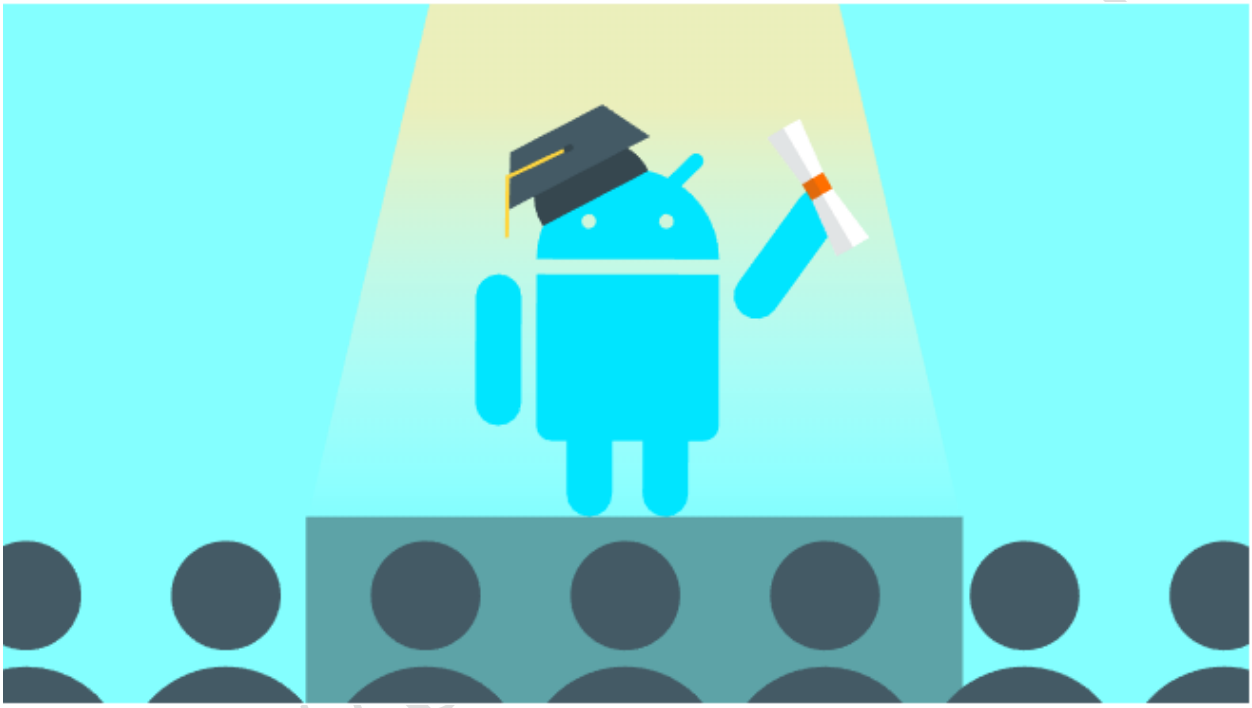
ANDROID TV TRAINING

Android Native C/C++ Libraries



Abstract

Android SDK provides API for interacting Android Application and Application Frameworks layers of Android Platform Architecture. It is enough to use this API for most common cases which developer faces every day. However, sometimes we need more low-level functionality which can be provided by C/C++ libraries. Android NDK we can use JNI to invoke native code from Java/Kotlin and vice versa. So, we can reuse existed C/C++ code in the form of pre-built libraries in our Android apps. This article explores the JNI workflow, provides code examples of how Java calls in both C and C++, and introduces the Android Native Development Kit (NDK), which compiles the C/C++ code into applications that can run on an Android device.

www.bbtechlab.com

bamboo@bbtechlab.com

1 Introduction

1.1 Document Organization

[This subsection of the Training Plan describes the document's organization.]

Version	Date	Author	Change
1.0	08th 01 2019	Bamboo Do, bamboo@bbtechlab.com	Initialize version

1.2 References

[1 Oracle Java™ Native Interface, https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/intro.html

[2]Android JNI tips, https://developer.android.com/training/articles/perf-jni

1.3 Security and the Privacy Act or data protection

[This document contains confidential and privileged information. The reproduction of any part of the document is strictly prohibited without the prior written consent of BBTechLab.]

1.4 Glossary

Acronym	Description
NDK	Native Development Kit
JNI	Java Native Interface

Contents

Tables

Table of Figures

Table of Codes

2 Background knowledge

2.1 Android NDK vs Android SDK

Figure 2-1. Android NDK vs Android SDK

Reasons to use NDK

- Great for CPU intensive operations: mobile videogames, signal processing or physics simulations. Run computationally intensive applications.

- Porting existing C/C++ code to Android.
- Developing a multiplatform application (iOS, Windows). (For cross-platform development)
- The native code is compiled to a binary code and run directly on OS, while Java code is translated into Java byte-code and interpreted by Virtual Machine.
- Native code allows developers to make use of some processor features that are not accessible at Android SDK.
- The opportunity to optimize the critical code at an assembly level.

Reasons to use SDK

- Ensured device portability despite processor architecture.
- Rich set of libraries.
- Automatic memory management.

2.2 JNI and NDK

JNI is part of Dalvik VM such as framework connecting the world of Java to the native code, it allows native code to access Java environment.

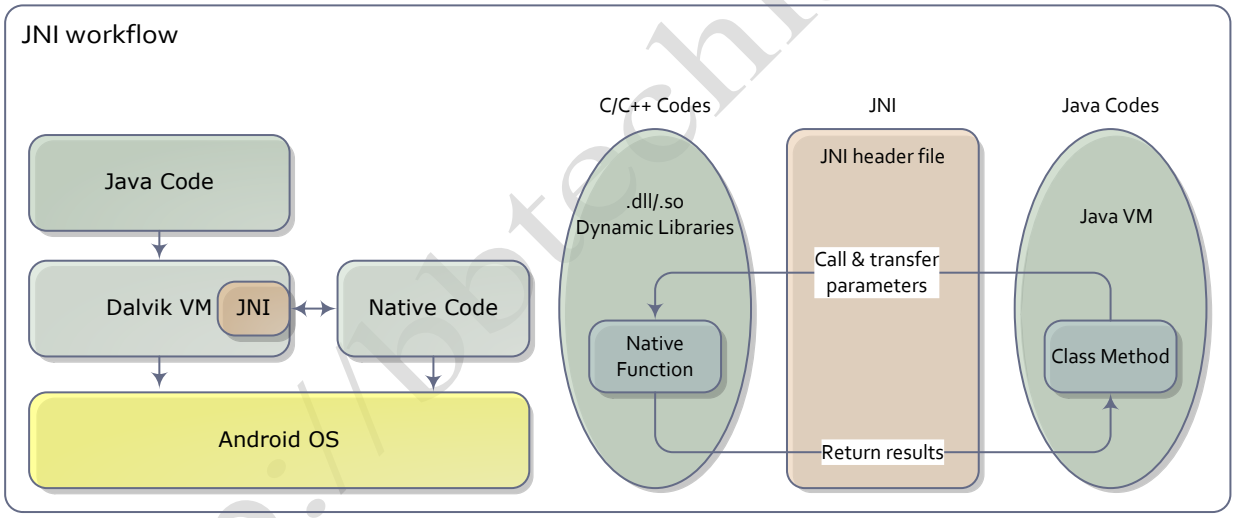


Figure 2-2. JNI general work flow

The general framework of a C/C++ function call via a JNI and Java program (especially Android application) is as follows

- The way of compiling native is declared in the Java class (C/C++ function).
- The .java source code file containing the native method is compiled (Build project in Android).
- The javah command generates an .h file, which corresponds to the native method according to the .class files.
- C/C++ methods are used to achieve the local method
- The recommended method for this step is first to copy the function prototypes into the .h file and then modify the function prototypes and add the function body. In this process, the following points should be noted:
    - The JNI function call must use the C function. If it is the C++ function, do not forget to add the extern "C" keyword;
    - The format of the method name should follow the following template: Java_pacakege_class_method, namely the Java_package name class name and function method name.
- The C or C++ file is compiled into a dynamic library (under Windows this is a .dll file, under Unix/Linux a .so file).

NDK is a toolchain from Android official, originally for developers who writes native C/C++ code as JNI library.

- Cross-compiler, linker to build for ARM, x86, MIPS, etc.
- Provides a way to bundle dynamic library into your APK.
- JNI headers, minimal C++ support headers, and android native app APIs.

NDK build process:

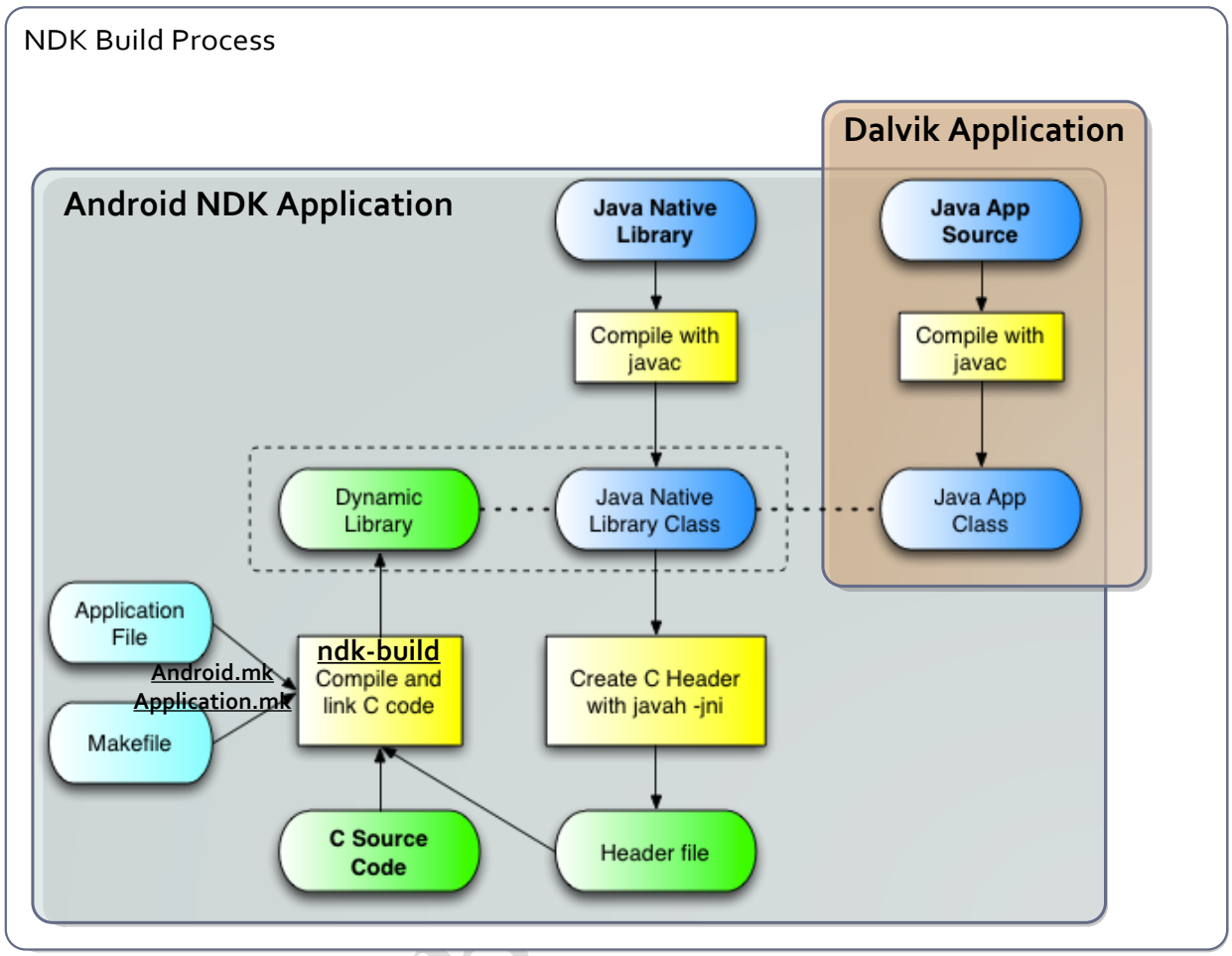


Figure 2-3. NDK build process

2.3 Deep Into JNI World

[JNI tips]

- General tips.
- JavaVM and JNIEnv.
- Threads
- jclass, jmethodID, and jfieldID.
- Local and global references.
- UTF-8 and UTF-16 strings.
- Primitive arrays.
- Region calls.
- Exceptions.
- Native libraries.
- 64-bit considerations.

2.3.1 JNIEnv Interface Pointer

[JNIEnv Interface Pointer]

JNI functions are available through an interface pointer. The JNIEnv interface pointer is pointing to thread-local data, which in turn points to a JNI function table shared by all threads.

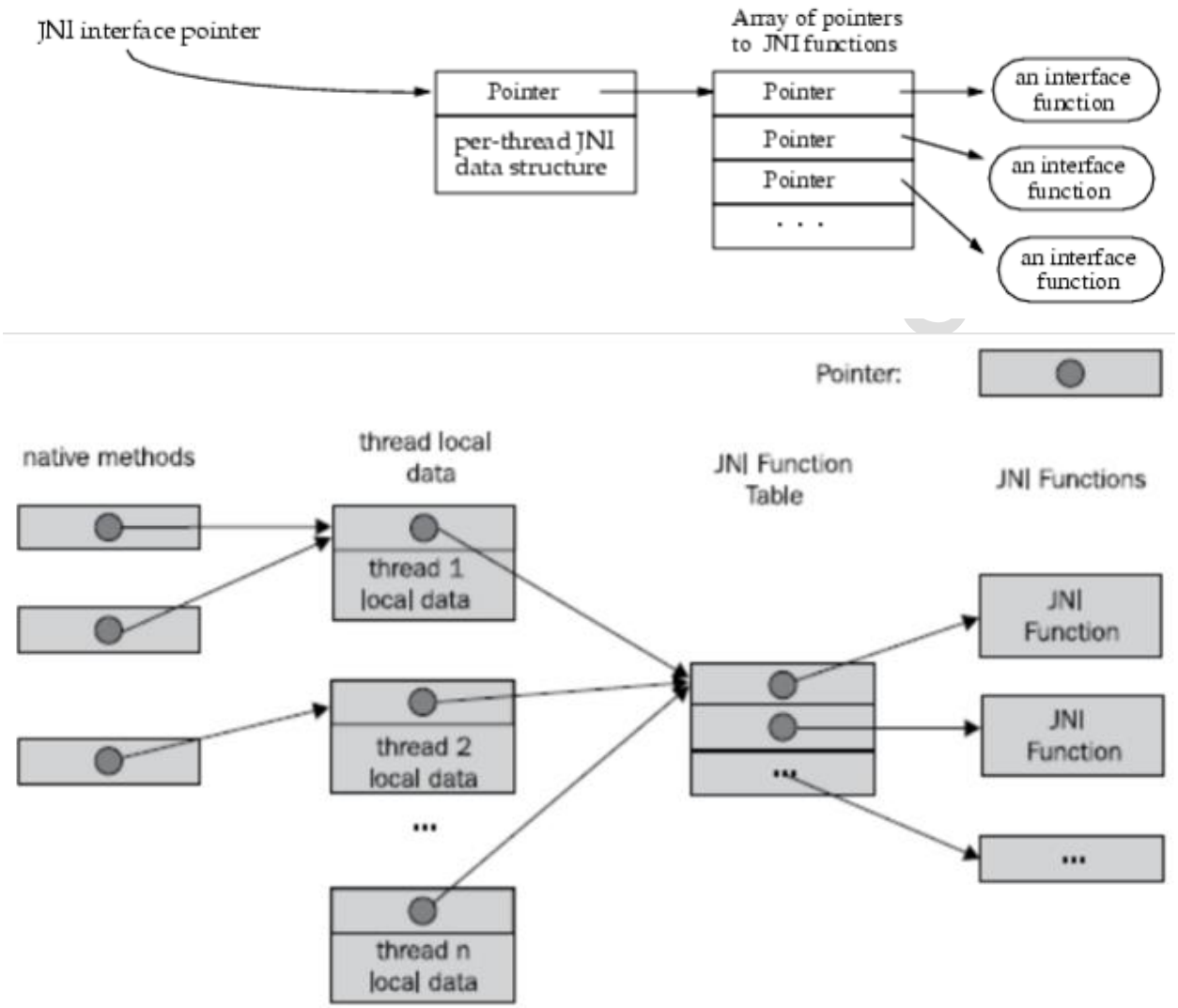


Figure 2-4. JNIEnv interface pointer

- Gateway to access all predefined JNI functions
- Access Java fields
- Invoke Java methods.
- It point to the thread's local data, so it cannot be shared.
- It can be accessible only by java threads.
- Native threads must call **AttachCurrentThread** to attach itself to VM and to obtain the JNIEnv interface pointer

Every native method receives a JNIEnv pointer as its first parameter; this pointer provides access to the JNI support functions.

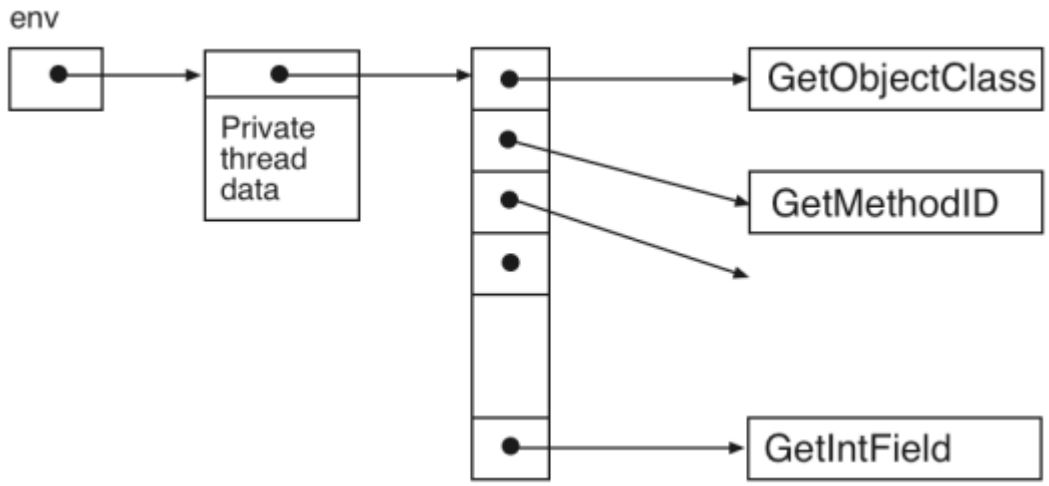


Figure 2-5. JNIEnv interface support

We have two ways to RegisterNatives

- Using javah tool
- Using RegisterNatives function

2.3.2 Loading Native Libraries

Native code is usually compiled into a shared library and loaded before the native methods can be called.

All the native methods are declared with ***native*** key word in Java.

```
1 static {
2     //use either of the two methods below
3     System.loadLibrary("nativelib");
4     System.load("/<path>/libNative.so");
5 }
```

Code 2-1. Loading Native Libraries

2.3.3 Registering native methods using RegisterNatives function

```
01 typedef struct {
02     char *name;
03     char *signature;
04     void *fnPtr;
05 } JNINativeMethod;
06
07 //
08 // Return 0 to indicates success, otherwise negative value
09 //
10 jint RegisterNatives(JNIEnv *env, jclass clazz, const JNINativeMethod
*methods, jint nMethods);
11
```

Code 2-2. Prototype of RegisterNatives

- Env: JNIEnv interface pointer
- The clazz argument is a reference to the class in which the native method is to be registered.
- Methods:
  - Name indicates the native method
  - Signature is the descriptor of the method's input argument data type and return value data type
  - fnPtr is function pointer pointing to the native method.
- nMethods indicates the number of methods to register.

2.3.4 JNI_OnLoad

Will be invoked when the native library is loaded. (system.loadLibray("nativelib");)

It is the right and safe place to register the native methods before their execution.

```
01 JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM* jvm, void* reserved)
02 {
03    JNIEnv *env = NULL;
04    if ((*jvm)->GetEnv(jvm, (void **)&env, JNI_VERSION_1_6) != JNI_OK)
{
05          return -1;
06    }
07    // Write your own code
08    // -> Get jclass with env->FindClass.
09    // -> Register methods with env->RegisterNatives.
10
11    return JNI_VERSION_1_6;
12 }
```

Code 2-3. JNI_OnLoad

2.3.5 JNI Datatypes and Data Structures

[JNI Types and Data Structures]

2.3.5.1 DATATYPES

The following definition is provided for convenience.

```
#define JNI_FALSE  0
#define JNI_TRUE   1
```

The jsize integer type is used to describe cardinal indices and sizes:

```
typedef jint jsize;
```

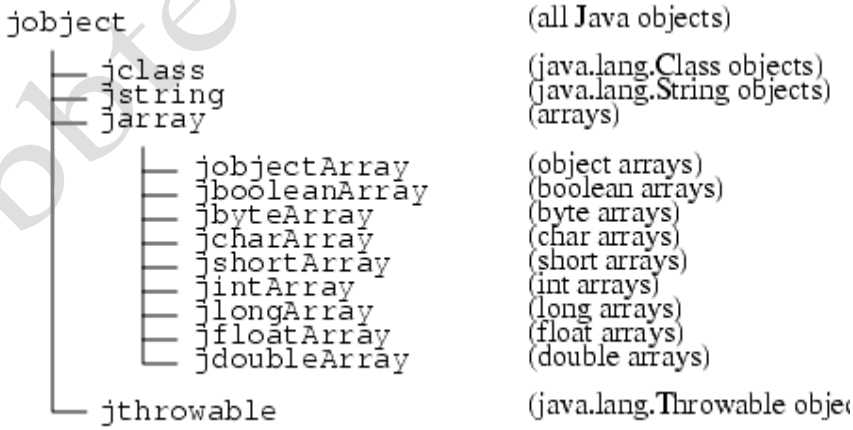
Table 2-1. Primitive Types

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

2.3.5.2 REFERENCES TYPES

In C, all other JNI reference types are defined to be the same as jobject (for example: typdef jobject jclass;)

Table 2-2. References Types



In C++, JNI modules a set of dummy classed to enforce the subtyping relationship. For example:

```
1 class _jobject {};
2 class _jclass : public _jobject {};
3 ...
4 typedef _jobject *jobject;
5 typedef _jclass *jclass;
```

2.3.5.3 FIELD AND METHOD IDS

```
1 struct _jfieldID;              /* opaque structure */
2 typedef struct _jfieldID *jfieldID;   /* field IDs */
3
4 struct _jmethodID;              /* opaque structure */
5 typedef struct _jmethodID *jmethodID; /* method IDs */
```

2.3.5.4 TYPE SIGNATURES

The JNI uses the Java VM's representation of type signatures

Table 2-3. Java VM Type Signatures

Type Signature/Field descriptor	Java Field Type
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L fully-qualified-class ; *Has following type signature:* *Ljava/lang/String;*	fully-qualified-class *example: String*
[type *Has following type signature:* *[I*	type[] *example: int[]*
(arg-types) ret-type *has following type signature:* *(ILjava/lang/String;[I)J*	method type *For example, the Java method:* *long f (int n, String s, int[] arr);*

2.3.6 JNI manipulating

[Oracle, JNI Functions]

2.3.6.1 MANIPULATING STRINGS IN JNI

Strings are complicated in JNI, because Java strings and C strings are internally different.

Java programming language uses UTF-16 to represent strings. If a character cannot fit in a 16-bit code value, a pair of code values named surrogate pair is used. [Modified UTF-8 Strings]

C strings are simply an array of bytes terminated by a null character.

[Oracle, String Operations]

2.3.6.2 MANIPULATING OBJECT IN JNI

[Oracle, Object Operations]

- The clazz argument is a reference to the Java class of which we want to create an instance object. It cannot be an array class, which has its own set of JNI functions
- methodID is the constructor method ID, which can be obtained using the GetMethodID JNI function.

```
1 jobject AllocObject(JNIEnv *env, jclass clazz);
2 jobject NewObject(JNIEnv *env, jclass clazz, jmethodID methodID, ...);
3 jobject NewObjectA(JNIEnv *env, jclass clazz, jmethodID methodID, const
jvalue *args);
4 jobject NewObjectV(JNIEnv *env, jclass clazz, jmethodID methodID,
va_list args);
```

2.3.6.3 MANIPULATING CLASSES IN JNI

[Oracle, Class Operations]

```
1 jclass DefineClass(JNIEnv *env, const char *name, jobject loader, const
jbyte *buf, jsize bufLen);
2 jclass FindClass(JNIEnv *env, const char *name);
3 jclass GetSuperclass(JNIEnv *env, jclass clazz);
```

2.3.6.4 ACCESSING JAVA STATIC AND INSTANCE FILEDS IN NATIVE CODE

- jfieldID data type: jfieldID is a regular C pointer pointing to a data structure with details hidden from developers. We should not confuse it with jobject or its subtypes. jobject is a reference type corresponding to Object in Java, while jfieldID doesn't have such a corresponding type in Java. However, JNI provides functions to convert the java.lang.reflect.Field instance to jfieldID and vice versa.
- Field descriptor: It refers to the modified UTF-8 string used to represent the field data type. (refer to type signatures)

[Oracle, Accessing Static Fields]

[Oracle, Accessing Fields of Objects]

2.3.6.5 CALLING STATIC AND INSTANCE METHODS FROM THE NATIVE CODE

- jmethodID data type: Similar to jfieldID, jmethodID is a regular C pointer pointing to a data structure with details hidden from the developers. JNI provides functions to convert the java.lang.reflect.Method instance to jmethodID and vice versa.
- Method descriptor: This is a modified UTF-8 string used to represent the input (input arguments) data types and output (return type) data type of the method. Method descriptors are formed by grouping all field descriptors of its input arguments inside a "()", and appending the field descriptor of the return type. If the return type is void, we should use "V". If there's no input arguments, we should simply use "()", followed by the field descriptor of the return type. For constructors, "V" should be used to represent the return type. The following table lists a few Java methods and their corresponding method descriptors:

Table 2-4. Example method descriptor by Java method

Java method	Method descriptor
`Dummy(int pValue)`	`(I)V`
`String getName()`	`()Ljava/lang/String;`
`void setName(String pName)`	`(Ljava/lang/String;)V`
`lont f(byte[] bytes, Dummy dummy)`	`([BLcookbook/chapter2/Dummy;)J`

[Oracle. Calling static methods]

[Oracle, Calling Instance Methods]

2.3.6.6 HANDLING EXCEPTIONS IN JNI

[Oracle, Exceptions]

2.3.6.7 DEBUG IN JNI

For Production Builds

- adb shell setprop debug.checkjni 1

For Engineering Builds:

- adb shell stop
- adb shell setprop dalvik.vm.checkjni true
- adb shell start

2.3.6.8 MEMORY ISSUES

Using Libc Debug Mode

- adb shell setprop libc.debug.malloc 1
- adb shell stop
- adb shell start

Supported libc debug mode values are

- 1: Perform leak detection.
- 5: Fill allocated memory to detect overruns.
- 10: Fill memory and add sentinel to detect overruns.

2.3.6.9 FURTHER

[Android Tips]

- Native Threads usage
- More about references
- JNI Graphics using OpenGL

- Audio using OpenSL apis.
- Etc….

3 Java programming tutorial with JNI

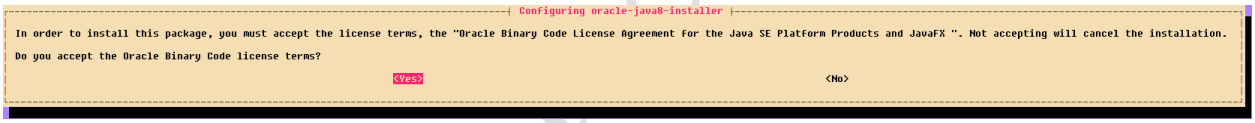
[https://www3.ntu.edu.sg/home/ehchua/programming/java/JavaNativeInterface.html]

3.1 Installation prerequisite environment

- Install Java JDK 8
    - [Java SE Development Kit 8]

Ubuntu

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt install oracle-java8-installer
```



Verify javac & javah installed already

```
bamboo@bbtechlab:~$ javac -version
javac 1.8.0_201
bamboo@bbtechlab:~$ javah -version
javah version "1.8.0_201"
```

Window

- Export environment PATH Java SDK 8 on Window 10: Control Panel > System and Security > System properties > Advanced system setting > Environment Variables.

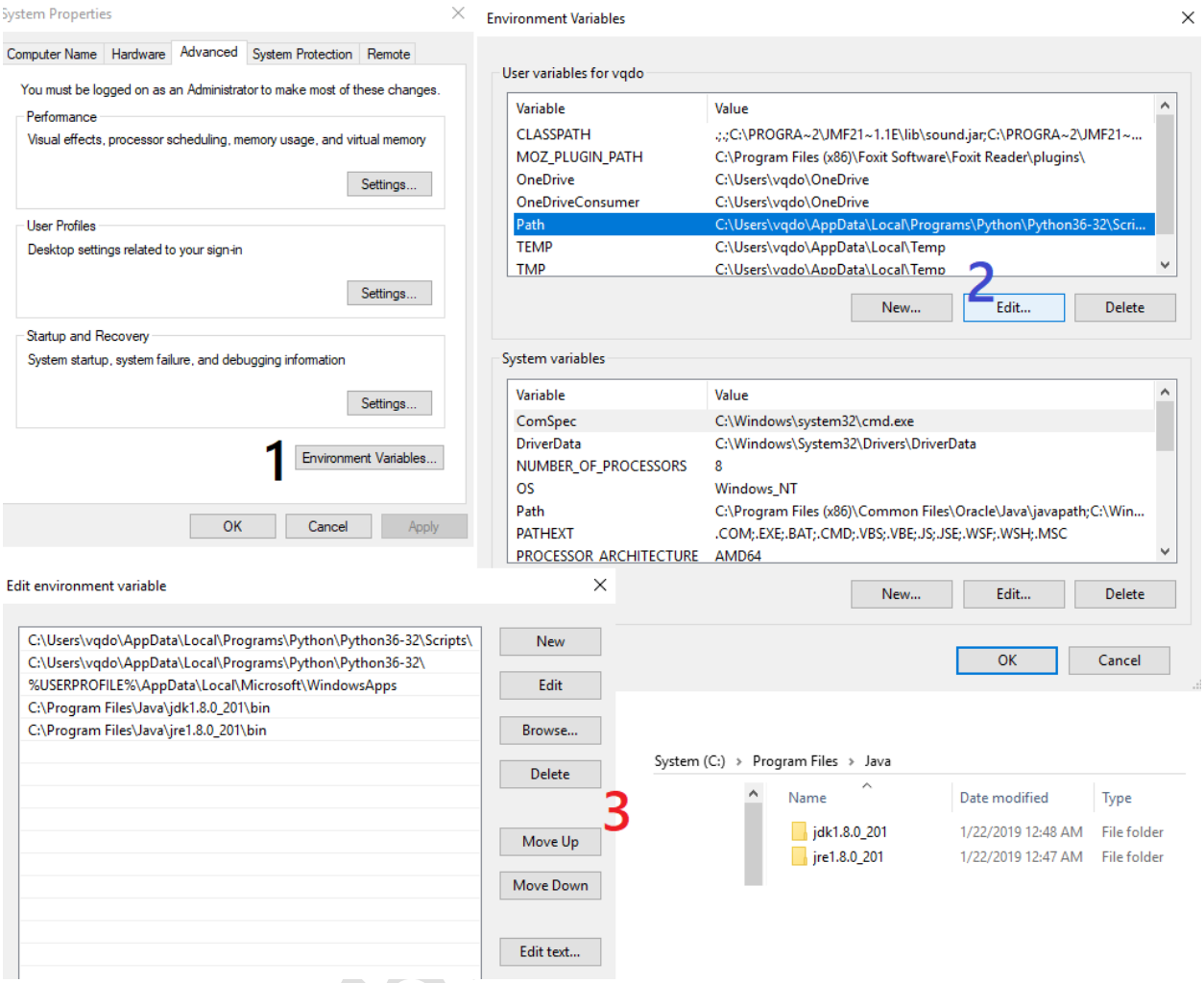


Figure 3-1. Export PATH environment Java SDK 8 on window

Verify javah & javac installed already

```
C:\Users\bamboo>javac.exe -version
javac 1.8.0_201
C:\Users\vqdo>javah.exe -version
javah version "1.8.0_201"
C:\Users\bamboo>
```

3.2 Hello JNI

3.2.1 Step 1: Write a Java Class HelloJNI.java that uses C Codes

```
01 public class HelloJNI {  // Save as HelloJNI.java
02    static {
03       System.loadLibrary("HelloJNI"); // Load native library
HelloJNI.dll (Windows) or libHelloJNI.so (Unixes)
04                                       //  at runtime
05                                       // This library contains a native
method called sayHello()
06    }
07
08    // Declare an instance native method sayHello() which receives no
parameter and returns void
09    private native void sayHello();
10
11    // Test Driver
12    public static void main(String[] args) {
13       new HelloJNI().sayHello();  // Create an instance and invoke the
native method
14    }
15 }
```

Code 3-1. Write a Java Class HelloJNI.java that uses C Codes

The ***static*** initializer invokes System.loadLibrary() to load library "helloJNI" that contains native method sayHello, this library is mapped to native dynamic library "helloJNI.dll" in Window or "helloJNI.so" in Unix.

Next, we declare the method sayHello() as a ***native*** instance method via keyword native which denotes that this method is implemented in native C/C++. A native method doesn't has body, the sayHello() shall be found in the native library loaded.

The main() method allocates an instance of HelloJNI and invoke the native method sayHello().

3.2.2 Step 2: Compile the Java Program HelloJNI.java & Generate C/C++ header.

```
bamboo@bbtechlab:~/work/jni$ javac -h . HelloJNI.java
bamboo@bbtechlab:~/work/jni$ ls -al
total 4
drwxrwxr-x 1 bamboo bamboo 512 Jan 26 16:33 .
drwxrwxr-x 1 bamboo bamboo 512 Jan 26 16:17 ..
-rw-rw-rw- 1 bamboo bamboo 452 Jan 26 16:33 HelloJNI.class
-rw-rw-rw- 1 bamboo bamboo 373 Jan 26 16:33 HelloJNI.h
-rwxrwxr-x 1 bamboo bamboo 620 Jan 26 16:25 HelloJNI.java
bamboo@bbtechlab:~/work/jni$
```

The "-h dir" option generates C/C++ header and places it in the directory specified.

HelloJNI.h looks like

```
01    /* DO NOT EDIT THIS FILE - it is machine generated */
02    #include <jni.h>
03    /* Header for class HelloJNI */
04
05    #ifndef  Included HelloJNI
06    #define _Included_HelloJNI
07    #ifdef __cplusplus
08    extern "C" {
09    #endif
10    /*
11     * Class:     HelloJNI
12     * Method:    sayHello
13     * Signature: ()V
14     */
15    JNIEXPORT void JNICALL Java_HelloJNI_sayHello
16      (JNIEnv *, jobject);
17
18    #ifdef __cplusplus
19    }
20    #endif
21    #endif
```

Code 3-2. HelloJNI.h

The header declares a C function

```
JNIEXPORT void JNICALL Java_HelloJNI_sayHello(JNIEnv *, jobject);
```

The naming convention for the C function is ***Java_{package_and_classname}_{function_name}{JNI_arguments}.*** The dot in package name is replaced by underscore.

The arguments are:

- JNIEnv*: reference to JNI environment, which let's you access all the native functions
- Jobject: reference to "this" java object which contains the native functions.

The extern "C" is recognized by C++ compiler only, it notifies the C++ compiler that these functions are to be compiled using C's function naming protocol instead of C++ naming protocol. C/C++ have different naming protocols as C++ support function overloading and uses a name mangling scheme to differentiate the overloaded functions.

3.2.3 Step 3: Implementing HelloJNI.c

```
01    // Save as "HelloJNI.c"
02    #include <jni.h>        // JNI header provided by JDK
03    #include <stdio.h>      // C Standard IO Header
04    #include "HelloJNI.h"   // Generated
05
06    // Implementation of the native method sayHello()
07    JNIEXPORT void JNICALL Java_HelloJNI_sayHello(JNIEnv *env, jobject
thisObj) {
08       printf("Hello World!\n");
09       return;
10    }
```

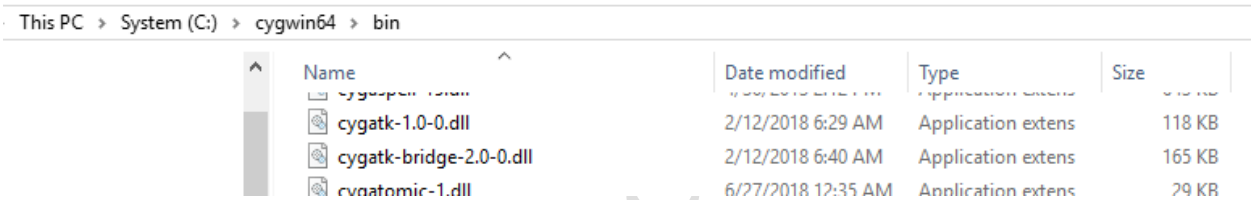
Code 3-3. HelloJNI.c

3.2.4 Compiling the C programming HelloJNI.c

Window

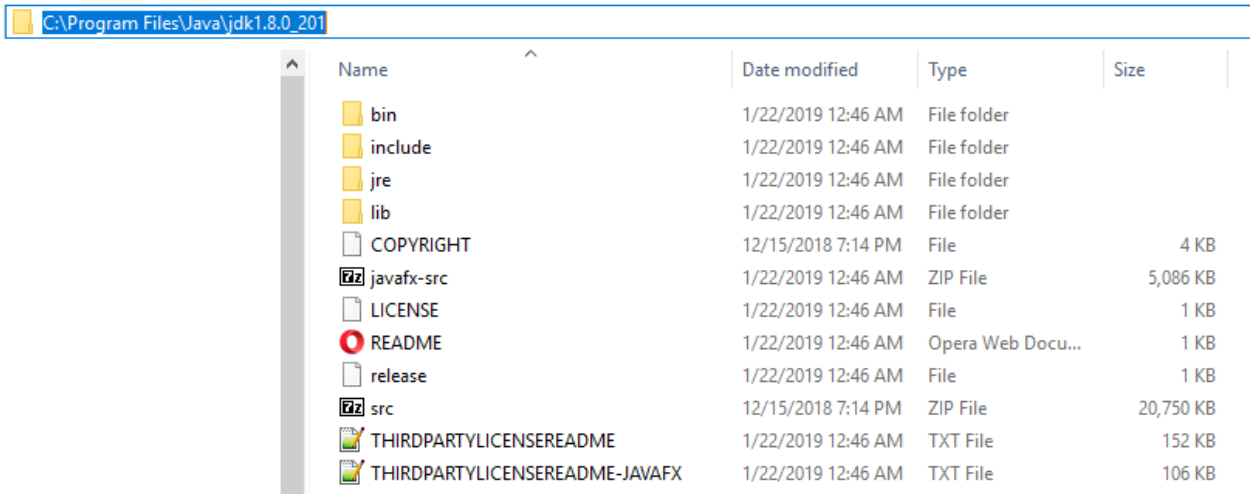
Installation window C compiler:

- https://www.cygwin.com/setup-x86_64.exe



To set the JAVA_HOME environment variable:

- Step 1: First, find your JDK installed directory, For JDK 8, the default is "C:\Program files\Java\jdk1.8.0_{x}"



- Step 2: Check if JAVA_HOME is already set. Start a CMD and issue: SET JAVA_HOME

```
C:\Users\bamboo>SET JAVA_HOME
Environment variable JAVA_HOME not defined
C:\Users\bamboo>
```

If you get a message "Environment variable JAVA_HOME not defined", proceed to the next step.

If you get "JAVA_HOME= Program files\Java\jdk1.8.0_{x}", verify that it is set correctly to your JDK directory. If not, proceed to the next step.

- Step 3: To set the environment variable JAVA_HOME in Windows: Launch "Control Panel" ⇒ (Optional) System and Security ⇒ System ⇒ Advanced system settings ⇒ Switch to "Advanced" tab ⇒ Environment Variables ⇒ System Variables (the bottom pane) ⇒ "New" (or look for "JAVA_HOME" and "Edit" if it is already set) ⇒ In "Variable Name", enter "JAVA_HOME" ⇒ In "Variable Value", enter your JDK installed directory you noted in Step 1.
- Step 4: Verify, RE-START a CMD (restart needed to refresh the environment) and issue: (example below)

```
C:\Users\bamboo>SET JAVA_HOME
JAVA_HOME=C:\Program Files\Java\jdk1.8.0_201
C:\Users\bamboo>
```

Run command below for compiling HelloJNI.c

```
// Compile-only "HelloJNI.c" with -c flag. Output is "HElloJNI.o"
C:\cygwin64\bin>gcc.exe   -D   __int64="long   long"   -c   -I"%JAVA_HOME%\include"   -
I"%JAVA_HOME%\include\win32" jni/HelloJNI.c

// Link "HelloJNO.o" into shared library "hello.dll"
C:\cygwin64\bin>gcc.exe -shared -o jni/HelloJNI.dll jni/HelloJNI.o
```

You need check the resultant file type via the "file" utility, which indicates "Hello.dll" is a 64-bit (x86_64) native Windows DLL.

```
C:\cygwin64\bin>file.exe jni/HelloJNI.dll
jni/HelloJNI.dll: PE32+ executable (DLL) (console) x86-64, for MS Windows
```

Try nm, which lists all the symbols in the shared library and look for the sayHello() function. Check for the function name Java_HelloJNI_sayHello with type "T" (defined).

```
C:\cygwin64\bin>nm.exe jni/HelloJNI.dll | grep say
0000000582271030 T Java_HelloJNI_sayHello
```

Ubuntu

Set environment variable JAVA_HOME to point to the JDK installed directory (which shall contains the include subdirectory to be used in the next step):

To find the Java installation path, run command below

```
bamboo@bbtechlab:~$ update-alternatives --config java
There is 1 choice for the alternative java (providing /usr/bin/java).

  Selection    Path                                            Priority   Status
------------------------------------------------------------
  0            /usr/lib/jvm/java-8-oracle/jre/bin/java   1081      auto mode
* 1            /usr/lib/jvm/java-8-oracle/jre/bin/java   1081      manual mode

Press <enter> to keep the current choice[*], or type selection number:
```

And copy the installation path – second column – under "Path". Next, open the file "/etc/environment" with a text editor

```
bamboo@bbtechlab:~$ sudo vim /etc/environment
```

And add the following line which you previously copied at the end of the file, example:

```
JAVA_HOME=”/usr/lib/jvm/java-8-oracle/jre/bin/java”
```

Compile the C program HelloJNI.c into share module libhello.so using gcc, which is included in all Unixes:

```
$ gcc -fPIC -I"$JAVA_HOME/include" -I"$JAVA_HOME/include/linux" -shared -o
libHelloJNI.so HelloJNI.c
```

Makefile

```
# Define a variable for classpath
CLASS_PATH = $(shell pwd)/bin

# Define include
CFLAGS = -I"${JAVA_HOME}/include" -I"${JAVA_HOME}/include/linux"

all : libHelloJNI.so

# $@ matches the target, $< matches the first dependency
libHelloJNI.so : HelloJNI.o
	gcc -fPIC -shared -o $@ $<

# $@ matches the target, $< matches the first dependency
HelloJNI.o : HelloJNI.c HelloJNI.h
	gcc $(CFLAGS) -c $< -o $@

# $* matches the target filename without the extension
HelloJNI.h : HelloJNI.class
	javah $*

HelloJNI.class : HelloJNI.java
	javac $<

clean :
	rm -rf *.o
	rm -rf *.so
	rm -rf *.class
```

Example

```
bamboo@bbtechlab:~/work/jni$ make all
javac HelloJNI.java
javah HelloJNI
gcc -I"/usr/lib/jvm/java-8-oracle/include" -I"/usr/lib/jvm/java-8-
oracle/include/linux" -c HelloJNI.c -o HelloJNI.o
gcc -fPIC -shared -o libHelloJNI.so HelloJNI.o
```

3.2.5 Run the Java program

You may need to explicitly specify the Java library path of the "HelloJNI.dll" (Windows), "libHelloJNI.so" (Unix). Option -Djava.library.path=/path/to/lib, as below. In this example, the native library is kept in the current directory '.'.

```
java -Djava.library.path=. HelloJNI
```

Example on Ubuntu

```
bamboo@bbtechlab:~/work/jni$ ls -al
total 16
drwxrwxr-x 1 bamboo bamboo  512 Feb  8 23:14 .
drwxrwxr-x 1 bamboo bamboo  512 Jan 26 16:17 ..
-rwxrwxrwx 1 bamboo bamboo 2968 Jan 28 00:30 HelloJNI.c
-rw-rw-rw- 1 bamboo bamboo  441 Feb  8 23:14 HelloJNI.class
-rw-rw-rw- 1 bamboo bamboo  373 Feb  8 23:14 HelloJNI.h
-rwxrwxr-x 1 bamboo bamboo  626 Feb  8 23:14 HelloJNI.java
-rwxrwxrwx 1 bamboo bamboo 7920 Feb  8 23:04 libHelloJNI.so
bamboo@bbtechlab:~/work/jni$ java -Djava.library.path=. HelloJNI
Hello World!
bamboo@bbtechlab:~/work/jni$
```

3.3 Passing Arguments and Result between Java & Native Programs

3.3.1 Passing Primitives

Very simple, just passing parameters according to datatypes in table 2-1 (i.e,. jint, jbyte, jshort, jlong, jfloat, jdouble, jchar and jboolean for each of the Java's primitives int, byte, short, long, float, double, char and boolean, respectively.)

HelloJNI.java

```
01 public class HelloJNI {  // Save as HelloJNI.java
02    static {
03       System.loadLibrary("HelloJNI");
04    }
05    // Passing Primitives
06    private native double average(int n1, int n2);
07
08    // Test Driver
09    public static void main(String[] args) {
10      // Create an instance and invoke the native method
11       HelloJNI thisObj = new HelloJNI();
12      // Average
13      System.out.println("In JAVA, the average is: " +
thisObj.average(4, 5));
14    }
15 }
```

HelloJNI.h

```
1 /*
2  * Class:     HelloJNI
3  * Method:    average
4  * Signature: (II)D
5  */
6 JNIEXPORT jdouble JNICALL Java_HelloJNI_average
7   (JNIEnv *, jobject, jint, jint);
```

HelloJNI.c

```
1 JNIEXPORT jdouble JNICALL Java_HelloJNI_average(JNIEnv *env, jobject
thisObj, jint n1, jint n2) {
2     jdouble result;
3
4     printf("In C, the numbers are %d and %d\n", n1, n2);
5
6     result = ((jdouble)n1 + n2)/2.0;
7
8     return result;
9 }
```

Code 3-4. Passing Primitives

3.3.2 Passing Strings

Passing strings is more complicate than passing primitives because Java's String is an object (reference type), while C's string is a NULL-termincated char array. Therefore we need to convert between them.

The JNI environment (JNIEnv *) provides a set of functions for convention:

```
// UTF-8 String (encoded to 1-3 byte, backward compatible with 7-bit
ASCII)
// Can be mapped to null-terminated char-array C-string
const char * GetStringUTFChars(JNIEnv *env, jstring string, jboolean
*isCopy);
   // Returns a pointer to an array of bytes representing the string in
modified UTF-8 encoding.
void ReleaseStringUTFChars(JNIEnv *env, jstring string, const char *utf);
   // Informs the VM that the native code no longer needs access to utf.
jstring NewStringUTF(JNIEnv *env, const char *bytes);
   // Constructs a new java.lang.String object from an array of
characters in modified UTF-8 encoding.
jsize GetStringUTFLength(JNIEnv *env, jstring string);
   // Returns the length in bytes of the modified UTF-8 representation of
a string.
void GetStringUTFRegion(JNIEnv *env, jstring str, jsize start, jsize
length, char *buf);
   // Translates len number of Unicode characters beginning at offset
start into modified UTF-8 encoding
   // and place the result in the given buffer buf.

// Unicode Strings (16-bit character)
const jchar * GetStringChars(JNIEnv *env, jstring string, jboolean
*isCopy);
   // Returns a pointer to the array of Unicode characters
void ReleaseStringChars(JNIEnv *env, jstring string, const jchar *chars);
   // Informs the VM that the native code no longer needs access to
chars.
jstring NewString(JNIEnv *env, const jchar *unicodeChars, jsize length);
   // Constructs a new java.lang.String object from an array of Unicode
characters.
jsize GetStringLength(JNIEnv *env, jstring string);
   // Returns the length (the count of Unicode characters) of a Java
string.
void GetStringRegion(JNIEnv *env, jstring str, jsize start, jsize length,
jchar *buf);
   // Copies len number of Unicode characters beginning at offset start
to the given buffer buf
```

Example)

- To get a C-string (char*) from JNI string (jstring), invoke method const char* GetStringUTFChars(JNIEnv*, jstring, jboolean*).
- To get a JNI string (jstring) from a C-string (char*), invoke method jstring NewStringUTF(JNIEnv*, char*).

The example as below implements

- Receives the JNI string (jstring), convert it into a C's string (char*) via GetStringUTFChars(), performs perations and displays.
- Requires to enter a C's string (char*), converts it into a JNI string (jstring) and returns.

HelloJNI.java

```
01 public class HelloJNI {  // Save as HelloJNI.java
02    static {
03       System.loadLibrary("HelloJNI");
04    }
05    // Passing Strings
06    private native String stringJNI(String msg);
07
08    // Test Driver
09    public static void main(String[] args) {
10          // Create an instance and invoke the native method
11          HelloJNI thisObj = new HelloJNI();
12          // Average
13          System.out.println("In JAVA, the returned string is: " +
thisObj.stringJNI("How to pass Strings"));
14    }
15 }
```

HelloJNI.h

```
1 /*
2  * Class:     HelloJNI
3  * Method:    stringJNI
4  * Signature: (Ljava/lang/String;)Ljava/lang/String;
5  */
6 JNIEXPORT jstring JNICALL Java_HelloJNI_stringJNI
7   (JNIEnv *, jobject, jstring);
```

HelloJNI.c

```
01 JNIEXPORT jstring JNICALL Java_HelloJNI_stringJNI(JNIEnv *env, jobject
thisObj, jstring str){
02    // Convert the JNI String (jstring) into C-string (char*)
03    const char *charArray = (*env)->GetStringUTFChars(env, str, NULL);
04    if (charArray == NULL) {
05          return NULL;
06    }
07
08    // Displaying the received string
09    printf("In C, the received string is: %s\n", charArray);
10    (*env)->ReleaseStringUTFChars(env, str, charArray); // Release
resourse
11
12    // Prompt users to enter string
13    char userString[128];
14    printf("In C, Enter a string:");
15    scanf("%s", userString);
16
17    // Convert C-string (char*) into JNI String (jstring) and return
18    return (*env)->NewStringUTF(env, userString);
19 }
```

Code 3-5. Passing strings

3.3.3 Passing Array of Primitives

In JAVA, array is reference type, simmilar to class, therefore we need to convert between native array & JNI array. JNI defines a type for each of JAVA primitive arrays, (i.e, jintArray, jbyteArray, jshortArray, jlongArray, jfloatArray, jdoubleArray, jcharArray, jbooleanArray for Java's primitive array of int, byte, short, long, float, double, char and boolean, respectively).

The JNI environment provides a set of functions for the convention

```
// ArrayType: jintArray, jbyteArray, jshortArray, jlongArray,
jfloatArray, jdoubleArray, jcharArray, jbooleanArray
// PrimitiveType: int, byte, short, long, float, double, char, boolean
// NativeType: jint, jbyte, jshort, jlong, jfloat, jdouble, jchar,
jboolean
NativeType * Get<PrimitiveType>ArrayElements(JNIEnv *env, ArrayType
array, jboolean *isCopy);

void Release<PrimitiveType>ArrayElements(JNIEnv *env, ArrayType array,
NativeType *elems, jint mode);

void Get<PrimitiveType>ArrayRegion(JNIEnv *env, ArrayType array, jsize
start, jsize length, NativeType *buffer);

void Set<PrimitiveType>ArrayRegion(JNIEnv *env, ArrayType array, jsize
start, jsize length, const NativeType *buffer);

ArrayType New<PrimitiveType>Array(JNIEnv *env, jsize length);

void * GetPrimitiveArrayCritical(JNIEnv *env, jarray array, jboolean
*isCopy);

void ReleasePrimitiveArrayCritical(JNIEnv *env, jarray array, void
*carray, jint mode);
```

Example)

- To get a C native jint[] from JNI jintArray, invoke jint* GetIntArrayElements()
- To get a JNI jintArray from C native jint[], first, invoke jintArray NewIntArray(JNIEnv *env, jsize len) to allocate , the use void SetIntArrayRegion(JNIEnv *env, jintArray a, jsize start, jsize len, const jint *buf) to copy the jint[] to jintArray

The example implements for

- Receive the incoming JNI array(jintArray), convert it to C's native array(jint[]).
- Perform its intended operations
- Convert the return C's native array(jdouble[]) to JNI array(jdoubleArray), and return the JNI array.

HelloJNI.java

```
01 public class HelloJNI {  // Save as HelloJNI.java
02    static {
03       System.loadLibrary("HelloJNI");
04    }
05    // Passing array of primitives
06    private native double[] sumAndAverage(int[] numbers);
07
08    // Test Driver
09    public static void main(String[] args) {
10      // Create an instance and invoke the native method
11      HelloJNI thisObj = new HelloJNI();
12      // Average
13      int[] numbers = {4, 7, 9};
14      double[] results = thisObj.sumAndAverage(numbers);
15      System.out.println("In JAVA, the sum is " + results[0]);
16      System.out.println("In JAVA, the average is " + results[1]);
17    }
18 }
```

HelloJNI.h

```
1 /*
2  * Class:     HelloJNI
3  * Method:    sumAndAverage
4  * Signature: ([I)[D
5  */
6 JNIEXPORT jdoubleArray JNICALL Java_HelloJNI_sumAndAverage
7   (JNIEnv *, jobject, jintArray);
```

HelloJNI.c

```
01 JNIEXPORT jdoubleArray JNICALL Java_HelloJNI_sumAndAverage (JNIEnv
*env, jobject thisObj, jintArray inJNIArray) {
02    // Convert the incoming JNI jintarray to C's jint[]
03    jint *intArray = (*env)->GetIntArrayElements(env, inJNIArray,
NULL);
04    if (intArray == NULL){
05          return NULL;
06    }
07    jsize length = (*env)->GetArrayLength(env, inJNIArray);
08
09    // Perform its intended operations
10    jint sum = 0;
11    int i;
12    for (i = 0; i < length; i++) {
13          sum += intArray[i];
14    }
15    jdouble average = (jdouble)sum / length;
16    (*env)->ReleaseIntArrayElements(env, inJNIArray, intArray, 0); //
Release resource
17
18    jdouble outArray[] = {sum, average};
19
20    // Conver the C's native jdouble[] to JNI jdoubleArray, and return
21    jdoubleArray outJNIArray = (*env)->NewDoubleArray(env, 2); //
Allocate
22    if (NULL == outArray) {
23          return NULL;
24    }
25
26    (*env)->SetDoubleArrayRegion(env, outJNIArray, 0, 2, outArray); //
Copy
27
28    return outJNIArray;
29 }
```

Code 3-6. Passing Array of Primitives

3.4 Accessing Object's Variables and Calling Back Methods

3.4.1 Accessing Object's Instance Variables

To access the instance variable of an object

- Get a reference to this object's class via GetObjectClass().
- Get the Field ID of the instance variable to be accessed via GetFieldID() from the class reference. You need to provide the variable name and its field descriptor (or signature, table 2-3).
- Based on the Field ID, retrieve the instance variable via GetObjectField() or Get<primitive-type>Field() function.
- To update the instance variable, use the SetObjectField() or Set<primitive-type>Field() function, providing the Field ID.

The JNI functions for accessing instance variable are:

```
01 jclass GetObjectClass(JNIEnv *env, jobject obj);
02    // Returns the class of an object.
03
04 jfieldID GetFieldID(JNIEnv *env, jclass cls, const char *name, const
char *sig);
05    // Returns the field ID for an instance variable of a class.
06
07 NativeType Get<type>Field(JNIEnv *env, jobject obj, jfieldID fieldID);
08 void Set<type>Field(JNIEnv *env, jobject obj, jfieldID fieldID,
NativeType value);
09    // Get/Set the value of an instance variable of an object
10    // <type> includes each of the eight primitive types plus Object.
```

The example as below

HelloJNI.java

```
01 public class HelloJNI {  // Save as HelloJNI.java
02    static {
03       System.loadLibrary("HelloJNI");  // Load native library
HelloJNI.dll (Windows) or libHelloJNI.so (Unixes)
04                                                              // at
runtime. This library contains a native method called sayHello()
05    }
06
07    // Instance variables
08    private int number = 8888;
09    private String message = "Hello from Java";
10
11    // Declare a native method that modifies the instance variables
12    private native void modifyInstanceVariable();
13
14    // Test Driver
15    public static void main(String[] args) {
16      // Create an instance and invoke the native method
17       HelloJNI thisObj = new HelloJNI();
18
19      // Test modifying the instance variables
20      thisObj.modifyInstanceVariable();
21      System.out.println("In Java, int is " + thisObj.number);
22      System.out.println("In Java, String is " + thisObj.message);
23    }
24 }
```

HelloJNI.h

```
1 /*
2  * Class:     HelloJNI
3  * Method:    modifyInstanceVariable
4  * Signature: ()V
5  */
6 JNIEXPORT void JNICALL Java_HelloJNI_modifyInstanceVariable
7   (JNIEnv *, jobject);
```

HelloJNI.c

```
01 JNIEXPORT void JNICALL Java_HelloJNI_modifyInstanceVariable(JNIEnv
*env, jobject thisObj) {
02    // Get a reference to this object's class
03    jclass thisClass = (*env)->GetObjectClass(env, thisObj);
04
05    // Get the Fileld ID of the instance variables "number"
06    jfieldID fidNumber = (*env)->GetFieldID(env, thisClass, "number",
"I");
07    if (NULL == fidNumber)
08          return;
09    // Get the int given the FieldID
10    jint number = (*env)->GetIntField(env, thisObj, fidNumber);
11    printf("In C, the int is %d\n", number);
12
13    // Change the value of variable
14    number = 9999;
15    (*env)->SetIntField(env, thisObj, fidNumber, number);
16
17    // Get the FieldID of the instance variable "message"
18    jfieldID fidMessage = (*env)->GetFieldID(env, thisClass, "message",
"Ljava/lang/String;");
19    if (NULL == fidMessage)
20          return;
21    // Get the object given the Field ID
22    jstring message = (*env)->GetObjectField(env, thisObj, fidMessage);
23    // Create a C-string with the JNI String
24    const char *cStr = (*env)->GetStringUTFChars(env, message, NULL);
25    if (NULL == cStr)
26          return;
27    printf("In C, the string is %s\n", cStr);
28    (*env)->ReleaseStringUTFChars(env, message, cStr); // Release
resource
29    // Create a new C-string and assign to the JNI string
30    message = (*env)->NewStringUTF(env, "Hello from C");
31    if (NULL == message)
32          return;
33    // modify the instance variables
34    (*env)->SetObjectField(env, thisObj, fidMessage, message);
35 }
```

Code 3-7. Accessing Object's Instance Variables

3.4.2 Accessing Class' Static Variables

Accessing static variables is similar to accessing instance variable, except that you use functions such as GetStaticFieldID(), Get|SetStaticObjectField(), Get|SetStatic<Primitive-type>Field().

3.4.3 Callback Instance Methods and Static Methods

3.4.4 Callback Overridden Superclass' Instance Method

3.5 Creating Objects and Object Arrays

3.5.1 Callback the Constructor to Create a New Java Object in the Native Code

3.5.2 Array of Objects

3.6 Local and Global References

3.7 Debugging JNI Programs

4 Developing Android NDK application

[Android* Application Development and Optimization on the Intel® Atom™ Platform]

4.1 Developing Android NDK Applications for Embedded Devices

NDK application development can be divided into five steps shown in following figure

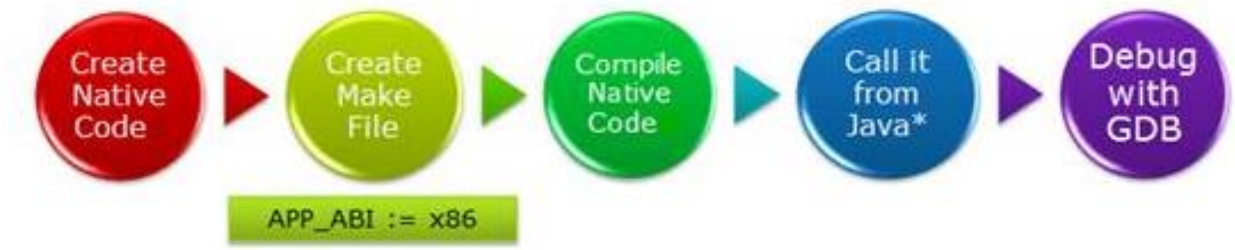


Figure 4-1. NDK Application Development Process

4.1.1 Developing Android NDK Applications with Android Studio

Required

- [Install JDK 8]
  - jdk1.8.0_201
- [Install Android Studio]
  - Make sure that plugin NDK installed: Tools > SDK Manager > Android SDK
    - Checked LLDB, CMake, NDK

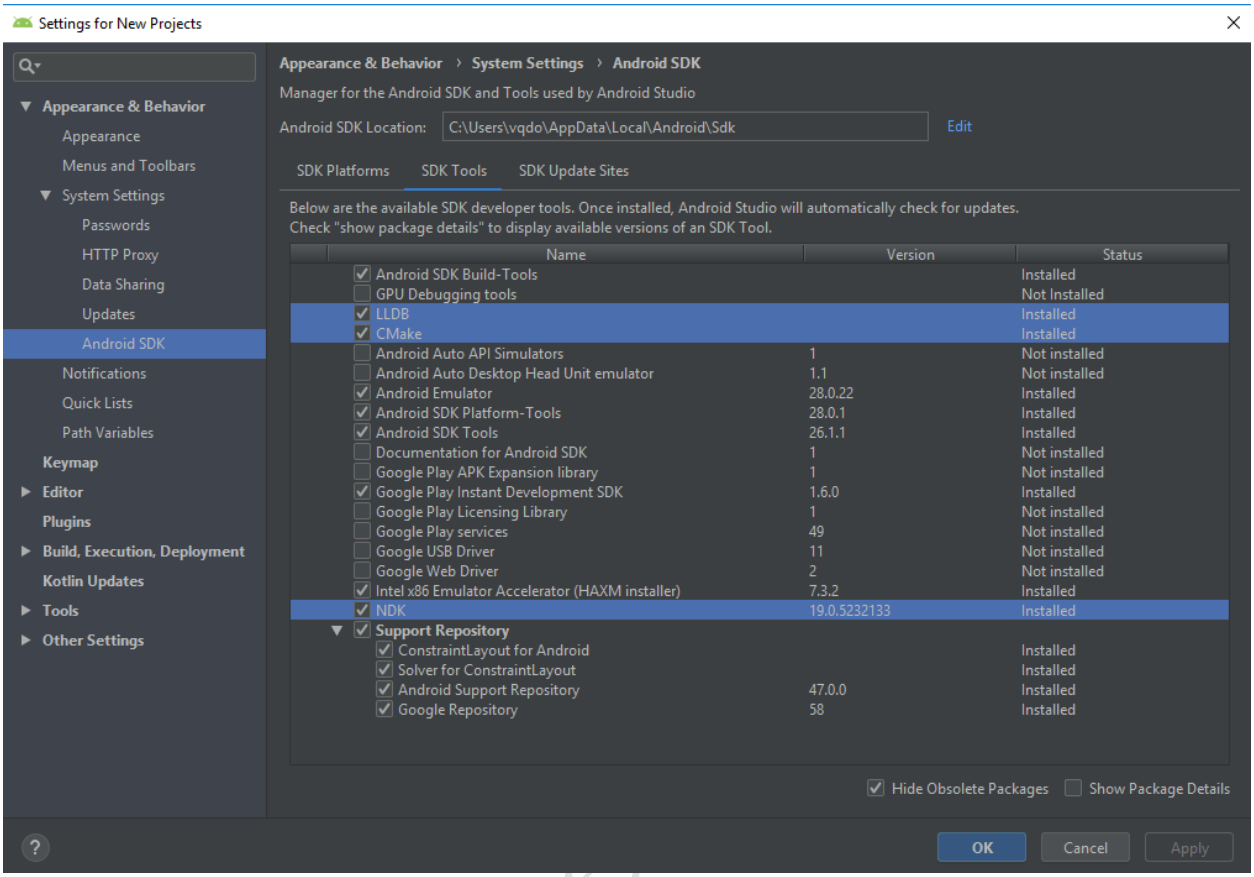


Figure 4-2. Install NDK plugin

4.1.1.1 STEP 1: CREATING A HELLOJNI PROJECT

- Open Android Studio IDE in your computer.
- Create a new project and Edit the Application name to "**HelloJNI**". (Optional) You can edit the company domain or select the suitable location for current project tutorial. Then click next button to proceed.
- Select Minimum SDK (API 15:Android 4.0.3 (IceCreamSandwich). I choose the API 15 because many android devices currently are support more than API 15. Click Next button.
- Choose "Empty Activity" and Click Next button
- Lastly, press finish button.

[*Note : You must download NDK package in the SDK Manager to proceed.*]

4.1.1.2 STEP 2: SETUP EXTERNAL TOOLS

In your android studio menu go to File > Settings. Expand the Tools section you will see "External Tools" and Click it. After that create two external tools which are javah and ndk-build.

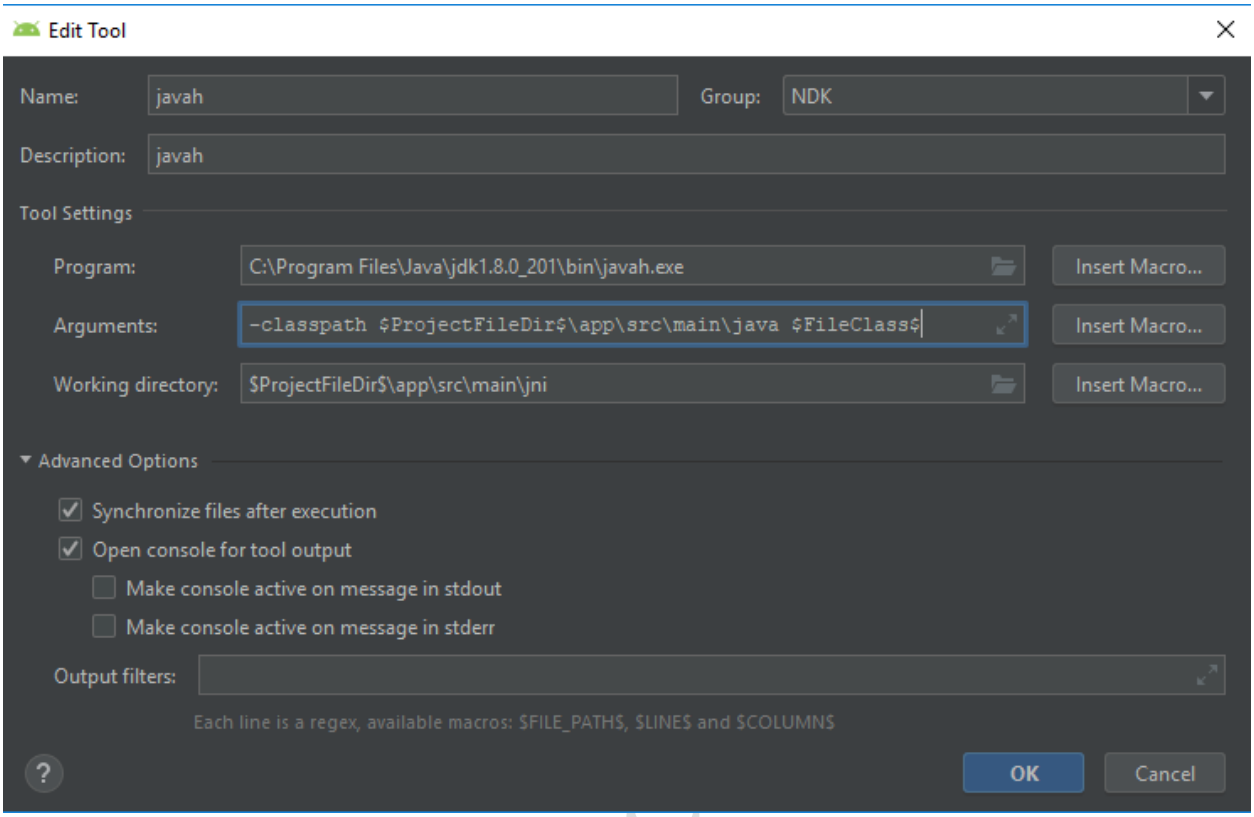


Figure 4-3. Setup external tools: javah

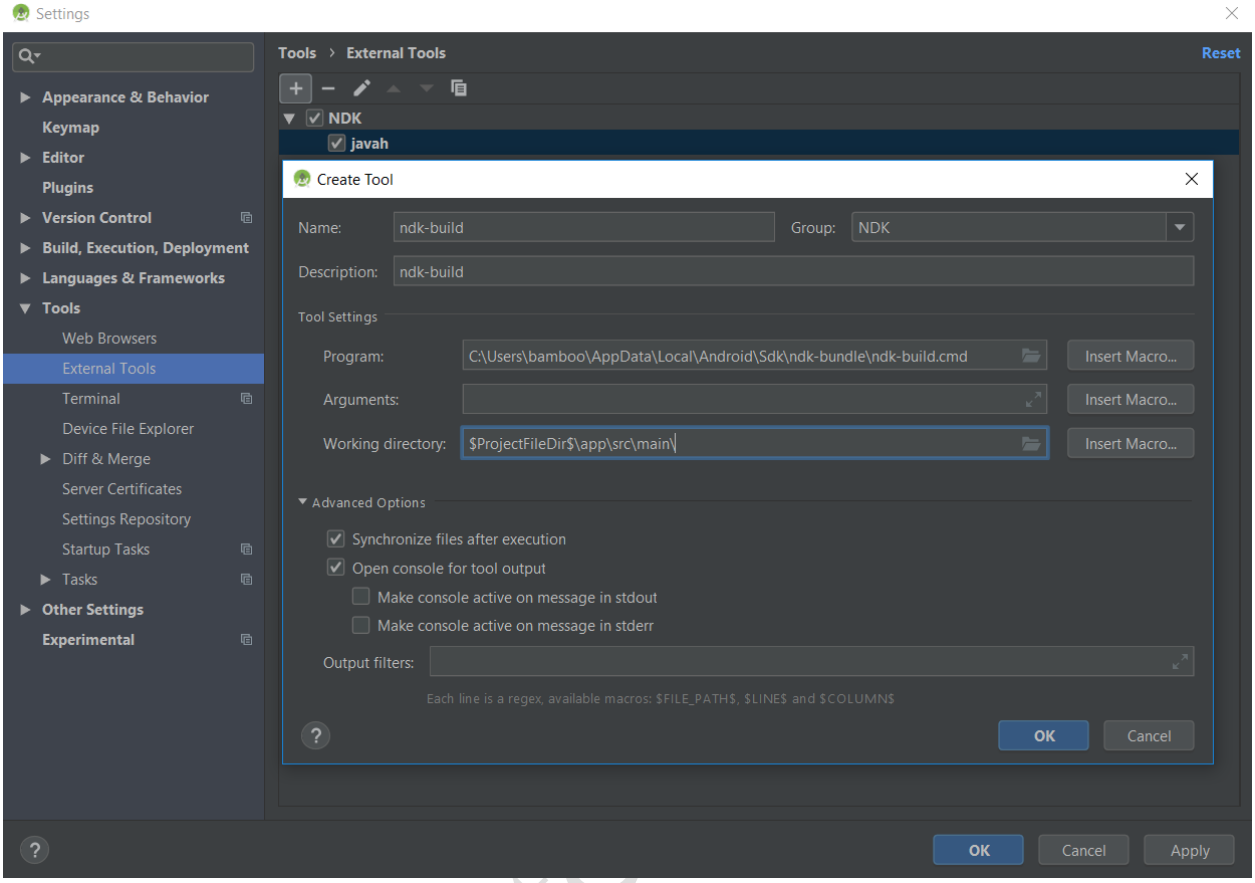


Figure 4-4. Setup external tools: ndk-build

4.1.1.3 STEP 3: ADD A JAVA CLASS FOR JAVA NATIVE INTERFACE

Right click package name > new > Java class and name it as “helloStringJNI“. This class will add static and load the library which name is “nativelib”. The library name is followed by the so file, we will compile .so file later. And the native method is to get the method from the C and C++ source code.

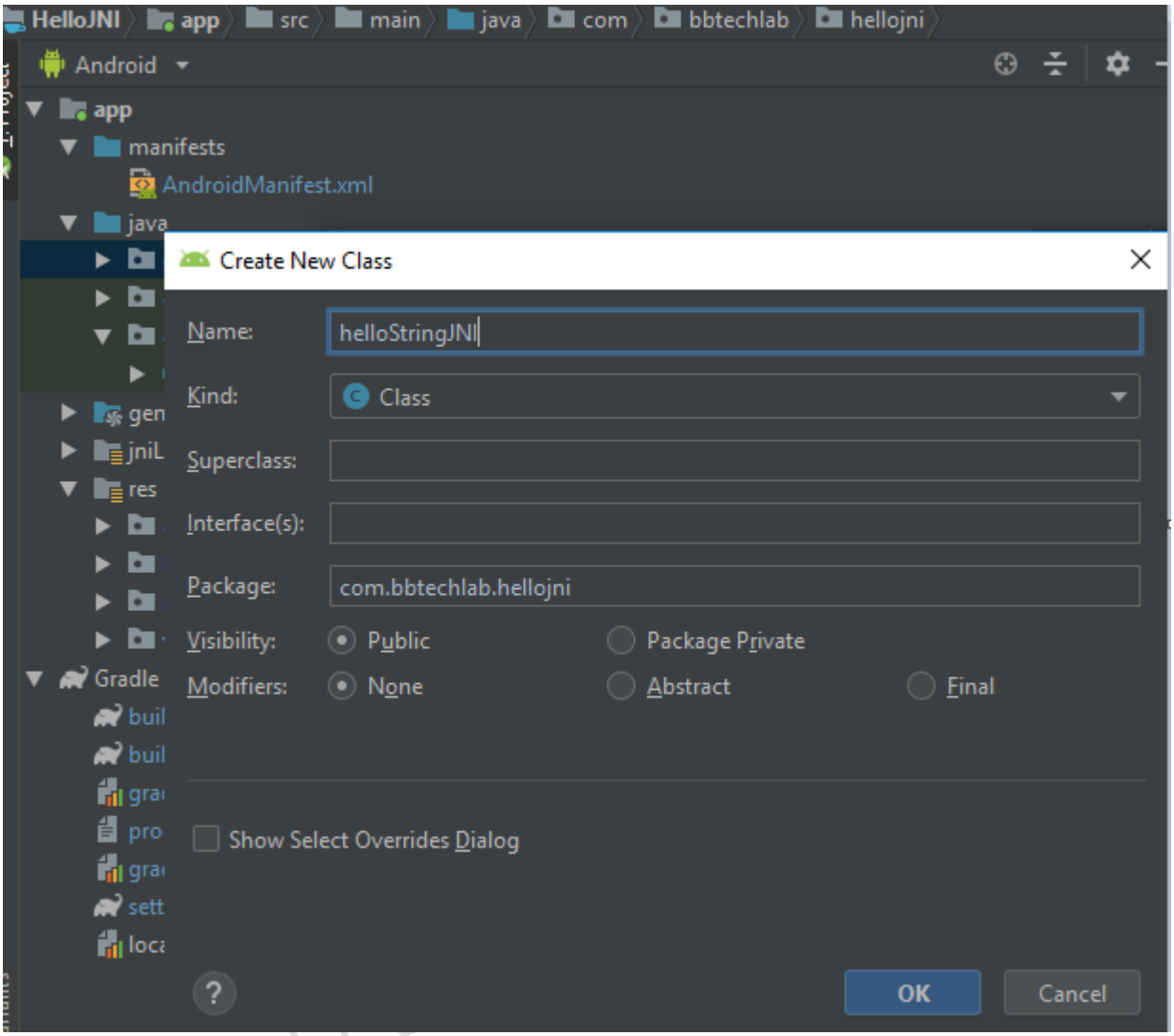


Figure 4-5. Add a Java class for JNI

Edit helloStringJNI.java class

Go to the file and copy the following code in your class.

```
01 package com.bbtechlab.hellojni;
02
03 public class helloStringJNI {
04     static {
05         System.loadLibrary("nativelib");
06     }
07
08     public native String getStringJNI();
09
10 }
```

Code 4-1. Add a Java class for JNI

4.1.1.4 STEP 4: EDIT BUILD.GRADLE (MODULE:APP)

Add ndk and sourceSets.main in the defaultConfig. NDK is to specific what module name you use, for example our module name will be "**nativelib**". The moduleName will follow by the C or C++ files so we will create later. In SourceSets.main section the jni.srcDirs = [] mean disable auto and jniLibs.src are specify which jni library located.

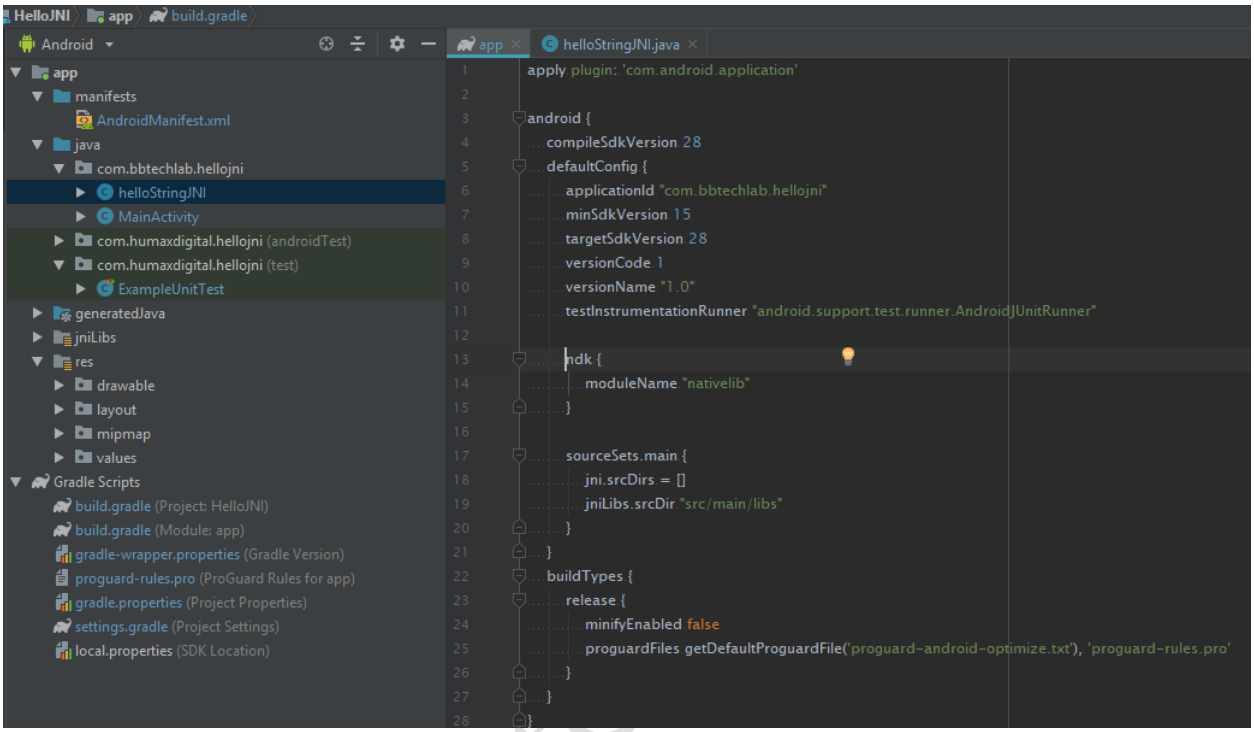


Figure 4-6. Edit build.gradle (module:app)

Edit gradle-properties

You will occur an error if you do not add the following code:

```
android.useDeprecatedNdk=true
```

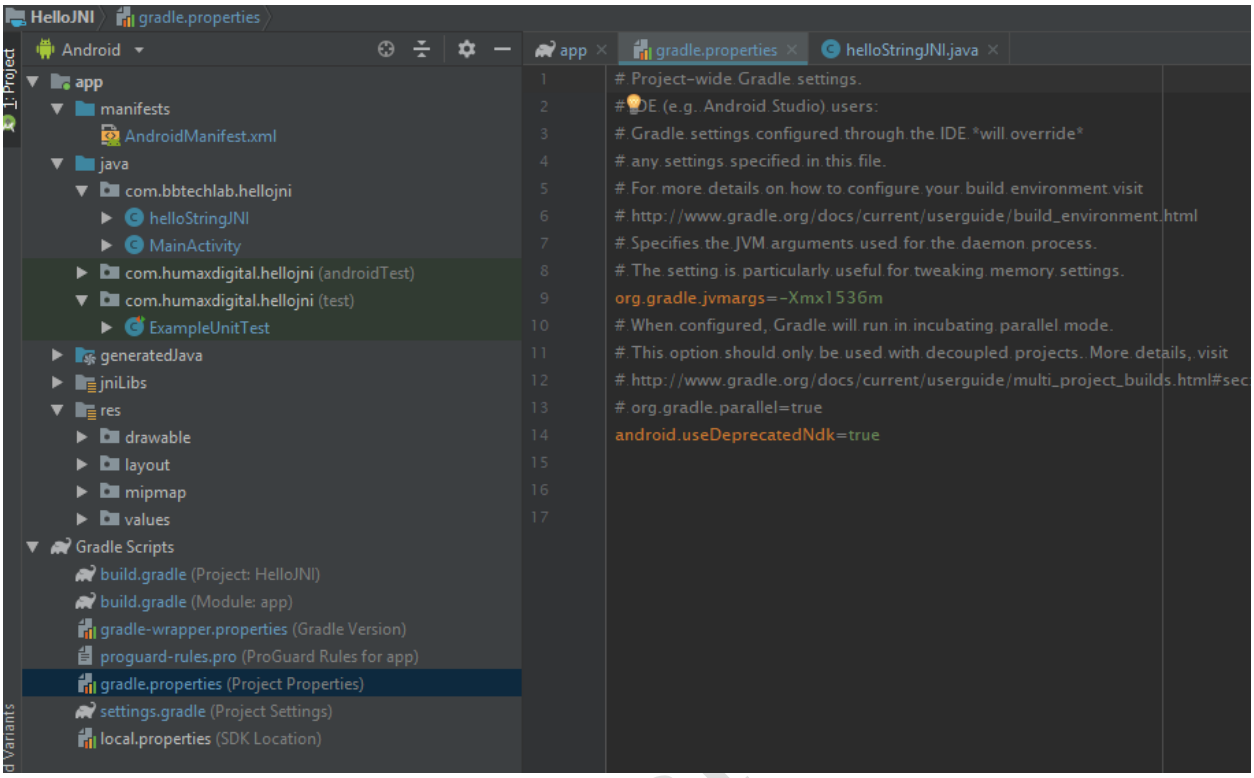


Figure 4-7. Edit gradle-properties

4.1.1.5 STEP 5: ADD JNI & IMPLEMENT C/C++ FOR NATIVELIB

From Android navigate to Project Files, after that right click main folder > New > Folder > JNI Folder. You will see a new JNI folder was added in.

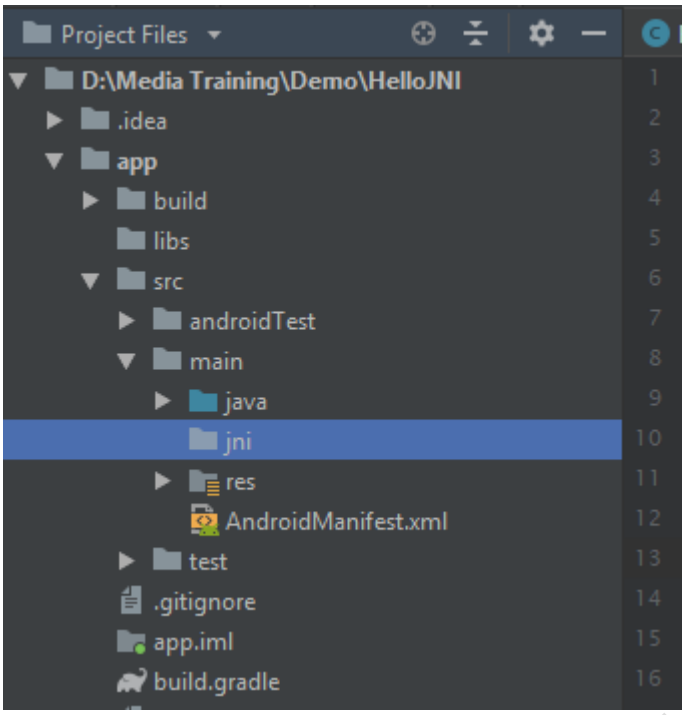


Figure 4-8. Add a JNI Folder

Right click your folder name jni > New > New C/C++ Source File and name it as “nativelib.cpp“. This name must be same as ModuleName in build.gradle.

Generate header files for nativelib.cpp.

- Go to your java folder and Right click helloJNI.java class > NDK >javah. It will automatically create a header file in your jni folder. For example, it will look like this.

```
01 /* DO NOT EDIT THIS FILE - it is machine generated */
02 #include <jni.h>
03 /* Header for class com_bbtechlab_hellojni_helloStringJNI */
04
05 #ifndef  Included com bbtechlab hellojni helloStringJNI
06 #define _Included_com_bbtechlab_hellojni_helloStringJNI
07 #ifdef __cplusplus
08 extern "C" {
09 #endif
10 /*
11  * Class:     com_bbtechlab_hellojni_helloStringJNI
12  * Method:    getStringJNI
13  * Signature: ()Ljava/lang/String;
14  */
15 JNIEXPORT jstring JNICALL
Java_com_bbtechlab_hellojni_helloStringJNI_getStringJNI
16   (JNIEnv *, jobject);
17
18 #ifdef __cplusplus
19 }
20 #endif
21 #endif
22
```

Code 4-2. Header of helloJNI.java class

Now, edit nativelib.cpp files. Copy the method header from Auto-generated header files and paste it into this file. After that, add the parameter variable and return a value by using C++ code style. You must include the header file.

```
01 //
02 // Created by vqdo on 1/22/2019.
03 //
04 #include "com_bbtechlab_hellojni_helloStringJNI.h"
05
06 JNIEXPORT jstring JNICALL
Java_com_bbtechlab_hellojni_helloStringJNI_getStringJNI (JNIEnv *env,
jobject obj) {
07     return (*env).NewStringUTF("hello JNI - Bamboo");
08 }
```

Code 4-3. nativelib.cpp

Compile nativelib.so (shared library) file by creating Android.mk

Right click jni folder > New > File and name it to Android.mk. Add the following code to your file.

[Android.mk]

```
1 LOCAL_PATH := $(call my-dir)
2 include  $(CLEAR_VARS)
3
4 LOCAL_MODULE := nativelib
5 LOCAL_SRC_FILES := nativelib.cpp
6 include $(BUILD_SHARED_LIBRARY)
```

Code 4-4. Android.mk for compiling all C/C++ source of nativelib

LOCAL_PATH := $(call my-dir)

An Android.mk file must begin defining the LOCAL_PATH variable, this is where the source files are. The macro ‘my-dir’ is the path where the Android.mk file is located.

include $(CLEAR_VARS)

Since all the building and parsing is done in the same context the variables called LOCAL_XXX is global and need to be cleared.

LOCAL_MODULE := nativelib

This is where you set the name used as the identifier for each module. Later used in java when loading the module. The system will add ‘lib’ before the module name when compiling into the .so file. So nativelib will become lib nativelib.so. The only exception is if you add ‘lib’ first in your module name then the system will not add it.

LOCAL_SRC_FILES := nativelib.cpp

Here you add a list of the files you need to compile your module. You do not need to add headers or include files the system will take care of that for you.

include $(BUILD_SHARED_LIBRARY)

The NDK provides you with two make files that parse and build everything accordingly to your Android.mk file. The two once are BUILD_STATIC_LIBRARY for building static library and BUILD_SHARED_LIBRARY for building shared library. For the example project here we use the BUILD_SHARED_LIBRARY.

Right click jni folder > New > File and name it to Application.mk. Add the following code to your file.

[Application.mk]

```
1 APP_MODULES := nativelib
2
3 APP_ABI := all
```

Code 4-5. Appkication.mk for compiling nativelib module

Right-click main folder > NDK > ndk-build. You will see new so files will appear in your libs folder as the picture below. The folders separate by different CPUs name.

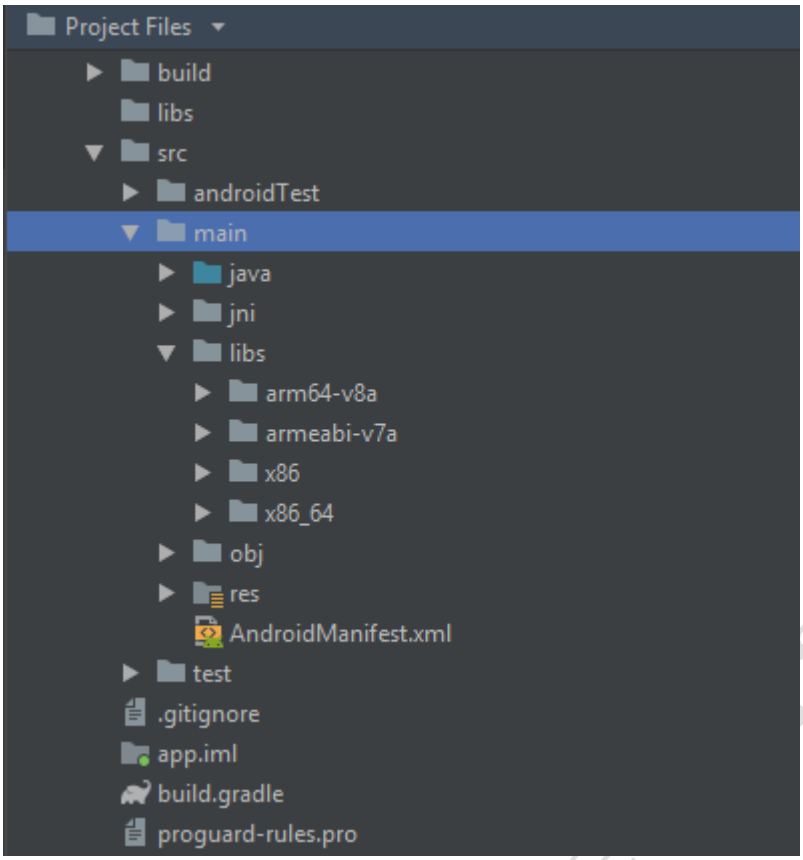


Figure 4-9. nativelib.so outputs separated by different CPUs name.

4.1.1.6 STEP 6: ACCESS NATIVELIB VIA MAIN ACTIVITY

Edit activity_main.xml layout

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
03     xmlns:app="http://schemas.android.com/apk/res-auto"
04     xmlns:tools="http://schemas.android.com/tools"
05     android:layout_width="match_parent"
06     android:layout_height="match_parent"
07     tools:context=".MainActivity">
08
09     <TextView
10         android:layout_width="wrap_content"
11         android:layout_height="wrap_content"
12         android:id="@+id/textView"
13         android:text="Hello World!"
14         app:layout_constraintBottom_toBottomOf="parent"
15         app:layout_constraintLeft_toLeftOf="parent"
16         app:layout_constraintRight_toRightOf="parent"
17         app:layout_constraintTop_toTopOf="parent" />
18
19 </android.support.constraint.ConstraintLayout>
```

Code 4-6. activity_main.xml layout

Edit MainActivity.java class

```
01 package com.bbtechlab.hellojni;
02
03 import android.support.v7.app.AppCompatActivity;
04 import android.os.Bundle;
05 import android.widget.TextView;
06
07 public class MainActivity extends AppCompatActivity {
08
09     @Override
10     protected void onCreate(Bundle savedInstanceState) {
11         super.onCreate(savedInstanceState);
12         setContentView(R.layout.activity_main);
13
14         TextView textView=(TextView)findViewById(R.id.textView);
15         helloStringJNI testStringJNI = new helloStringJNI();
16         textView.setText("" + testStringJNI.getStringJNI());
17     }
18 }
```

Code 4-7. MainActivity.java class

Now to try to run your project, you shall see the output like as below

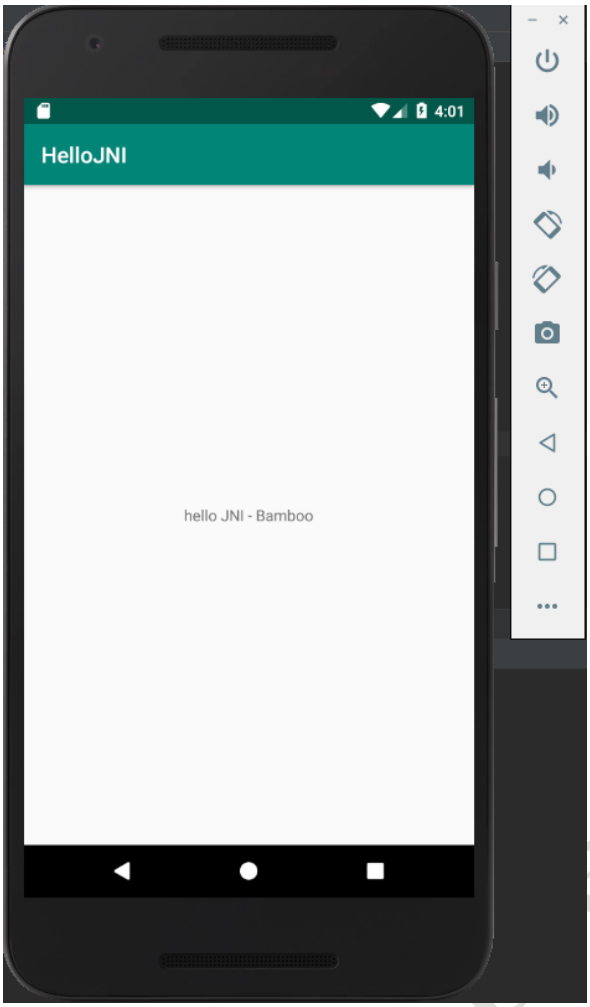


Figure 4-10. Demo application of nativelib

4.1.2 Developing Android NDK Applications with Eclipse

Required

- Install Eclipse IDE for Java Developers, Android Development Tools (ADT) Eclipse Plugin.
  - [Eclipse IDE 2018-12]
  - [Installing the Eclipse Plugin]
- Install Android SDK
  - [Command line tools only]
- [Install NDK]

```
bamboo@DESKTOP-9NDTRKT:~/work/C/android-ndk-r10e$ tree -L 1 ./
./
├── GNUmakefile
├── README.TXT
├── RELEASE.TXT
├── build
├── docs
├── find-win-host.cmd
├── ndk-build
├── ndk-build.cmd
├── ndk-depends.exe
├── ndk-gdb
├── ndk-gdb-py
├── ndk-gdb-py.cmd
├── ndk-gdb.py
├── ndk-stack.exe
├── ndk-which
├── platforms
├── prebuilt
├── remove-windows-symlink.sh
├── samples
├── sources
├── tests
└── toolchains

8 directories, 14 files
bamboo@DESKTOP-9NDTRKT:~/work/C/android-ndk-r10e$
```

Figure 4-11. Install NDK to C:\android-ndk-r10e

- [Install JDK 8]
    - jdk1.8.0_201

4.2 Porting Existing Android NDK applications to Embedded Devices

All NDK applications can be divided into three types based on the following properties of the native code:

- Consists of C/C++ code only that is not related to hardware
- Uses a third-party dynamic linked library
- Includes assembly code that is highly related to non-embedded platform.(example: non-Intel Atom platform)

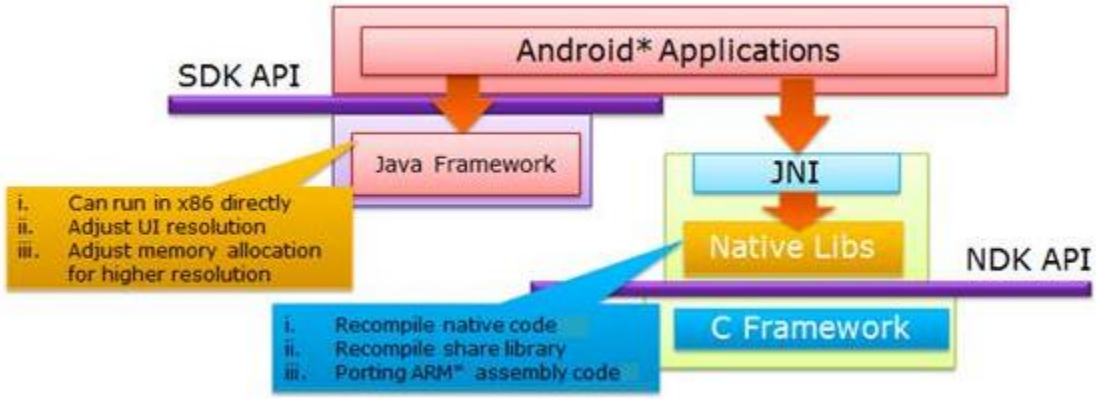


Figure 4-12. Example for porting existing Android NDK application to embedded devices

Native code that consists of C/C++ code only that is not related to hardware

- Recompile the native code to run the application on embedded platform successfully.
- Open the NDK project and search for Android.mk file and add APP_ABI:=armeabi armeabi-v7a x86 in Android.mk and recompile the native code with ndk-build.
- If the Android.mk file is not found, use the ndk-build APP_ABI="armeabi armeabi-v7a x86" command to build the project.
- Package the application again with supported x86 platforms.

If native code uses a third-party dynamic linked library, the shared library must be recompiled into embedded platform version (example: x86 version for the Intel Atom platform).

If native code includes assembly code that is highly related to non-embedded platform (example: non-IA platforms), code must be rewritten with IA assembly or C/C++.

5 Integrate pre-built Native libraries to android projects

https://proandroiddev.com/android-ndk-interaction-of-kotlin-and-c-c-5e19e35bac74