java Application 실행 과정

자바로 만든 Application

Dalvik VM이 해석 할 수 있는 바이트코드가 포함된 실행파일.
Application이 실행 될 때, 이 실행파일은 Dalvik VM 안에 있는 컴파일러 (VIT)에 의해 native code 로 변환 된 후, CPU에 의해 실행된다

02

Dalvik VM의 장점

Dalvik VM이 동작하는 환경이라면 ARM 이나 x86 같은 CPU 아키텍처와 관계없이 Application을 실행할수 있다

03

실행 속도

'JIT 컴파일러에서 실행한다'라는 말은 Application 실행 속도가 느려진다고 생각할 수 있지만, Dalvik VM은 native Code로 변환된 실행파일을 캐시하는 등 Application이 고속으로 실행되는 구조를 가진다

01

04



NDK Application 실행 과정

NDK Build -> .SO 파일

안드로이드 NDK를 이용할 경우 C/C++로 만든 코드는 컴파일러와 링커를 거쳐 최종적으로 모듈(.so 파일)로 만들어 진다.

.SO 파일 호출

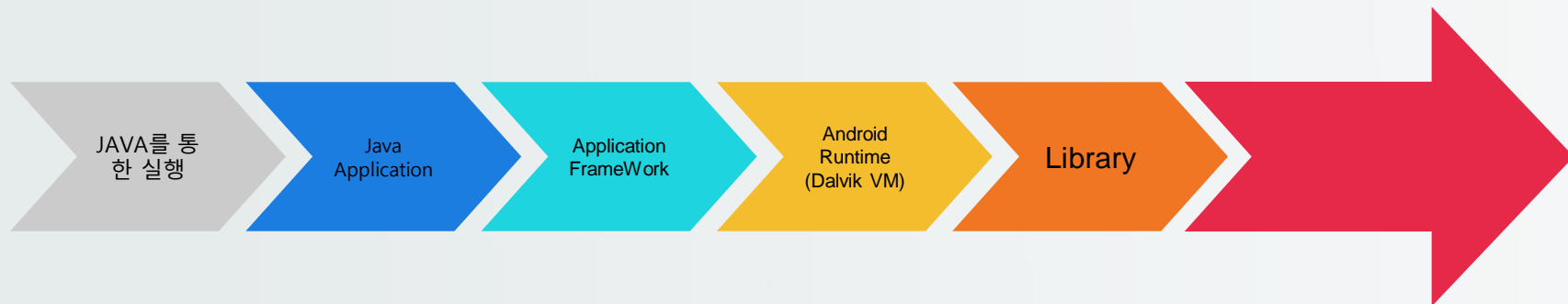
.so 파일은 공유라이브러리 이므로, 단독으로 동작이 불가능하다. 자바 코드(.dex)로 부터 호출되어 이용된다. 이처럼 자바 코드가 추가되고, C/C++ 파일이 종이 되어 실행된다.

Build 순서

실제로 C/C++ 코드를 빌드한 후에 자바 코드를 빌드해야한다. 자바 코드를 빌드 할때 모듈(.so)을 이용해 실행파일(.apk)을 생성하기 때문이다.



실행파일 생성까지 과정 비교





NDK 장단점

Android NDK Strength / Weakness



Android NDK 장점

Native Code 사용 가능

Native Code (CPU 가 이해할 수 있는 코드, 기계어) 이용이 가능하여 CPU를 직접 제어할 수 있다.

C/C++ 라이브러리, 코드 이용가능

C/C++ 소프트웨어 자산을 가진 기업, 업계에서는 이를 활용할 수 있다.

고속 처리

프로그램을 어떻게 만드냐에 따라 네이티브로 만드는 편이 훨씬 빠른 속도로 데이터 처리가 가능하다. 이미지/ 게임 등 고속처리가 필요한 부분에서 사용된다.



Android NDK 단점

실행파일이 CPU에 의존

자바로 만들면 Dalvik VM 이 CPU 아키텍처 차이를 해결해 주므로 개발자나 사용자는 하드웨어 차이를 신경 쓸 필요가 없었다. 하지만, 안드로이드 NDK를 이용한 애플리케이션은 실행파일을 CPU마다 준비해야 한다. 따라서, 사용자는 자신의 하드웨어에 따라 실행 파일을 선택/실행 해야 한다.



디버깅의 어려움

자바 디버깅 환경에 비해서 디버깅이 어렵다.

예를들어, 접근해서는 안될 메모리 영역에 접근할때 자바 앱이라면 "Out Of Range" 등의 예외를 발생시키며 종료된다. 그러나, 네이티브 코드에서는 자바와 같은 예러가 전혀 발생하지 않고 앱이 종료된다.

소스코드의 어떤 행에서 예러가 발생했는지도 표시되지 않는다.



JNI

Java Native Interface






JNI

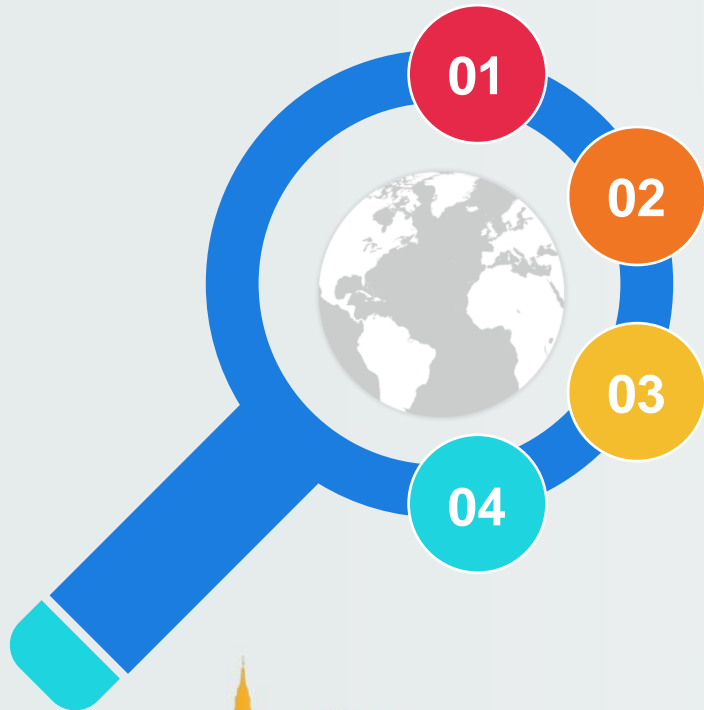
Java Native Interface

자바에서 C/C++ 함수를 호출하거나, 반대로 C/C++ 에서 자바의 클래스와 메서드를 사용하게 해주는 인터페이스.
JNI를 이용하면 자바에서 처리하는 특정 부분을 C/C++로 만들 수 있다. 예를들어 사용자 인터페이스를 자바로 만들고, 실제 이미지 처리하는 부분을 C/C++로 만들면 모든 과정을 자바로 처리할 때 보다 훨씬 빠른 속도로 영상처리하는 애플리케이션이 된다.



JNI 의 규약 (Memory)

자바와 C/C++의 문법은 다르다. 동작환경도 다르다. (VM에서 동작, CPU에서 직접 동작) 이런 차이를 JNI가 어느정도 해결해 주지만, 서로 다른 상호 연결을 하려면 여러 규약을 지켜줘야 한다.



1. 메모리 관리

자바와 C/C++의 큰 차이점은 메모리 관리하는 방법이다.
자바로 만든 애플리케이션의 메모리는 전적으로 달빅 VM이 일괄적으로 관리하며, 개발자는 메모리 관리에 신경 쓸 필요가 없다.
하지만, C/C++에서는 할당한 메모리를 모두 개발자가 책임지고 관리한다.
그러므로 메모리 관리할 때는 C/C++로 프로그래밍 할때처럼 메모리 할당/해제를 명확히 해줘야 한다.

JAVA : 가바지 컬렉션이 Dalvik VM에 있어 가비지 컬렉션이 사용하지 않는 메모리를 해제시켜줌.

C/C++ : 할당한 메모리 영역을 해제하지 않았을 경우, 메모리 누수가 발생하게 됨. 결국 애플리케이션이 강제 종료 됨.

JNI 의 규약 (타입 선언)



2. 타입 선언

자바에서 건네받은 변수는 C/C++에서 사용할때 자바에서 선언한 변수의 타입에 따라 C/C++에서 변수를 사용하는 방법이 다르다.

	JAVA	JNI(C)
01	boolean	jboolean
02	byte	jbyte
03	char	jchar
04	short	jshort
05	int	jint
06	long	jlong
07	float	jfloat
08	double	jdouble
09	object	jobject

JNI 의 규약 (메서드 정의)



3. 메서드의 정의

호출하는 JAVA 쪽과 호출되는 C/C++ 쪽은 JNI 규정에 따라 정의 되어야 한다.

1) JAVA Side

- 호출하는 함수를 선언할 때는 Native를 추가한다.
- loadLibrary()를 이용해 호출할 라이브러리를 지정한다.

```
Public class MainActivity extend Activity{  
    public native String stringFromJNI(); // native를 붙임  
    static{  
        System.loadLibrary("native-li"); // native-li.so를 로드  
    }  
}
```

2) C/C++ Side

- 함수명은 Java로 시작해야 하며, 패키지명, 클래스명, 메서드명을 연결해서 기술해야 한다.
- 선두에 설정하는 2개의 인수는 JNIEnv 타입과 jobject 타입으로 한다.

```
#include<jni.h>  
jint Java_com_example_jeon_jniproject_MainActivity_addVals(JNIEnv* env,  
    jobject /* this */,jint a, jint b) {  
    return a+b;  
}
```

JNI 의 규약 (메서드 정의)



3. 메서드의 정의

2) C/C++ Side

- 함수명은 Java로 시작해야 하며, 패키지명, 클래스명, 메서드명을 연결해서 기술해야 한다.
- 선두에 설정하는 2개의 인수는 JNIEnv 타입과 jobject 타입으로 한다.

```
#include<jni.h>
jint Java_com_example_jeon_jniproject_MainActivity_addVals(JNIEnv* env,
    jobject /* this */,jint a, jint b) {
    return a+b;
}
```

- jni.h : JNI를 이용하는데 필요한 함수와 구조체가 정의되어 있음.
- 함수명 : Java_패키지명_클래스명_메서드명 (밑줄 이용)
- JNIEnv* 변수, jobject 변수 : JNI로 부터 건네받는 변수로 JNI에서 선언한 멤버 함수는 반드시 이 두개의 변수를 인수로 받아야 한다.
- JNIEnv : JNI의 환경정보가 들어있다. Env 변수를 통해 자바츠 실행정보를 취득해 설정함.
- jobject : 이 함수가 포함된 클래스 정보가 들어있다.

JNI 의 규약 (NDK Build Tool)



1. CMake

Android Studio의 기본 빌드 툴, CMake를 권장
Android, Linux, Windows, IOS 등 모든 타겟이서 빌드 가능.
CMakeLists.txt 파일 필요.
Android Studio 2.2 이상에서 지원
so 파일 생성 이름 : liblib-name.so

2. NDK-Build

legacy 프로젝트들이 아직 많아 Android Studio에서 NDK-Build를 지원
Android.mk, Application.mk 파일 필요.

JNI 의 규약 (CMakeLists.txt)



4. CMakeLists.txt 파일

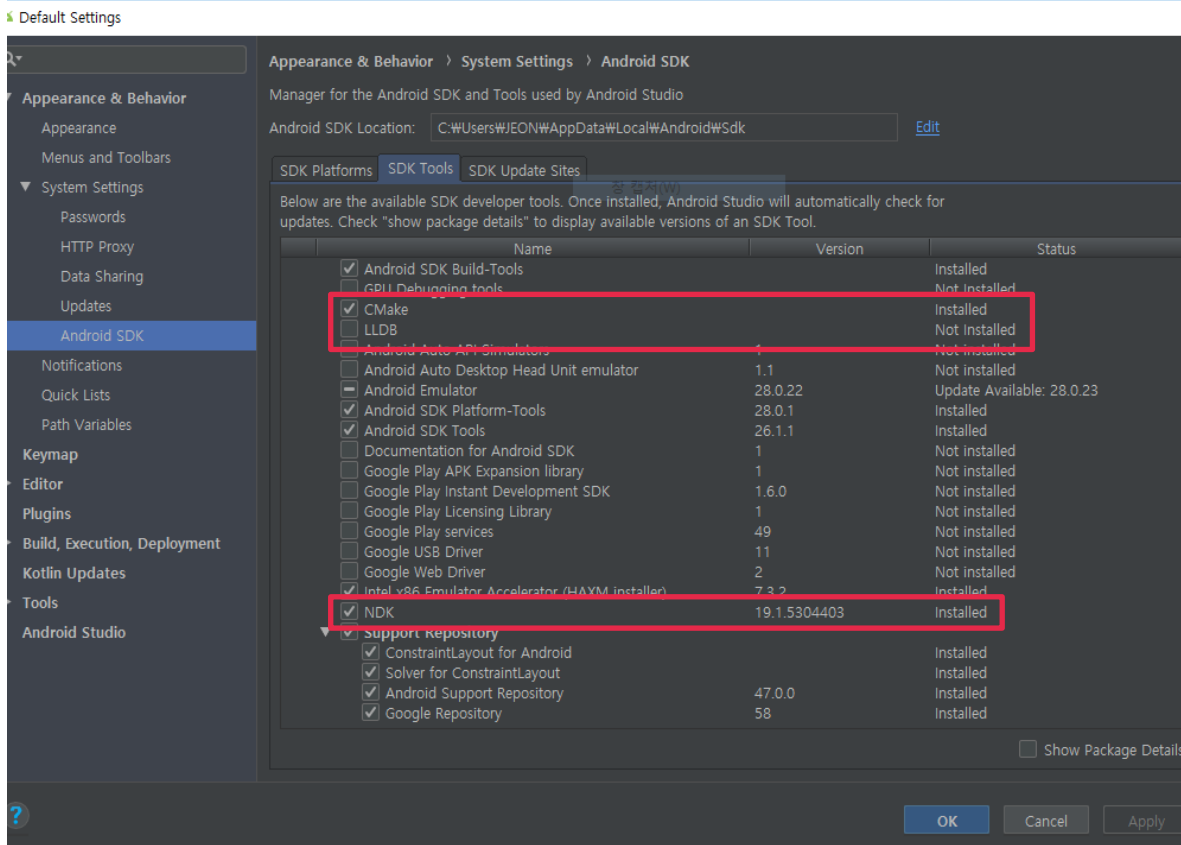
- 1) cmake_minimum_required(VERSION 3.4.1)
 - 사용할 CMake의 최소 버전 설정
- 2) add_library(native-lib
 SHARED
 src/main/cpp/native-lib.cpp)
 - 라이브러리 이름 : native-lib
 - 결과물 이름 : libnative-lib.so
 - 소스 경로 코드 : src/main/cpp/native-lib.cpp
- 3) include_directories(src/main/cpp/include/)
 - 소스 코드에서 사용하는 헤더 파일이 있는 디렉토리 설정
 - 설정해 주지 않아도 헤더 파일이 인식 됨.
- 4) find_library(log-lib
 log)
 - 이미 존재하는 안드로이드 NDK 기본 라이브러리 사용



Practice

Native Code를 활용한 문자열 출력





실습

Step 1 Install NDK

Install CMake

CMake Build 사용을 위해 CMake
를 설치한다

Install NDK

NDK를 설치한다

Intall LLDB

LLDB를 디버깅을 위해 설치한다

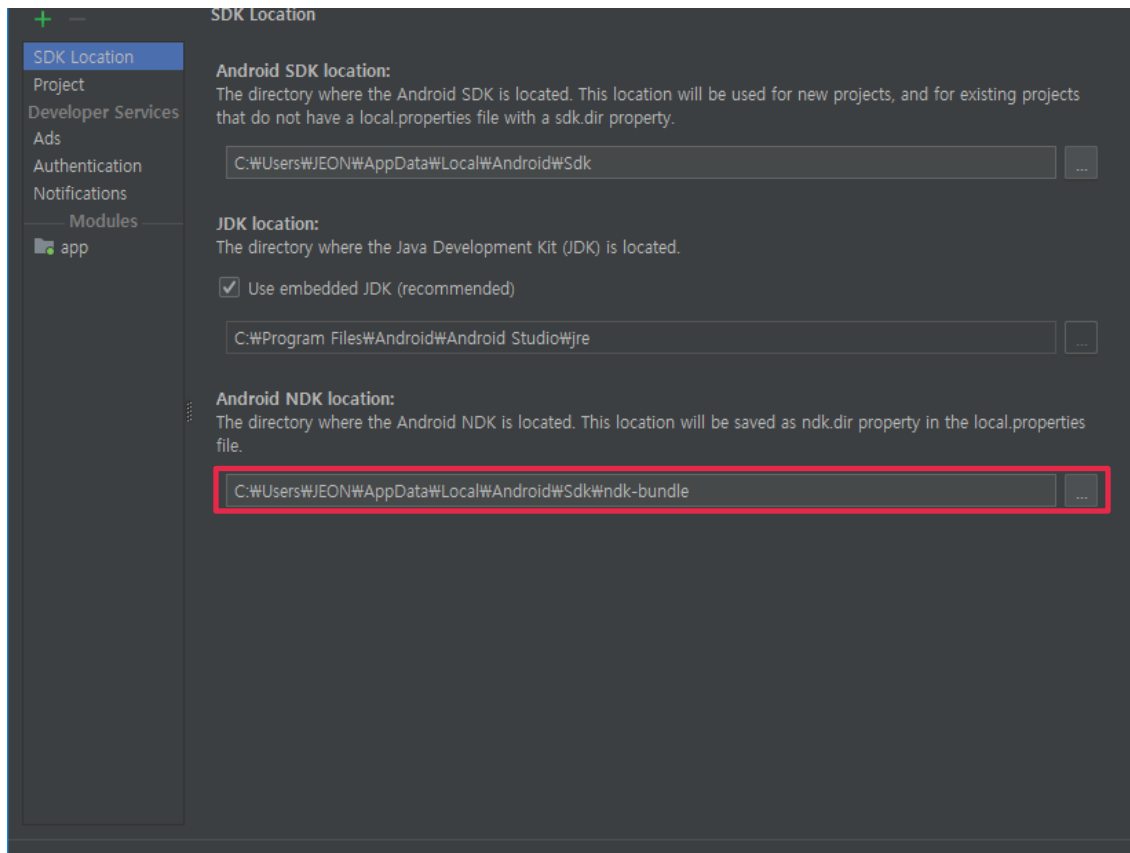
실습

Step 2

Set NDK Location

Set NDK Location

Project Structure Setting에서 ndk
설치 위치를 set 한다



Create New Project

Create Android Project

Application name
My Application

Company domain
jeon.example.com

Project location
C:\Users\WJEON\AndroidStudioProjects\MyApplication2

Package name
com.example.jeon.myapplication Edit

☒ Include C++ support
☐ Include Kotlin support

Previous **Next** Cancel Finish

실습

Step 3

Create Project

Check Include C++ support
C++ support 옵션을 체크 한다

Click Next Button
기존 프로젝트 생성과 동일하게
Next 버튼으로 프로젝트를 생성

실습

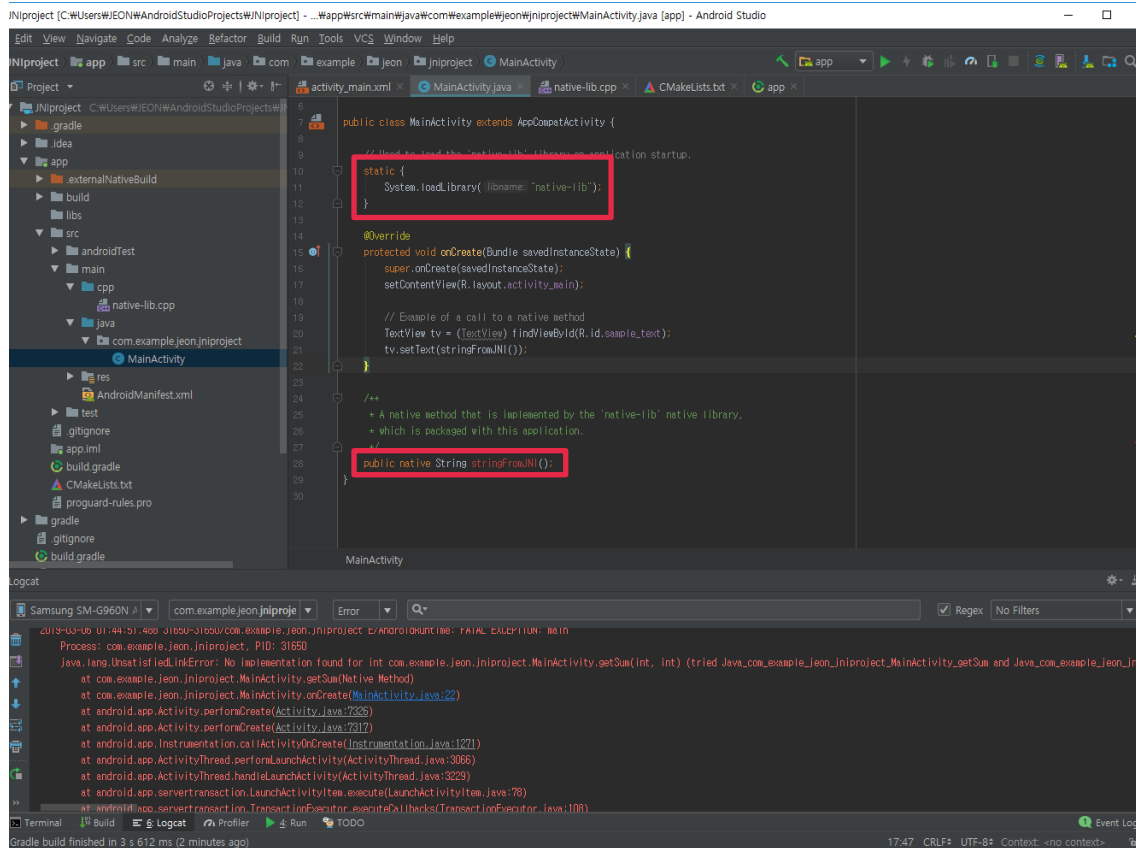
Step 4 MainActivity

System.loadLibrary(native-lib)

System.loadLibrary를 통해
'native-lib'라는 이름의 native
library를 load 한다

**public native String
stringFormJni()**

호출할 함수 선언 시 native 이름
을 붙여 선언한다



실습

Step 4

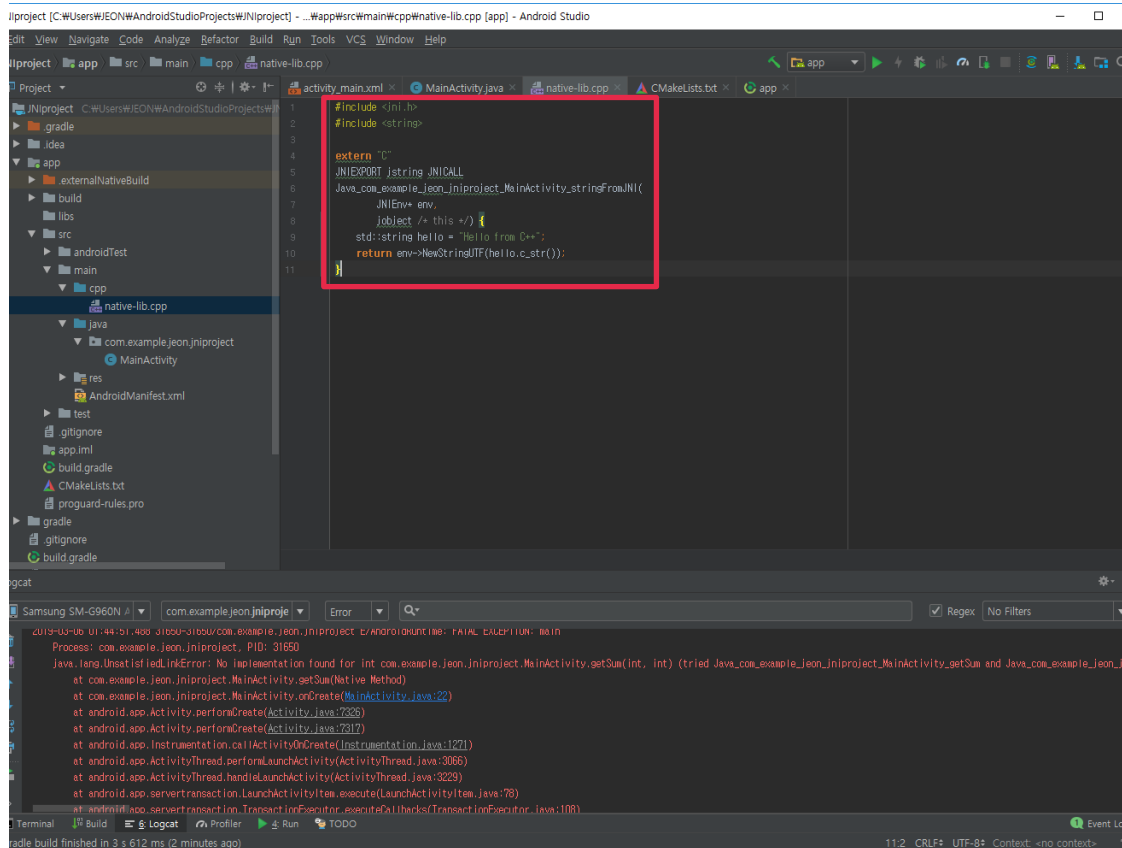
C++

native code

native code 함수를 작성

Java_com_example_jeon_jniproj
ect_mainAcitvty_stringFormJNI

함수명 : Java_패키지명_클래스명
_메서드명



```
1 #include <jni.h>
2 #include <string>
3
4 extern "C"
5 JNIEXPORT jstring JNICALL
6 Java_com_example_jeon_jniproject_MainActivity_stringFromJNI(
7     JNIEnv* env,
8     jobject /* this */) {
9     std::string hello = "Hello from C++";
10    return env->NewStringUTF(hello.c_str());
11 }
```

Logcat output:

```
2019-03-06 01:44:51.400 31600-31600/com.example.jeon.jniproject E/AndroidRuntime: FATAL EXCEPTION: main
Process: com.example.jeon.jniproject, PID: 31600
java.lang.UnsatisfiedLinkError: No implementation found for int com.example.jeon.jniproject.MainActivity.getSum(int, int) (tried Java_com_example_jeon_jniproject_MainActivity_getSum and Java_com_example_jeon_jniproject_MainActivity_getSum(Native Method))
at com.example.jeon.jniproject.MainActivity.getSum(Native Method)
at com.example.jeon.jniproject.MainActivity.onCreate(MainActivity.java:22)
at android.app.Activity.performCreate(Activity.java:7326)
at android.app.Activity.performCreate(Activity.java:7317)
at android.os.Bundle.run(Bundle.java:1271)
at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:3066)
at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:3229)
at android.app.servertransaction.LaunchActivityItem.execute(LaunchActivityItem.java:78)
at android.app.servertransaction.TransactionExecutor.execute(TransactionExecutor.java:108)
at android.os.HandlerThread.run(HandlerThread.java:61)
Gradle build finished in 3 s 612 ms (2 minutes ago)
```


실습

Step 5 CMakeLists

check CMakeLists

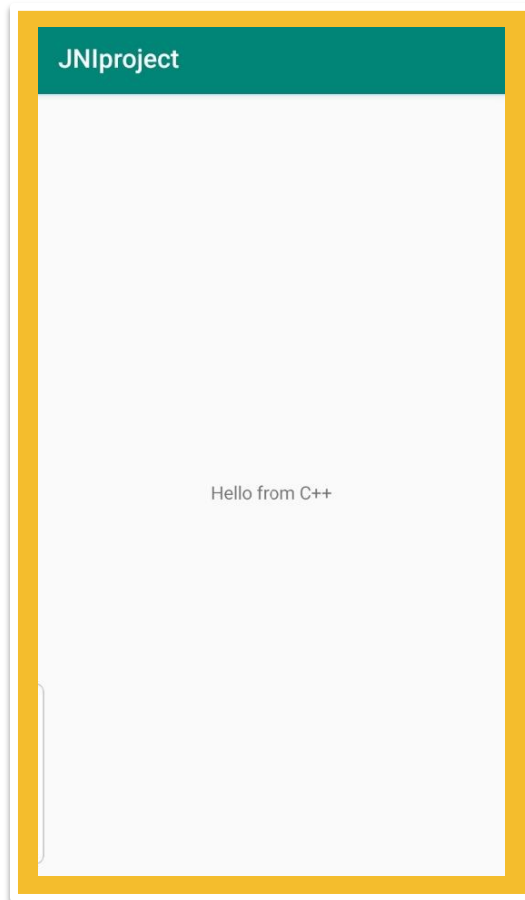
library name 과 source file name
확인

```
ject] - ...Wapp#CMakeLists.txt [app] - Android Studio
tor Build Run Tools VCS Window Help

activity_main.xml MainActivity.java app .gitignore android.toolchain.cmake ExampleCpp.cpp ExampleCpp.h native-lib.cpp ExampleInstrume

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
6 cmake_minimum_required(VERSION 3.4.1)
7
8 # Creates and names a library, sets it as either STATIC
9 # or SHARED, and provides the relative paths to its source code.
10 # You can define multiple libraries, and CMake builds them for you.
11 # Gradle automatically packages shared libraries with your APK.
12
13 add_library( # Sets the name of the library.
14             native-lib
15
16             # Sets the library as a shared library.
17             SHARED
18
19             # Provides a relative path to your source file(s).
20             src/main/cpp/native-lib.cpp
21             src/main/cpp/ExampleCpp.cpp
22             )
23
24 # Searches for a specified prebuilt library and stores the path as a
25 # variable. Because CMake includes system libraries in the search path by
26 # default, you only need to specify the name of the public NDK library
27 # you want to add. CMake verifies that the library exists before
28 # completing its build.
29
30 find_library( # Sets the name of the path variable.
31             log-lib
32
33             # Specifies the name of the NDK library that
34             # you want CMake to locate.
35             log )
36
37 # Specifies libraries CMake should link to your target library. You
38 # can link multiple libraries, such as libraries you define in this
39 # build script, prebuilt third-party libraries, or system libraries.
40
41 target_link_libraries( # Specifies the target library.
42                       native-lib
43
44                       # Links the target library to the log library
45                       # included in the NDK.
46                       ${log-lib} )

Windows 정품
[설정]으로 이동하기
```



실습

Step 6

실행



Thank you

감사합니다

