Android Native Development Kit & Java Native Interface

Chapter xx

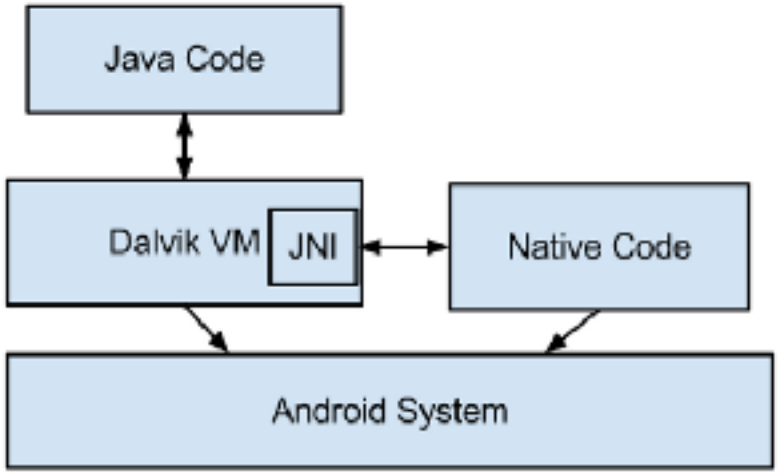
Do Trong Tuan

Content

What is the Android NDK?

- Android provides Native Development Kit (NDK) to support native development in C/C++.
- The NDK provides all the tools (compilers, libraries, and header files) to build apps that access the device natively.
- Native code (in C/C++) is necessary for high performance to overcome the limitations in Java's memory management and performance.

What is the Android NDK?



- Supports “classic” C/C++
- Executed natively, without Interpretation
- Can call and be called from Java
- Can execute assembly code
- Gives access to a few Android API

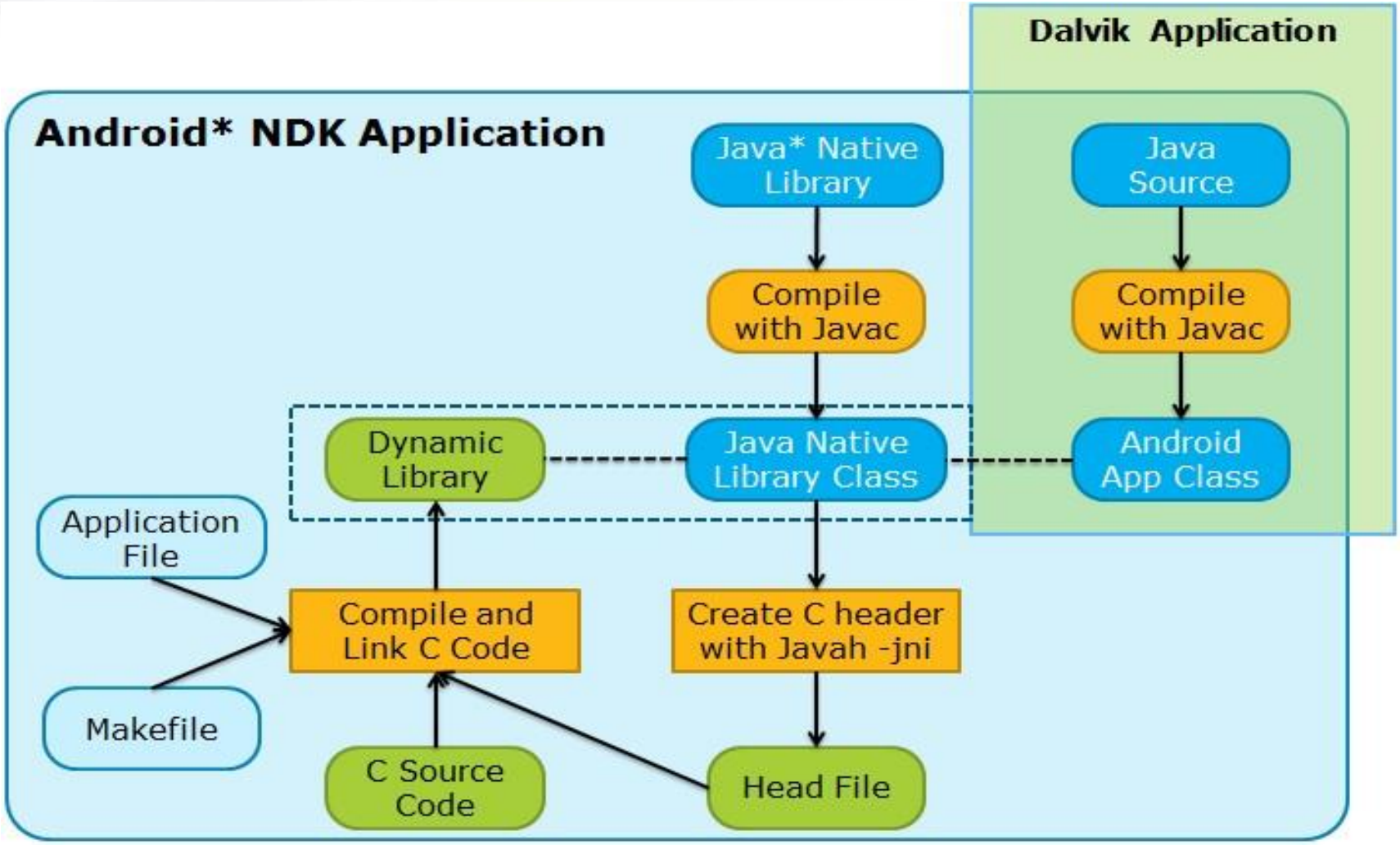
When NDK should be used?

- Need performance(Intensive Apps or Game)
- Port C/C++ code
  - ✓ Existing libraries from the C/C++ ecosystem
  - ✓ Share code with other systems

When NDK should be used?

- ✓ Only when its benefits outrages its drawbacks
- ❖ Disadvantages:
  - Always increases application complexity
  - Very difficult debugging
- ❖ Advantages:
  - + Increases speed
  - + Enable us to port most of the libraries

Two types of Android applications



Installing the NDK

- Setting up all the necessary tools for Android programming, such as JDK, Eclipse, Android SDK, Eclipse ADT
- Download the Android NDK from http://developer.android.com/tools/sdk/ndk/index.html#Installing (e.g., android-ndk-r10c-windows-x86.exe).
- Unzip the downloaded zip file into a directory of your choice (e.g., d:\AndroidNDK). The NDK will be unzipped as d:\AndroidNDK\android-ndk-r10c. The installed directory path is denoted as <NDKROOT>.
- Include the NDK installed directory in the PATH environment variable.

Installing the NDK

Android NDK | Android De

developer.android.com/tools/sdk/ndk/index.html

Training API Guides Reference Tools Google Services Samples

- Download
- Android Studio
- Workflow
- Support Library
- Tools Help
- Revisions
- NDK
- ADK

Android NDK

The NDK is a toolset that allows you to implement parts of your app using native-code languages such as C and C++. For certain types of apps, this can be helpful so you can reuse existing code libraries written in these languages, but most apps do not need the Android NDK.

Before downloading the NDK, you should understand that **the NDK will not benefit most apps**. As a developer, you need to balance its benefits against its drawbacks. Notably, using native code on Android generally does not result in a noticable performance improvement, but it always increases your app complexity. In general, you should only use the NDK if it is essential to your app—never because you simply prefer to program in C/C++.

Typical good candidates for the NDK are CPU-intensive workloads such as game engines, signal processing, physics simulation, and so on. When examining whether or not you should develop in native code, think about your requirements and see if the Android framework APIs provide the functionality that you need.

IN THIS DOCUMENT

Downloads

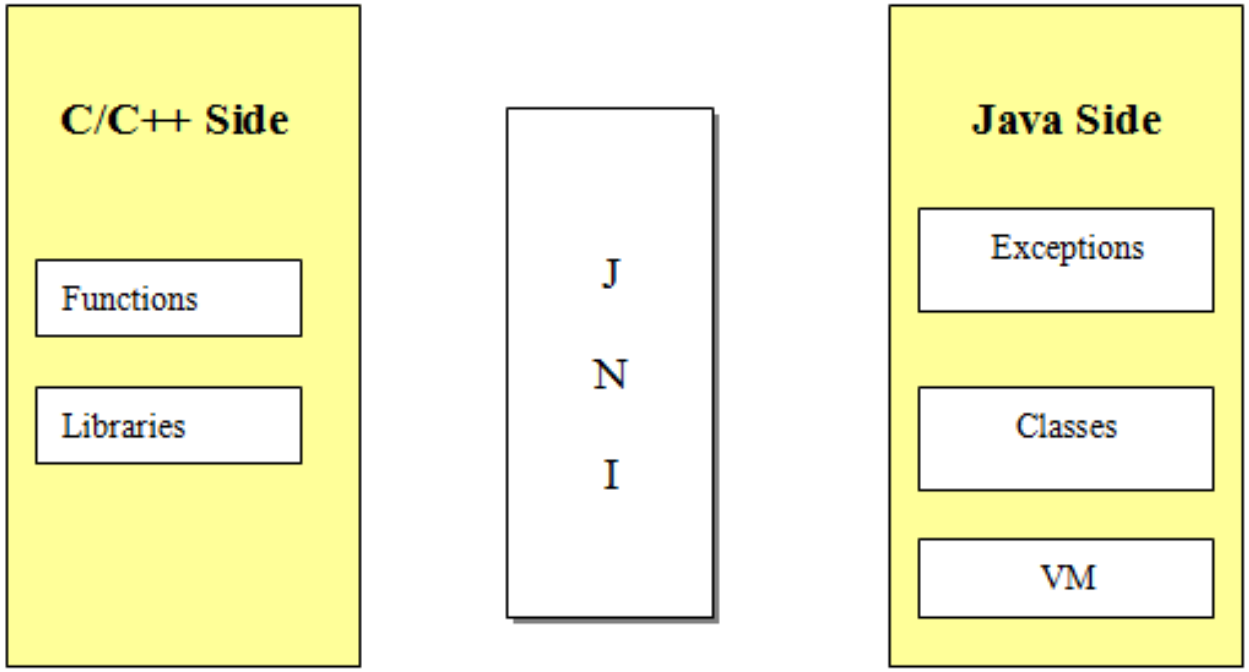
Platform	Package	Size (Bytes)	MD5 Checksum
Windows 32-bit	android-ndk-r10c-windows-x86.exe	433102815	805a04810719886674d3c7bff5eca53f
Windows 64-bit	android-ndk-r10c-windows-x86_64.exe	458925419	af8edf5d316e1bf1a5a72e04a9faec41

EN 6:03 PM 11/30/2014

JNI

JAVA NATIVE INTERFACE

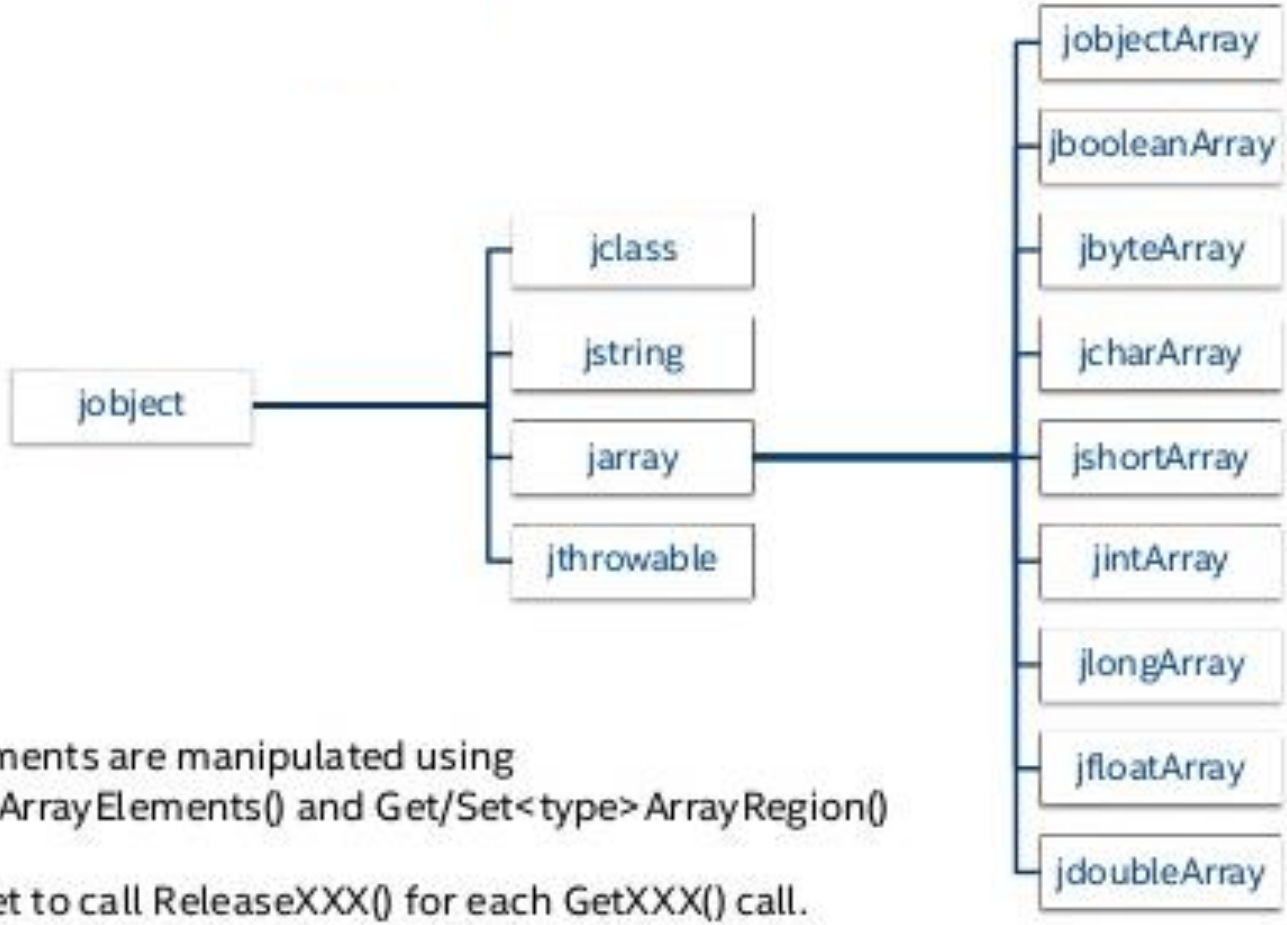
JNI serves as a gateway between native code and Java



JNI 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

JNI Reference Types



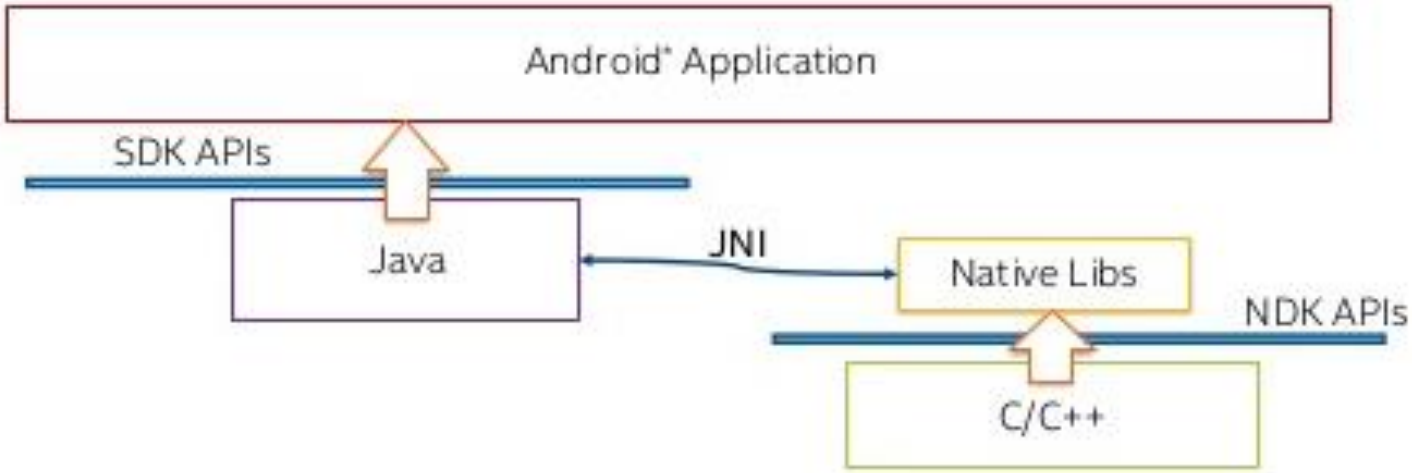
Arrays elements are manipulated using
Get<type>ArrayElements() and Get/Set<type>ArrayRegion()

Don't forget to call ReleaseXXX() for each GetXXX() call.

JNI strategy

- Create the native methods in java class
- Load the library in java class
- Create the function prototypes using javah in a .h file
- Create the final native source file!
- Use your native methods inside Android activity

NDK Application Development

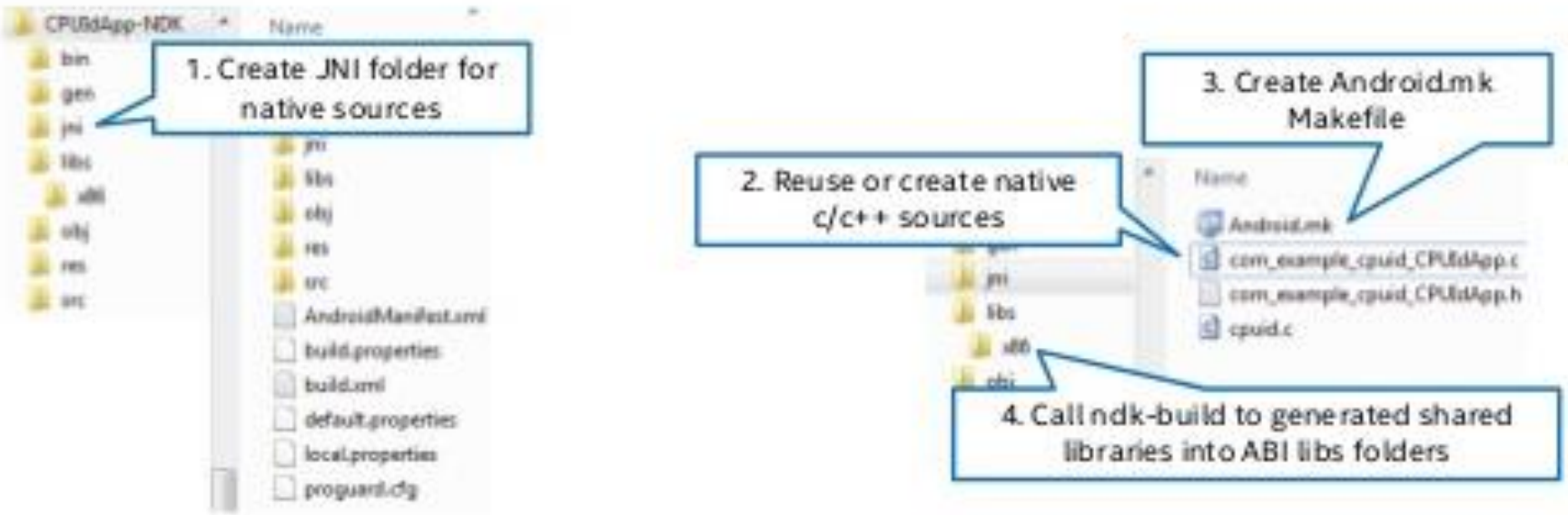


Steps in building an Android NDK App

The steps in building an Android NDK app are:

1. **Create a sub-directory called "jni"** and place all the native sources here.
2. **Create a "Android.mk"** to describe your native sources to the NDK build system.
3. Write Native code for a library
4. **Build your native code by running the "ndk-build"** (in NDK installed directory) script from your project's directory. The build tools copy the stripped, shared libraries needed by your application to the proper location in the application's project directory.
4. Finally, **compile and run your application using the SDK tools** in the usual way. The SDK build tools will package the shared libraries in the application's deployable ".apk" file.

Adding Native Code to Android Project



Integrating Native Function with Java

Declare native methods in your Android™ application (Java™) using the 'native' keyword:

```
public native String stringFromJNI();
```

Provide a native shared library built with the NDK that contains the methods used by your application:

```
libMyLib.so
```

Your application must load the shared library (before use... during class load for example):

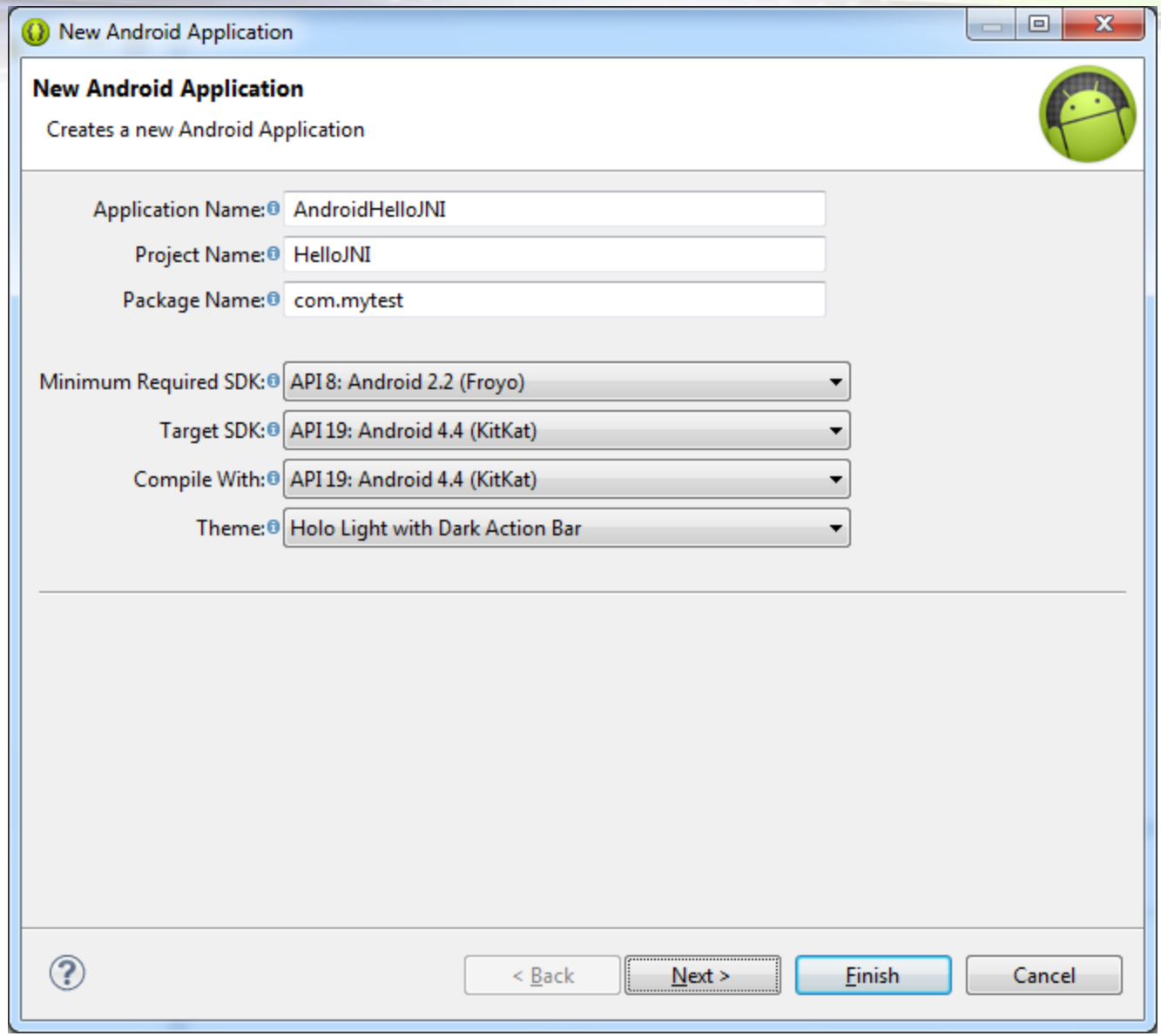
```
static {
    System.loadLibrary("MyLib");
}
```

Write an Android JNI program

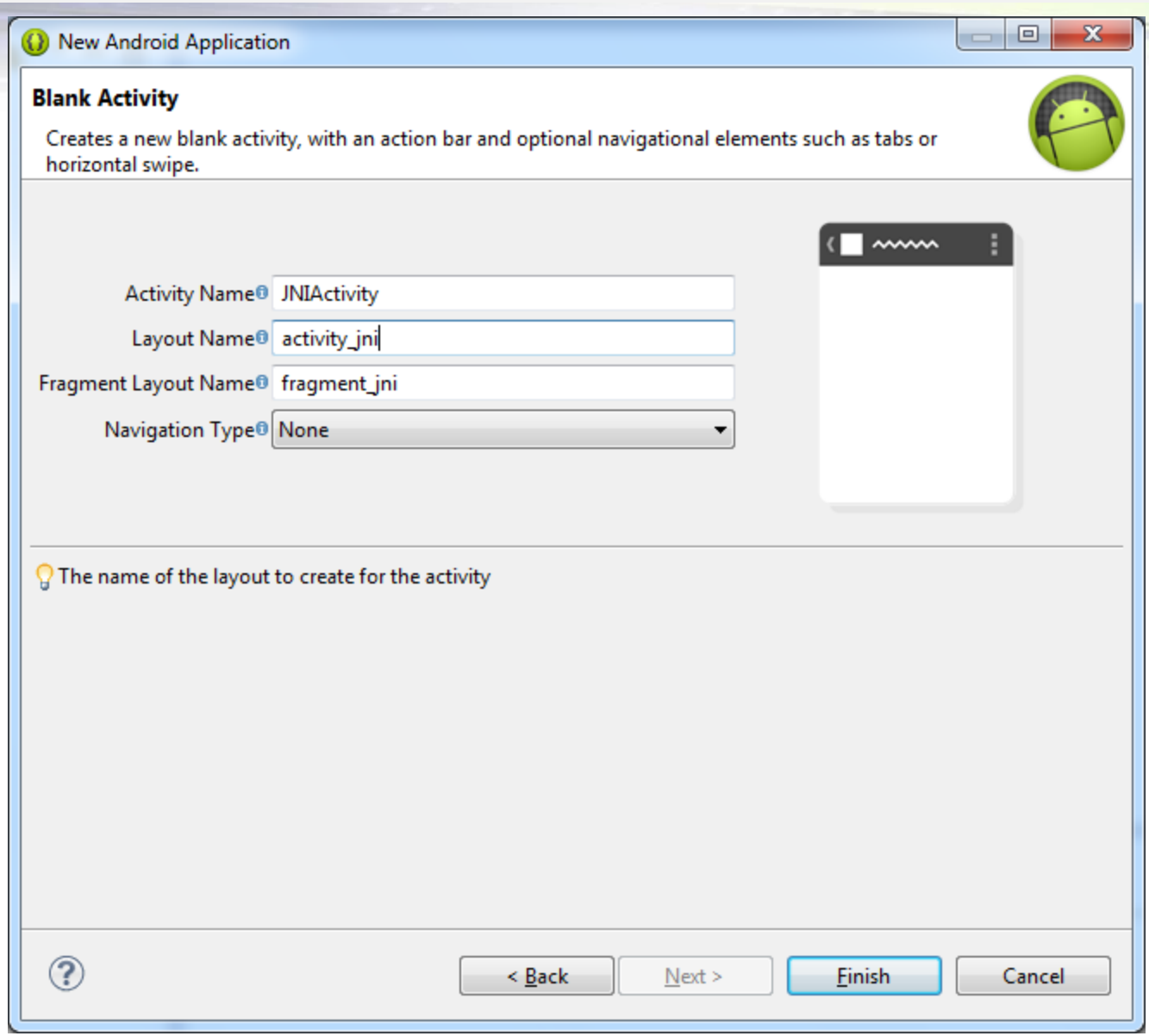
In this example, we shall **create an activity**, that **calls a native method** to **obtain** a string **and displays** the **string** on a **TextView**.

- Create an **Android project** called "**AndroidHelloJNI**", with **application name** "**Hello JNI**" and **package** "**com.mytest**"
- Create an **activity** called "**JNIActivity**" with **Layout** name "**activity_jni**"

Create New Android Project



Set Activity Name



New Project Created

Java - HelloJNI/res/layout/fragment_jni.xml - ADT

File Edit Refactor Source Navigate Search Project Run Window Help

Quick Access | Java

Package Explorer

- appcompat_v7
- HelloJNI
  - src
  - gen [Generated Sources]
  - Android 4.4.2
  - Android Private Libraries
  - assets
  - bin
  - libs
  - res
    - drawable-hdpi
    - drawable-ldpi
    - drawable-mdpi
    - drawable-xhdpi
    - drawable-xxhdpi
    - layout
      - activity_jni.xml
      - fragment_jni.xml
    - menu
    - values
    - values-v11
    - values-v14
    - values-w820dp
  - AndroidManifest.xml
  - ic_launcher-web.png
  - proguard-project.txt
  - project.properties

Error's mark may exist!

JNIActivity.java | fragment_jni.xml

Palette

Nexus One | AppTheme | PlaceholderFragment | 19

Custom Widgets: Large Text, Button

Text Fields | Layouts | Composite | Images & Media | Time & Date | Transitions | Advanced | Other | Custom & Library Views

AndroidHelloJNI

Hello world!

Graphical Layout | fragment_jni.xml

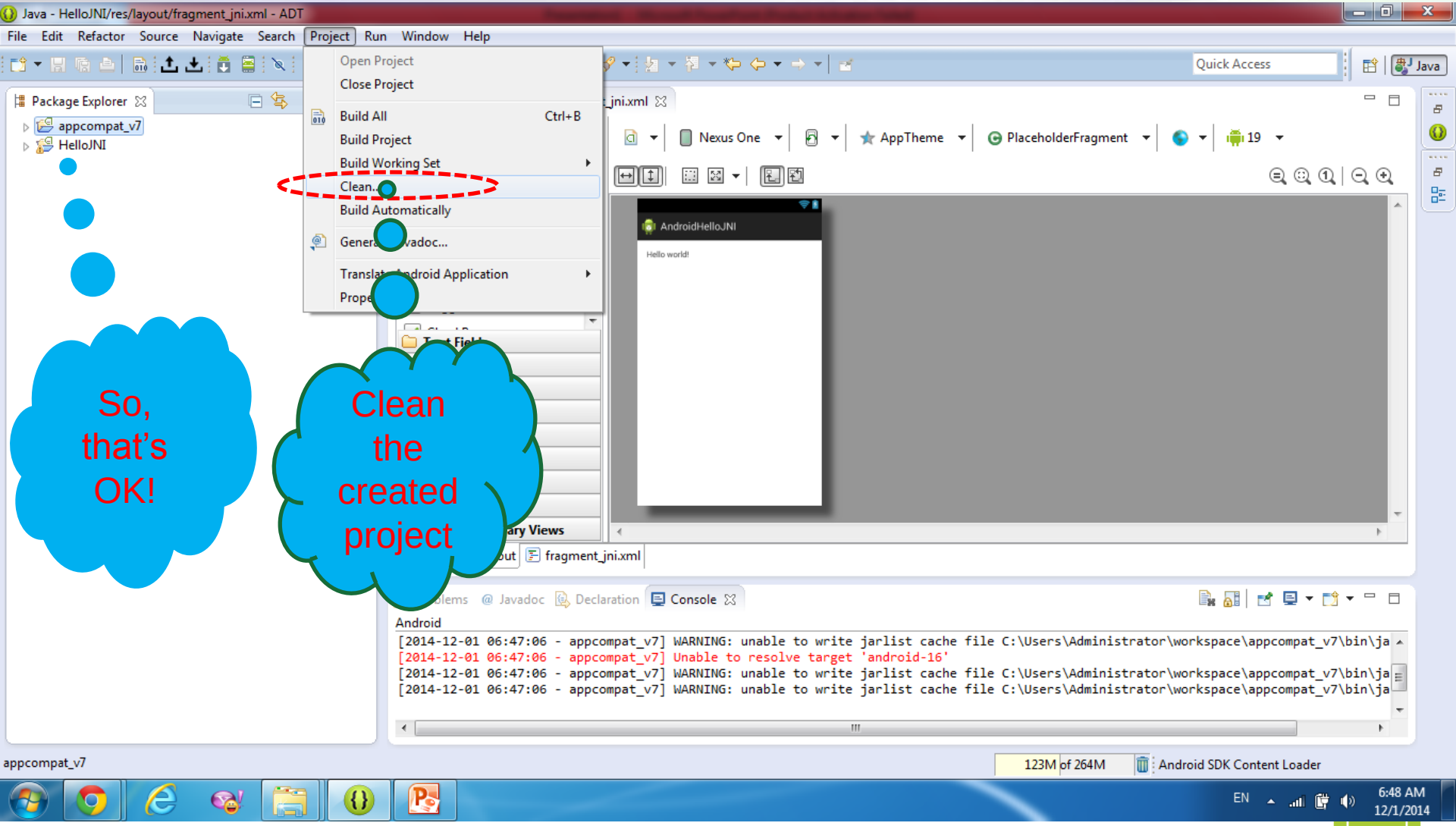
Problems | Javadoc | Declaration | Console

Android

```
[2014-12-01 06:47:06 - appcompat_v7] WARNING: unable to write jarlist cache file C:\Users\Administrator\workspace\appcompat_v7\bin\ja
[2014-12-01 06:47:06 - appcompat_v7] Unable to resolve target 'android-16'
[2014-12-01 06:47:06 - appcompat_v7] WARNING: unable to write jarlist cache file C:\Users\Administrator\workspace\appcompat_v7\bin\ja
[2014-12-01 06:47:06 - appcompat_v7] WARNING: unable to write jarlist cache file C:\Users\Administrator\workspace\appcompat_v7\bin\ja
```

HelloJNI | 205M of 264M | Android SDK Content Loader

Create New Android Project



Create New Android Project

Java - HelloJNI/res/layout/fragment_jni.xml - ADT

File Edit Refactor Source Navigate Search Project Run Window Help

Package Explorer
- appcompat_v7
- HelloJNI
  - src
  - gen [Generated Java Files]
  - Android 4.4.2
  - Android Private Libraries
  - Android Dependencies
  - assets
  - bin
  - libs
  - res
  - AndroidManifest.xml
  - ic_launcher-web
  - proguard-project.txt
  - project.properties

Pure Java Code

JNIActivity.java | fragment_jni.xml

Palette
- Form Widgets: TextView, Large Text, Medium Text, Small Text, Button, Small Button, ToggleButton
- Text Fields
- Layouts
- Composite
- Images & Media
- Time & Date
- Transitions
- Advanced
- Other
- Custom & Library Views

Nexus One | AppTheme | PlaceholderFragment | 19

AndroidHelloJNI

Hello world!

Graphical Layout | fragment_jni.xml

Problems | Javadoc | Declaration | Console

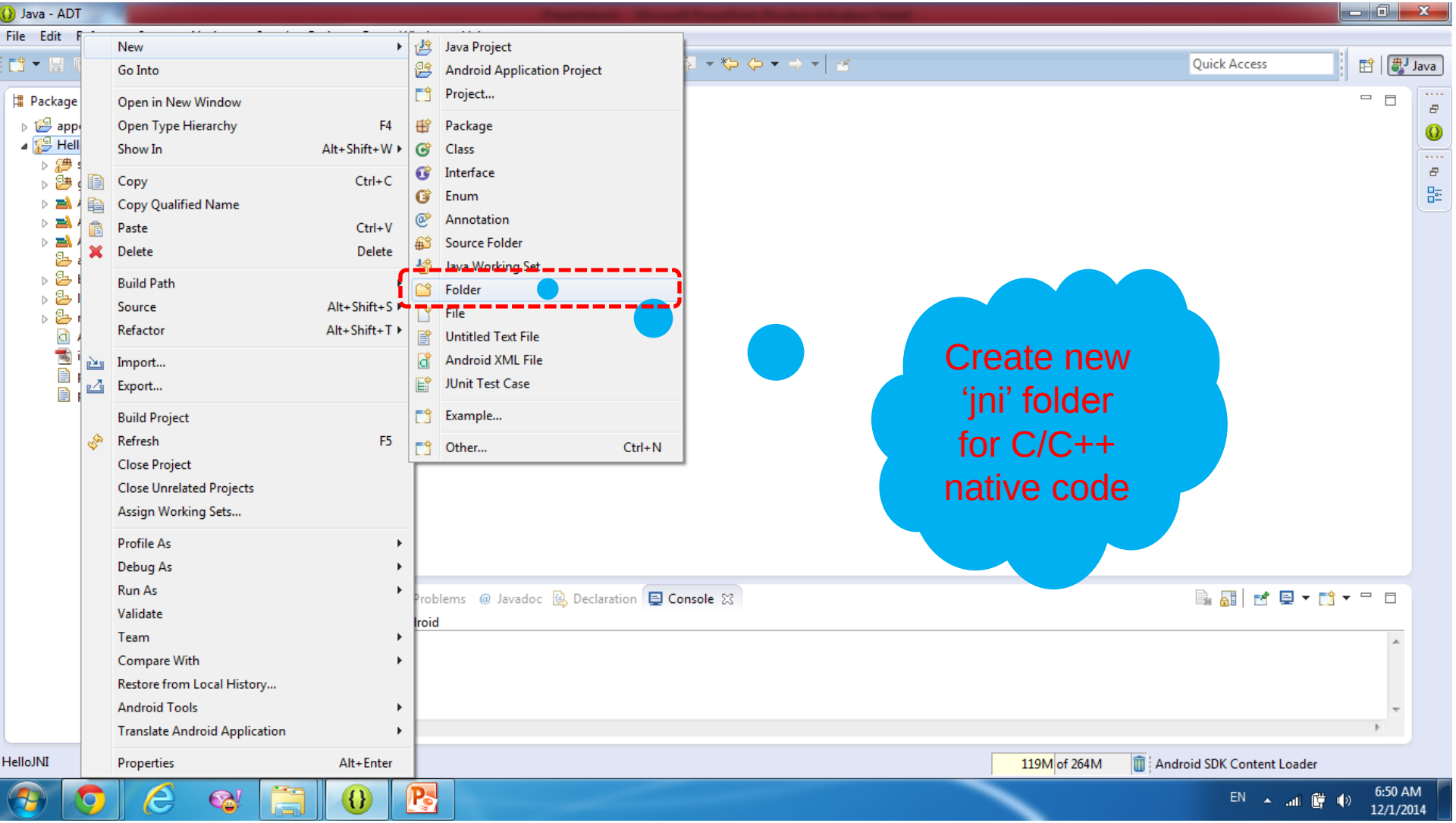
Android

```
[2014-12-01 06:47:06 - appcompat_v7] WARNING: unable to write jarlist cache file C:\Users\Administrator\workspace\appcompat_v7\bin\ja
[2014-12-01 06:47:06 - appcompat_v7] Unable to resolve target 'android-16'
[2014-12-01 06:47:06 - appcompat_v7] WARNING: unable to write jarlist cache file C:\Users\Administrator\workspace\appcompat_v7\bin\ja
[2014-12-01 06:47:06 - appcompat_v7] WARNING: unable to write jarlist cache file C:\Users\Administrator\workspace\appcompat_v7\bin\ja
```

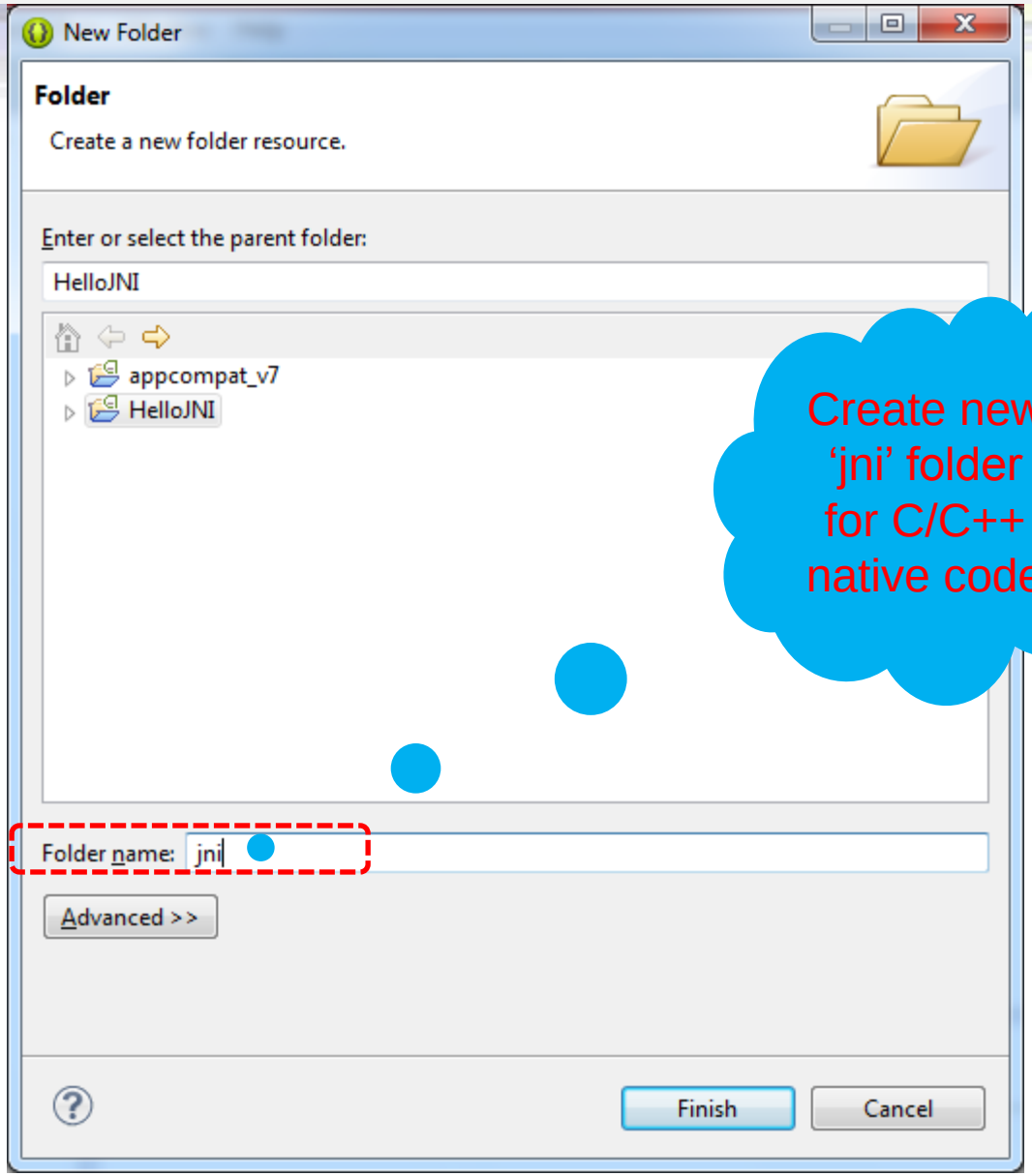
HelloJNI | 100M of 264M | Android SDK Content Loader

EN 6:49 AM 12/1/2014

Create New Android Project

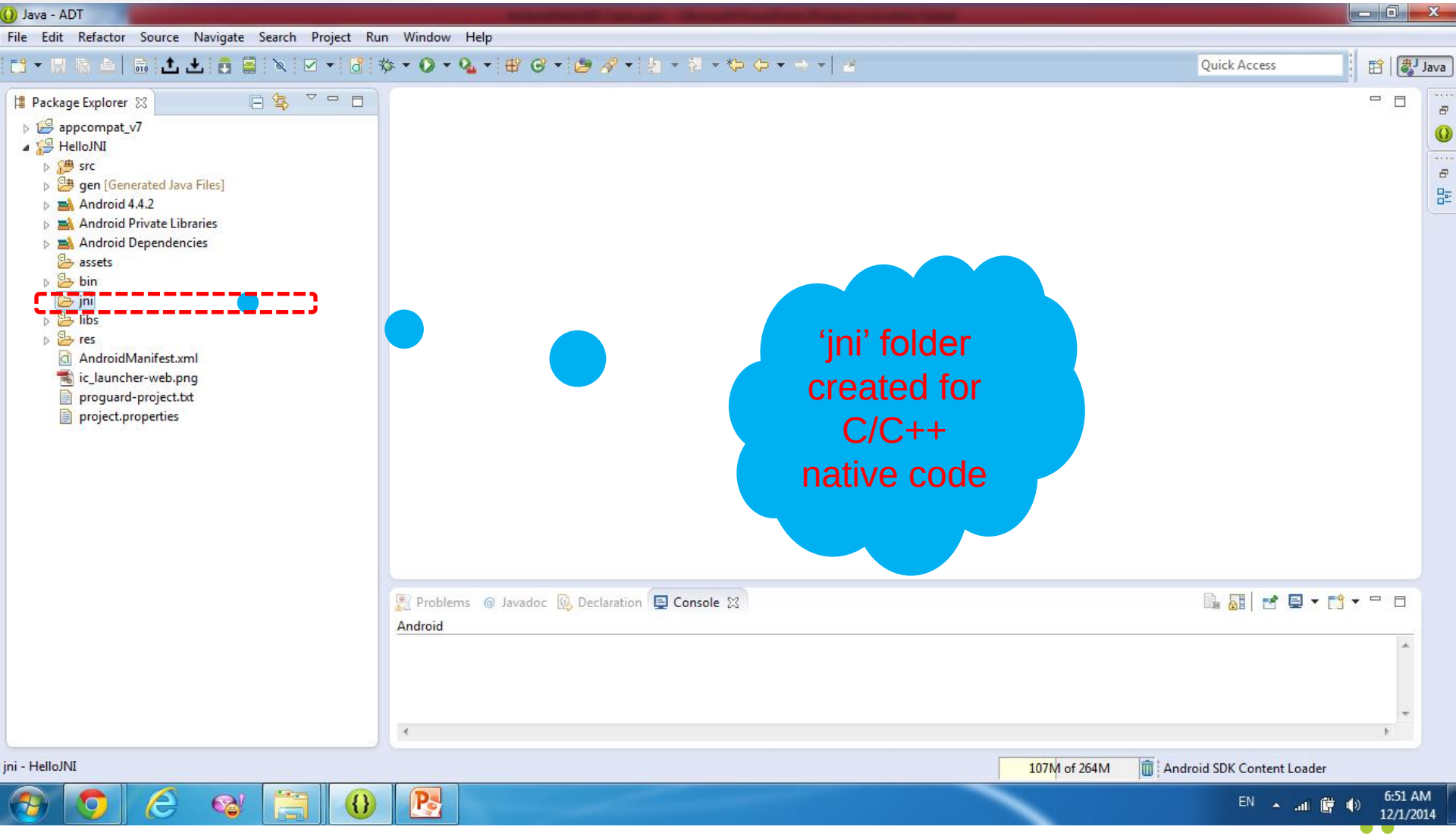


Create New Android Project



Create new 'jni' folder for C/C++ native code

Create New Android Project



Create New Android Project

Java - ADT

File Edit Refactor Source Navigate Search Project Run Window Help

Quick Access | Java

Package Explorer

- appcompat_v7
- HelloJNI
  - src

New
- Open in New Window
- Show In Alt+Shift+W
- Copy Ctrl+C
- Copy Qualified Name
- Paste Ctrl+V
- Delete Delete
- Build Path
- Refactor Alt+Shift+T
- Import...
- Export...
- Refresh F5
- Assign Working Sets...
- Validate
- Profile As
- Debug As
- Run As
- Team
- Compare With
- Restore from Local History...
- Source
- Translate Android Application
- Properties Alt+Enter

Java Project
Android Application Project
Project...
Package
Class
Interface
Enum
Annotation
Source Folder
Java Working Set
Folder
File
Untitled Text File
Android XML File
JUnit Test Case
Example...
Other... Ctrl+N

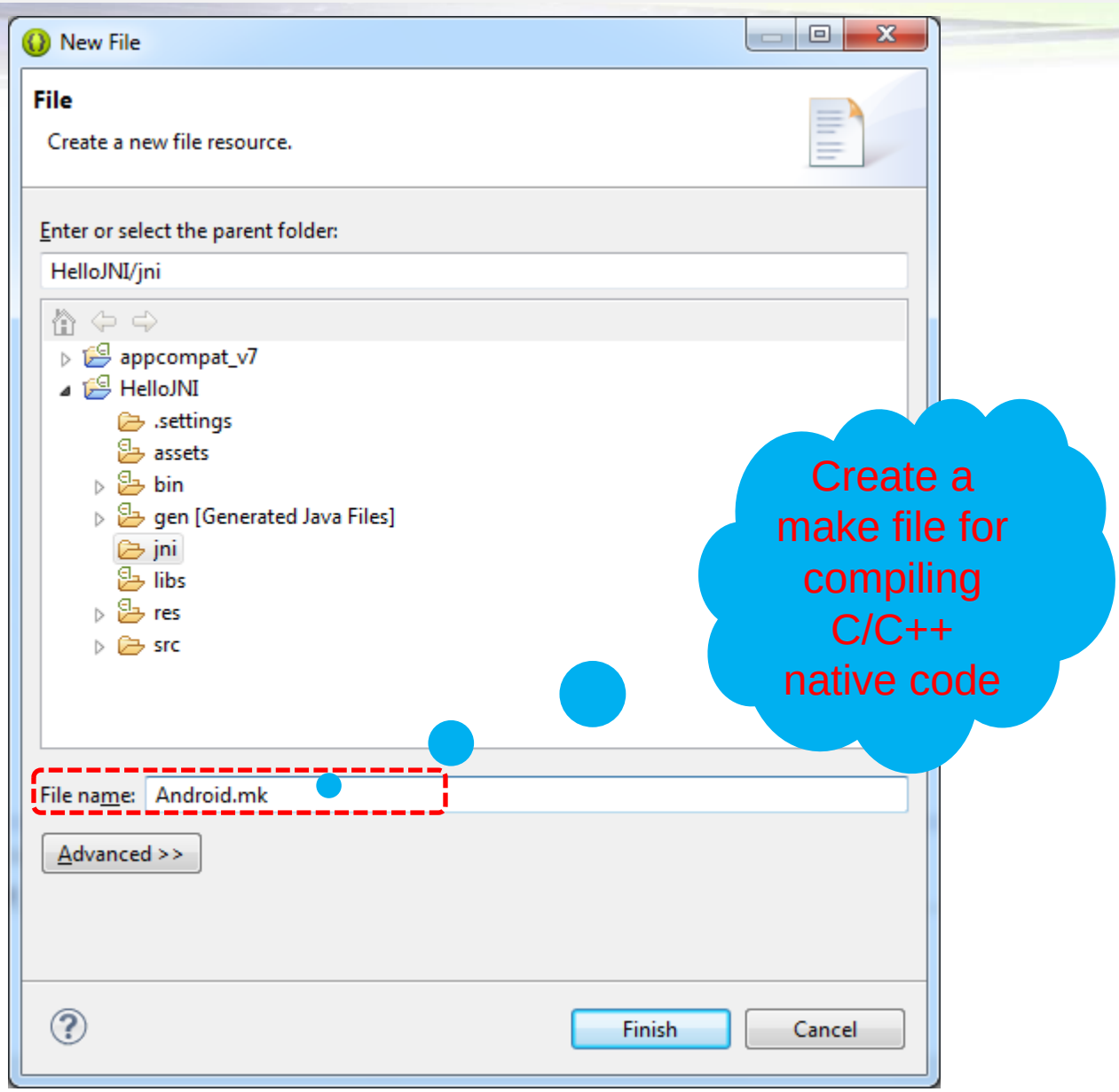
Create a make file for compiling C/C++ native code

jni - HelloJNI

96M of 264M | Android SDK Content Loader

EN 6:52 AM 12/1/2014

Create New Android Project



Create a make file for compiling C/C++ native code

Create New Android Project

Java - HelloJNI/jni/Android.mk - ADT

File Edit Refactor Navigate Search Project Run Window Help

Package Explorer

- appcompat_v7
- HelloJNI
  - src
  - gen [Generated Java Files]
  - Android 4.4.2
  - Android Private Libraries
  - Android Dependencies
  - assets
  - bin
  - jni
    - Android.mk
  - libs
  - res
  - AndroidManifest.xml
  - ic_launcher-web.png
  - proguard-project.txt
  - project.properties

*Android.mk

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE := myjni
LOCAL_SRC_FILES := HelloJNI.c

include $(BUILD_SHARED_LIBRARY)
```

'Android.mk' created for compiling C/C++ native code

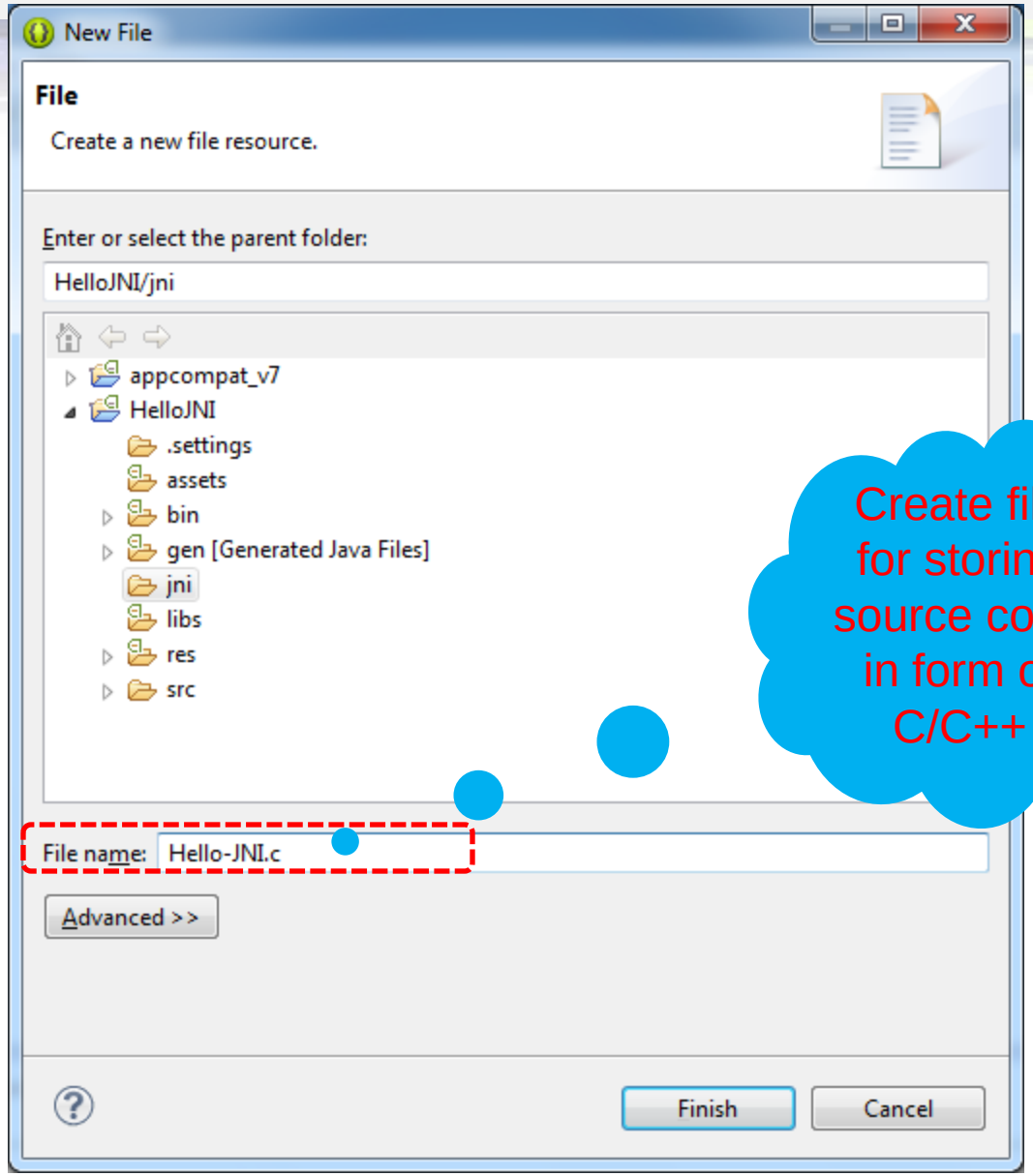
Problems | Javadoc | Declaration | Console

Android

Writable | Smart Insert | 8 : 1 | 93M of 264M | Android SDK Content Loader

EN 6:53 AM 12/1/2014

Create New Android Project



Create file for storing source code in form of C/C++

Create New Android Project

Java - HelloJNI/jni/Hello-JNI.c - ADT

File Edit Refactor Source Navigate Search Project Run Window Help

Package Explorer

- appcompat_v7
- HelloJNI
  - src
  - gen [Generated Java Files]
  - Android 4.4.2
  - Android Private Libraries
  - Android Dependencies
  - assets
  - bin
  - jni
    - Android.mk
    - Hello-JNI.c
  - libs
  - res
  - AndroidManifest.xml
  - ic_launcher-web.png
  - proguard-project.txt
  - project.properties

*Hello-JNI.c

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

JNIEXPORT jstring JNICALL Java_com_mytest_JNIActivity_getMessage(JNIEnv *env, jobject thisObj){
    return (*env)->NewStringUTF(env,"Hello Android from Native Code!");
}
```

'Hello-JNI.c' created for storing source code in form of C/C++

Problems | Javadoc | Declaration | Console

Android

Writable | Smart Insert | 7 : 1 | 95M of 264M | Android SDK Content Loader

EN 6:55 AM 12/1/2014

Create New Android Project

Compile Native code in command line using syntax: ndk-build

Java - HelloJNI/jni/Android.mk - ADT

File Edit Refactor Navigate Search Project Run Window Help

Package Explorer

- appcompat_v7
- HelloJNI
  - src
  - gen [Generated Java Files]
  - Android 4.4.2
  - Android Private Libraries
  - Android Dependencies
  - assets
  - bin
  - jni
    - Android.mk
    - Hello-JNI.c
  - libs
  - res
  - AndroidManifest.xml
  - ic_launcher-web.png
  - proguard-project.txt
  - project.properties

*Hello-JNI.c *Android.mk

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE := myjni
LOCAL_SRC_FILES := HelloJNI.c

include $(BUILD_SHARED_LIBRARY)
```

Administrator: C:\Windows\System32\cmd.exe

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Administrator\workspace\HelloJNI\jni>ndk-build
make.exe: *** No rule to make target `C:/Users/Administrator/workspace/HelloJNI/
/jni/HelloJNI.c', needed by `C:/Users/Administrator/workspace/HelloJNI//obj/loca
l/armeabi/objs/myjni/HelloJNI.o'.  Stop.

C:\Users\Administrator\workspace\HelloJNI\jni>
```

Check to find posible solution!

Error's message appeared ???

Problems @ Javad

Android

Writable | Smart Insert | 8 : 1 | 88M of 264M | Android SDK Content Loader

EN 6:57 AM 12/1/2014

Create New Android Project

Re- compile Native code in command line using syntax: ndk-build

Java - HelloJNI/jni/Android.mk - ADT

File Edit Refactor Navigate Search Project Run Window Help

Package Explorer

- appcompat_v7
- HelloJNI
  - src
  - gen [Generated Java Files]
  - Android 4.4.2
  - Android Private Libraries
  - Android Dependencies
  - assets
  - bin
  - jni
    - Android.mk
    - HelloJNI.c
  - libs
  - res
  - AndroidManifest.xml
  - ic_launcher-web.png
  - proguard-project.txt
  - project.properties

HelloJNI.c | Android.mk

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE := myjni
LOCAL_SRC_FILES := HelloJNI.c

include $(BUILD_SHARED_LIBRARY)
```

Administrator: C:\Windows\System32\cmd.exe

```
C:\Users\Administrator\workspace\HelloJNI\jni>ndk-build
[armeabi] Compile thumb  : myjni <= HelloJNI.c
[armeabi] SharedLibrary  : libmyjni.so
[armeabi] Install        : libmyjni.so => libs/armeabi/libmyjni.so

C:\Users\Administrator\workspace\HelloJNI\jni>_
```

Change the file name to 'HelloJNI.c'

So, the library is compiled successfully!!

Problems | Javadoc

Android

Writable | Smart Insert | 8 : 1 | 91M of 264M | Android SDK Content Loader

EN 6:58 AM 12/1/2014

Create New Android Project

Java - HelloJNI/jni/Android.mk - ADT

File Edit Refactor Source Navigate Search Project Run Window Help

Package Explorer

- appcompat_v7
- HelloJNI
  - src
  - gen [Generated Java Files]
  - Android 4.4.2
  - Android Private Libraries
  - Android Dependencies
  - assets
  - bin
  - jni
    - Android.mk
    - HelloJNI.c
  - libs
  - obj
    - local
      - armeabi
        - objs
        - libmyjni.so
  - res
  - AndroidManifest.xml
  - ic_launcher-web.png
  - proguard-project.txt
  - project.properties

HelloJNI.c | Android.mk

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE := myjni
LOCAL_SRC_FILES := HelloJNI.c

include $(BUILD_SHARED_LIBRARY)
```

Administrator: C:\Windows\System32\cmd.exe

```
C:\Users\Administrator\workspace\HelloJNI\jni>ndk-build
[armeabi] Compile thumb  : myjni <= HelloJNI.c
[armeabi] SharedLibrary  : libmyjni.so
[armeabi] Install        : libmyjni.so => libs/armeabi/libmyjni.so

C:\Users\Administrator\workspace\HelloJNI\jni>_
```

The new library "libmyjni.so" created

Problems | Java

Android

HelloJNI

98M of 264M | Android SDK Content Loader

EN 6:59 AM 12/1/2014

Create New Android Project

Now, we integrate native library into Android Project!

```
package com.mytest;

import android.support.v7.app.ActionBarActivi
import android.support.v4.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

public class JNIActivity extends ActionBarActivity {
    static{
        System.loadLibrary("myjni");    // Load native library into Android Project
    }
    public native String getMessage();  // A native method that returns a Java String

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

        TextView textView = new TextView(this);
        textView.setText(getMessage());
        setContentView(textView);
```

1. Load the native library
2. Declare native method(s)

Create New Android Project

Java - HelloJNI/src/com/mytest/JNIActivity.java - ADT

File Edit Refactor Source Navigate Search Project Run Window Help

Package Explorer

- appcompat_v7
- HelloJNI
  - src
    - com.mytest
      - JNIActivity.java
  - gen [Generated Java Files]
  - Android 4.4.2
  - Android Private Libraries
  - Android Dependencies
  - assets
  - bin
  - jni
  - libs
  - obj
  - res
  - AndroidManifest.xml
  - ic_launcher-web.png
  - proguard-project.txt
  - project.properties

HelloJNI.c | Android.mk | *JNIActivity.java

```
package com.mytest;

import android.support.v7.app.ActionBarActivi
import android.support.v4.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

public class JNIActivity extends ActionBarActivity {
    static{
        System.loadLibrary("myjni");    // Load native library into Android Project
    }
    public native String getMessage();  // A native method that returns a Java String

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

        TextView textView = new TextView(this);
        textView.setText(getMessage());
        setContentView(textView);
```

Problems | Javadoc | Declaration | Console

Android

Writable | Smart Insert | 27 : 9 | 97M of 264M | Android SDK Content Loader

Now, we integrate native library into Android Project!

Implement operations/actions using native method:
-> getMessage()
To feed predefined string into a TextView

EN 7:03 AM 12/1/2014

Create New Android Project

Then, we clean project to make sure no error exist!

Java - HelloJNI/src/com/mytest/JNIActivity.java - ADT

File Edit Refactor Source Navigate Search Project Run Window Help

Open Project
Close Project
Build All Ctrl+B
Build Project
Build Working Set
Clean...
Build Automatically
Generate Javadoc...
Translate Android Application
Properties

Package Explorer
- appcompat_v7
- HelloJNI
  - src
    - com.mytest
      - JNIActivity.java
  - gen [Generated Java Files]
  - Android 4.4.2
  - Android Private Libraries
  - Android Dependencies
  - assets
  - bin
  - jni
  - libs
  - obj
  - res
  - AndroidManifest.xml
  - ic_launcher-web.png
  - proguard-project.txt
  - project.properties

*JNIActivity.java

```
public class JNIActivity extends ActionBarActivity {
    static{
        System.loadLibrary("myjni");    // Load native library into Android Project
    }
    public native String getMessage();  // A native method that returns a Java String

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

        TextView textView = new TextView(this);
        textView.setText(getMessage());
        setContentView(textView);
```

Problems | Javadoc | Declaration | Console

Android

Writable | Smart Insert | 14 : 12 | 137M of 264M | Android SDK Content Loader

EN 7:04 AM 12/1/2014

Create New Android Project

Java - HelloJNI/src/com/mytest/JNIActivity.java - ADT

File Edit Ref ... ow Help

Quick Access | Java

Package Ex

appco
HelloJ
src
ger
An
An
An
ass
bin
jni
libs
obj
res
An
ic_
pro
pro

HelloJNI

New
Go Into
Open in New Window
Open Type Hierarchy F4
Show In Alt+Shift+W
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Build Path
Source Alt+Shift+S
Refactor Alt+Shift+T
Import...
Export...
Build Project
Refresh F5
Close Project
Close Unrelated Projects
Assign Working Sets...
Profile As
Debug As
Run As
Validate
Team
Compare With
Restore from Local History...
Android Tools
Translate Android Application
Properties Alt+Enter

1 Android Application
2 Android JUnit Test
3 Android Native Application
4 Java Applet Alt+Shift+D, A
5 Java Application Alt+Shift+D, J
6 JUnit Test Alt+Shift+D, T
Debug Configurations...

loJNI.c | Android.mk | *JNIActivity.java

```
package com.mytest;

import android.support.v7.app.ActionBarActivity;
import android.support.v4.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

public class JNIActivity extends ActionBarActivity {
    static{
        System.loadLibrary("myjni");    // Load native library into A
    }
    public native String getMessage();  // A native method that re

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

        TextView textView = new TextView(this);
        textView.setText(getMessage());
        setContentView(textView);
```

Finally, we build and run the integrated application which is written by Java and Native (C/C++) code !!

92M of 264M | Android SDK Content Loader

EN 7:05 AM 12/1/2014

Step 1: Write an Android JNI program

- This JNI program uses a static initializer to load a shared library ("myjni.dll" in Windows or "libmyjni.so" in Unixes).
- It declares a native method called getMessage(), which returns a String to be as the TextView's message.
- The onCreate() method declares a TextView, and invokes the native method getMessage() to set its text.

Step 2: C Implementation - HelloJNI.c

- Create the following C program called "**HelloJNI.c**" under the **"jni" directory** (right-click on the "jni" folder ⇒ New ⇒ File):

```c
#include <jni.h>
#include "include/HelloJNI.h"

JNIEXPORT jstring JNICALL Java_com_mytest_JNIActivity_getMessage
          (JNIEnv *env, jobject thisObj) {
   return (*env)->NewStringUTF(env, "Hello from native code!");
}
```

- The native program gets and returns a JNI jstring via JNI environment interface function **NewStringUTF()** with an input C-string "Hello from native code!"

Step 3: Create an Android makefile - Android.mk

- Create an Android makefile called "**Android.mk**" under the **"jni" directory** (right-click on "jni" folder ⇒ New ⇒ File), as follows:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := myjni
LOCAL_SRC_FILES := HelloJNI.c

include $(BUILD_SHARED_LIBRARY)
```

- In the above makefile, "**myjni**" is the name of our shared library (**used in System.loadLibrary()**), and "**HelloJNI.c**" is the **source file**.

Step 4: Build NDK

- **Start a CMD shell**
- **Change directory** to the project's root directory
- **Run "ndk-build"** script provided by Android NDK (the Android NDK installed directory shall be in the PATH).

```
// Change directory to <project-root>
> ndk-build
Compile thumb : myjni <= HelloJNI.c
SharedLibrary  : libmyjni.so
Install        : libmyjni.so => libs/armeabi/libmyjni.so
```

NOTES:

- Use "**ndk-build --help**" to display the command-line options.
- Use "**ndk-build V=1**" to display the build messages.
- Use "**ndk-build -B**" to perform a force re-built.

Step 5: Run the Android App

- Run the android app, via "Run As" ⇒ "Android Application". You shall see the message from the native program appears on the screen.
- Check the "LogCat" panel to confirm that the shared library "libmyjni.so" is loaded.

```
...: Trying to load lib /data/data/com.example.androidhellojni/lib/libmyjni.so ...

...: Added shared lib /data/data/com.example.androidhellojni/lib/libmyjni.so ...
```

Java integrated with C via JNI

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

```java
public class HelloJNI {
   static {
      System.loadLibrary("hello"); // Load native library at runtime
                                   // hello.dll (Windows) or libhello.so (Unixes)
   }

   // Declare a native method sayHello() that receives nothing and returns void
   private native void sayHello();

   // Test Driver
   public static void main(String[] args) {
      new HelloJNI().sayHello();  // invoke the native method
   }
}
```

- The static initializer invokes **System.loadLibrary()** to **load** the **native library "hello"** (which contains the native method sayHello()) during the class loading. It will be mapped to "hello.dll" in Windows; or "libhello.so" in Unixes.

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

- Next, we **declare** the **method** **sayHello()** as a native instance method, via keyword native, which denotes that this method is implemented in another language.
- A **native method** does not **contain a body**. The sayHello() is contained in the native library loaded.

- The **main() method** allocate an instance of HelloJNI and invoke the native method sayHello().
- Compile the "HelloJNI.java" into "HelloJNI.class"

```
> javac HelloJNI.java
```

Step 2: Create the C/C++ Header file - HelloJNI.h

- **Run** javah utility on the class file to create a header file for C/C++ programs:

```
> javah HelloJNI
```

- The output is HelloJNI.h as follows:

```c
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloJNI */

#ifndef _Included_HelloJNI
#define _Included_HelloJNI
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloJNI
 * Method:    sayHello
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloJNI_sayHello(JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Step 2: Create the C/C++ Header file - HelloJNI.h

- The header declares a C function Java_HelloJNI_sayHello as follows:

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

- The naming convention for C function is **Java_{package_and_classname}_{function_name}**(***JNI arguments***).
- The dot in package name shall be replaced by underscore.

The arguments:

- JNIEnv*: reference to JNI environment, which lets you access all the JNI functions.
- jobject: reference to "this" Java object.

Step 2: Create the C/C++ Header file - HelloJNI.h

- 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 and C++ have different function naming protocols as C++ support function overloading and uses a name mangling scheme to differentiate the overloaded functions.

Step 3: C Implementation - HelloJNI.c

```c
#include <jni.h>
#include <stdio.h>
#include "HelloJNI.h"

// Implementation of native method sayHello() of HelloJNI class
JNIEXPORT void JNICALL Java_HelloJNI_sayHello(JNIEnv *env, jobject thisObj) {
   printf("Hello World!\n");
   return;
}
```

- **Save the C program** as "**HelloJNI.c**".
- The header "jni.h" is available under the "<JAVA_HOME>\include" and "<JAVA_HOME>\include\win32" directories, where <JAVA_HOME> is your JDK installed directory (e.g., "c:\program files\java\jdk1.7.0").
- The C function simply prints the message "Hello world!" to the console.
- **Compile the C program** - this depends on the C compiler you used.

Step 4: Run the Java Program

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

- You may need to specify the library path of the "hello.dll" via VM option -Djava.library.path=<path_to_lib>, as shown above.

Java integrated C/C++ Mixture via JNI

Step 1: Write a Java Class that uses Native Codes - HelloJNICpp.java

```java
public class HelloJNICpp {
   static {
      System.loadLibrary("hello"); // hello.dll (Windows) or libhello.so (Unixes)
   }

   // Native method declaration
   private native void sayHello();

   // Test Driver
   public static void main(String[] args) {
      new HelloJNICpp().sayHello();  // Invoke native method
   }
}
```

- Compile the HelloJNICpp.java into HelloJNICpp.class

```
> javac HelloJNICpp.java
```

Step 2: Create the C/C++ Header file - HelloJNICpp.h

```
> javah HelloJNICpp
```

- The resultant header file "HelloJNICpp.h" declares the native function as:

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

Step 3: C/C++ Implementation - HelloJNICppImpl.h, HelloJNICppImpl.cpp, and HelloJNICpp.c

- We shall implement the program in C++ (in "HelloJNICppImpl.h" and "HelloJNICppImpl.cpp"), but use a C program ("HelloJNICpp.c") to interface with Java.

- C++ Header - "HelloJNICppImpl.h"

```
#ifndef _HELLO_JNI_CPP_IMPL_H
#define _HELLO_JNI_CPP_IMPL_H

#ifdef __cplusplus
        extern "C" {
#endif
        void sayHello ();
#ifdef __cplusplus
        }
#endif

#endif
```

Step 3: C/C++ Implementation - HelloJNICppImpl.h, HelloJNICppImpl.cpp, and HelloJNICpp.c

- We shall implement the program in C++ (in "HelloJNICppImpl.h" and "HelloJNICppImpl.cpp"), but use a C program ("HelloJNICpp.c") to interface with Java.

- C++ Implementation - "HelloJNICppImpl.cpp"

```
#include "HelloJNICppImpl.h"
#include  <iostream>

using namespace std;

void sayHello () {
    cout << "Hello World from C++!" << endl;
    return;
}
```

Step 3: C/C++ Implementation - HelloJNICppImpl.h, HelloJNICppImpl.cpp, and HelloJNICpp.c

- C Program interfacing with Java - "HelloJNICpp.c"

```
#include <jni.h>
#include "HelloJNICpp.h"
#include "HelloJNICppImpl.h"

JNIEXPORT void JNICALL Java_HelloJNICpp_sayHello (JNIEnv *env, jobject thisObj) {
    sayHello();  // invoke C++ function
    return;
}
```

- Compile the C/C++ programs into shared library ("hello.dll" for Windows).

Step 4: Run the Java Program

```
> java HelloJNICpp
or
> java -Djava.library.path=. HelloJNICpp
```

- You may need to specify the library path of the "hello.dll" via VM option -Djava.library.path=<path_to_lib>, as shown above.

End of Lecture

Q&A