



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Gradle for Android

Automate the build process for your Android projects with Gradle

Kevin Pelgrims

[PACKT] open source*
PUBLISHING community experience distilled

Gradle for Android

Automate the build process for your Android projects with Gradle

Kevin Pelgrims



BIRMINGHAM - MUMBAI

Gradle for Android

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: July 2015

Production reference: 1140715

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78398-682-8

www.packtpub.com

Credits

Author

Kevin Pelgrims

Reviewers

Peter Friese

Jonathan H. Kau

Takuya Miyamoto

Marco A. Rodriguez-Suarez

Felix Schulze

Hugo Visser

Commissioning Editor

Amarabha Banerjee

Acquisition Editor

Nikhil Karkal

Content Development Editor

Prachi Bisht

Technical Editor

Pankaj Kadam

Copy Editors

Puja Lalwani

Laxmi Subramanian

Project Coordinator

Shipra Chawhan

Proofreader

Safis Editing

Indexer

Tejal Soni

Production Coordinator

Manu Joseph

Cover Work

Manu Joseph

About the Author

Kevin Pelgrims started his career as a .NET developer in Belgium. After some years working on Windows and web development for corporate clients, he moved to Copenhagen to become part of its start-up community. There, he began working on mobile platforms, and within a year, he was the lead developer of several Android and Windows Phone apps. Kevin is also a regular speaker at the Android developers group in Copenhagen. He maintains multiple Android apps in his spare time and likes to experiment with server-side technology. When he is not busy writing code, he is most likely playing the guitar with his wife and their cat.

To follow what Kevin is up to, you can take a look at his blog at <http://www.kevinpelgrims.com> or follow him on Twitter at @kevinpelgrims.

I could not have written this book without the support of my brilliant wife, Louise. Not only did she put up with me spending entire evenings and weekends with my computer, she also proofread the entire book, improving the language and the grammar.

I would like to thank Packt Publishing for giving me the opportunity to write this book, especially Nikhil Karkal and Prachi Bisht, for believing in me and guiding me throughout the process. I appreciate the help of my friend Koen Metsu, whose comments helped in making sure that the explanations and examples are easy to understand. Thanks to Anders Bo Pedersen and Emanuele Zattin for coming up with several improvements.

I also want to thank the reviewers: Hugo Visser, Peter Friese, Felix Schulze, Takuya Miyamoto, Jonathan H. Kau, and Marco Rodriguez-Suarez. Their input has significantly improved the content of this book. Last but not least, special thanks to our cat, Jigsaw, who encouraged me to stay focused and keep going, by sitting on my lap so I couldn't get up.

About the Reviewers

Peter Friese works as a developer advocate at Google in the Developer Relations team in London, UK. He is a regular open source contributor, blogger, and public speaker. He is on Twitter at @peterfrieese and on Google+ at +PeterFriese. You can find his blog at <http://www.peterfrieese.de>.

Jonathan H. Kau is an experienced Android developer, having published several standalone applications to Google Play and worked on numerous hackathon projects leveraging Android.

In the past, he has worked on the mobile team at Yelp and as a contractor for Propeller Labs. Jonathan is currently working in the engineering team at Shyp, both on the Android application and the corresponding backend API.

I'd like to thank Packt Publishing for giving me the opportunity to review this educational book that will hopefully simplify the use of Gradle for many developers.

Takuya Miyamoto is a full-stack developer with 6 years of experience in designing, implementing, and maintaining various types of web services and APIs such as e-commerce and SNS. He also has some experience as an Android developer. He has worked as a senior engineer in an enterprise, as a lead engineer in start-ups, and is currently working as an individual developer.

Marco A. Rodriguez-Suarez is the head of mobile at Snapwire, a collaborative platform and marketplace that connects photographers with brands and businesses around the world. Prior to that, he worked in mobile consulting on diverse projects, ranging from topics such as video streaming to game emulation. He has been working with Android since the release of the first public device in 2008 and has been hooked on mobile development ever since. He is passionate about build systems and has extensive experience with Gradle, Maven, and Ant. Marco received his master's degree in electrical engineering from the University of California at Santa Barbara.

Felix Schulze is leading the mobile software development department at AutoScout24 and is responsible for the Android and iOS development. He also gives a lot of talks about continuous integration in app development and contributes to open source tools. His Twitter handle is @x2on; you can also check out his website at www.felixschulze.de.

Hugo Visser has many years of experience as a software engineer, ranging from server-side to desktop and from web to mobile. Since the original announcement, he has closely followed the development of the Android platform, which resulted in his first app in 2009, *Rainy Days*, which has since been downloaded over 1,000,000 times worldwide.

He runs his own company, Little Robots, which focuses on apps and other clever uses of the Android platform. He has been named the Google Developer Expert for Android by Google and is an organizer of The Dutch Android User Group, which is a community in the Netherlands where Android professionals can meet and share knowledge with each other during monthly meetings.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	vii
Chapter 1: Getting Started with Gradle and Android Studio	1
Android Studio	2
Staying up to date	3
Understanding Gradle basics	3
Projects and tasks	4
The build lifecycle	4
The build configuration file	4
Creating a new project	7
Getting started with the Gradle Wrapper	10
Getting the Gradle Wrapper	10
Running basic build tasks	12
Migrating from Eclipse	13
Using the import wizard	14
Migrating manually	15
Keeping the old project structure	15
Converting to the new project structure	17
Migrating libraries	18
Summary	18
Chapter 2: Basic Build Customization	19
Understanding the Gradle files	19
The settings file	20
The top-level build file	20
The module build file	21
Plugin	22
Android	22
Dependencies	24

Getting started with tasks	24
Base tasks	24
Android tasks	25
Inside Android Studio	26
Customizing the build	28
Manipulating manifest entries	29
Inside Android Studio	29
BuildConfig and resources	30
Project-wide settings	31
Project properties	32
Default tasks	33
Summary	33
Chapter 3: Managing Dependencies	35
Repositories	35
Preconfigured repositories	37
Remote repositories	37
Local repositories	38
Local dependencies	39
File dependencies	39
Native libraries	40
Library projects	41
Creating and using library project modules	41
Using .aar files	42
Dependency concepts	42
Configurations	42
Semantic versioning	43
Dynamic versions	44
Inside Android Studio	44
Summary	46
Chapter 4: Creating Build Variants	47
Build types	48
Creating build types	48
Source sets	50
Dependencies	52
Product flavors	52
Creating product flavors	52
Source sets	53
Multiflavor variants	53
Build variants	54
Tasks	55

Source sets	56
Resource and manifest merging	56
Creating build variants	57
Variant filters	58
Signing configurations	59
Summary	61
Chapter 5: Managing Multimodule Builds	63
The anatomy of a multimodule build	64
The build lifecycle revisited	65
Module tasks	66
Adding modules to a project	67
Adding a Java library	68
Adding an Android library	68
Integrating Android Wear	69
Using Google App Engine	70
Analyzing the build file	71
Using the backend in an app	72
Custom tasks	73
Tips and best practices	73
Running module tasks from Android Studio	74
Speeding up multimodule builds	74
Module coupling	75
Summary	76
Chapter 6: Running Tests	77
Unit tests	77
JUnit	78
Robolectric	81
Functional tests	82
Espresso	82
Test coverage	86
Jacoco	87
Summary	88
Chapter 7: Creating Tasks and Plugins	89
Understanding Groovy	89
Introduction	90
Classes and members	91
Methods	92
Closures	93
Collections	93
Groovy in Gradle	94

Getting started with tasks	95
Defining tasks	95
Anatomy of a task	98
Using a task to simplify the release process	100
Hooking into the Android plugin	104
Automatically renaming APKs	104
Dynamically creating new tasks	105
Creating your own plugins	107
Creating a simple plugin	107
Distributing plugins	108
Using a custom plugin	111
Summary	111
Chapter 8: Setting Up Continuous Integration	113
Jenkins	114
Setting up Jenkins	114
Configuring the build	115
TeamCity	117
Setting up TeamCity	118
Configuring the build	118
Travis CI	119
Configuring the build	119
Further automation	122
The SDK manager plugin	122
Running tests	122
Continuous deployment	123
Beta distribution	125
Summary	126
Chapter 9: Advanced Build Customization	127
Reducing the APK file size	127
ProGuard	128
Shrinking resources	129
Automatic shrinking	129
Manual shrinking	130
Speeding up builds	131
Gradle properties	131
Android Studio	133
Profiling	134
Jack and Jill	135
Ignoring Lint	136

Using Ant from Gradle	136
Running Ant tasks from Gradle	136
Importing an entire Ant script	137
Properties	139
Advanced app deployment	139
Split APK	139
Summary	141
Index	143

Preface

The build process for an Android app is an incredibly complex procedure that involves many tools. To begin with, all the resource files are compiled and referenced in a R.java file, before the Java code is compiled and then converted to Dalvik bytecode by the dex tool. These files are then packaged into an APK file, and that APK file is signed with a debug or release key, before the app can finally be installed onto a device.

Going through all these steps manually would be a tedious and time-consuming undertaking. Luckily, the Android Tools team has continued to provide developers with tools that take care of the entire process, and in 2013, they introduced Gradle as the new preferred build system for Android apps. Gradle is designed in a way that makes it easy to extend builds and plug into the existing build processes. It provides a Groovy-like DSL to declare builds and create tasks, and makes dependency management easy. Additionally, it is completely free and open source.

By now, most Android developers have switched to Gradle, but many do not know how to make the best of it, and are unaware of what can be achieved with just a few lines of code. This book aims to help those developers, and turn them into Gradle power users. Starting with the basics of Gradle in an Android context, this book goes on to cover dependencies, build variants, testing, creating tasks, and more.

What this book covers

Chapter 1, Getting Started with Gradle and Android Studio, explains why Gradle is useful, how to get started with Android Studio, and what the Gradle Wrapper is.

Chapter 2, Basic Build Customization, goes into detail about the Gradle build files and tasks, and shows how to do simple customizations to the build process.

Chapter 3, Managing Dependencies, shows how to use dependencies, both local and remote ones, and explains dependency-related concepts.

Chapter 4, Creating Build Variants, introduces build types and product flavors, explains what the difference between them is, and shows how to use signing configurations.

Chapter 5, Managing Multimodule Builds, explains how to manage app, library, and test modules, and mentions how to integrate them into the build.

Chapter 6, Running Tests, introduces several testing frameworks for unit tests and functional tests, how to automate testing and how to get test coverage reports.

Chapter 7, Creating Tasks and Plugins, explains the basics of Groovy, and shows how to create custom tasks and how to hook them into the Android build process. This chapter also explains how to create a reusable plugin.

Chapter 8, Setting Up Continuous Integration, provides guidance on automating builds using the most commonly used CI systems.

Chapter 9, Advanced Build Customization, shows some tips and tricks to shrink APKs, speed up the build process, and split up APKs based on density or platform.

What you need for this book

To follow all the examples, you will need to have access to a computer with Microsoft Windows, Mac OS X, or Linux. You will also need to have the Java Development Kit installed, and it is recommended that you install Android Studio, as it is mentioned in most chapters.

Who this book is for

This book is for Android developers who want to get a better understanding of the build system and become masters of the build process. Throughout the book, we will go from the basics of Gradle, to creating custom tasks and plugins, and automating multiple parts of the build process. You are assumed to be familiar with developing for the Android platform.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Every `build.gradle` file represents a project."


A block of code is set as follows:


```
buildscript {  
    repositories {  
        jcenter()  
    }  
    dependencies {  
        classpath 'com.android.tools.build:gradle:1.2.3'  
    }  
}
```

Any command-line input or output is written as follows:

```
$ gradlew tasks
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "You can start a new project in Android Studio by clicking on **Start a new Android Studio project** on the start screen."

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

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.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

Getting Started with Gradle and Android Studio

When Google introduced Gradle and Android Studio, they had some goals in mind. They wanted to make it easier to reuse code, create build variants, and configure and customize the build process. On top of that, they wanted good IDE integration, but without making the build system dependent on the IDE. Running Gradle from the command line or on a continuous integration server will always yield the same results as running a build from Android Studio.

We will refer to Android Studio occasionally throughout the book, because it often provides a simpler way of setting up projects, dealing with changes, and so on. If you do not have Android Studio installed yet, you can download it from the Android developer website (<http://developer.android.com/sdk/index.html>).

In this chapter, we will cover the following topics:

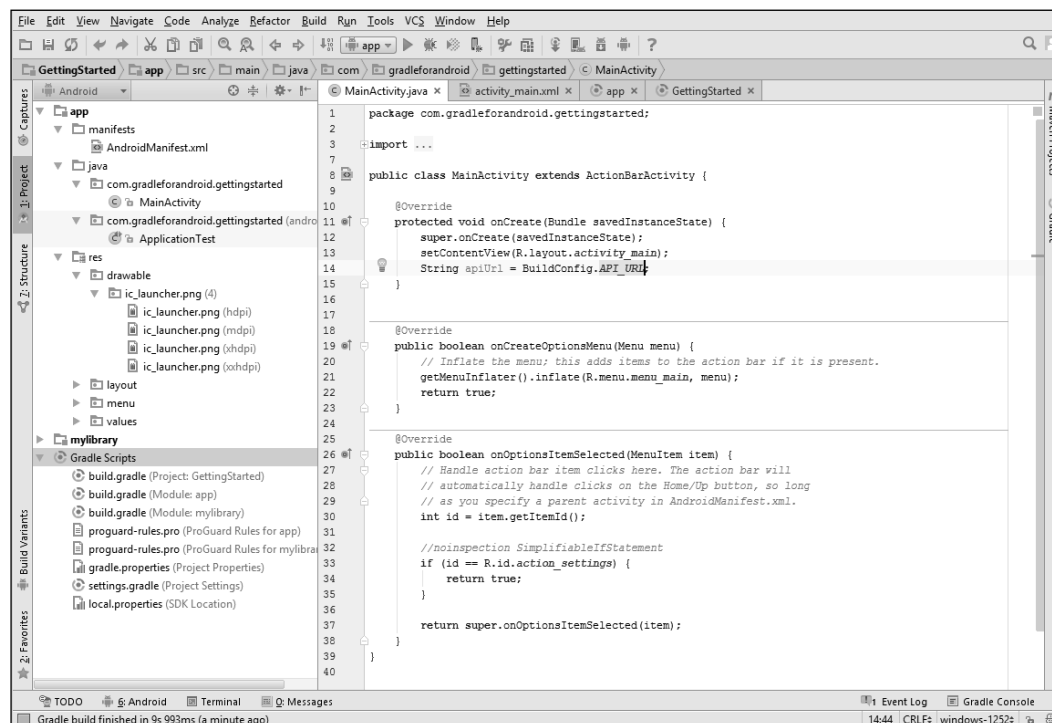
- Getting to know Android Studio
- Understanding Gradle basics
- Creating a new project
- Getting started with the Gradle wrapper
- Migrating from Eclipse

Android Studio

Android Studio was announced and released (as an early access preview) by Google in May 2013, alongside support for Gradle. Android Studio is based on JetBrains' IntelliJ IDEA, but is designed specifically with Android development in mind. It is available for free for Linux, Mac OS X, and Microsoft Windows.

Compared to Eclipse, Android Studio has an improved user interface designer, a better memory monitor, a nice editor for string translation, warnings for possible Android-specific issues and a lot more features aimed at Android developers. It also features a special project structure view for Android projects, besides the regular Project view and Packages view that exist in IntelliJ IDEA. This special view groups Gradle scripts, drawables, and other resources in a convenient way. As soon as the stable version 1.0 of Android Studio was released, Google retired the **Android Developer Tools (ADT)** for Eclipse and recommended all developers to switch to Android Studio. This means that Google will not provide new features for Eclipse anymore, and all IDE-related tool development is now focused on Android Studio. If you are still using Eclipse, it is time to change if you do not want to be left behind.

This screenshot shows what Android Studio looks like for a simple Android app project:



Staying up to date

There are four different update channels for Android Studio:

- Canary brings bleeding-edge updates, but might contain some bugs
- The Dev channel gets an update more or less every month
- Beta is used feature complete updates that might still contain bugs
- The Stable channel, which is the default, features thoroughly tested releases that should be bug-free

By default, Android Studio checks every time it starts if there any updates available and notifies you.

When you launch Android Studio for the first time, it starts a wizard to set up your environment and to make sure you have the latest Android SDK and the necessary Google repositories. It also gives you the option to create an **Android Virtual Device (AVD)**, so you can run apps on the emulator.

Understanding Gradle basics

In order for an Android project to be built using Gradle, you need to set up a build script. This will always be called `build.gradle`, by convention. You will notice, as we go through the basics, that Gradle favors convention over configuration and generally provides default values for settings and properties. This makes it a lot easier to get started with a lot less configuration than that found in systems such as Ant or Maven, which have been the de facto build systems for Android projects for a long time. You do not need to absolutely comply with these conventions though, as it is usually possible to override them if needed.

Gradle build scripts are not written in the traditional XML, but in a **domain-specific language (DSL)** based on Groovy, a dynamic language for the **Java Virtual Machine (JVM)**. The team behind Gradle believes that using a declarative, DSL-style approach based on a dynamic language has significant advantages over using the more procedural, free-floating style of Ant, or any XML-based approach used by many other build systems.

That does not mean you need to know Groovy to get started with your build scripts. It is easy to read, and if you already know Java, the learning curve is not that steep. If you want to start creating your own tasks and plugins (which we will talk about in later chapters), it is useful to have a deeper understanding of Groovy. However, because it is based on the JVM, it is possible to write code for your custom plugins in Java or any other JVM-based language.

Projects and tasks

The two most important concepts in Gradle are projects and tasks. Every build is made up of at least one project, and every project contains one or more tasks. Every `build.gradle` file represents a project. Tasks are then simply defined inside the build script. When initializing the build process, Gradle assembles `Project` and `Task` objects based on the build file. A `Task` object consists of a list of `Action` objects, in the order they need to be executed. An `Action` object is a block of code that is executed, similar to a method in Java.

The build lifecycle

Executing a Gradle build is, in its simplest form, just executing actions on tasks, which are dependent on other tasks. To simplify the build process, the build tools create a dynamic model of the workflow as a **Directed Acyclic Graph (DAG)**. This means all the tasks are processed one after the other and loops are not possible. Once a task has been executed, it will not be called again. Tasks without dependencies will always be run before the others. The dependency graph is generated during the configuration phase of a build. A Gradle build has three phases:

- **Initialization:** This is where the `Project` instance is created. If there are multiple modules, each with their own `build.gradle` file, multiple projects will be created.
- **Configuration:** In this phase, the build scripts are executed, creating and configuring all the tasks for every project object.
- **Execution:** This is the phase where Gradle determines which tasks should be executed. Which tasks should be executed depends on the arguments passed for starting the build and what the current directory is.

The build configuration file

In order to have Gradle build a project, there always needs to be a `build.gradle` file. A build file for Android has a few required elements:

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.2.3'
    }
}
```

This is where the actual build is configured. In the repositories block, the JCenter repository is configured as a source of dependencies for the build script. JCenter is a preconfigured Maven repository and requires no extra setup; Gradle has you covered. There are several repositories available straight from Gradle and it is easy to add your own, either local or remote.

The build script block also defines a dependency on Android build tools as a classpath Maven artifact. This is where the Android plugin comes from. The Android plugin provides everything needed to build and test applications. Every Android project needs to apply the Android plugin using this line:

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

Plugins are used to extend the capabilities of a Gradle build script. Applying a plugin to a project makes it possible for the build script to define properties and use tasks that are defined in the plugin.



If you are building a library, you need to apply 'com.android.library' instead. You cannot use both in the same module because that would result in a build error. A module can be either an Android application or an Android library, not both.

When using the Android plugin, Android-specific conventions can be configured and tasks only applicable to Android will be generated. The Android block in the following snippet is defined by the plugin and can be configured per project:

```
android {  
    compileSdkVersion 22  
    buildToolsVersion "22.0.1"  
}
```

This is where the Android-specific part of the build is configured. The Android plugin provides a DSL tailored to Android's needs. The only required properties are the compilation target and the build tools. The compilation target, specified by `compileSdkVersion`, is the SDK version that should be used to compile the app. It is good practice to use the latest Android API version as the compilation target.

There are plenty of customizable properties in the `build.gradle` file. We will discuss the most important properties in *Chapter 2, Basic Build Customization*, and more possibilities throughout the rest of the book.

The project structure

Compared to the old Eclipse projects, the folder structure for Android projects has changed considerably. As mentioned earlier, Gradle favors convention over configuration and this also applies to the folder structure.

This is the folder structure that Gradle expects for a simple app:

```
MyApp
├── build.gradle
├── settings.gradle
└── app
    ├── build.gradle
    ├── build
    ├── libs
    └── src
        └── main
            ├── java
            │   └── com.package.myapplication
            └── res
                ├── drawable
                ├── layout
                └── etc.
```

Gradle projects usually have an extra level at the root. This makes it easier to add extra modules at a later point. All source code for the app goes into the `app` folder. The folder is also the name of the module by default and does not need to be named `app`. If you use Android Studio to create a project with both a mobile app and an Android Wear smartwatch app, for example, the modules are called `application` and `wearable` by default.

Gradle makes use of a concept called source set. The official Gradle documentation explains that a source set is *a group of source files, which are compiled and executed together*. For an Android project, `main` is the source set that contains all the source code and resources for the default version of the app. When you start writing tests for your Android app, you will put the source code for the tests inside a separate source set called `androidTest`, which only contains tests.

Here is a short overview of the most important folders of an Android app:

Directory	Content
<code>/src/main/java</code>	The source code for the app
<code>/src/main/res</code>	These are app-related resources (drawables, layouts, strings, and so on)
<code>/libs</code>	These are external libraries (<code>.jar</code> or <code>.aar</code>)
<code>/build</code>	The output of the build process

Creating a new project

You can start a new project in Android Studio by clicking on **Start a new Android Studio project** on the start screen or by navigating to **File | New Project...** in the IDE itself.

Creating a new project in Android Studio starts with a wizard that helps set everything up. The first screen is for setting up the application name and the company domain. The application name is the name that will be used as the name of the app when it is installed and is used as the toolbar title by default. The company domain is used in combination with the application name to determine the package name, which is the unique identifier for any Android app. If you prefer a different package name, you can still change it by clicking on **Edit**. You can also change the location of the project on your hard drive.

The screenshot shows the 'New Project' dialog in Android Studio. The title bar says 'New Project' and 'Android Studio'. Below that, it says 'Configure your new project'. There are four input fields: 'Application name' (GettingStarted), 'Company Domain' (gradleforandroid.com), 'Package name' (com.gradleforandroid.gettingstarted) with an 'Edit' link, and 'Project location' (GettingStarted) with a browse button. At the bottom, there are four buttons: 'Previous', 'Next', 'Cancel', and 'Finish'.

No files are generated before going through all the steps in the wizard, because the next few steps will define which files need to be created.

Android does not only run on phones and tablets, but also supports a broad range of form factors, such as TV, watches, and glasses. The next screen helps you set up all the form factors you want to target in your project. Depending on what you choose here, dependencies and build plugins are included for development. This is where you decide if you just want to make a phone and tablet app or whether you also want to include an Android TV module, an Android Wear module, or a Google Glass module. You can still add these later, but the wizard makes it easy by adding all the necessary files and configurations. This is also where you choose what version of Android you want to support. If you select an API version below 21, the Android Support Library (including the appcompat library) is automatically added as a dependency.

Target Android Devices

Select the form factors your app will run on

Different platforms require separate SDKs

☒ **Phone and Tablet**

Minimum SDK: API 14: Android 4.0 (IceCreamSandwich)

Lower API levels target more devices, but have fewer features available. By targeting API 14 and later, your app will run on approximately 90.4% of the devices that are active on the Google Play Store. Help me choose..

☐ **TV**

Minimum SDK: API 21: Android 5.0 (Lollipop)

☐ **Wear**

Minimum SDK: API 20: Android 4.4 (KitKat Wear)

☐ **Glass (Not Installed)**

Minimum SDK:

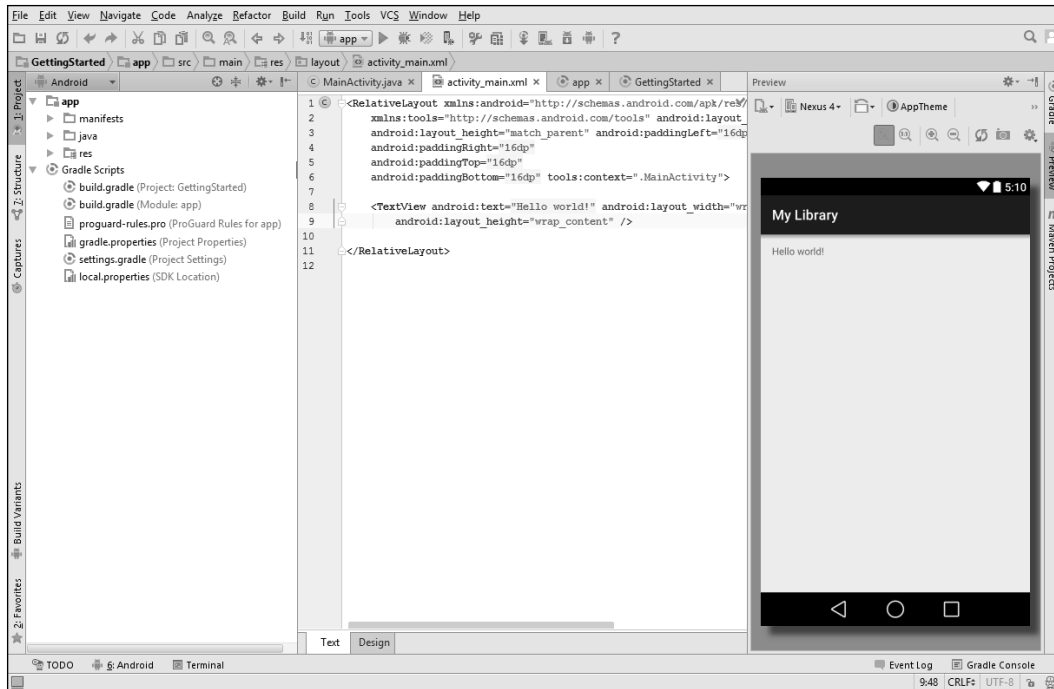
Previous Next Cancel Finish



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.

The following screen suggests adding an activity and provides a lot of options, all of which result in generated code that makes it easier to get started. If you choose to have Android Studio generate an activity for you, the next step is to enter a name for the activity class, the layout file and the menu resource, and also to give the activity a title:



After you go through the entire wizard, Android Studio generates source code for an activity and a fragment, depending on the choices you made during the wizard. Android Studio also generates the basic Gradle files to make the project build. You will find a file called `settings.gradle` and one called `build.gradle` on the top level of the project. Inside the `app` module folder, there is another `build.gradle` file. We will go into more detail about the content and the purpose of these files in *Chapter 2, Basic Build Customization*.

You now have several options to trigger a build from inside Android Studio:

- Inside the **Build** menu, you can click on **Make Project**, or you can use the keyboard shortcut, which is `Ctrl + F9` on a PC and `Cmd + F9` on Mac OS X
- The toolbar has a shortcut for the same **Make Project**
- The Gradle tool window, which lists all the available Gradle tasks

In the Gradle tool window, you can try to execute `assembleDebug` to build, or `installDebug` to install the app on a device or emulator. We will discuss these tasks in the next part of this chapter, which deals with the Gradle wrapper.

Getting started with the Gradle Wrapper

Gradle is a tool that is under constant development, and new versions could potentially break backward compatibility. Using the Gradle Wrapper is a good way to avoid issues and to make sure builds are reproducible.


The Gradle Wrapper provides a batch file on Microsoft Windows and a shell script on other operating systems. When you run the script, the required version of Gradle is downloaded (if it is not present yet) and used automatically for the build. The idea behind this is that every developer or automated system that needs to build the app can just run the wrapper, which will then take care of the rest. This way, it is not required to manually install the correct version of Gradle on a developer machine or build server. Therefore, it is also recommended to add the wrapper files to your version control system.

Running the Gradle Wrapper is not that different from running Gradle directly. You just execute `gradlew` on Linux and Mac OS X and `gradlew.bat` on Microsoft Windows, instead of the regular `gradle` command.

Getting the Gradle Wrapper

For your convenience, every new Android project includes the Gradle Wrapper, so when you create a new project, you do not have to do anything at all to get the necessary files. It is, of course, possible to install Gradle manually on your computer and use it for your project, but the Gradle Wrapper can do the same things and guarantee that the correct version of Gradle is used. There is no good reason not to use the wrapper when working with Gradle outside of Android Studio.

You can check if the Gradle Wrapper is present in your project by navigating to the project folder and running `./gradlew -v` from the terminal or `gradlew.bat -v` from Command Prompt. Running this command displays the version of Gradle and some extra information about your setup. If you are converting an Eclipse project, the wrapper will not be present by default. In this case, it is possible to generate it using Gradle, but you will need to install Gradle first to get the wrapper.


 The Gradle download page (<http://gradle.org/downloads>) has links to binaries and the source code, and it is possible to use a package manager such as Homebrew if you are on Mac OS X. All the instructions for installation are on the installation page (<http://gradle.org/installation>).

After you have downloaded and installed Gradle and added it to your `PATH`, create a `build.gradle` file containing these three lines:

```
task wrapper(type: Wrapper) {
    gradleVersion = '2.4'
}
```

After that, run `gradle wrapper` to generate the wrapper files.

In recent versions of Gradle, you can also run the wrapper task without modifying the `build.gradle` file, because it is included as a task by default. In that case, you can specify the version with the `--gradle-version` parameter, like this:

```
$ gradle wrapper --gradle-version 2.4
```

If you do not specify a version number, the wrapper is configured to use the Gradle version that the task is executed with.

These are all the files generated by the wrapper task:

```
myapp/
├─ gradlew
├─ gradlew.bat
└─ gradle/wrapper/
    ├─ gradle-wrapper.jar
    └─ gradle-wrapper.properties
```

You can see here that the Gradle Wrapper has three parts:

- A batch file on Microsoft Windows and a shell script on Linux and Mac OS X
- A JAR file that is used by the batch file and shell script
- A properties file

The `gradle-wrapper.properties` file is the one that contains the configuration and determines what version of Gradle is used:

```
#Sat May 30 17:41:49 CEST 2015
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
```



```
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/
gradle-2.4-all.zip
```

You can change the distribution URL if you want to use a customized Gradle distribution internally. This also means that any app or library that you use could have a different URL for Gradle, so be sure to check whether you can trust the properties before you run the wrapper.

Android Studio is kind enough to display a notification when the Gradle version used in a project is not up to date and will suggest automatically updating it for you. Basically, Android Studio changes the configuration in the `gradle-wrapper.properties` file and triggers a build, so that the latest version gets downloaded.



Android Studio uses the information in the properties to determine which version of Gradle to use, and it runs the wrapper from the Gradle Wrapper directory inside your project. However, it does not make use of the shell or bash scripts, so you should not customize those.

Running basic build tasks

In the terminal or command prompt, navigate to the project directory and run the Gradle Wrapper with the `tasks` command:

```
$ gradlew tasks
```

This will print out a list of all the available tasks. If you add the `--all` parameter, you will get a more detailed overview with the dependencies for every task.



On Microsoft Windows, you need to run `gradlew.bat`, and on Linux and Mac OS X, the full command is `./gradlew`. For the sake of brevity, we will just write `gradlew` throughout this book.

To build the project while you are developing, run the `assemble` task with the debug configuration:

```
$ gradlew assembleDebug
```

This task will create an APK with the debug version of the app. By default, the Android plugin for Gradle saves the APK in the directory `MyApp/app/build/outputs/apk`.



Abbreviated task names

To avoid a lot of typing in the terminal, Gradle also provides abbreviated Camel case task names as shortcuts. For example, you can execute `assembleDebug` by running `gradlew assDeb`, or even `gradlew aD`, from the command-line interface.

There is one caveat to this though. It will only work as long as the Camel case abbreviation is unique. As soon as another task has the same abbreviation, this trick does not work anymore for those tasks.

Besides `assemble`, there are three other basic tasks:

- `check` runs all the checks, this usually means running tests on a connected device or emulator
- `build` triggers both `assemble` and `check`
- `clean` cleans the output of the project

We will discuss these tasks in detail in *Chapter 2, Basic Build Customization*.

Migrating from Eclipse

There are two ways to take on migration from an Eclipse project to a Gradle-based project:

- Use the import wizard in Android Studio to handle migration automatically
- Add a Gradle script to the Eclipse project and set everything up manually

Most projects are simple enough for the import wizard to be able to convert everything automatically. If there is something the wizard cannot figure out, it might even give you hints as to what you need to change for it to work.

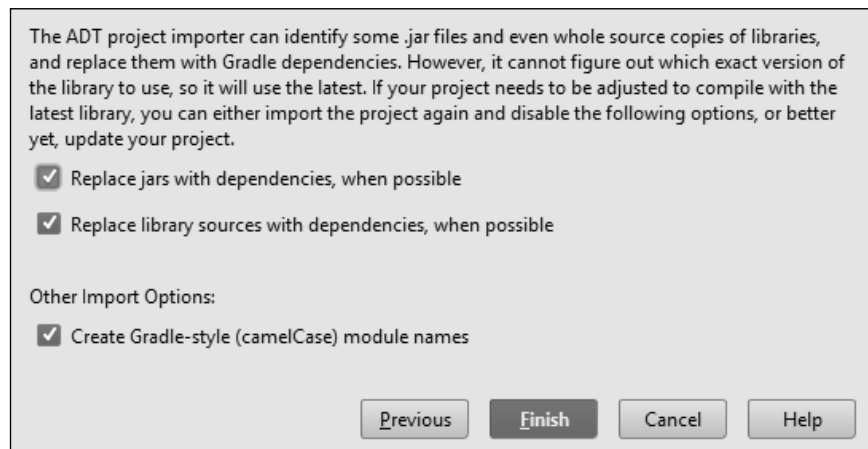
Some projects, though, might be extremely complicated and require a manual conversion. If you have a huge project and you prefer to convert the project in steps, instead of all at once, it is possible to execute Ant tasks, or even entire Ant builds from Gradle. Doing that, you can do the transition at the pace you prefer and convert all the components slowly.

Using the import wizard

To start the import wizard, you need to open Android Studio, click on the **File** menu and then on **Import Project...**, or on the Android Studio start screen, click on **Import Non-Android Studio project**.

If you convert a project with JAR files or library sources, the import wizard will suggest replacing those with Gradle dependencies. These dependencies can come from local Google repositories (such as the Android Support Library) or even from a known online repository central. If no matching Google or online dependencies are found, the JAR file is used, as it was before. The import wizard creates at least one module for your app. If you have libraries with source code in your project, those are converted to modules as well.

This is what the import wizard looks like:



Studio creates a new folder to make sure you do not lose anything when you convert, and you can easily compare the outcome of the import wizard with the original. When the conversion is done, Android Studio opens the project and shows an import summary.

The summary lists any files that the import wizard decided to ignore and did not copy to the new project. If you want to include those anyway, you have to manually copy them to the new project. Right below the ignored files, the summary shows any JAR files that the import wizard was able to replace with Gradle dependencies. Android Studio tries to find those dependencies on JCenter. If you are using the Support Library, it is now included in the Google repository that is downloaded to your machine using the SDK manager, instead of a JAR file. Finally, the summary lists all the files that the import wizard has moved, showing their origin and destination.

The import wizard also adds three Gradle files: `settings.gradle` and `build.gradle` on the root, and another `build.gradle` in the module.

If you have any libraries that include source code, the import wizard turns those into Gradle projects as well and links everything together as necessary.

The project should now build without any issues, but keep in mind that you might need an Internet connection to download some of the necessary dependencies.

Projects that are more complicated might require extra work though, so next we will take a look at how to do the conversion manually.



The Eclipse export wizard

There is an export wizard in Eclipse as well, but it is completely outdated because the Android Tools team at Google stopped working on the Android Developer Tools for Eclipse. Therefore, it is recommended to always use the import wizard in Android Studio instead.

Migrating manually

There are multiple ways to go about manually migrating to a Gradle-based Android project. It is not required to change to the new directory structure, and it is even possible to run Ant scripts from your Gradle scripts. This makes the process of migrating very flexible, and it can make the transition easier for larger projects. We will look at running Ant tasks in *Chapter 9, Advanced Build Customization*.

Keeping the old project structure

If you do not want to move files around, it is possible to keep the Eclipse folder structure in your project. To do that, you need to change the source set configuration. We mentioned source sets earlier when talking about the project structure. Gradle and the Android plugin have their defaults, as usual, but it is possible to override those.

The first thing you need to do is to create a `build.gradle` file in the project directory. This file should apply the Android plugin and define the required properties for Gradle and the Android plugin. In its simplest form, it looks like this:

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
```

```
        classpath 'com.android.tools.build:gradle:1.2.3'
    }
}

apply plugin: 'com.android.application'

android {
    compileSdkVersion 22
    buildToolsVersion "22.0.1"
}
```

Then you can start by changing the source set. Usually, overriding the main source set to comply with the Eclipse structure looks like this:

```
android {
    sourceSets {
        main {
            manifest.srcFile 'AndroidManifest.xml'
            java.srcDirs = ['src']
            resources.srcDirs = ['src']
            aidl.srcDirs = ['src']
            renderscript.srcDirs = ['src']
            res.srcDirs = ['res']
            assets.srcDirs = ['assets']
        }

        androidTest.setRoot('tests')
    }
}
```

In the Eclipse folder structure, all source files will reside in the same folder, so you need to tell Gradle that all these components can be found in the `src` folder. You only need to include the components that are in your project, but adding them all does no harm.

If you have any dependencies on JAR files, you need to tell Gradle where the dependencies are located. Assuming the JAR files are in a folder called `libs`, the configuration looks like this:

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
}
```

This one-liner includes every file with the extension `.jar` inside the `libs` directory as a dependency.

Converting to the new project structure

If you decide to convert to the new project structure manually, you need to create a few folders and move some files. This table shows an overview of the most important files and folders, and where you need to move them to convert to the new project structure:

Old location	New location
src/	app/src/main/java/
res/	app/src/main/res/
assets/	app/src/main/assets/
AndroidManifest.xml	app/src/main/AndroidManifest.xml

If you have any unit tests, you need to move the source code for those to `app/src/test/java/` to have Gradle recognize them automatically. Functional tests belong in the `app/src/androidTest/java/` folder.

The next step is to create a `settings.gradle` file in the root of the project. This file needs to contain only one line, and its purpose is to tell Gradle to include the app module in the build:

```
include: ':app'
```

When that is ready, you need two `build.gradle` files for a successful Gradle build. The first one belongs in the root of the project (on the same level as `settings.gradle`) and is used to define project-wide settings:

```
buildscript {  
    repositories {  
        jcenter()  
    }  
    dependencies {  
        classpath 'com.android.tools.build:gradle:1.2.3'  
    }  
}
```

This sets up a few properties for all modules in the project. The second `build.gradle` goes in the app folder and contains module-specific settings:

```
apply plugin: 'com.android.application'  
  
android {  
    compileSdkVersion 22  
    buildToolsVersion "22.0.1"  
}
```

These are the absolute basics. If you have a simple Android app that does not depend on third-party code, this will suffice. If you have any dependencies, you need to migrate those to Gradle as well.

Migrating libraries

If you have any libraries in your project that contain Android-specific code, those also need to use Gradle in order for them to play nice with the app module. The same basics apply, but you need to use the Android library plugin instead of the Android application plugin. The details of this process are discussed in *Chapter 5, Managing Multimodule Builds*.

Summary

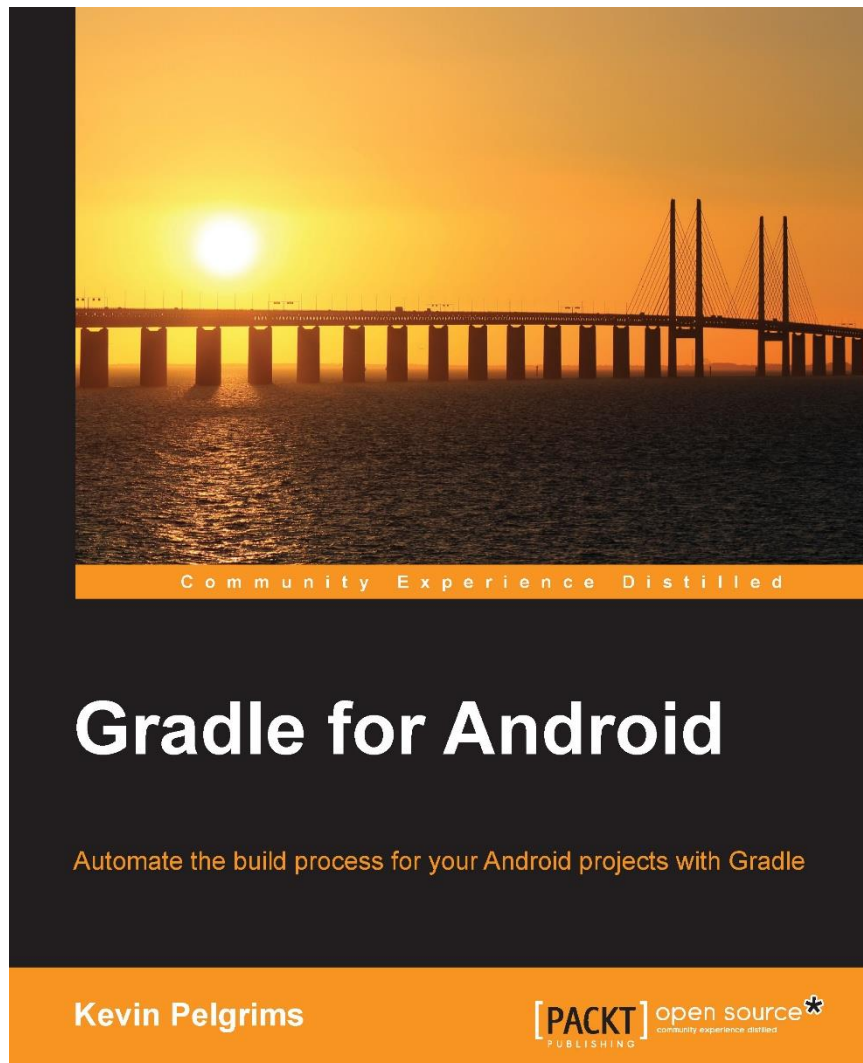
We started the chapter by looking at the advantages of Gradle and why it is more useful than other build systems currently in use. We briefly looked at Android Studio and how it can help us by generating build files.

After the introduction, we took a look at the Gradle Wrapper, which makes maintenance and sharing projects a lot easier. We created a new project in Android Studio, and you now know how to migrate an Eclipse project to Android Studio and Gradle, both automatically and manually. You are also capable of building projects with Gradle in Android Studio, or straight from the command-line interface.

In the next few chapters, we will look at ways to customize the build, so you can further automate the build process and make maintenance even easier. We will start by examining all the standard Gradle files, exploring basic build tasks, and customizing parts of the build in the next chapter.

Purchase the full book

Get 50% discount on the eBook format using coupon code **GRADLE50**



Packt

 **Buy Now**