
Android NDK is all about injecting high-performance and portable code into your mobile apps by exploiting the maximum speed of the device they run on. This book will show you how to create C/C++-enabled mobile applications and integrate them with Java. The books starts with teaching you how to access native API and port libraries used in some of the most successful Android applications. Next, you will move on to create a real native application project through the complete implementation of a native API and porting existing third-party libraries. Moving forward, you will learn how to access the keyboard and input peripherals and read accelerometer or orientation sensors. Finally, you will dive into more advanced topics such as RenderScript.

What this book will do for you...

- Build your first Android native project from scratch
- Communicate with Java through Java Native Interfaces
- Learn the key design intricacies of creating a native OpenGL ES 2.0 graphics application
- Initialize, play, and record sound and music with OpenSL ES
- Handle input events and sensors to create different interaction types
- Port an existing library on Android by compiling most common C++ frameworks on Android
- Interface and optimize the existing code with RenderScript
- Combine graphics, sound, input, sensors, and physics in your application

The Beginner’s Guide approach...

- Clear step-by-step instructions for the most useful tasks
- Learn by doing – start working right away
- Leave out the boring bits
- Inspiring, realistic examples that give you ideas for your own work
- Tasks and challenges to encourage experimentation


$ 49.99 US
£ 32.99 UK

Sylvain Ratabouil

Android NDK
Second Edition

Discover the native side of Android and inject the power of C/C++ in your applications

Beginner’s Guide
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 2 'Starting a Native Android Project'
- A synopsis of the book’s content

About the Author

Sylvain Ratabouil is an IT consultant, experienced in Android, Java, and C/C++. He has contributed to the development of digital and mobile applications for large companies as well as industrial projects for the space and aeronautics industries. As a technology lover, he is passionate about mobile technologies and cannot live without his Android smartphone.
Android NDK Beginner's Guide

Second Edition

Android NDK is all about injecting high performance and portable code into your mobile apps by exploiting the maximum speed of these mobile devices. Android NDK allows you to write fast code for intensive tasks and port existing code to Android and non-Android platforms. Alternatively, if you have an application with multiple lines of C code, using NDK can considerably reduce the project development process. This is one of the most efficient operating systems for multimedia and games.

This Beginner's Guide will show you how to create applications enabled by C/C++ and integrate them with Java. By using this practical step-by-step guide, and gradually practicing your new skills using the tutorials, tips, and tricks, you will learn how to run C/C++ code embedded in a Java application or in a standalone application.

The books starts by teaching you how to access native API and port libraries used in some of the most successful Android applications. Next, you will move on to create a real native application project through the complete implementation of a native API and porting existing third-party libraries. As we progress through the chapters, you will gain a detailed understanding of rendering graphics and playing sound with OpenGL ES and OpenSL ES, which are becoming the new standard in mobility. Moving forward, you will learn how to access the keyboard and input peripherals, and read accelerometer or orientation sensors. Finally, you will dive into more advanced topics, such as RenderScript.

By the end of the book, you will be familiar enough with the key elements to start exploiting the power and portability of native code.
What This Book Covers

Chapter 1, Setting Up Your Environment, covers all the prerequisite packages installed on our system. This chapter also covers installing the Android Studio bundle, which contains both the Android Studio IDE and the Android SDK.

Chapter 2, Starting a Native Android Project, discusses how to build our first sample application using command-line tools and how to deploy it on an Android device. We also create our first native Android projects using Eclipse and Android Studio.

Chapter 3, Interfacing Java and C/C++ with JNI, covers how to make Java communicate with C/C++. We also handle Java object references in native code using Global references, and we learn the differences of Local references. Finally, we raise and check Java exceptions in native code.

Chapter 4, Calling Java Back from Native Code, calls Java code from native code with the JNI Reflection API. We also process bitmaps natively with the help of JNI and decode a video feed by hand.

Chapter 5, Writing a Fully Native Application, discusses creating NativeActivity that polls activity events to start or stop native code accordingly. We also access the display window natively, such as a bitmap to display raw graphics. Finally, we retrieve time to make the application adapt to device speed using a monotonic clock.

Chapter 6, Rendering Graphics with OpenGL ES, covers how to initialize an OpenGL ES context and bind it to an Android window. Then, we see how to turn libpng into a module and load a texture from a PNG asset.

Chapter 7, Playing Sound with OpenSL ES, covers how to initialize OpenSL ES on Android. Then, we learn how to play background music from an encoded file and in-memory sounds with a sound buffer queue. Finally, we discover how to record and play a sound in a way that is thread-safe and non-blocking.

Chapter 8, Handling Input Devices and Sensors, discusses multiple ways to interact with Android from native code. More precisely, we discover how to attach an input queue to the Native App Glue event loop.

Chapter 9, Porting Existing Libraries to Android, covers how to activate the STL with a simple flag in the NDK makefile system. We port the Box2D library into an NDK module that is reusable among Android projects.

Chapter 10, Intensive Computing with RenderScript, introduces RenderScript, an advanced technology to parallelize intensive computation tasks. We also see how to use predefined RenderScript with built-in Intrinsics, which is currently mainly dedicated to image processing.
Starting a Native Android Project

A man with the most powerful tools in hand is unarmed without the knowledge of their usage. Make, GCC, Ant, Bash, Eclipse...—any new Android programmer needs to deal with this technological ecosystem. Luckily, some of these names may already sound familiar. Indeed, Android is based on many open source components, laid together by the Android Development Kits and their specific tool-set: ADB, AAPT, AM, NDK-Build, NDK-GDB... Mastering them will give us the power to create, build, deploy and debug our own Android applications.

Before diving deeper into native code in the next chapter, let’s discover these tools by starting a new concrete Android project that includes native C/C++ code. Despite Android Studio being the new official Android IDE, its lack of support for native code encourages us to focus mainly on Eclipse.

Therefore, in this chapter, we are going to:

- Build an official sample application and deploy it on an Android device
- Create our first native Android project using Eclipse
- Interface Java with C/C++ using Java Native Interfaces
- Debug a native Android application
- Analyze a native crash dump
- Set up a Gradle project with native code

By the end of this chapter, you should know how to start a new native Android project on your own.
Building NDK sample applications

The simplest way to get started with your new Android development environment is to compile and deploy some of the samples provided with the Android NDK. A possible (and polygonfull) choice is the San Angeles demo, created in 2004 by Jetro Lauha and later ported to OpenGL ES (more information at http://jet.ro/visuals/4k-intros/san-angeles-observation/).

Time for action – compiling and deploying San Angeles sample

Let’s use Android SDK and NDK tools to build a working APK:

1. Open a command-line prompt and go to the San Angeles sample directory inside the Android NDK. All further steps have to be performed from this directory.

   Generate San Angeles project files with the android command:

   ```sh
   cd $ANDROID_NDK/samples/san-angeles
   android update project -p ./
   ```

   ![Command output]

   You may get the following error upon executing this command:

   Error: The project either has no target set or the target is invalid.

   Please provide a --target to the 'android update' command.

   This means that you have not installed all the Android SDK platforms as specified in Chapter 1, Setting Up Your Environment. In which case, either install them using the Android manager tool or specify your own project target, for example, android update project --target 18 -p ./.
2. Compile San Angeles native library with `ndk-build`:

```
root@computer:~/android/ndk/samples/san-angeles
$ make
```

Updated and renamed default.properties to project.properties
Updated local.properties
No project name specified, using Activity name 'DemoActivity'.
If you wish to change it, edit the first line of build.xml.
Added file ./build.xml
Added file ./proguard-project.txt

3. Build and package San Angeles application in **Debug** mode:

```
root@computer:~/android/ndk/samples/san-angeles
$ ant debug
```

Buildfile: /home/root/android/ndk/samples/san-angeles/build.xml

- set-mode-check:
- set-debug-impl:
- pre-compile:
- compile:
  [javac] Compiling 3 source files to /home/root/android/ndk/samples/san-angeles/bin/classes
- package:
  [apkbuilder] Current build type is different than previous build: forced apkbuilder run.
  [apkbuilder] Creating DemoActivity-debug-unaligned.apk and signing it with a debug key...
- debug:

BUILD SUCCESSFUL
Starting a Native Android Project

4. Make sure your Android device is connected or the emulator is started. Then deploy the generated package:

ant install

```
[Android SDK Manager] packt@computer:~$ android ndk install
```

5. Launch SanAngeles application on your device or emulator:

```
adb shell am start -a android.intent.action.MAIN -n com.example.SanAngeles/com.example.SanAngeles.DemoActivity
```

```bash
packt@computer:~$ android ndk install
```

BUILD SUCCESSFUL
Total time: 4 seconds

```
packt@computer:~$ android ndk install
```

Starting: Intent { act=android.intent.action.MAIN cmp=com.example.SanAngeles/.DemoActivity }

--- Downloading the example code ---

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
What just happened?

The old-school San Angeles demo, full of flat-shaded polygons and nostalgia, is now running on your device. With only a few command lines, involving most of the tools needed for the Android development, a full application including native C/C++ code has been generated, compiled, built, packaged, deployed, and launched.

Let’s see this process in detail.

Generating project files with Android manager

We generated project files from an existing code base thanks to the Android manager. The following bullet points give more information regarding this process:

- **build.xml**: This is the Ant file that describes how to compile and package the final application APK file (which stands for *Android PacKage*). This build file contains mainly links to properties and core Android Ant build files.
- **local.properties**: This file contains the Android SDK location. Every time your SDK location changes, this file should be regenerated.
- **proguard-project.txt**: This file contains a default configuration for *Proguard*, a code optimizer and obfuscator for Java code. More information about it can be found at [http://developer.android.com/tools/help/proguard.html](http://developer.android.com/tools/help/proguard.html).
Starting a Native Android Project

- project.properties: This file contains the application target Android SDK version. This file is generated by default from a pre-existing default.properties file in the project directory. If no default.properties exists, then an additional --target <API Target> flag (for example, --target 4 for Android 4 Donut) must be appended to the android create command.

Target SDK version is different from the minimum SDK version. The first version describes the latest Android version for which an application is built, whereas the latter indicates the minimum Android version on which the application is allowed to run. Both can be declared optionally in AndroidManifest.xml file (clause <uses-sdk>) but only the target SDK version is "duplicated" in project.properties.

When creating an Android application, choose carefully the minimum and target Android API you want to support, as this can dramatically change your application capabilities as well as your audience wideness. Indeed, as a result of fragmentation, targets tend to move a lot and faster in Android!

An application that does not target the latest Android version does not mean it will not run on it. However, it will not have access to all the latest features nor all of the latest optimizations.

The Android manager is the main entry point for an Android developer. Its responsibilities are bound to SDK version updates, virtual devices management, and projects management.

They can be listed exhaustively from the command line by executing android -help. Since we have already looked at SDK and AVD management in Chapter 1, Setting Up Your Environment, let’s focus on its project management capabilities:

1. android create project allows creating new Android projects ex-nihilo from the command line. Generated projects contain only Java files but no NDK-related files. A few additional options must be specified to allow for proper generation, such as:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a</td>
<td>Main activity name</td>
</tr>
<tr>
<td>-k</td>
<td>Application package</td>
</tr>
<tr>
<td>-n</td>
<td>Project name</td>
</tr>
<tr>
<td>-p</td>
<td>Project path</td>
</tr>
<tr>
<td>-t</td>
<td>Target SDK version</td>
</tr>
<tr>
<td>-g and -v</td>
<td>To generate Gradle build file instead of Ant and specifying its plugin version</td>
</tr>
</tbody>
</table>
An example of command line to create a new project is as follows:

```
android create project -p ./MyProjectDir -n MyProject -t android-8 -k com.mypackage -a MyActivity
```

2. `android update project` creates project files from existing sources, as shown in the previous tutorial. However, if they already exist it can also upgrade the project target to new SDK versions (that is, the `project.properties` file) and update the Android SDK location (that is, the `local.properties` file). The available flags are slightly different:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-l</td>
<td>Library projects to add</td>
</tr>
<tr>
<td>-n</td>
<td>Project name</td>
</tr>
<tr>
<td>-p</td>
<td>Project path</td>
</tr>
<tr>
<td>-t</td>
<td>Target SDK version</td>
</tr>
<tr>
<td>-s</td>
<td>To update projects in subfolders</td>
</tr>
</tbody>
</table>

We can also append a new library project with the `-l` flag, for example:

```
android update project -p ./ -l ../MyLibraryProject
```

3. `android create lib-project` and `android update lib-project` manage library projects. These kinds of projects are not well adapted for native C/C++ development, especially when it comes to debugging, since NDK has its own way of reusing native libraries.

4. `android create test-project`, `android update test-project`, and `android create uitest-project` manage unit test and UI test projects.

More details about all these options can be found on the Android developer website at http://developer.android.com/tools/help/android.html.

**Compiling native code with NDK-Build**

After generating project files, we then compile our first native C/C++ library (also called *module*) using `ndk-build`. This command, the most essential one to know for NDK development, is basically a Bash script, which:

- Sets up the Android native compilation toolchain based on either GCC or CLang.
- Wraps Make to control native code construction with the help of user-defined Makefiles: `Android.mk` and optional `Application.mk`. By default, NDK-
- Build looks for in the `jni` project directory, where native C/C++ are often located by convention.
NDK-Build generates intermediate object files from C/C++ source files (in the obj directory) and produces the final binary library (.so) in the libs directory. NDK-related build files can be erased with the following command:

```
ndk-build clean
```

For more information about NDK-Build and Makefiles, see Chapter 9, Porting Existing Libraries to Android.

## Building and packaging an application with Ant

An Android application is not composed of native C/C++ code only, but also of Java code. Thus, we have:

- Built Java sources located in the src directory with `javac` (Java Compiler).
- Dexed generated Java bytecode, that is, transforming it into Android Dalvik or ART bytecode with `DX`. Indeed, both Dalvik and ART Virtual Machines (more about these later in this chapter) operate on a specific bytecode, which is stored in an optimized format called `Dex`.
- Packaged Dex files, Android manifest, resources (images, and so on), and native libraries in the final APK file with `AAPT`, also known as the Android Asset Packaging Tool.

All these operations are summarized in one call to Ant: `ant debug`. The result is an APK packaged in debug mode and generated in the bin directory. Other build modes are available (for example, release mode) and can be listed with `ant help`. If you would like to erase temporary Java-related build files (for example, the Java .class), then simply run the following command line:

```
ant clean
```

## Deploying an application package with Ant

A packaged application can be deployed as is with Ant through ADB. The available options for deployment are as follows:

- `ant installd` for debug mode
- `ant installr` for release mode

Beware that an APK cannot overwrite an older APK of the same application if they come from a different source. In such a case, remove the previous application first by executing the following command line:

```
ant uninstall
```
Installation and uninstallation can also be performed directly through ADB, for example:

- `adb install <path to application APK>`: For installing an application for the first time (for example, `bin/DemoActivity-debug.apk` for our sample).
- `adb install -r <path to application APK>`: For reinstalling an application and to keep its data stored on the device.
- `adb uninstall <application package name>`: For uninstalling an application identified by its Application package name (for example, `com.example.SanAngeles` for our sample).

### Launching an application with ADB Shell

Finally, we launched the application thanks to the **Activity Manager (AM)**. AM command parameters that are used to start San Angeles come from the `AndroidManifest.xml` file:

- `com.example.SanAngeles` is the application package name (the same we use to uninstall an application as previously shown).
- `com.example.SanAngeles.DemoActivity` is the launched Activity canonical class name (that is, a simple class name concatenated to its package). Here is a brief example of how these are used:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.SanAngeles"
    android:versionCode="1"
    android:versionName="1.0">
    ...
    <activity android:name=".DemoActivity"
        android:label="@string/app_name">
        ...
    </activity>
</manifest>
```

Because it is located on your device, AM needs to be run through ADB. To do so, the latter features a limited Unix-like shell, which features some classic commands such as `ls`, `cd`, `pwd`, `cat`, `chmod`, or `ps` as well as a few Android specific ones as shown in the following table:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>am</code></td>
<td>The Activity Manager which not only starts Activities but can also kill them, broadcast intent, start/stop profiler, and so on.</td>
</tr>
<tr>
<td><code>dmesg</code></td>
<td>To dump kernel messages.</td>
</tr>
<tr>
<td><code>dumpsys</code></td>
<td>To dump the system state.</td>
</tr>
<tr>
<td><code>logcat</code></td>
<td>To display device log messages.</td>
</tr>
</tbody>
</table>
Starting a Native Android Project

<table>
<thead>
<tr>
<th>run-as &lt;user id&gt; &lt;command&gt;</th>
<th>To run a command with the user id privilege. user id can be an application package name, which gives access to application files (for example, run-as com.example.SanAngeles ls).</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqlite3 &lt;db file&gt;</td>
<td>To open an SQLite Database (it can be combined with run-as).</td>
</tr>
</tbody>
</table>

ADB can be started in one of the following ways:

- With a command in parameter, as shown in step 5 with AM, in which case Shell runs a single command and immediately exits.
- With the adb shell command without a parameter, in which case you can use it as a classic Shell (and, for example, call am and any other command).

ADB Shell is a real 'Swiss Army knife', which allows advanced manipulations on your device, especially with the root access. For example, it becomes possible to observe applications deployed in their "sandbox" directory (that is, the /data/data directory) or to list and kill the currently running processes. Without root access to your phone, possible actions are more limited. For more information, have a look at http://developer.android.com/tools/help/adb.html.

If you know a bit about the Android ecosystem, you may have heard about rooted phones and non-rooted phones. **Rooting** a phone means getting administrative privilege, generally using hacks. Rooting a phone is useful to install a custom ROM version (optimized or modified, for example, Cyanogen) or to perform any sort of (especially dangerous) manipulations that a root user can do (for example, accessing and deleting any file). Rooting is not an illegal operation as such, as you are modifying YOUR device. However, not all manufacturers appreciate this practice, which usually voids the warranty.

**More about Android tooling**

Building San Angeles sample application gives you a glimpse of what Android tools can do. However, behind their somewhat 'rustic' look, more is possible. Information can be found on the Android developer website at http://developer.android.com/tools/help/index.html.
Creating your first native Android project

In the first part of the chapter, we saw how to use Android command-line tools. However, developing with Notepad or VI is not really attractive. Coding should be fun! And to make it so, we need our preferred IDE to perform boring or unpractical tasks. So now we will see how to create a native Android project using Eclipse.

The resulting project is provided with this book under the name Store_Part1.

Time for action – creating a native Android project

Eclipse provides a wizard to help us set up our project:

1. Launch Eclipse. In the main menu, go to File | New | Project....
2. Then, in the opened New project wizard, go to Android | Android Application Project and click on Next.
3. In the next screen, enter project properties as follows and click on Next again:
4. Click on **Next** twice, leaving default options, to go to the **Create activity** wizard screen. Select **Blank activity with Fragment** and click on **Next**.

5. Finally, in the **Blank Activity** screen, enter activity properties as follows:

![Blank Activity wizard](image)

6. Click on **Finish** to validate. After a few seconds, the wizard disappears and the project **Store** is displayed in Eclipse.

7. Add native C/C++ support to the project. Select the project **Store** in the **Package Explorer** view and from its right-click context menu, go to **Android Tools | Add Native Support**....

8. In the opened **Add Android Native Support** popup, set the library name to `com_packtpub_store_Store` and click on **Finish**.

![Add Android Native Support](image)
9. The \texttt{jni} and \texttt{obj} directories are created in the project directory. The first directory contains one makefile \texttt{Android.mk} and one C++ source file \texttt{com_packtpub_store_Store.cpp}.

After adding native support, Eclipse may automatically switch your perspective to C/C++. Therefore, in case your development environment does not look as usual, simply check your perspective in the Eclipse's top-right corner. You can work on an NDK project from either a Java or C/C++ perspective without any trouble.

10. Create a new Java class \texttt{Store} in \texttt{src/com/packtpub/store/Store.java}. From within a static block, load the \texttt{com_packtpub_store_Store} native library:

   ```java
   package com.packtpub.store;

   public class Store {
       static {
           System.loadLibrary("com_packtpub_store_Store");
       }
   }
   ```

11. Edit \texttt{src/com/packtpub/store/StoreActivity.java}. Declare and initialize a new instance of \texttt{Store} in \texttt{activity's onCreate()}. Since we do not need them, remove the \texttt{onCreateOptionsMenu()} and \texttt{onOptionsItemSelected()} methods that may have been created by the Eclipse project creation wizard:

   ```java
   package com.packtpub.store;

   public class StoreActivity extends Activity {
       private Store mStore = new Store();

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

           if (savedInstanceState == null) {
               getFragmentManager().beginTransaction()
                   .add(R.id.container, new PlaceholderFragment())
                   .commit();
           }
   }
   ```
12. Connect your device or emulator and launch the application. Select Store in the Package Explorer view and then navigate to Run | Run As | Android Application from the Eclipse main menu. Alternatively, click on the Run button in the Eclipse toolbar.

13. Select the application type Android Application and click on OK to get the following screen:
What just happened?

In only a few steps, our first native Android project has been created and launched thanks to Eclipse.

1. The Android project creation wizard helps get you started quickly. It generates the minimum code for a simple Android application. However, by default, new Android projects support Java and only Java.

2. With the help of ADT, an Android Java project is easily turned into a hybrid project with native C/C++ support. It generates the minimum files necessary for an NDK-Build to compile a native library:
   - Android.mk is a Makefile describing which source files to compile and how to generate the final native library.
   - com_packtpub_store_Store.cpp is an almost empty file containing a single include. We are going to explain this in the next part of this chapter.

3. Once the project is set up, dynamically loading a native library is done in a single call to System.loadLibrary(). This is easily done in a static block, which ensures that the library is loaded once and for all, before a class is initialized. Beware that this works only if the container class is loaded from a single Java ClassLoader (which is usually the case).

Working with an IDE like Eclipse really offers a huge productivity boost and makes programming much more comfortable! But if you are a command-line aficionado or would like to train your command-line skills, the first part, Building NDK sample applications, can easily be applied here.

Introducing Dalvik and ART

It is not possible to talk about Android without mentioning a few words about Dalvik and ART.

Dalvik is a Virtual Machine on which the Dex bytecode is interpreted (not native code!). It is at the core of any application running on Android. Dalvik has been conceived to fit the constrained requirements of mobile devices. It is specifically optimized to use less memory and CPU. It sits on top of the Android kernel, which provides the first layer of abstraction over the hardware (process management, memory management, and so on).
ART is the new Android runtime environment, which has replaced Dalvik since the Android 5 Lollipop. It has improved performances a lot compared to Dalvik. Indeed, where Dalvik interprets bytecode Just-In-Time upon application startup, ART, on the other hand, precompiles bytecode Ahead-Of-Time into native code during application installation. ART is backward compatible with applications packaged for former Dalvik VMs.

Android has been designed with speed in mind. Because most users do not want to wait for their application to be loaded while others are still running, the system is able to instantiate multiple Dalvik or ART VMs quickly, thanks to the Zygote process. Zygote, (whose name comes from the very first biologic cell of an organism from which daughter cells get reproduced), starts when the system boots up. It preloads (or "warms up") all core libraries shared among applications as well as the Virtual Machine instance. To launch a new application, Zygote is simply forked and the initial Dalvik instance gets copied as a consequence. Memory consumption is lowered by sharing as many libraries as possible between processes.

Dalvik and ART are themselves made of native C/C++ code compiled for the target Android platform (ARM, X86, and so on). This means that interfacing these VMs with native C/C++ libraries is easily possible provided that it is compiled with the same Application Binary Interface (ABI) (which basically describes the application or library binary format). This is the role devoted to the Android NDK. For more information, have a look at the Android Open Source Project (AOSP), that is, the Android source code at https://source.android.com/.

Interfacing Java with C/C++

Native C/C++ code has the ability to unleash the power of your application. To do so, Java code needs to invoke and run its native counterpart. In this part, we are going to interface Java and native C/C++ code together.

The resulting project is provided with this book under the name Store_Part2.
**Time for action – calling C code from Java**

Let’s create our first native method and call it from the Java side:

1. Open `src/com/packtpub/store/Store.java` and declare one native method to query the Store. This method returns `int` with the number of entries in it. There is no need to define a method body:

   ```java
   package com.packtpub.store;

   public class Store {
       static {
           System.loadLibrary("com_packtpub_store_Store");
       }

       public native int getCount();
   }
   ```

2. Open `src/com/packtpub/store/StoreActivity.java` and initialize the store. Use its `getCount()` method value to initialize the application title:

   ```java
   public class StoreActivity extends Activity {
       ...;

       public static class PlaceholderFragment extends Fragment {
           private Store mStore = new Store();
           ...

           public PlaceholderFragment() {
           }

           @Override
           public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
               View rootView = inflater.inflate(R.layout.fragment_store, container, false);
               updateTitle();
               return rootView;
           }

           private void updateTitle() {
               int numEntries = mStore.getCount();
               String title = String.format("Store (%1$s)", numEntries);
               setTitle(title);
           }
       }
   }
   ```
3. Generate a JNI header file from the Store class. Go to the Eclipse main menu and go to Run | External Tools | External Tools Configurations.... Create a new Program configuration with the following parameters described in the following screenshot:

![External Tools Configurations](image)

**Location** refers to the `javah` absolute path, which is OS specific. On Windows, you can enter `${env_var:JAVA_HOME}\bin\javah.exe`. On Mac OS X and Linux, it is usually `/usr/bin/javah`.

4. In the **Refresh** tab, check **Refresh resources upon completion** and select **Specific resources**. Using the **Specify Resources...** button, select the jni folder. Finally, click on **Run** to execute `javah`. A new file `jni/com_packtpub_store_Store.h` will then be generated. This contains a prototype for the native method `getCount()` expected on the Java side:

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

[62]
5. We can now implement jni/com_packtpub_store_Store.cpp so that it returns 0 when invoked. The method signature originates from the generated header file (you can replace any previous code) except that the parameter names have been explicitly specified:

```cpp
#include "com_packtpub_store_Store.h"

JNIEXPORT jint JNICALL Java_com_packtpub_store_Store_getCount
(JNIEnv * pEnv, jobject pObject)
{
    return 0;
}
```

6. Compile and run the application.

**What just happened?**

Java now talks C/C++! In the previous part, we created a hybrid Android project. In this part, we interfaced Java with native code. This cooperation is established through Java Native Interfaces (JNI). JNI is the bridge, which binds Java to C/C++. This occurs in three main steps.

Defining native method prototypes on the Java side, marked with the native keyword. Such methods have no body, like an abstract method, because they are implemented on the native side. Native methods can have parameters, a return value, visibility (private, protected, package protected, or public), and can be static: such as the usual Java methods.
Native methods can be called from anywhere in Java code, provided that containing a native library has been loaded before they are called. Failure to do so results in an exception of type java.lang.UnsatisfiedLinkError, which is raised when the native method is invoked for the first time.

Using javah to generate a header file with corresponding native C/C++ prototypes. Although it is not compulsory, the javah tool provided by the JDK is extremely useful to generate native prototypes. Indeed, the JNI convention is tedious and error-prone (more about this in Chapter 3, Interfacing Java and C/C++ with JNI). The JNI code is generated from the .class file, which means your Java code must be compiled first.

Writing native C/C++ code implementation to perform expected operations. Here, we simply return 0 when the Store library is queried. Our native library is compiled in the libs/armeabi directory (the one for ARM processors) and is named libcom_packtpub_store_Store.so. Temporary files generated during compilation are located in the obj/local directory.

Despite its apparent simplicity, interfacing Java with C/C++ is much more involved than what it seems superficially. How to write JNI code on the native side is explored in more detail in Chapter 3, Interfacing Java and C/C++ with JNI.

### Debugging native Android applications

Before diving deeper into JNI, there is one last important tool that any Android developer needs to know how to use: the Debugger. The official NDK one is the GNU Debugger also known as GDB.

The resulting project is provided with this book under the name Store_Part3.

#### Time for action – debugging a native Android application

1. Create file jni/Application.mk with the following content:

```
APP_PLATFORM := android-14
APP_ABI := armeabi armeabi-v7a x86
```

These are not the only ABIs provided by the NDK; more processor architectures such as MIPS or variants such as 64 bits or hard floats exist. The ones used here are the main ones you should be concerned with. They can easily be tested on an emulator.
2. Open **Project Properties**, go to **C/C++ Build**, uncheck **Use default build command** and enter `ndk-build NDK_DEBUG=1`:

3. In `jni/com_packtpub_store_Store.cpp`, place a breakpoint inside the `Java_com_packtpub_store_Store_getCount()` method by double-clicking on the Eclipse editor gutter.

4. Select the **Store** project in the **Package Explorer** or **Project Explorer** view and go to **Debug As | Android Native Application**. The application starts, but you will probably find that nothing happens. Indeed, the breakpoint is likely to be reached before the GDB Debugger could attach to the application process.
5. Leave the application and reopen it from your device application menu. This time, Eclipse stops at the native breakpoint. Look at your device screen. The UI should be frozen because the main application thread is paused in native code.

6. Inspect variables in the Variables view and check the call stack in the Debug view. In the Expressions view, enter \(*pEnv.functions\) and open result expression to see the various functions provided by the JNIEnv object.

7. Step Over current instruction from the Eclipse toolbar or with the shortcut, F6 (you can also use Step Into with the shortcut, F7). The following instructions will be highlighted:
   - **Resume** the execution from the Eclipse toolbar or with the shortcut, F8. The application screen is displayed on your device again.
   - **Terminate** the application from the Eclipse toolbar or with the shortcut, Ctrl+F2. The application is killed and the Debug view is emptied.

**What just happened?**

This useful productivity tool that is a debugger is now an asset in our toolbox. We can easily stop or resume program execution at any point, step into, over or out of native instructions, and inspect any variable. This ability is made available to developers thanks to NDK-GDB, which is a wrapper script around the command-line debugger GDB (which can be cumbersome to use by hand). Hopefully, GDB is supported by Eclipse CDT and by extension Eclipse ADT.
On Android, and more generally on embedded devices, GDB is configured in client/server mode, while a program runs on a device as a server (gdbserver, which is generated by NDK-Build in the `libs` directory). A remote client, that is, a developer's workstation with Eclipse, connects and sends remote debugging commands to it.

**Defining NDK application-wide settings**

To help NDK-Build and NDK-GDB do their work, we created a new `Application.mk` file. This file should be considered as a global Makefile defining application-wide compilation settings, such as the following:

- **APP_PLATFORM**: Android API that the application targets. This information should be a duplication of `minSdkVersion` in the `AndroidManifest.xml` file.
- **APP_ABI**: CPU architectures that the application targets. An Application Binary Interface specifies the binary code format (instruction set, calling conventions, and so on) that makes executable and library binaries. ABIs are thus strongly related to processors. ABI can be tweaked with additional settings such as `LOCAL_ARM_CODE`.

The main ABIs that are currently supported by the Android NDK are as shown in the following table:

<table>
<thead>
<tr>
<th>ABI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>armeabi</td>
<td>This is the default option, which should be compatible with all ARM devices. Thumb is a special instruction set that encodes instructions on 16 bits instead of 32 to improve code size (useful for devices with constrained memory). The instruction set is severely restricted compared to ArmEABI.</td>
</tr>
<tr>
<td>armeabi</td>
<td>(Or Arm v5) Should run on all ARM devices. Instructions are encoded on 32 bits but may be more concise than Thumb code. Arm v5 does not support advanced extensions such as floating point acceleration and is thus slower than Arm v7.</td>
</tr>
<tr>
<td>armeabi-v7a</td>
<td>Supports extensions such as Thumb-2 (similar to Thumb but with additional 32-bit instructions) and VFP, plus some optional extensions such as NEON. Code compiled for Arm V7 will not run on Arm V5 processors.</td>
</tr>
<tr>
<td>armeabi-v7a-hard</td>
<td>This ABI is an extension of the armeabi-v7a that supports hardware floats instead of soft floats.</td>
</tr>
<tr>
<td>arm64-v8a</td>
<td>This is dedicated to the new 64-bit processor architecture. 64-bit ARM processors are backward compatible with older ABIs.</td>
</tr>
</tbody>
</table>
Starting a Native Android Project

<table>
<thead>
<tr>
<th>ABI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>x86 and x86_64</strong></td>
<td>For &quot;PC-like&quot; processor architectures (that is, Intel/AMD). These are the ABIs used on the emulator in order to get hardware acceleration on a PC. Although most Android devices are ARM, some of them are now X86-based. The x86 ABI is for 32-bit processors and x86_64 is for 64-bit processors.</td>
</tr>
<tr>
<td><strong>mips and mips 64</strong></td>
<td>For processors made by MIPS Technologies, now property of Imagination Technologies well-known for the PowerVR graphics processors. Almost no device uses these at the time of writing this book. The mips ABI is for 32-bit processors and mips64 is for 64-bit processors.</td>
</tr>
<tr>
<td><strong>all, all32 and all64</strong></td>
<td>This is a shortcut to build an ndk library for all 32-bit or 64-bit ABIs.</td>
</tr>
</tbody>
</table>

Each library and intermediate object file is recompiled for each ABI. They are stored in their own respective directory which can be found in the obj and libs folders.

A few more flags can be used inside Application.mk. We will discover more about this in detail in Chapter 9, Porting Existing Libraries to Android.

The Application.mk flags are not the only ones necessary to ensure the NDK debugger work; NDK_DEBUG=1 must also be passed manually to NDK-Build so that it compiles Debug binaries and generates GDB setup files (gdb.setup and gdbserver) correctly. Note that this should probably be considered more as a defect in Android development tools rather than a real configuration step, since it should normally handle the debugging flag automatically.

**NDK-GDB day-to-day**

Debugger support in the NDK and Eclipse is quite recent and has improved a lot among NDK releases (for example, debugging purely native threads was not working before). However, although it is now quite usable, debugging on Android can sometimes be buggy, unstable, and rather slow (because it needs to communicate with the remote Android device).

NDK-GDB might sometimes appear crazy and stop at a breakpoint with a completely unusual stack trace. This could be related to GDB not being able to correctly determine current ABI while debugging. To fix this issue, put only your corresponding device ABI in the APP_ABI clause and remove or comment any other.
Chapter 2

NDK Debugger can also be tricky to use, such as when debugging native startup code. Indeed, GDB does not start fast enough to activate breakpoints. A simple way to overcome this problem is to make native code sleep for a few seconds when an application starts. To leave GDB enough time to attach an application process, we can do, for example, the following:

```c
#include <unistd.h>
...
sleep(3); // in seconds.
```

Another solution is to launch a Debug session and then simply leave and re-launch the application from your device, as we have seen in the previous tutorial. This is possible because the Android application life cycle is such that an application survives when it is in the background, until the memory is needed. This trick only works if your application does not crash during startup though.

Analyzing native crash dumps

Every developer has one day experienced an unexpected crash in its application. Do not be ashamed, it has happened to all of us. And as a newcomer in Android native development, this situation will happen again, many times. Debuggers are a tremendous tool to look for problems in your code. Sadly, however they work in "real-time", when a program runs. They become sterile with fatal bugs that cannot be reproduced easily. Hopefully, there is a tool for that: NDK-Stack. NDK-Stack helps you read a crash dump to analyze an application’s stack-trace at the moment it crashed.

The resulting project is provided with this book under the name Store_Crash.

Time for action – analyzing a native crash dump

Let’s make our application crash to see how to read a crash dump:

1. Simulate a fatal bug in jni/com_packtpub_store_Store.cpp:
   ```c
   #include "com_packtpub_store_Store.h"
   
   JNIEXPORT jint JNICALL Java_com_packtpub_store_Store_getCount
     (JNIEnv* pEnv, jobject pObject) {
     pEnv = 0;
     return pEnv->CallIntMethod(0, 0);
   }
   ```
2. Open the LogCat view in Eclipse, select the All Messages (no filter) option, and then run the application. A crash dump appears in the logs. This is not pretty! If you look carefully through it, you should find a backtrace section with a snapshot of the call-stack at the moment the application crashed. However, it does not give the line of code involved:

3. From a command-line prompt, go to the project directory. Find the line of code implied in the crash by running NDK-Stack with logcat as the input. NDK-Stack needs the obj files corresponding to the device ABI on which the application crashed, for example:

   cd <project directory>

   adb logcat | ndk-stack -sym obj/local/armeabi-v7a
NDK-Stack utility provided with the Android NDK can help you locate the source of an application crash. This tool is an inestimable help and should be considered as your first-aid kit when a bad crash happens. However, if it can point you toward the where, it is another kettle of fish to find out the why.

Stack-trace is only a small part of a crash dump. Deciphering the rest of a dump is rarely necessary but understanding its meaning is good for general culture.

Deciphering crash dumps
Crash dumps are not only dedicated to overly talented developers seeing a red-dressed girl in binary code, but also to those who have a minimum knowledge of assemblers and the way processors work. The goal of this trace is to give as much information as possible on the current state of the program at the time it crashed. It contains:

- 1st line: Build Fingerprint is a kind of identifier indicating the device/Android release currently running. This information is interesting when analyzing dumps from various origins.
- 3rd line: The PID or process identifier uniquely identifies an application on the Unix system, and the TID, which is the thread identifier. The thread identifier can be the same as the process identifier when a crash occurs on the main thread.
- 4th line: The crash origin represented as a Signal is a classic segmentation fault (SIGSEGV).
Starting a Native Android Project

- **Processor Register** values. A register holds values or pointers on which the processor can work immediately.
- **Backtrace** (that is the stack-trace) with the method calls that lead to the crash.
- **Raw stack** is similar to the backtrace but with stack parameters and variables.
- **Some Memory Words** around the main register (provided for ARM processors only). The first column indicates memory-line locations, while others columns indicate memory values represented in hexadecimal.

Processor registers are different between processor architectures and versions. ARM processors provide:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rX</td>
<td>Integer Registers where a program puts values it works on.</td>
</tr>
<tr>
<td>dX</td>
<td>Floating Point Registers where a program puts values it works on.</td>
</tr>
<tr>
<td>fp (or r11)</td>
<td>Frame Pointer holds the current stack frame location during a routine call (in conjunction with the Stack Pointer).</td>
</tr>
<tr>
<td>ip (or r12)</td>
<td>Intra Procedure Call Scratch Register may be used with some sub-routine calls; for example, when the linker needs a veneer (a small piece of code) to aim at a different memory area when branching. Indeed, a branch instruction to jump somewhere else in memory requires an offset argument relative to the current location, allowing a branching range of a few MB only, not the full memory.</td>
</tr>
<tr>
<td>sp (or r13)</td>
<td>Stack Pointer holds the location of the top of the stack.</td>
</tr>
<tr>
<td>lr (or r14)</td>
<td>Link Register saves a program counter value temporarily so that it can restore it later. A typical example of its use is as a function call, which jumps somewhere in the code and then goes back to its previous location. Of course, several chained sub-routine calls require the Link Register to be stacked.</td>
</tr>
<tr>
<td>pc (or r15)</td>
<td>Program Counter holds the address of the next instruction to be executed. The program counter is just incremented when executing a sequential code to fetch the next instruction but it is altered by branching instructions (if/else, a C/C++ function calls, and so on).</td>
</tr>
<tr>
<td>cpsr</td>
<td>Current Program Status Register contains a few flags about the current processor working mode and some additional bit flags for condition codes (such as N for an operation that resulted in a negative value, Z for a 0 or equality result, and so on), interrupts, and instruction sets (Thumb or ARM).</td>
</tr>
</tbody>
</table>

Remember that the use of registers is mainly a convention. For example, Apple iOS uses r7 as a Frame Pointer instead of r12 on ARMs. So always be very careful when writing or reusing assembly code!
On the other hand, X86 processors provide:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>Accumulator Register is used, for example, for arithmetic or I/O operations.</td>
</tr>
<tr>
<td>ebx</td>
<td>Base Register is a data pointer for memory access.</td>
</tr>
<tr>
<td>ecx</td>
<td>Counter Register is used for iterative operations such as loop counter.</td>
</tr>
<tr>
<td>edx</td>
<td>Data Register is a secondary Accumulator Register used in conjunction with eax.</td>
</tr>
<tr>
<td>esi</td>
<td>Source Index Register is used for memory array copying in conjunction with edi.</td>
</tr>
<tr>
<td>edi</td>
<td>Destination Index Register is used for memory array copying in conjunction with esi.</td>
</tr>
<tr>
<td>eip</td>
<td>Instruction Pointer holds offset of the next instruction.</td>
</tr>
<tr>
<td>ebp</td>
<td>Base Pointer holds the current stack frame location during a routine call (in conjunction with the Stack Pointer).</td>
</tr>
<tr>
<td>esp</td>
<td>Stack Pointer holds the location of the top of the stack.</td>
</tr>
<tr>
<td>xcs</td>
<td>Code Segment helps in addressing the memory segment in which the program runs.</td>
</tr>
<tr>
<td>xds</td>
<td>Data Segment helps addressing a data memory segment.</td>
</tr>
<tr>
<td>xes</td>
<td>Extra Segment is an additional register to address a memory segment.</td>
</tr>
<tr>
<td>xfs</td>
<td>Additional Segment which is a general purpose data segment.</td>
</tr>
<tr>
<td>xss</td>
<td>Stack segment holds the Stack memory segment.</td>
</tr>
</tbody>
</table>

Many X86 registers are a legacy, which means that they lost the initial purpose they were created for. Take their descriptions with some caution.

Deciphering stack-traces is not an easy task and requires time and expertise. Don’t bother too much if you do not understand every part of it yet. This is necessary as a last resort only.

**Setting up a Gradle project to compile native code**

Android Studio is now the new officially supported Android IDE, in place of Eclipse. It comes with Gradle, which is the new official Android build system. Gradle introduces a Groovy-based specific language to define the project configuration easily. Although its support of the NDK is still preliminary, it keeps improving and is becoming more and more useable.

Let’s now see how to create an Android Studio project with Gradle that compiles native code.

The resulting project is provided with this book under the name Store_Gradle_Auto.
Starting a Native Android Project

**Time for action – creating a native Android project**

Gradle-based projects can be created easily through Android Studio:

1. Launch Android Studio. On the welcome screen, select New Project... (or go to File | New Project... if a project is already opened).
2. From the New Project wizard, enter the following configuration and click on Next:

![New Project wizard](image)

3. Then, select the minimum SDK (for example, API 14: Ice Scream Sandwich) and click on Next.
4. Select Blank Activity with Fragment and click on Next.
5. Finally, enter Activity Name and Layout Name names as follows and click on Finish:
6. Android Studio should then open the project:
Starting a Native Android Project

7. Modify StoreActivity.java and create Store.java in the same way as we did in the Interfacing Java with C/C++ section in this chapter (Step 1 and 2).

8. Create the app/src/main/jni directory. Copy the C and Header files we created in the Interfacing Java with C/C++ section in this chapter (Step 4 and 5).

9. Edit app/build.gradle that has been generated by Android Studio. In defaultConfig, insert a ndk section to configure the module (that is, a library) name:

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

   android {
      compileSdkVersion 21
      buildToolsVersion "21.1.2"

      defaultConfig {
         applicationId "com.packtpub.store"
         minSdkVersion 14
         targetSdkVersion 21
         versionCode 1
         versionName "1.0"
         ndk {
            moduleName "com_packtpub_store_Store"
         }
      }

      buildTypes {
         release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
         }
      }
   }

   dependencies {
      compile fileTree(dir: 'libs', include: ['*.jar'])
      compile 'com.android.support:appcompat-v7:21.0.3'
   }
   ```

10. Compile and install the project on your device by clicking on installDebug in the Gradle tasks view of Android Studio.
If Android Studio complains that it cannot find the NDK, make sure the `local.properties` file in the project's root directory contains both `sdk.dir` and `ndk.dir` properties that can point to your Android SDK and NDK location.

**What just happened?**

We created our first Android Studio project that compiles native code through Gradle. NDK properties are configured in a section specific to `ndk` in the `build.gradle` file (for example, the module name).

Multiple settings are available as shown in the following table:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abiFilter</td>
<td>The list of ABIs to compile for; by default, all.</td>
</tr>
<tr>
<td>cFlags</td>
<td>Custom flags to pass to the compiler. More about this in Chapter 9,</td>
</tr>
<tr>
<td></td>
<td>Porting Existing Libraries to Android.</td>
</tr>
<tr>
<td>ldLibs</td>
<td>Custom flags to pass to the linker. More about this in Chapter 9,</td>
</tr>
<tr>
<td></td>
<td>Porting Existing Libraries to Android.</td>
</tr>
<tr>
<td>moduleName</td>
<td>This is the name of the module to be built.</td>
</tr>
<tr>
<td>stl</td>
<td>This is the STL library to use for compilation. More about this in Chapter 9,</td>
</tr>
<tr>
<td></td>
<td>Porting Existing Libraries to Android.</td>
</tr>
</tbody>
</table>

You might have noticed that we have not reused the `Android.mk` and `Application.mk` files. This is because Gradle generates the build files automatically if given an input to `ndk-build` at compilation time. In our example, you can see the generated `Android.mk` for the `Store` module in the `app/build/intermediates/ndk/debug` directory.

NDK automatic Makefile generation makes it easy to compile native NDK code on simple projects. However, if you want more control on your native build, you can create your own Makefiles like the ones created in the Interfacing Java with C/C++ section in this chapter. Let's see how to do this.

The resulting project is provided with this book under the name `Store_Gradle_Manual`. 
Time for action – using your own Makefiles with Gradle

Using your own handmade makefiles with Gradle is a bit tricky but not too complicated:

1. Copy the Android.mk and Application.mk files we created in the Interfacing Java with C/C++ section in this chapter into the app/src/main/jni directory.

2. Edit app/build.gradle.

3. Add an import for the OS "Class" and remove the first ndk section we created in the previous section:

   ```java
   import org.apache.tools.ant.taskdefs.condition.Os
   ```

   apply plugin: 'com.android.application'

   android {
       compileSdkVersion 21
       buildToolsVersion "21.1.2"
   }

   defaultConfig {
       applicationId "com.packtpub.store"
       minSdkVersion 14
       targetSdkVersion 21
       versionCode 1
       versionName "1.0"
   }

   buildTypes {
       release {
           minifyEnabled false
           proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
       }
   }

4. Still in the android section of app/build.gradle, insert a sourceSets.main section with the following:

   - jniLibs.srcDir, which defines where Gradle will find the generated libraries.
   - jni.srcDirs, which is set to an empty array to disable native code compilation through Gradle.

   ```java
   ...  
   sourceSets.main {
       jniLibs.srcDir 'src/main/libs'
       jni.srcDirs = []
   }
   ```
5. Finally, create a new Gradle task `ndkBuild` that will manually trigger the `ndk-build` command, specifying the custom directory `src/main` as the compilation directory.

Declare a dependency between the `ndkBuild` task and the Java compilation task to automatically trigger native code compilation:

```groovy
... 

task ndkBuild(type: Exec) { 
    if (Os.isFamily(Os.FAMILY_WINDOWS)) { 
        commandLine 'ndk-build.cmd', '-C', file('src/main').absolutePath
    } else { 
        commandLine 'ndk-build', '-C', file('src/main').absolutePath
    }
}

tasks.withType(JavaCompile) { 
    compileTask -> compileTask.dependsOn ndkBuild
}
}

dependencies { 
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:21.0.3'
}
```

6. Compile and install the project on your device by clicking on `installDebug` in the Gradle tasks view of Android Studio.

**What just happened?**

The Makefile generation and native source compilation performed by the Android Gradle plugin can easily be disabled. The trick is to simply indicate that no native source directory is available. We can then use the power of Gradle, which allows defining easily custom build tasks and dependencies between them, to execute the `ndk-build` command. This trick allows using our own NDK makefiles, giving us more flexibility in the way we build native code.
Summary

Creating, compiling, building, packaging, and deploying an application project are not the most exciting tasks, but they cannot be avoided. Mastering them will allow you to be productive and focused on the real objective: **producing code**.

In summary, we built our first sample application using command-line tools and deploying it on an Android device. We also created our first native Android project using Eclipse and interfaced Java with C/C++ using Java Native Interfaces. We debugged a native Android application with NDK-GDB and analyzed a native crash dump to find its origin in the source code. Finally, we created a similar project using Android Studio and built it with Gradle.

This first experiment with the Android NDK gives you a good overview of the way native development works. In the next chapter, we are going to focus on the code and dive more deeply into the JNI protocol.
Where to buy this book


Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.