Up to date for Android Studio 3.3 & Kotlin 1.3

Android Apprentice

SECOND EDITION

Beginning Android Development with Kotlin

By the raywenderlich.com Tutorial Team

Namrata **Bandekar**, Darryl **Bayliss**, Tom **Blankenship** & Fuad **Kamal**

Table of Contents: Overview

Table of Contents: Extended

About This Book Sample

The book *Android Apprentice* is your introduction to building great apps in Android, using the Kotlin language. Whether you still consider yourself a novice programmer, or have extensive experience programming for iOS or other platforms, this is the book for you!

The book shows you how to build four complete apps from scratch — each app is a little more complicated than the previous one. Together, these apps will teach you how to work with the most common controls and APIs used by Android developers around the world.

It also includes some bonus sections on handling the Android fragmentation problem, how to keep your app up-to-date, preparing your app for release, testing your app, and publishing it for the world to enjoy!

This book sample contains the first five chapters of the full book. With this sample you'll learn how to build your first Android app using Kotlin, the hot new programming language for Android development. Everybody likes games, right? So, this first app is a simple but fun Android game named Timefighter which will teach you the basics of Android programming.

We hope that this hands-on look inside the book will give you a flavor of what's available in the full version and that it encourages you to write your own Android apps and release them on the Google Play Store!

The full book is available for purchase at:

- https://store.raywenderlich.com/products/android-apprentice

Enjoy!

— Darryl, Tom, Namrata, Fuad and the *Android Apprentice* team

Android Apprentice, Second Edition

Darryl Bayliss, Tom Blankenship, Fuad Kamal & Namrata Bandekar

Notice of Rights

Notice of Liability

Trademarks

About the Authors

Darryl Bayliss is an author of this book. Darryl is a Software Engineer from Liverpool, focusing on Mobile Development. Away from programming he is usually reading or playing some fantastical video game involving magic and dragons. You can say hello on Twitter over at @dazindustries

Tom Blankenship is an author on this book. Tom has been addicted to coding since he was a young teenager, writing his first programs on Atari home computers. He currently runs his own software development company focused on native iOS and Android app development. He enjoys playing tennis, guitar, and drums, and spending time with his wife and two children.

Fuad Kamal is an author on this book. He provides mobile strategy, architecture and development for the Health & Fitness markets. If you've ever been to an airport, you've likely seen his work — the flight arrival and departure screens are a Flash 7 interface he wrote towards the beginning of the millennium. He can be contacted through anaara.com.

Namrata Bandekar is one of the authors and a tech editor of this book. She is a Software Engineer focusing on native Android and iOS development. When she's not developing apps, she enjoys spending her time travelling the world with her husband, SCUBA diving and hiking with her dog. She has also spoken at a number of international conferences on mobile development. Say hi to Namrata on Twitter: @NamrataCodes.

About the Editors

Kevin Moore is a technical editor for this book. He has been developing Android apps for over 8 years and at many companies. He's written several articles at www.raywenderlich.com and created the "Programming in Kotlin" video series. He enjoys creating apps for fun and teaching others how to write Android apps. In addition to programming, he loves playing volleyball and running the sound system at church.

Vijay Sharma is a tech editor of this book. Vijay is a husband, a father and a senior mobile engineer. Based out of Canada's capital, Vijay has worked on dozens of apps for both Android and iOS. When not in front of his laptop, you can find him in front of a TV, behind a book, or chasing after his kids. You can reach out to him on Twitter: @v_sharm

Ellen Shapiro is a tech editor on this book. Ellen is an iOS developer for Bakken & Bæck's Amsterdam office who also occasionally writes Android apps. She is working in her spare time to help bring songwriting app Hum to life. She's also developed several independent applications through her personal company, Designated Nerd Software. When she's not writing code, she's usually tweeting about it at @designatednerd

Tammy Coron is an editor of this book. Tammy is an independent creative professional and the host of Roundabout: Creative Chaos. She's also the co-founder of Day Of The Indie and the founder of Just Write Code. For more information visit TammyCoron.com.

Eric Soto is the final pass editor of this book. Eric is a Professional Software Engineer, certified Agile-Scrum Master and Mac fanatic. Focusing on Apple iOS, Android Apps, NodeJS and APIs, Eric works with clients all across the US including many big-name brands. During his 30+ year career, Eric has also worked with web applications, server back-end systems, automated infrastructure deployments and more. Follow Eric on Twitter @ericwastaken or on his website ericsoto.net.

Book License

By purchasing *Android Apprentice*, you have the following license:

- You are allowed to use and/or modify the source code in *Android Apprentice* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *Android Apprentice* in as many apps as you want, but must include this attribution line somewhere inside your app: "Artwork/images/designs: from *Android Apprentice*, available at www.raywenderlich.com".
- The source code included in *Android Apprentice* is for your personal use only. You are NOT allowed to distribute or sell the source code in *Android Apprentice* without prior authorization.
- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

All materials provided with this book are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the properties of their respective owners.

Section I: Your First Android App

This is your introduction to creating apps in Android. This section will take you step-by-step through installing Android Studio and working inside the IDE and visual designer while you build **TimeFighter**, a simple game that uses many common Android components.

Chapter 1: Setting Up Android Studio

By Darryl Bayliss

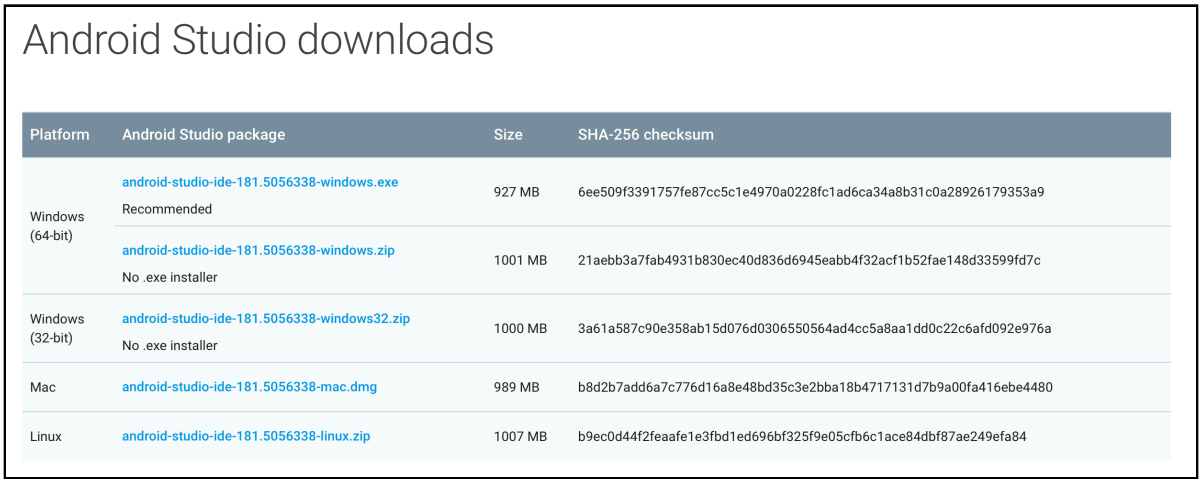
To create that killer Android App, you'll need to install the tools that you need as a young apprentice. Android development happens inside **Android Studio**, a customized IDE, based on IntelliJ, that gives you a powerful set of tools to work with.

In this chapter you'll learn:

- How to set up Android Studio on your machine.
- How to set up a physical and emulated device for development.
- How to run an app on a device.

Getting started

Open up your favorite web browser and navigate to https://developer.android.com/studio/#downloads.



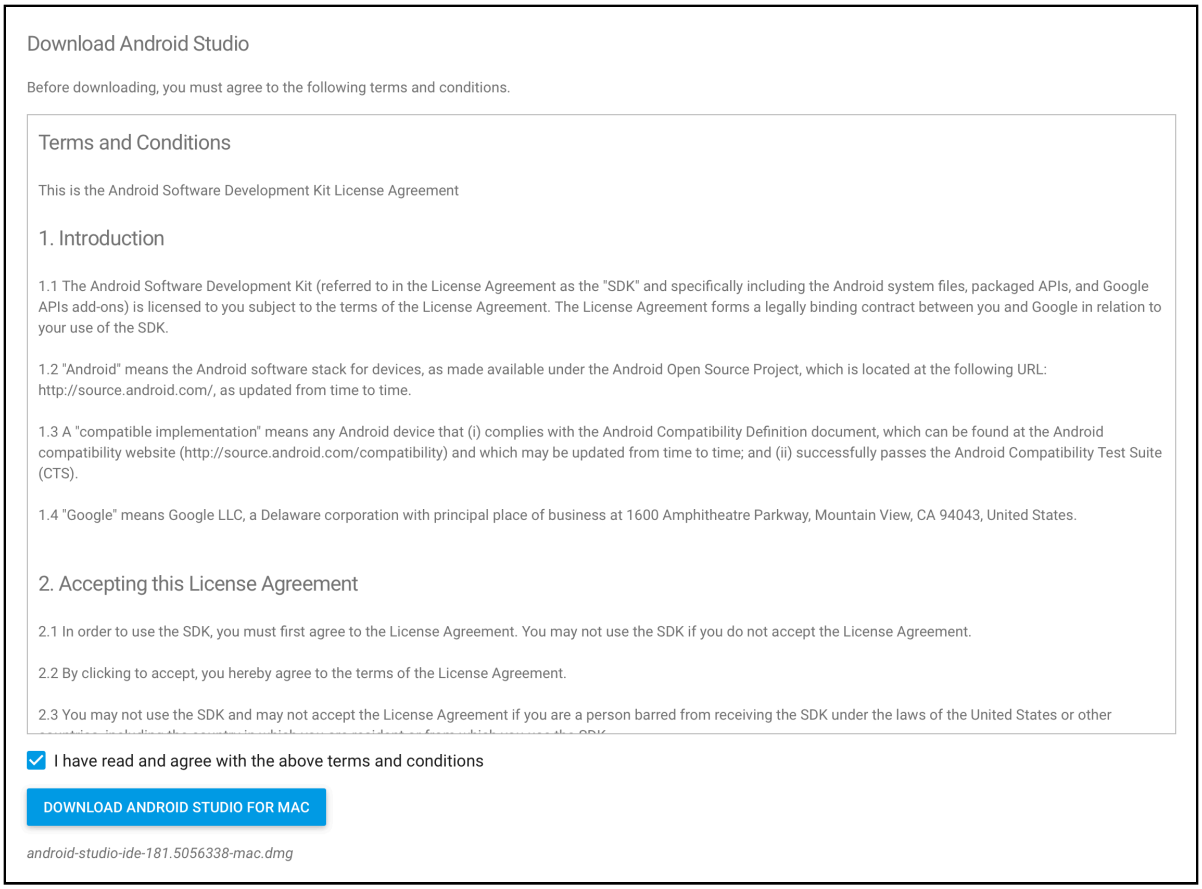
Android Studio downloads

Platform	Android Studio package	Size	SHA-256 checksum
Windows (64-bit)	android-studio-ide-181.5056338-windows.exe Recommended	927 MB	6ee509f3391757fe87cc5c1e4970a0228fc1ad6ca34a8b31c0a28926179353a9
	android-studio-ide-181.5056338-windows.zip No .exe installer	1001 MB	21aebb3a7fab4931b830ec40d836d6945eabb4f32acf1b52fae148d33599fd7c
Windows (32-bit)	android-studio-ide-181.5056338-windows32.zip No .exe installer	1000 MB	3a61a587c90e358ab15d076d0306550564ad4cc5a8aa1dd0c22c6afd092e976a
Mac	android-studio-ide-181.5056338-mac.dmg	989 MB	b8d2b7add6a7c776d16a8e48bd35c3e2bba18b4717131d7b9a00fa416ebe4480
Linux	android-studio-ide-181.5056338-linux.zip	1007 MB	b9ec0d44f2feaafe1e3fbd1ed696bf325f9e05cfb6c1ace84dbf87ae249efa84

You have a number of download options, as Android Studio can run on a variety of Operating Systems. **Click** the package for the operating system your computer uses.

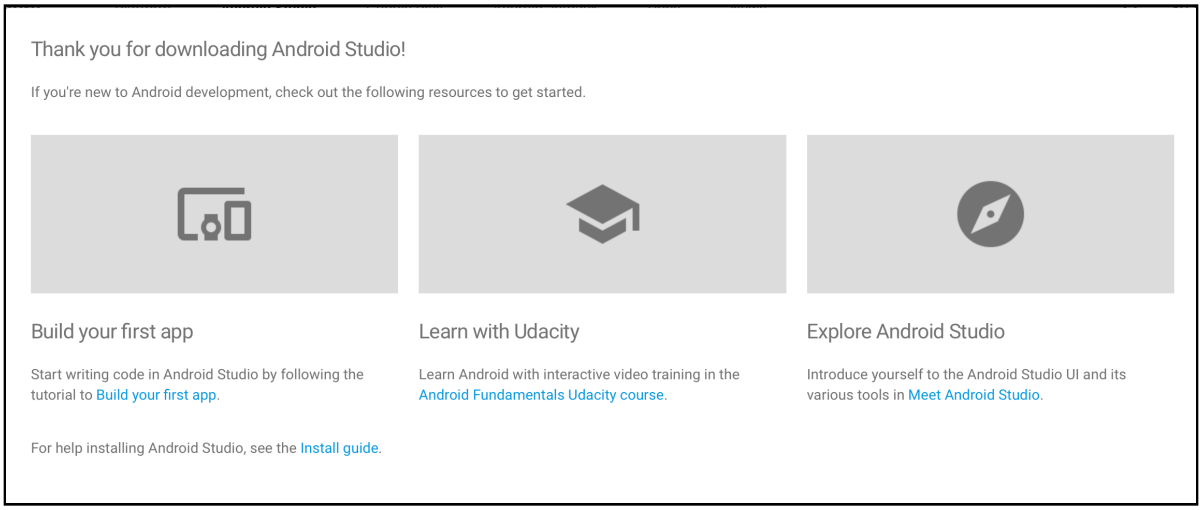
> **Note**: This chapter assumes that your computer is running macOS; however, as Android Studio supports Windows and Linux, we'll provide instructions for those operating systems as well.

The downloads page will pop up a terms and conditions screen.

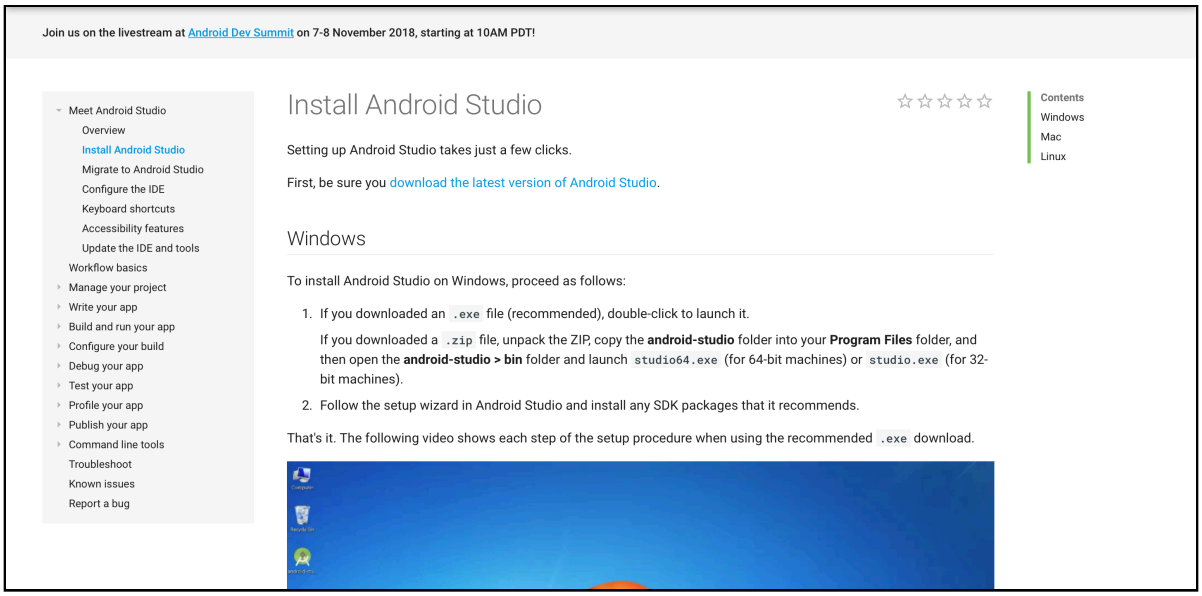


Check the checkbox at the bottom of the screen if you agree to the terms and then **Click** the Download Android Studio button.

As your computer begins to download Android Studio, your browser will show a pop-up window with some suggestions on what to do next.

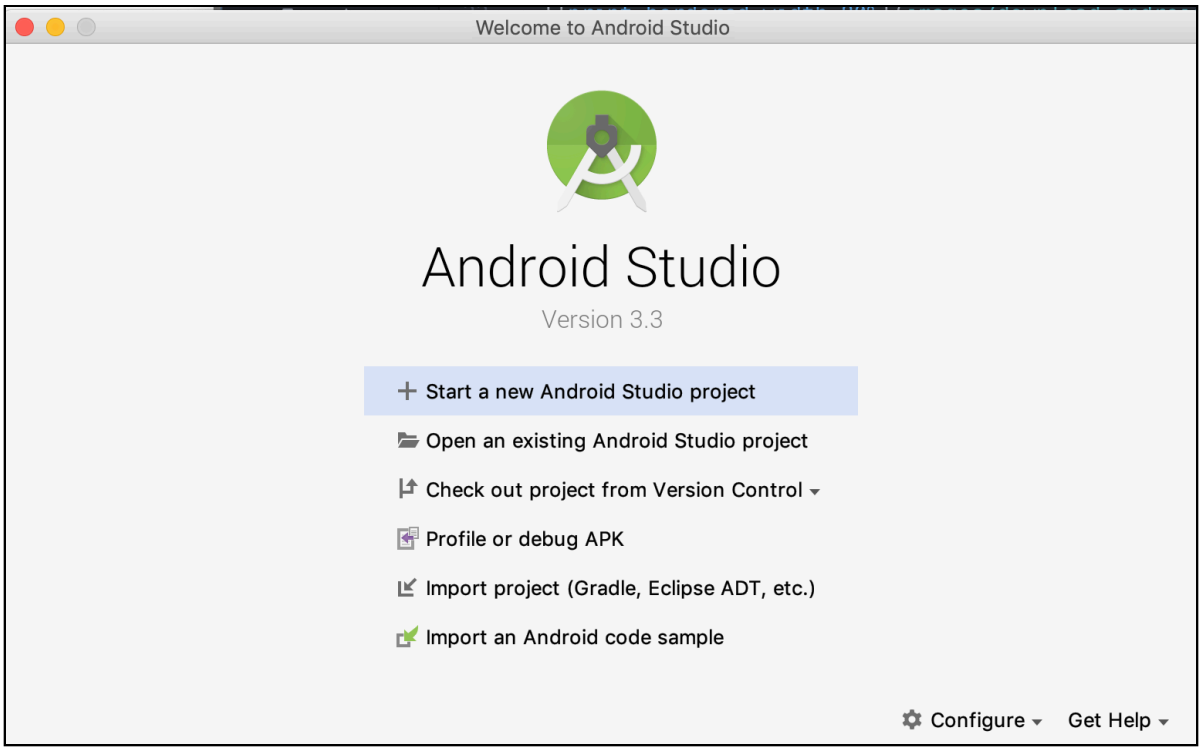


Click the blue install guide link at the bottom of the pop-up. The browser will open up a new page containing setup videos for Android Studio for each operating system.



Follow the instructions and/or video relevant to your operating system until the welcome to Android Studio window is open.

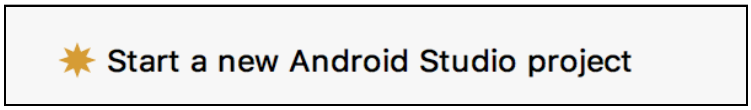
> **Note**: Unless you have an extremely fast internet connection, it will take a while for all of the components you need to download. Depending on how your system is set up, you may need to enter your password or an administrator's password to allow installation to complete.



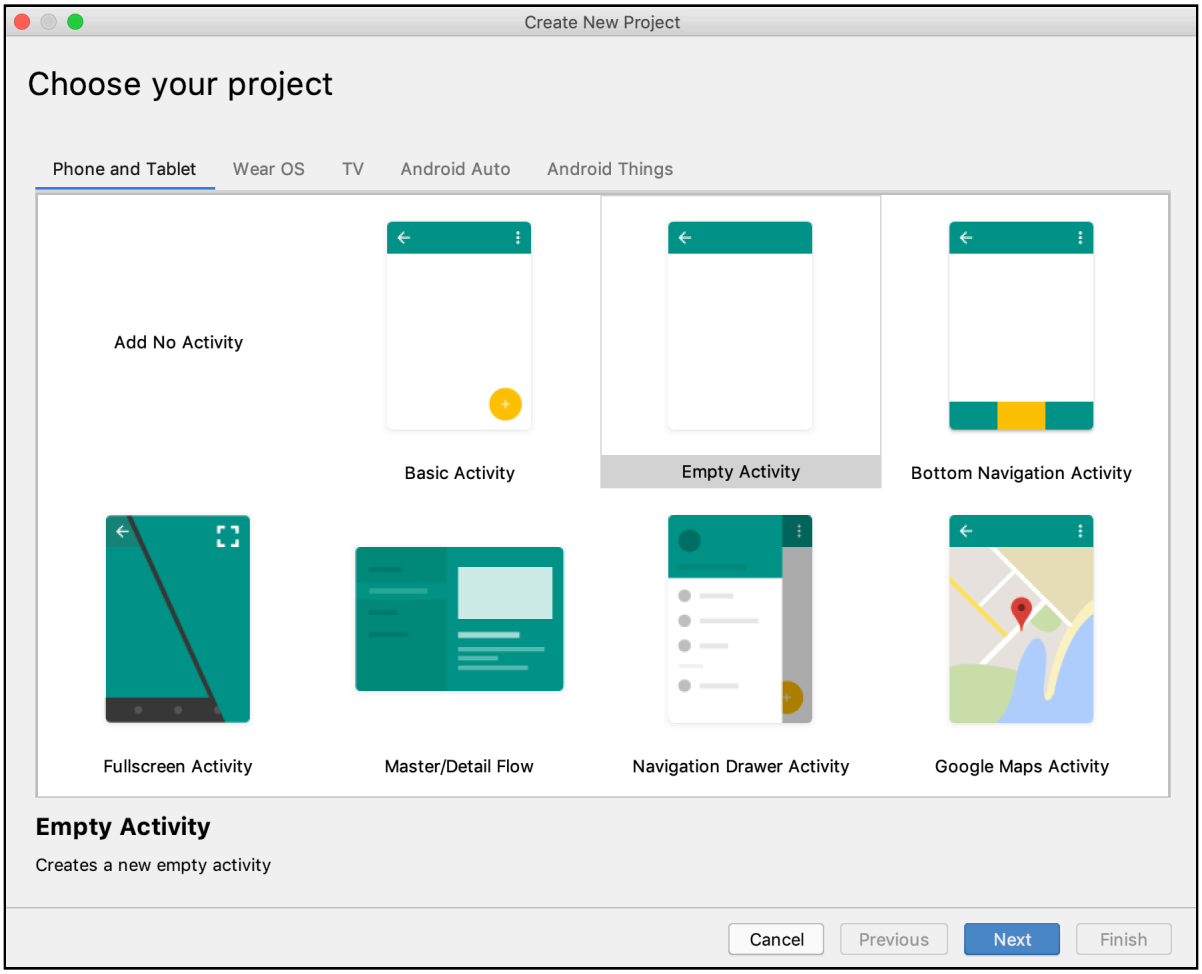
Your first Android project

Now that you have Android Studio installed, it's time to create your first project. This chapter focuses on getting your app running as quickly as possible on a device. Along the way, you'll encounter a few screens that you won't quite understand at first, but don't worry: you'll get a chance to experience these screens in detail in the full book. For this sample, just enjoy the ride!

On the welcome screen, click **Start a new Android Studio project**:



The welcome window will disappear and a new window will take its place. This is where you can set up a few key elements of your app.



Here's what each field means:

1. The first and most important thing in any app is its name. In the **Application name** textfield, enter **Timefighter**.
2. The **Company domain** textfield provides your app with a package name, a concept you should be familiar with from Java or Kotlin. Here, simply enter **raywenderlich.com**.
3. The **Project location** textfield tells Android Studio where to create the directory containing your project.

> **Note**: Feel free to create your project anywhere you want. The ellipsis button to the right provides you with a system navigator to easily find the place to create your project.

4. The **Include Kotlin Support** checkbox informs Android Studio that your project requires the Kotlin libraries to be added so you can write your app in Kotlin. It also ensures the starter project code is all Kotlin. Check the box if it isn't already checked.

5. The **Next** button in the bottom right of the window moves you to the target device section.

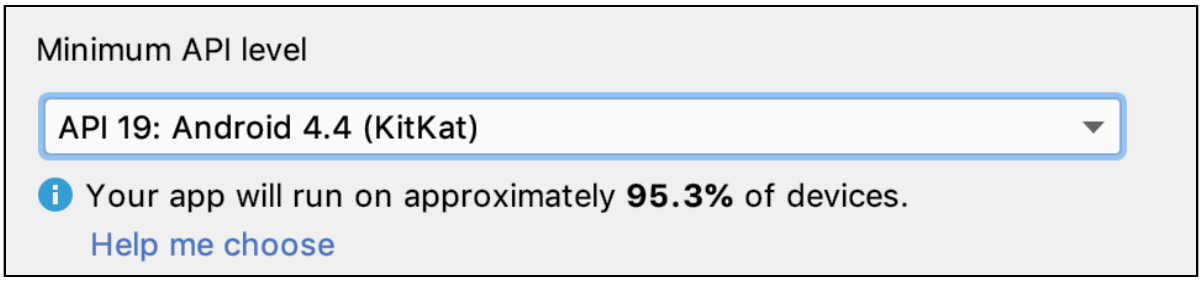
The next window provides a variety of checkboxes and dropdowns for you to configure which versions of Android you want to support:

On this screen:

1. Android supports a variety of devices: phones, watches, cars and even refrigerators! If you want to configure a new app to run on these devices, this is the screen to do it.

   Timefighter will only run on a phone, and Android Studio has already checked that option for you. As well, it has set a minimum Android version for the app in the dropdown box.

   This book requires your apps to run on API 19, or Android KitKit in English. So click the dropdown and click **API 19: Android 4.4 (KitKat)**.



2. The **Next** button in the bottom-right of the window moves you to the project template section.

The next window provides a variety of preset projects you can choose from to set up the foundation of your app. Each choice provides your starter project with different code and resources generated by Android Studio.

The **Empty Activity** is already selected for you by Android Studio, which is exactly what you want. Don't worry about what an Activity is right now; all will be revealed in the chapters to come.

Go to the bottom of the window and press the **Next** button.

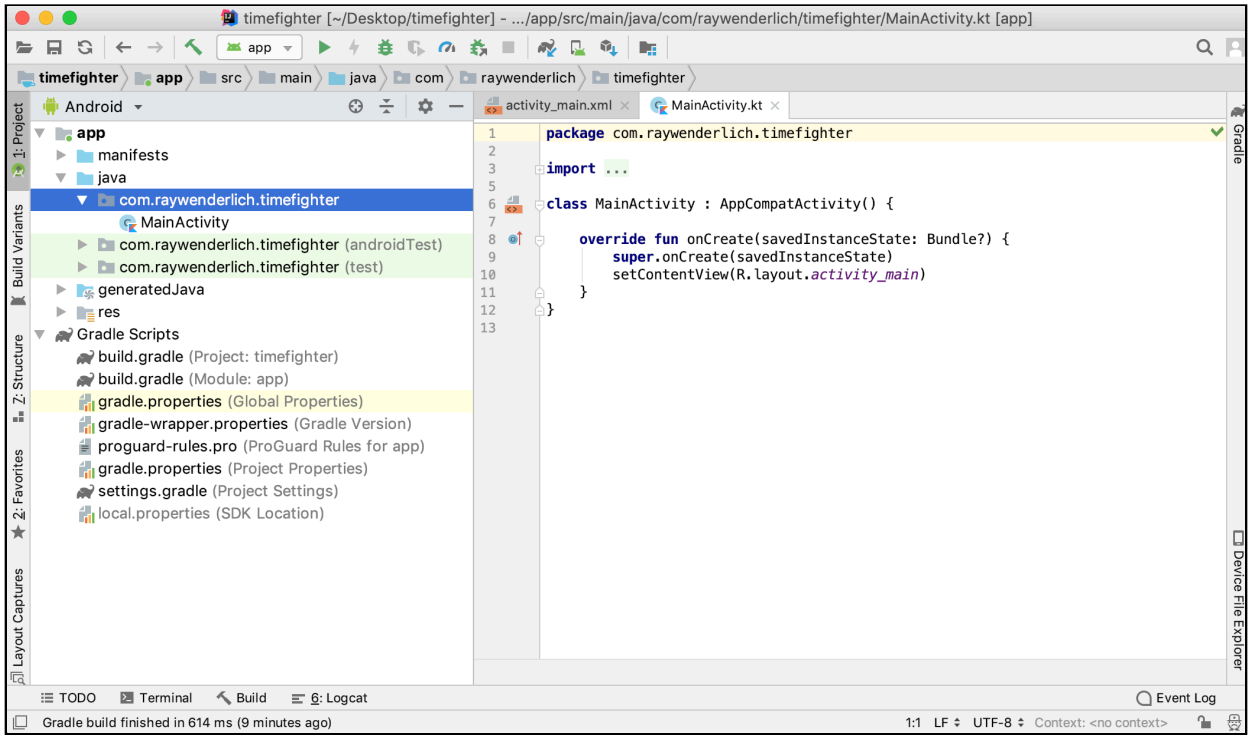
The final window is dedicated to setting up your empty Activity — in particular, giving it a name.

1. In the **Activity Name** textfield, enter the name **GameActivity**. You'll notice that the name of the layout automatically updates itself to `activity_game`.
2. Skip past the rest of the options and click **Finish**.

With the project creation taken care of, Android Studio will take all the information you provided, and gather the required libraries and resources to generate a fresh project for you.

> **Note**: The "gathering" phase may take a while, depending on the speed of your internet connection.

When that process finishes, Android Studio will show you your created project, with the `GameActivity.kt` and `activity_game.xml` files already open for you:

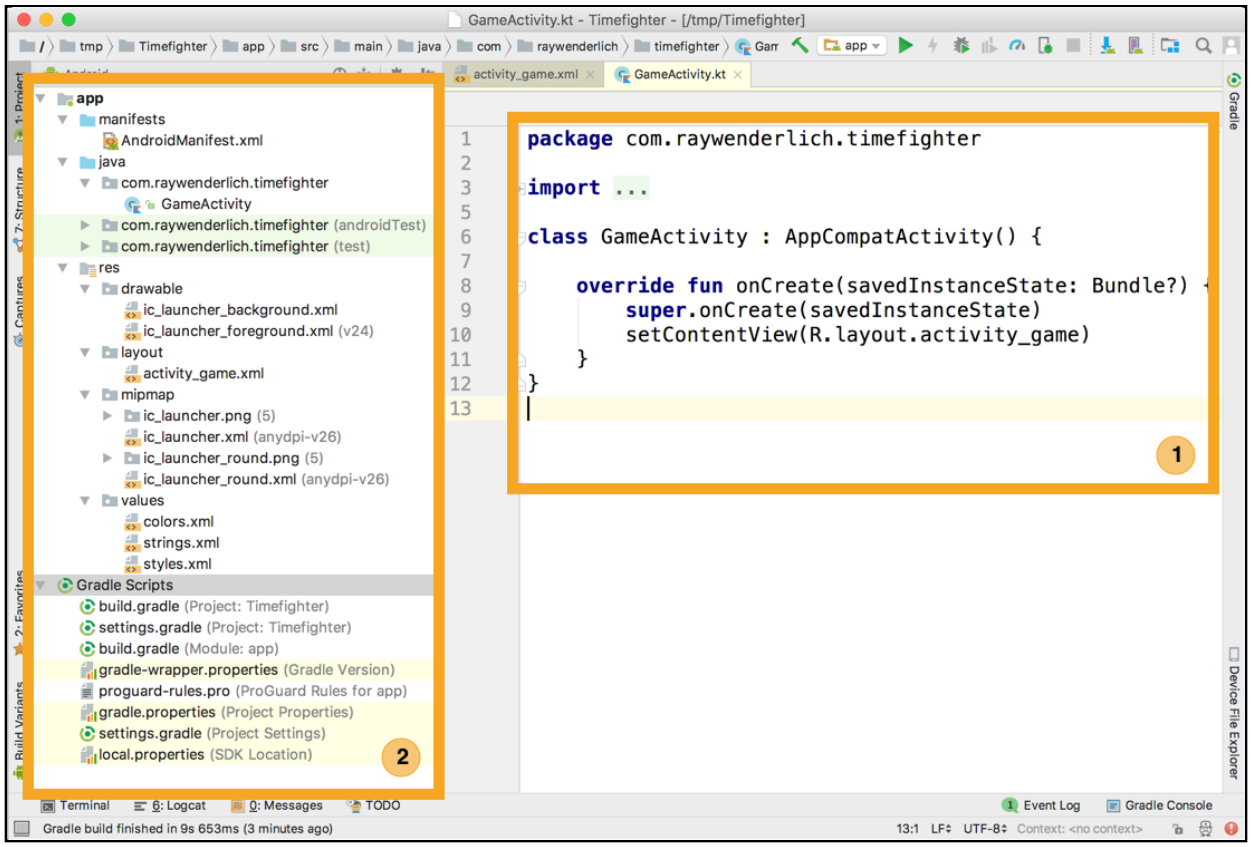


Android Studio

With your project created, you are now free to work on your project as you please.

Android Studio is a large and complex piece of software and, if you dive in without a good map, you may find yourself lost!

Before you start to build you app, take a look at what Android Studio has to offer as part of your app development experience.



1. The most obvious window that first appears to you is the **Editor**.

   This window provides you with space to edit your app's source code. It provides syntax highlighting, auto completion for methods and objects as well as the ability to drop breakpoints into your code while debugging. You'll learn more about breakpoints and debugging in "Chapter 4: Debugging." You'll spend most of your development time using the Editor to code your app to work exactly the way you intend.

2. The other window that you'll spend most of your time with is the **Project navigator**, to the left of the Editor.

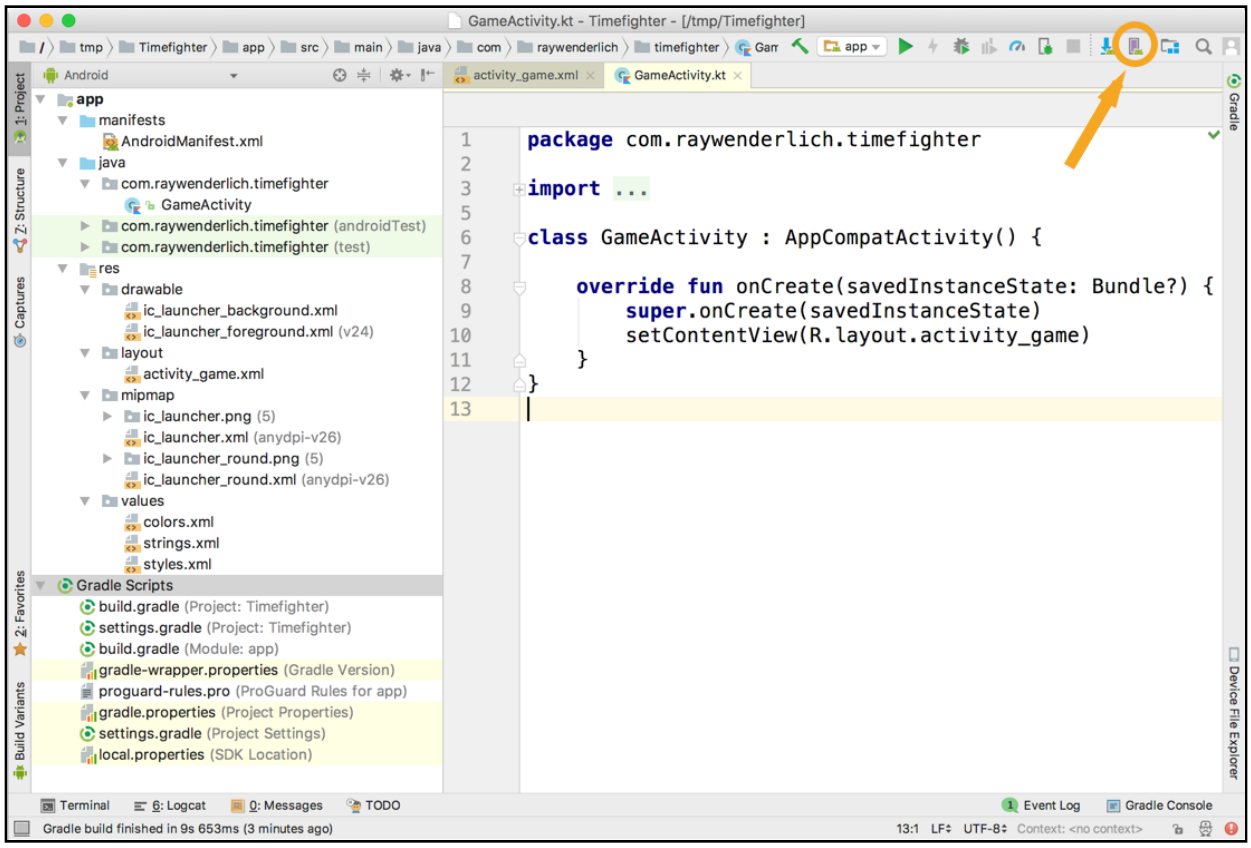
   This window shows you everything your project contains; from code to image assets, you can find it here. Android Studio already provides you with a lot to begin with. You can see this by left-clicking on the arrow to the left of the items in the Project navigator.

   Don't worry about these files for now. You'll become well-acquainted with them in the chapters to come.

Creating an Android virtual device

> **Note**: If you have a physical Android device you want to use for development, feel free to skip to the next section.

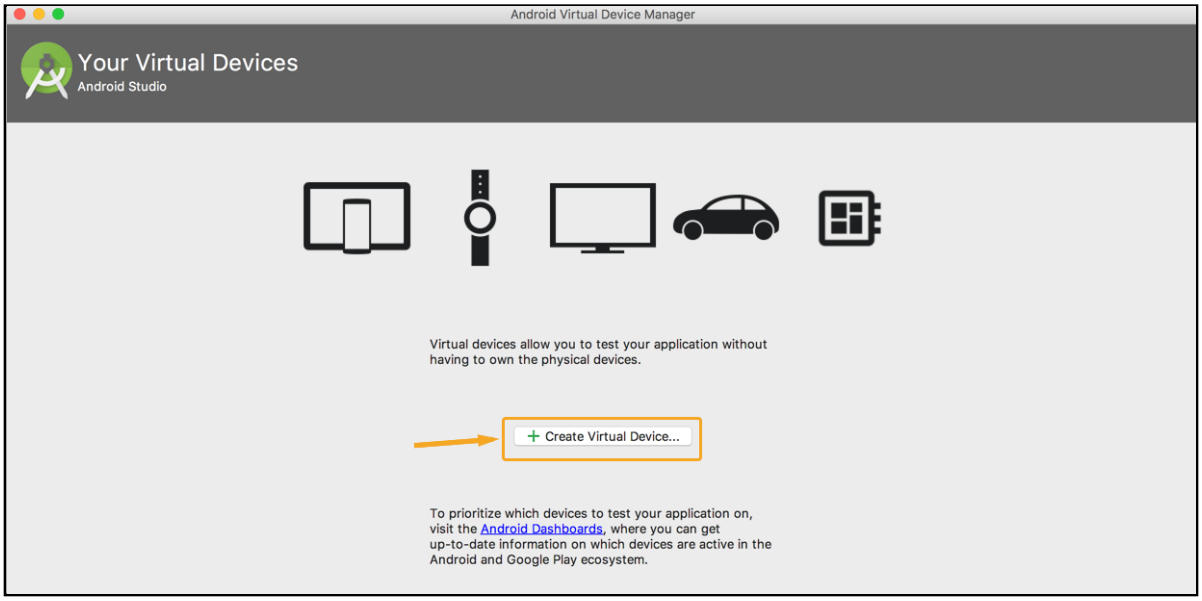
Looking at editors and files is great, but after you're done writing code, you'll likely want to run your app. But before you can run your app, you need a device — real or virtual — on which to run it! Take a look at the button highlighted in the following image.



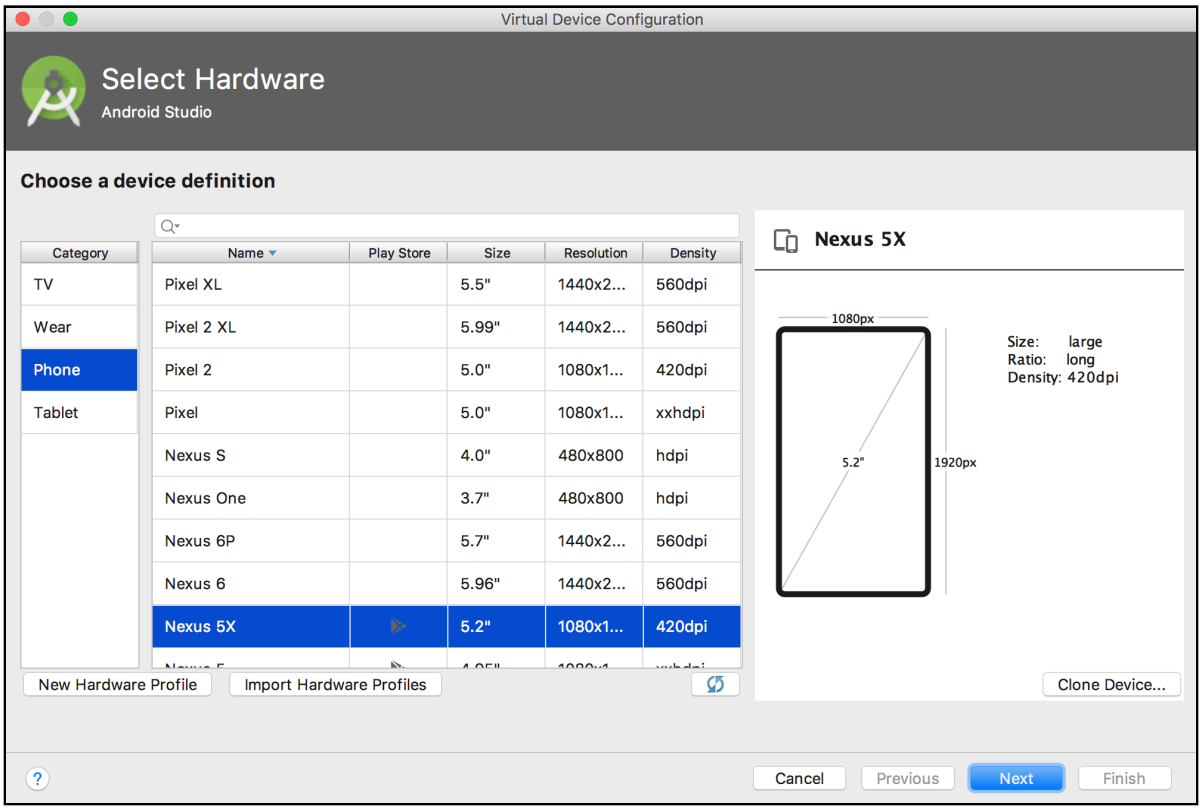
This button allows you to create an **Android Virtual Device**, or AVD for short. This is an emulator that pretends to be a device on your computer, which lets you test your app without requiring a physical device. If you don't have a physical device to test your app with, you'll need to create a virtual device before you can run your app.

Click the Android Virtual Device button, and a new window will appear.

This window shows all the available AVDs that exist on your machine. Since you've just installed Android Studio, no AVDs will be available yet.

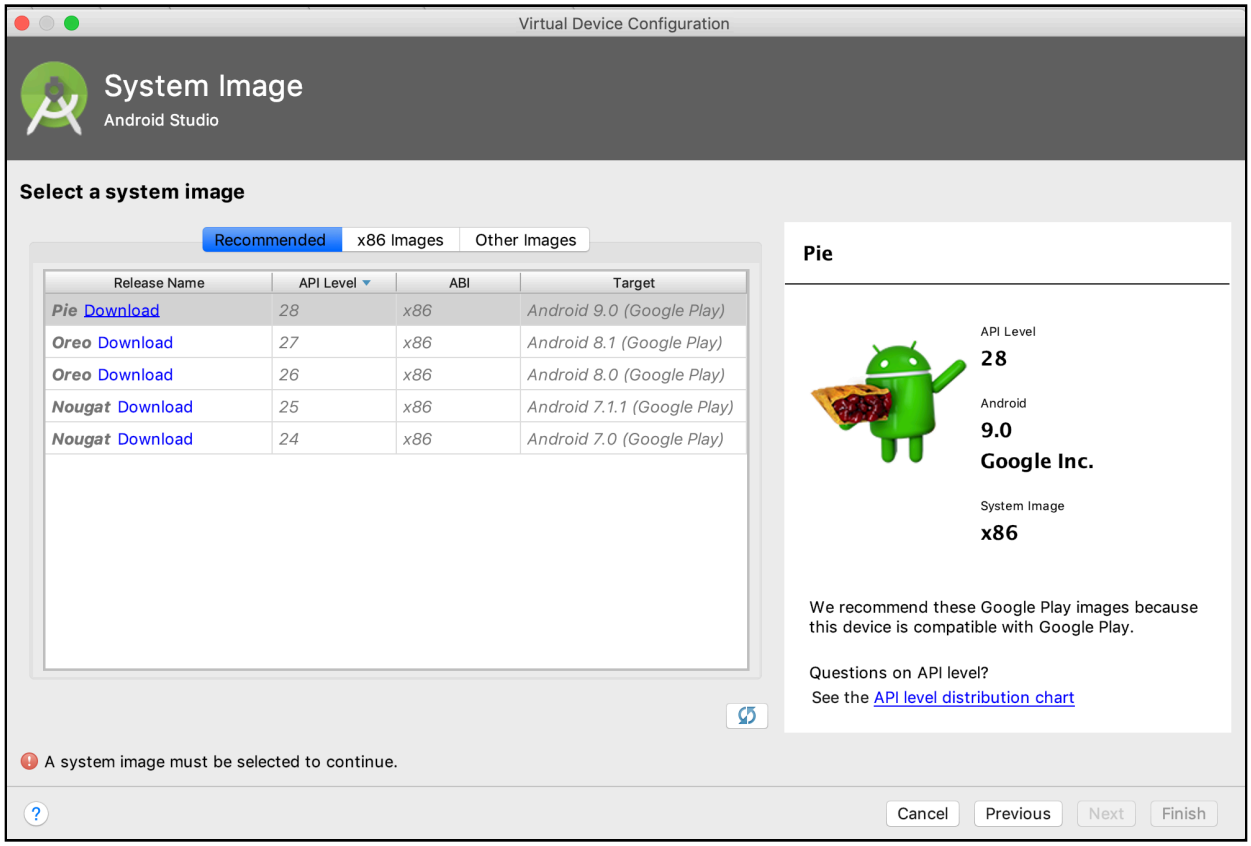


Click the **Create Virtual Device** button in the middle of the screen and the **Select Hardware** window. This window allows you to select what kind of device you want your AVD to emulate.



You will already have a device selected for you: a Nexus 5X. You'll use this device since it closely emulates a real device used by many people.

In the bottom-right of the window, click **Next**. This will display the **System Image** window, where you can choose the version of Android that runs on your emulator.

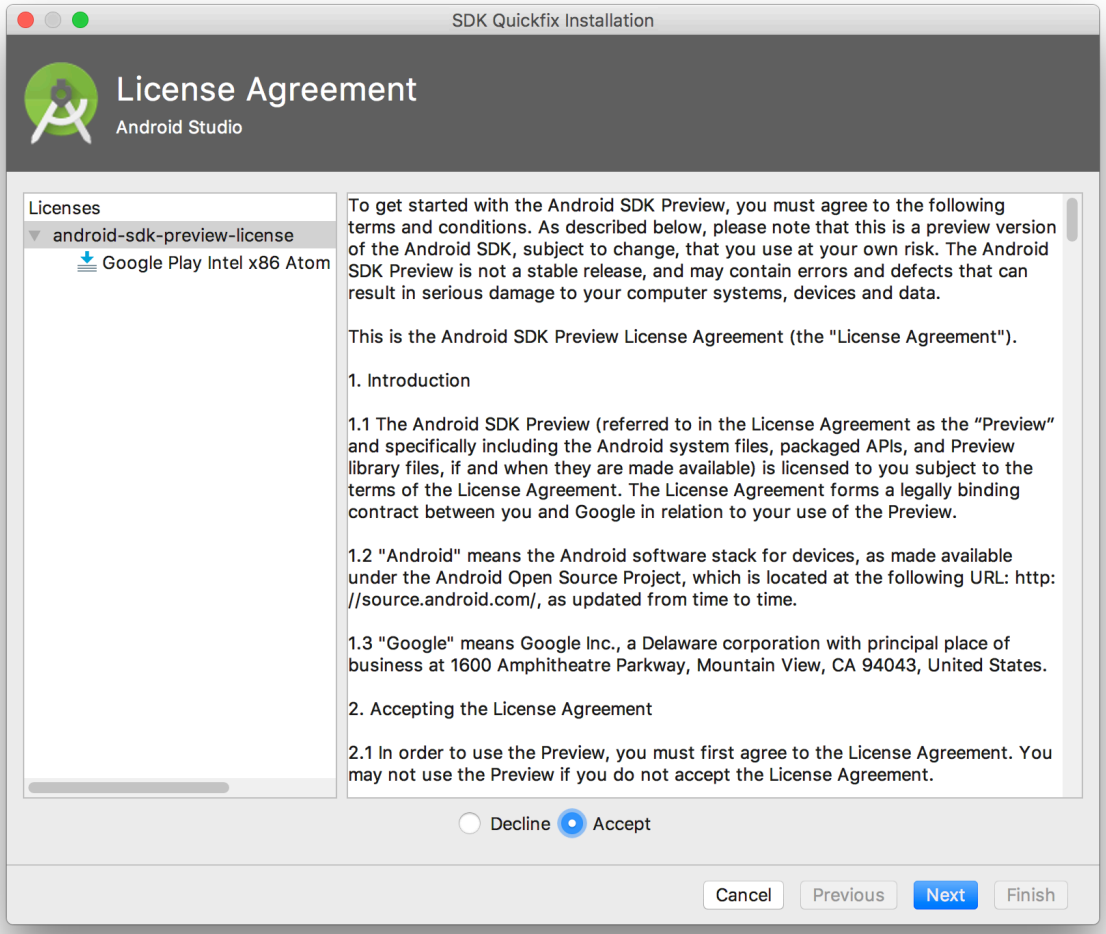


A number of tabs run along the top of the list within the window. The **Recommended** tab is a list of Android versions that Google recommend you use to test your apps.

At the moment, those versions are grayed out. That's because you haven't installed any of them onto your machine.

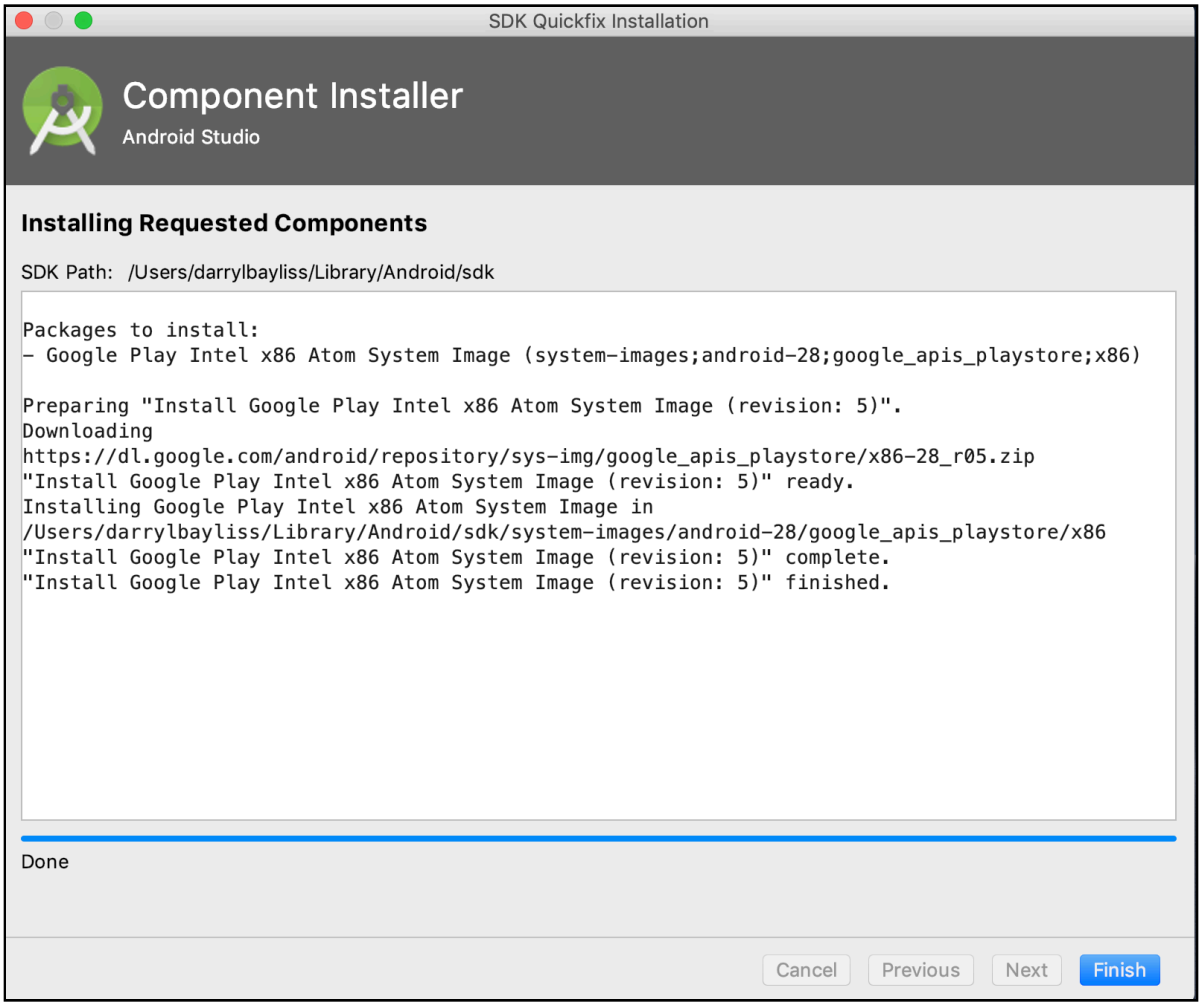
You'll download the latest and greatest recommended by Android Studio. Select the top item in the table and click the Download button in the **Release Name** column.

You'll see another legal agreement you need to agree to:



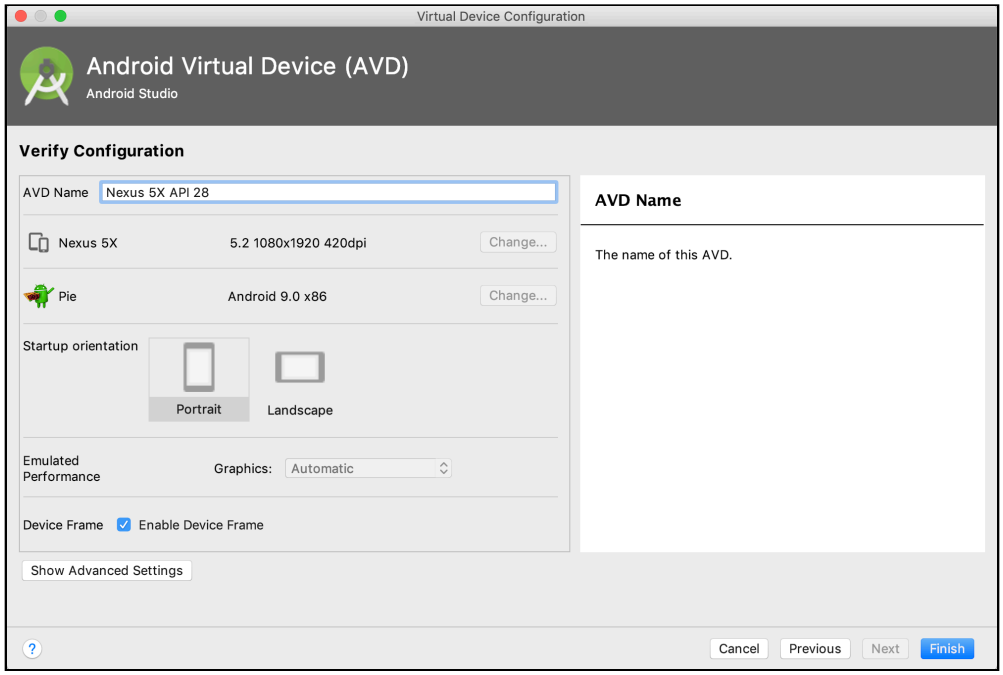
Select **Accept**, then click **Next**.

The **Component Installer** window will appear and automatically download the version of Android you selected.



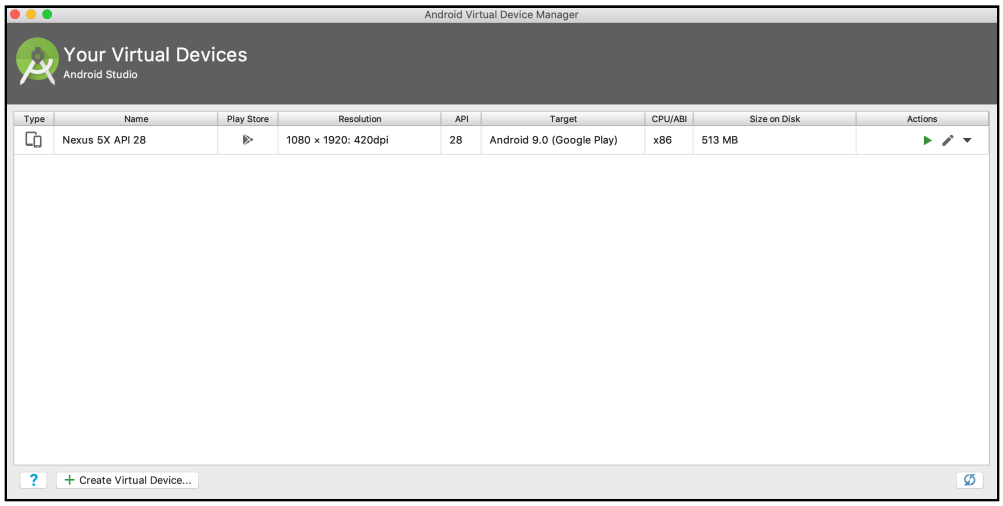
Once the download has finished, **Click** the Finish button in the bottom-right. The component installer window will disappear and the System Image window will appear again. At this point, your Android version is ready to use. To move on, **Click** the Next button in the bottom-right of the window.

The next and final window for creating your emulated device is a summary of the characteristics your device will have.



This window gives you the opportunity to give your AVD a name and to confirm other aspects of the device such as the Android version. You don't need to do anything here, so **Click** Finish at the bottom-right of the screen to create your AVD.

The current window will disappear. In the original AVD window that listed all available AVDs, you'll see your freshly created AVD ready for use:



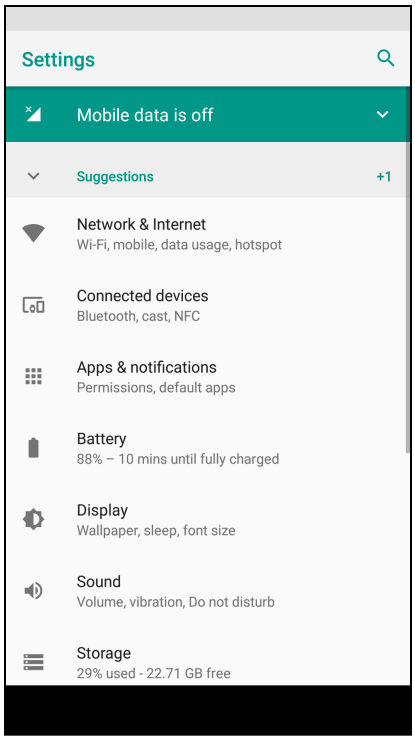
Setting up an Android device

> **Note**: If you don't have an Android device to use for development, read the previous section on how to set up an Android Virtual Device.

One of the joys of Android development is having your app working on your own device to show to your friends. But before you can install Timefighter onto your device, you need to get your device up for use with Android Studio. The first thing to do is to connect your Android device to your machine via a USB cable.

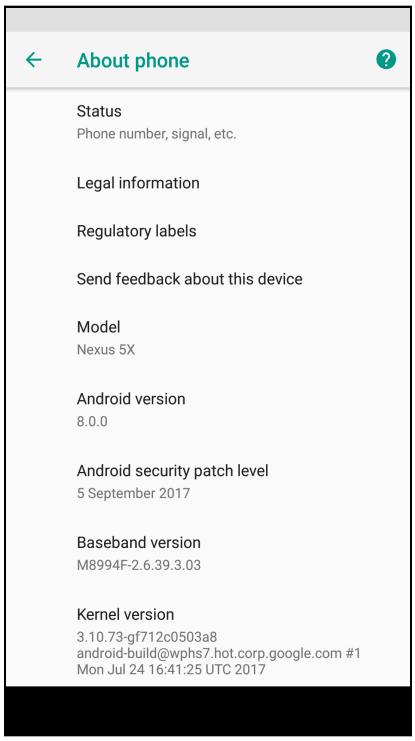
> **Note**: If you're using a Windows machine, you'll need to download a USB driver for your device first. You can download the driver and find instructions on installing it at https://developer.android.com/studio/run/oem-usb.html.

On your device, open the **Settings** app.

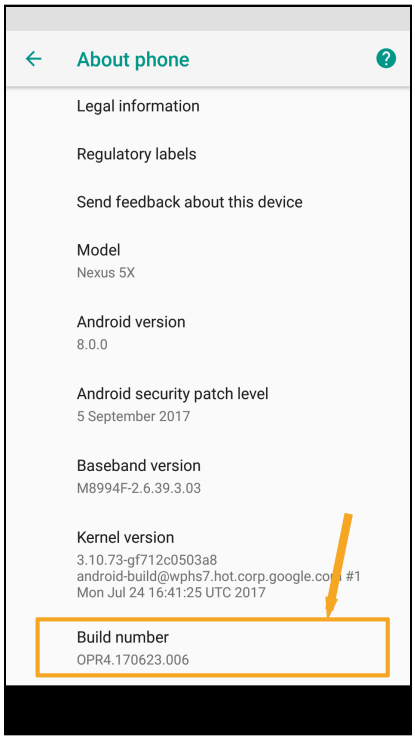


> **Note**: If your device is running Android 8.0 (Oreo) or above, you need to tap **System** first to find the **About Phone** section.

Scroll through the settings until you find the **About Phone** button and tap it.



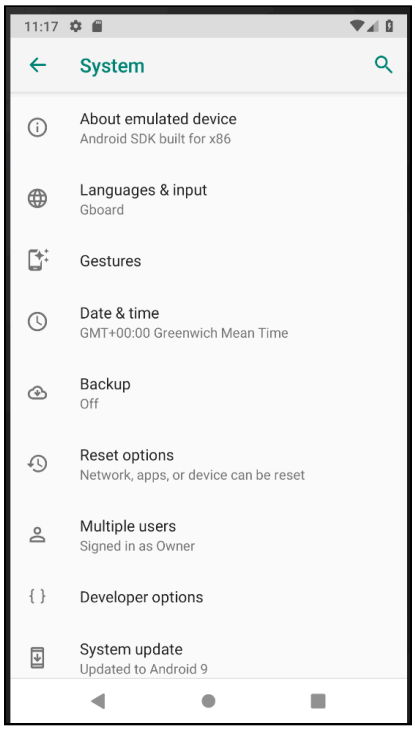
Now for the magical part! Scroll to the bottom of the **About Phone** screen, until the build number item appears:



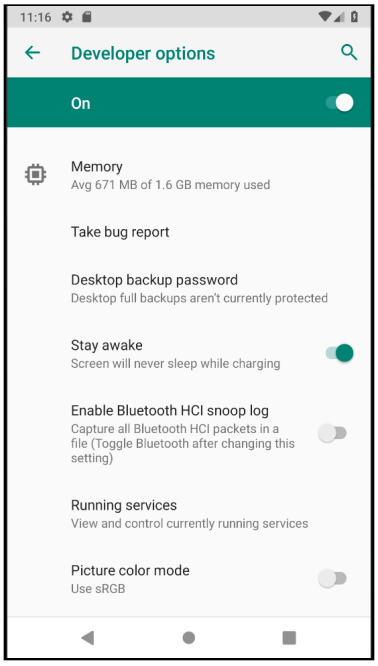
When you see the build number item, tap it several times until you see a toast message appear, informing you are a few steps away from being a developer. Keep tapping away until you get another toast message telling you that you've become a developer.

> **Note**: If your device is locked with a PIN, you will need to enter it first before you can reach this stage.

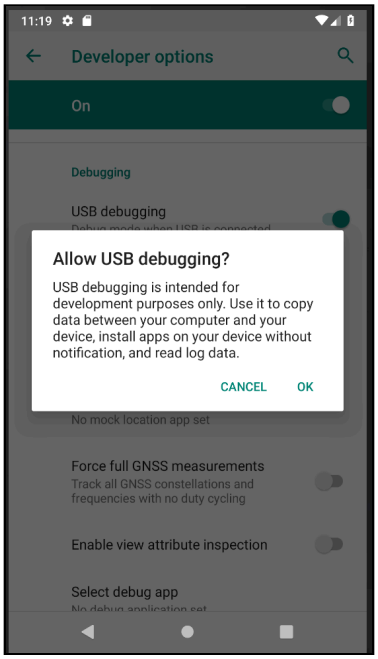
So what did this magical button do? Tap the Back button to go to the previous Settings page. Notice anything different?



A new item has appeared called **Developer Options**! Tap the option to check out all the developer features available to you.

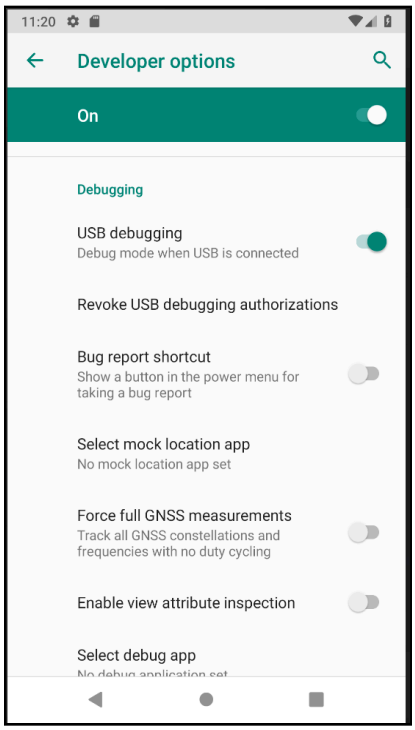


There's a lot, here, but there is only one option you need right now: **USB Debugging**. Scroll down to the option and toggle it on. A dialog will appear informing you of the intended usage of USB debugging.



Granting **USB debugging** privileges is a potential security hole, so most devices have this turned off by default. Since you will need to install apps over USB as a developer, you need to turn this on.

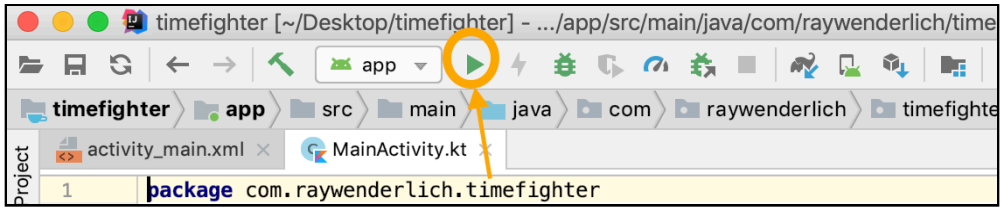
When you are ready, tap **OK** and the USB Debugging toggle will enable.



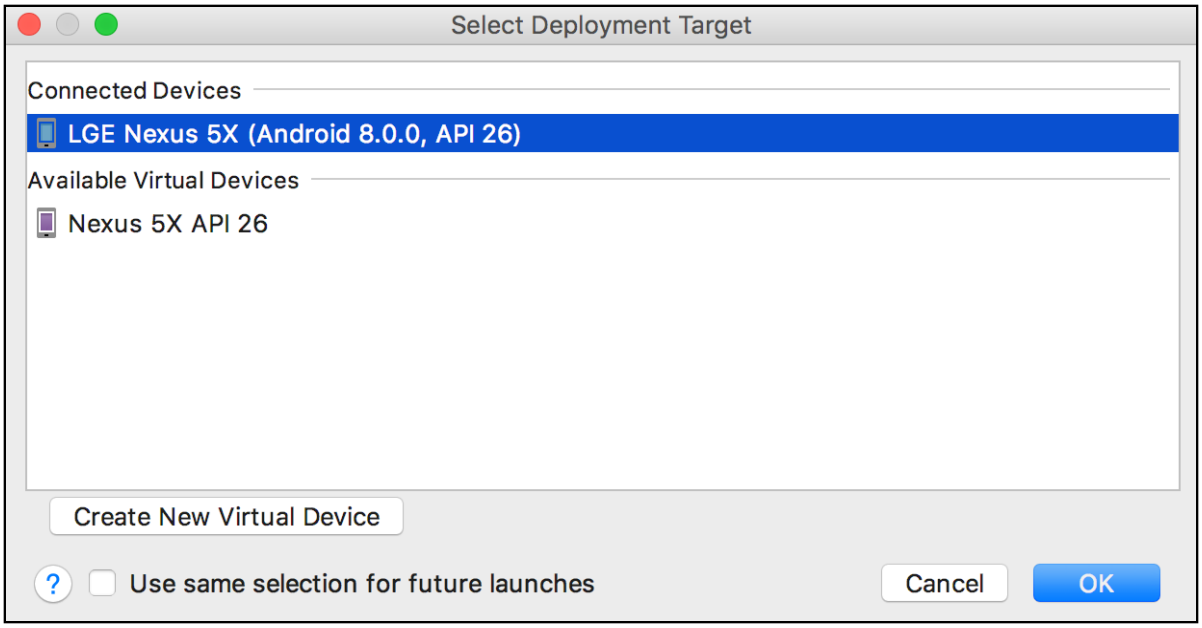
Congratulations: your device is now set up for development!

Running the app

It's time to run Timefighter! Along the top of Android studio, there is a button that looks like a green Play button:



Click the button, and a new window will appear over Android Studio.

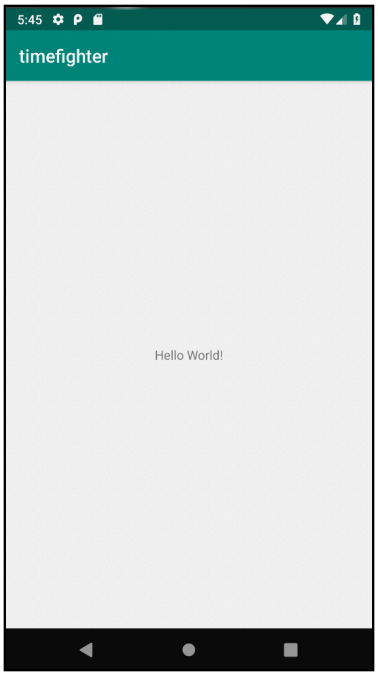


This is the **Select Deployment Target** window, showing you all available devices and emulators you can use to run your app.

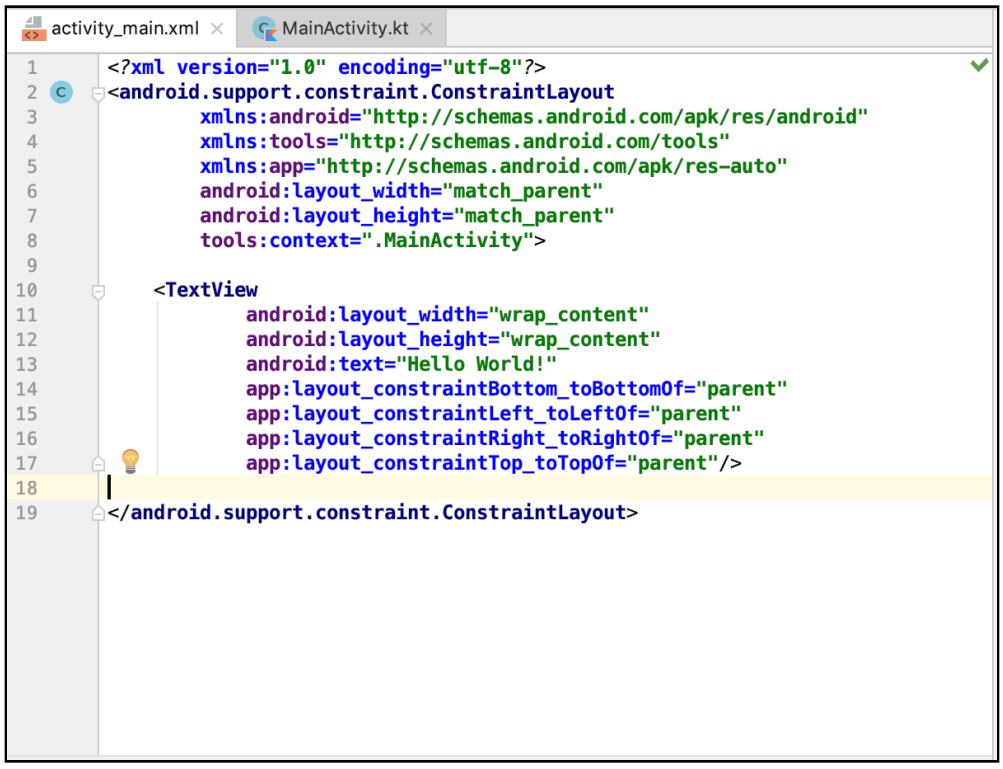
If you have a physical device with developer mode enabled, this will appear in the **Connected Devices** section. If not, then the emulator you set up will be available in the **Available Virtual Devices** section. If you have both, then well done for tackling both sections!

Select either of the devices available and click OK on the bottom-right of the window. Android Studio will begin building Timefighter and installing the built app on your device. You can see this happening at the bottom of Android Studio.

When Android Studio finishes, Timefighter will appear on your device:



You've just built your very first Android App! To celebrate, let's make it a little more personal. Head back to Android Studio, and in the project navigator open **app** ▸ **res** ▸ **layout** ▸ **activity_game.xml**, then switch to the `Text` tab on the bottom.

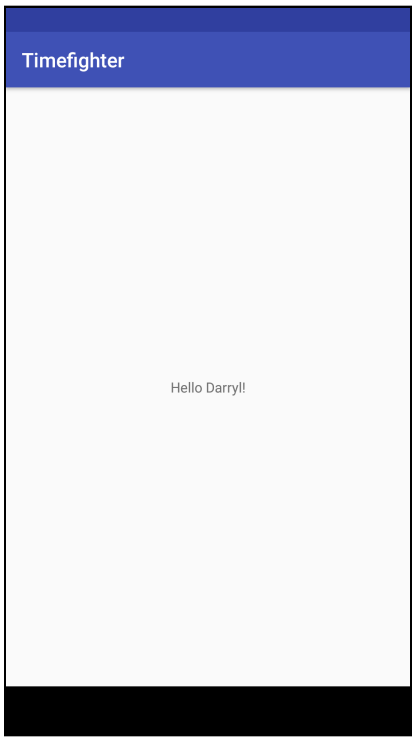


Don't worry too much now about everything you see in this file. All you need to know is that it represents the app screen that appeared on your device earlier. You'll learn more about this in the next chapter.

For now, inside the **TextView** tag, update `android:text` property to greet you with your own name:

```
android:text="Hello Darryl!"
```

Click the green Play button again to run your app:



Installing new versions of Android studio

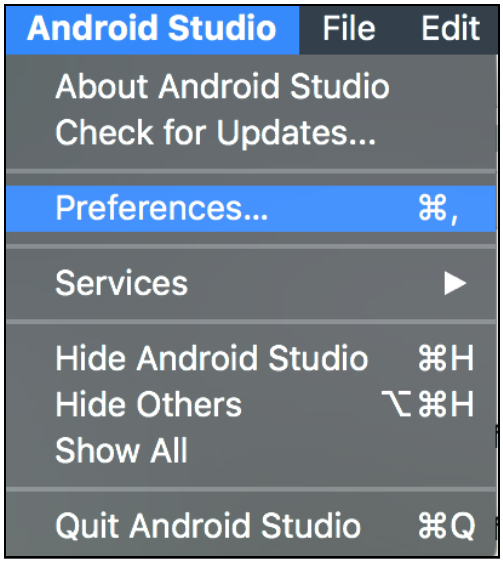
This book was written assuming a specific version of the Android SDK; as of this writing, we used Android Pie API version 28 as a baseline. However, you might be reading this book long after that version has been superseded. In that case, you may need to install the latest versions of Android Studio and the Android SDK.

> **Note**: Google engineers have decoupled Android Studio from versions of Android. This means you can build apps in Android Studio with any version of the Android operating system you want, including any future versions of the Android SDK.

Android Studio will do its best to prompt you when new versions of either Android Studio or Android SDK are available; however, you don't have to wait on Android Studio to do that for you.

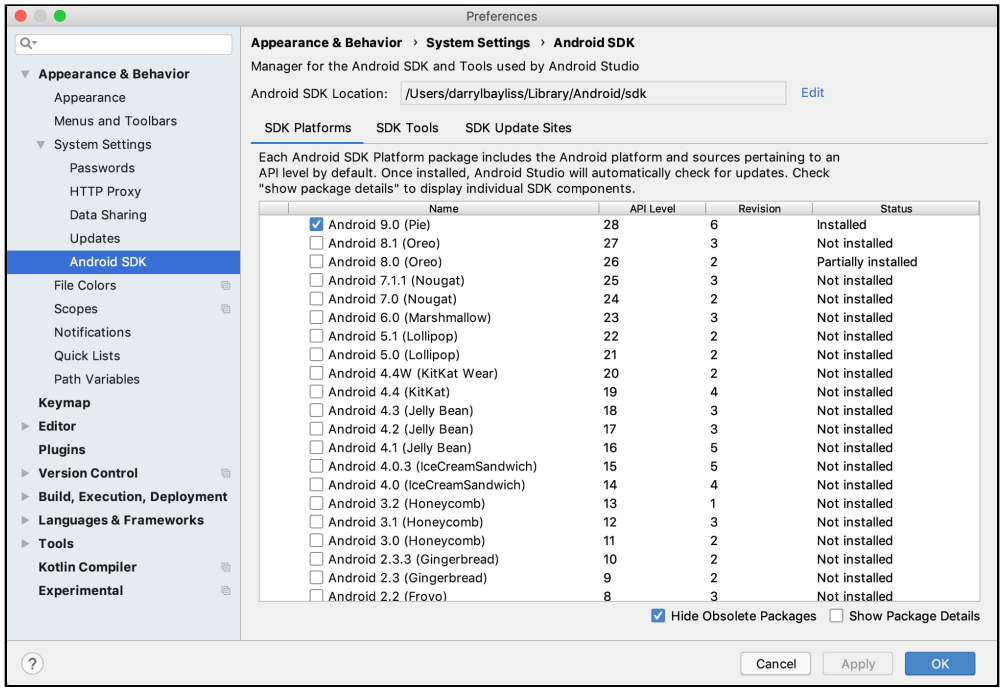
In the **Android Studio** menu, selecting **Check for Updates** will give you a dialog with all things that can be updated on your machine, or which lets you know you're up to date already.

If you'd like to download a newer (or older) version of the Android SDK, in the same menu, select the **Preferences...** menu item.



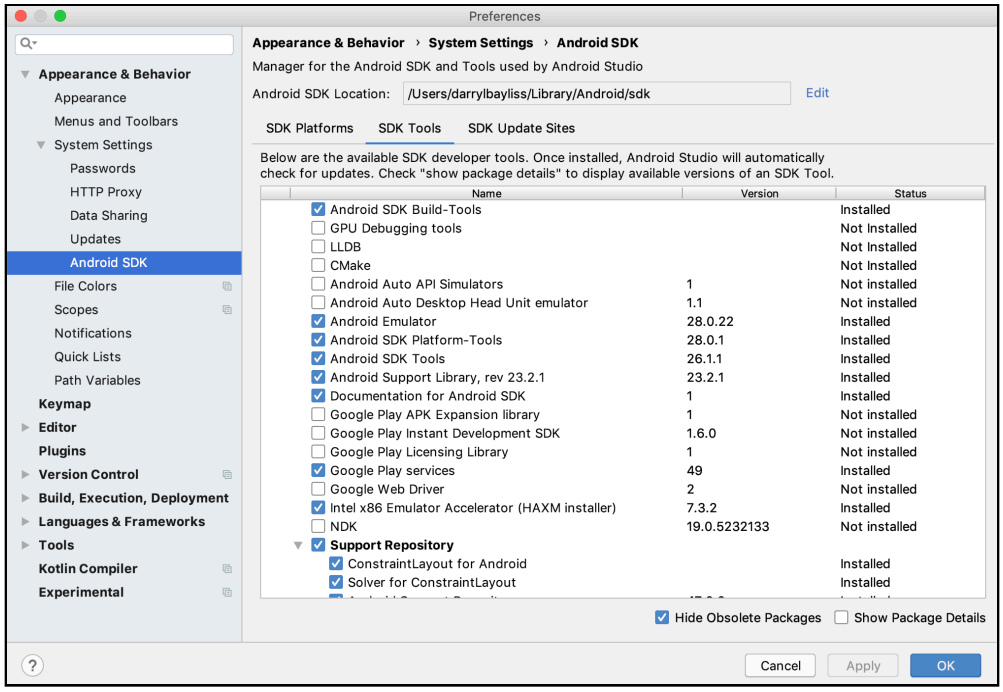
In the **Preferences** dialog, drill down through the menu items in the tree to **Appearance & Behavior ▸ System Settings ▸ Android SDK**.

In this window, there are two tabs to note: **SDK Platforms** and **SDK Tools**. In **SDK Platforms**, you should see a list of all the available Android SDK.



Clicking on any one of the SDK in the list, and then clicking **OK**, will install that SDK.

In **SDK Tools**, you will see a list of all the available build tools that Android Studio and your app have access to.



Clicking on any one of them, and then clicking **OK** will install that SDK.

At this point in the book, don't worry too much about why you would need to download any of these items. You'll learn more about each of these topics throughout the book. However, it's worth seeing these things now so that you'll recognize them as you encounter them in later chapters.

Where to go from here?

Well done getting your first app up and running! This is just the beginning; the next few chapters in this section will teach you even more about the basics of Android development. As you work through the chapters in this book, you'll end up with a fully featured app! Head on into the next chapter to start building out your app!

Chapter 2: Layouts

By Darryl Bayliss

If bricks and mortar are the foundation of a sturdy building, then **Layouts** are the Android equivalent of a sturdy app. Layouts are incredibly flexible. They let you define how to present your app's user interface on the device. You can create Layouts in one of two ways:

1. Using an XML file to declare the user interface ahead of time.
2. Writing Kotlin code to create the layout at runtime programmatically.

In this book, you'll define your Layouts ahead of time using XML — that's because Android Studio has a powerful Layout editor that covers 90% of the cases you'll ever need when creating a user interface.

Getting started

Before diving into the wonderful world of Layouts, take a few moments to think about what makes up an app. Most often, an app is a self-contained program that lets its users perform one or more tasks. When you build an app, you want your users to accomplish those tasks quickly and intuitively, which is why having a well-thought-out user interface is so important.

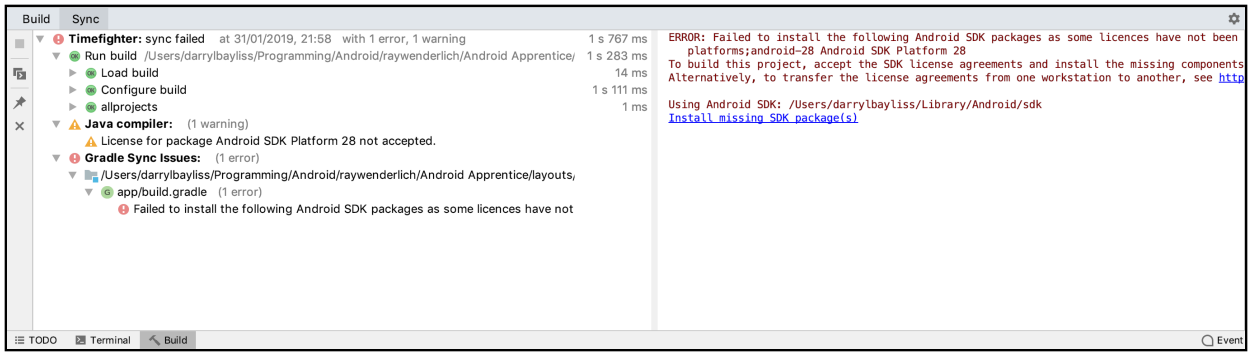
The app you'll build in this section, TimeFighter, is no different. It's minimal in its design, so usability isn't an issue.

Your first task is to set up the user interface, which has two **TextViews** and one **Button**.

Locate the **projects** folder for this chapter and open the TimeFighter app inside **starter**. The first time you open the project, Android Studio will take a few minutes to set up the environment and update its dependencies. After that, you're ready to rock and roll!

These are not the SDKs you're looking for

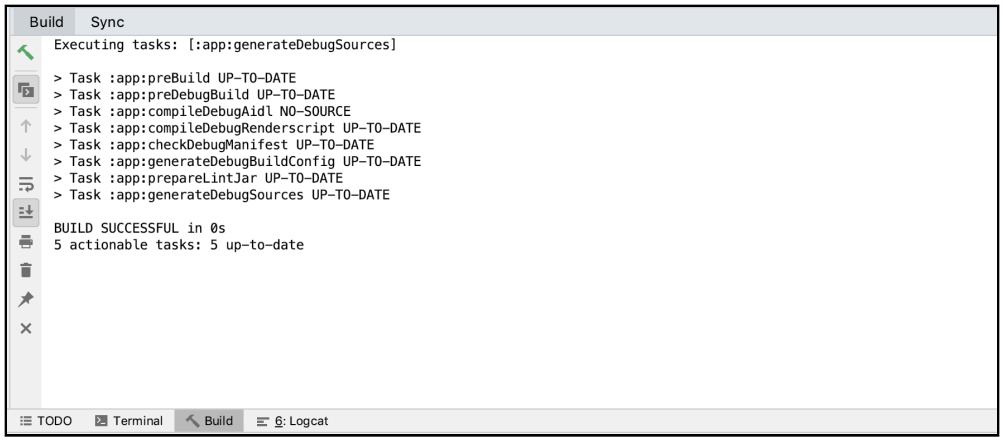
When you open the project, you might get the following error in the **Build** tab:



If you followed along in the previous chapter and installed a fresh version of Android Studio, you may not see this error. However, if you're already running Android Studio, it's possible that you don't have the version of the Android SDK that was used to create this project.

Do not fret young padawan learner; Android Studio will always do its best to help resolve these sorts of issues for you. On the right, Android Studio provides you with a convenient link, that when clicked, will install the required version of the Android SDK *and* rebuild your project.

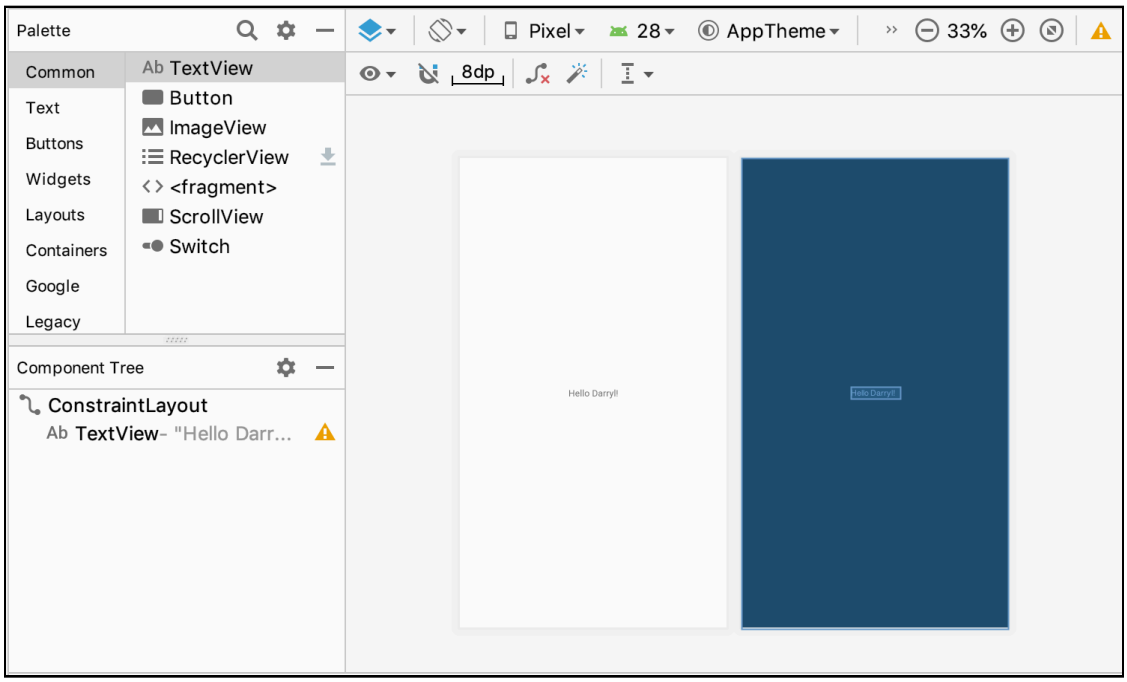
After this error disappears, you'll see the following in the **Build** tab:



Excellent! It's time to get comfy with everything Android Studio has to offer.

The Visual editor

In the project structure sidebar on the left of Android Studio, expand **app**, **res**, and **layout**. Then, double-click **activity_main.xml** and you'll see a screen that looks like this:



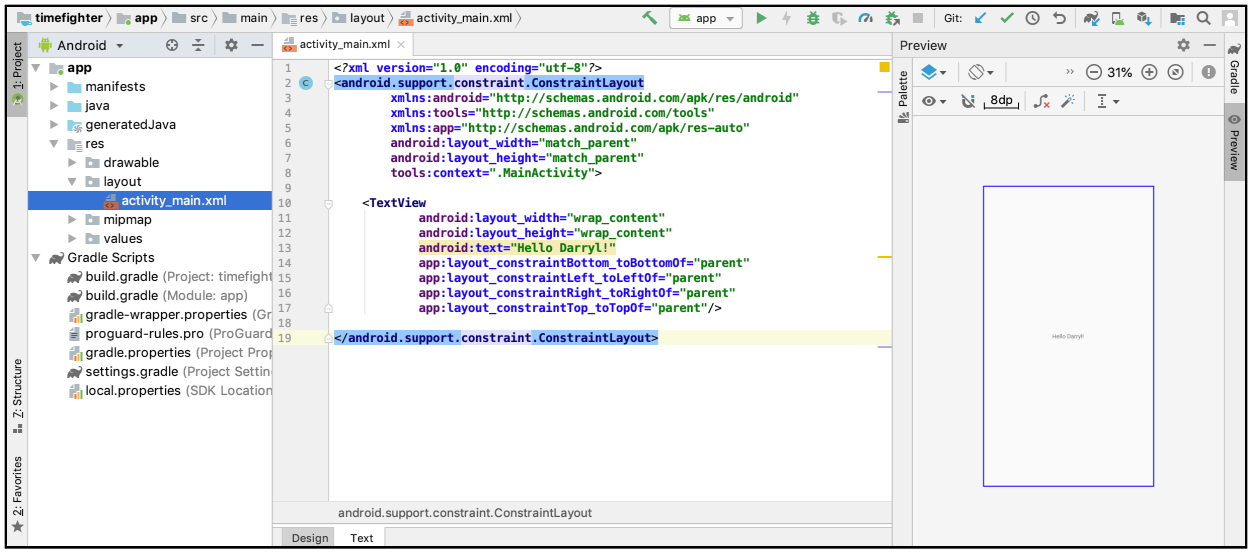
Editing activity_main.xml in Visual Editor

Behold the **Visual editor**!

In **design mode**, the middle of Android Studio shows a few different screens.

The first screen, located in the middle next to what looks like a blueprint, is the preview area. This is where you'll begin to build the user interface.

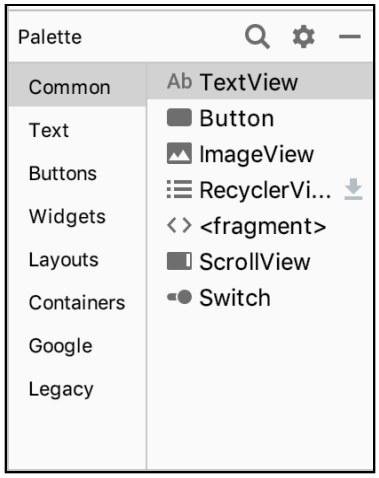
At the bottom of the Visual editor, you'll find two tabs: **Design** and **Text**. Click **Text**, and you'll see this:



Editing activity_main.xml in XML Editor

In the middle section of Android Studio, you'll see the **Text editor**. This shows the XML representation of the app's first screen. You can create the interface here if you like, but using the Design tab provides you with more visuals.

Click **Design** to switch back to design mode. You'll start by adding a **TextView** to the user interface. In the top-left of the middle section of Android Studio, you'll see the **Palette**:

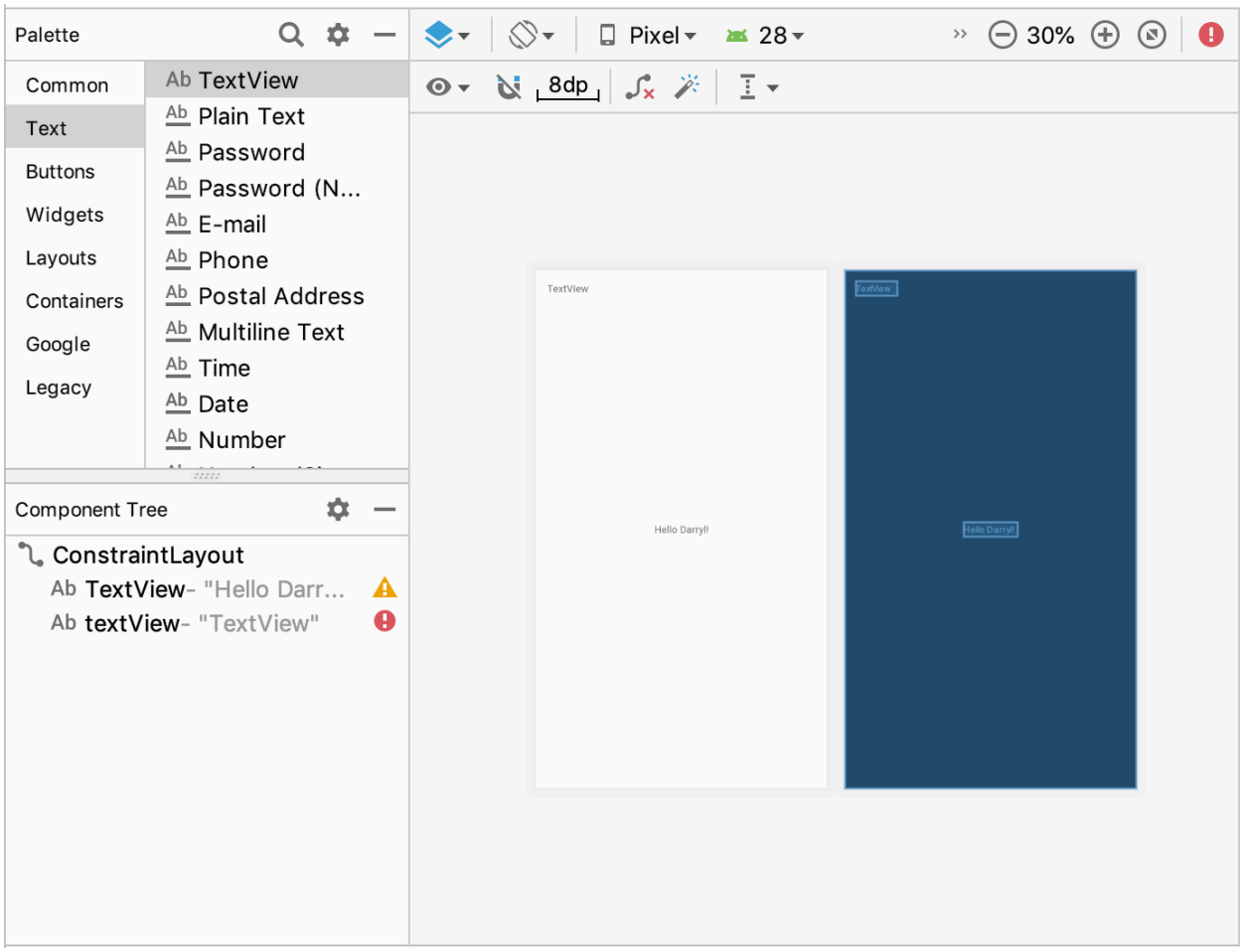


Palette of interface components

This contains all of the built-in user interface components that you can use to build the screens of your Android app. What's even more useful is that you can drag and drop from this palette directly into the Preview screen to add a component.

Open the **Palette** and select **Text**. The palette changes and shows everything text-related.

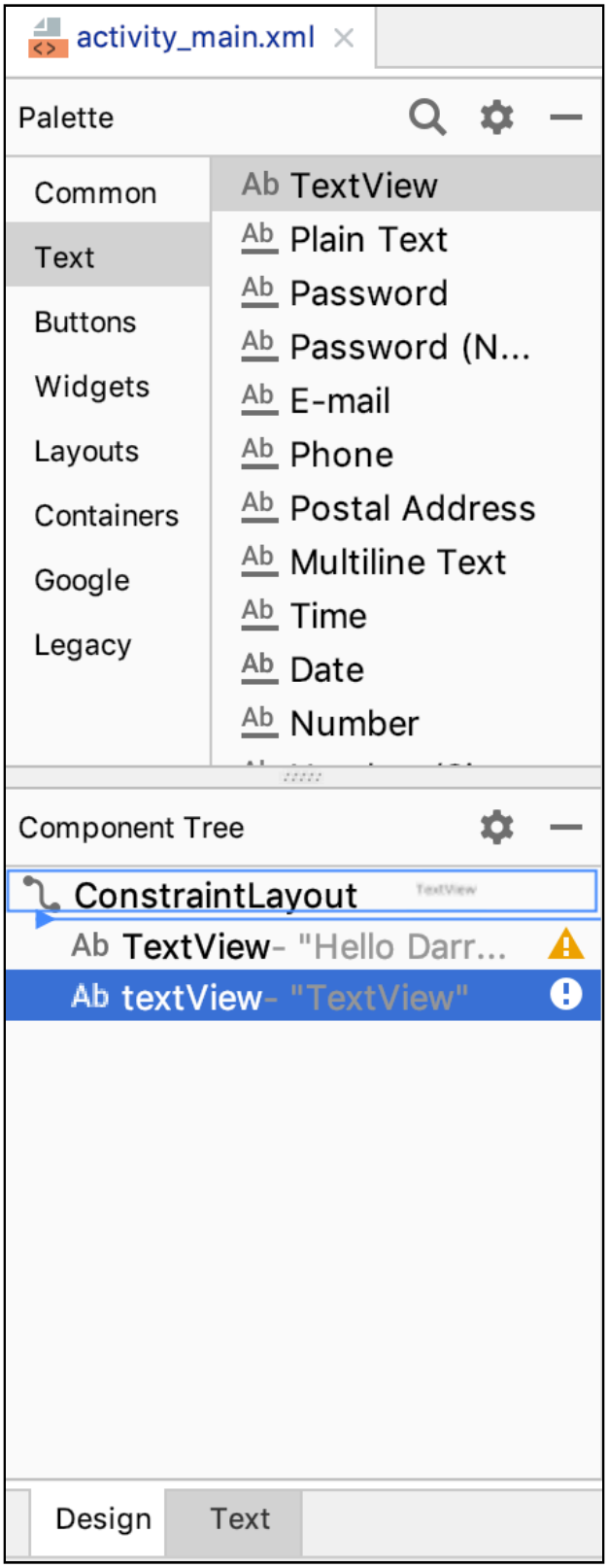
Next, drag a **TextView** from the Palette — this is for your score label — and drop it in the top-left of the Preview screen.



Component tree view

Before moving on, it's worth noting that although dragging and dropping components into the Preview area is relatively easy to do, it can be tricky to get things to show up in the right spot, especially when you're dealing with projects that have many views.

As an alternative, you can drag components from the Palette directly into a **Component Tree**, dropping it underneath the desired parent component.



Keep that little feather in your cap as you progress through this book, because you may find it easier to drop components in this way, and then deal with positioning them later.

Positioning your views

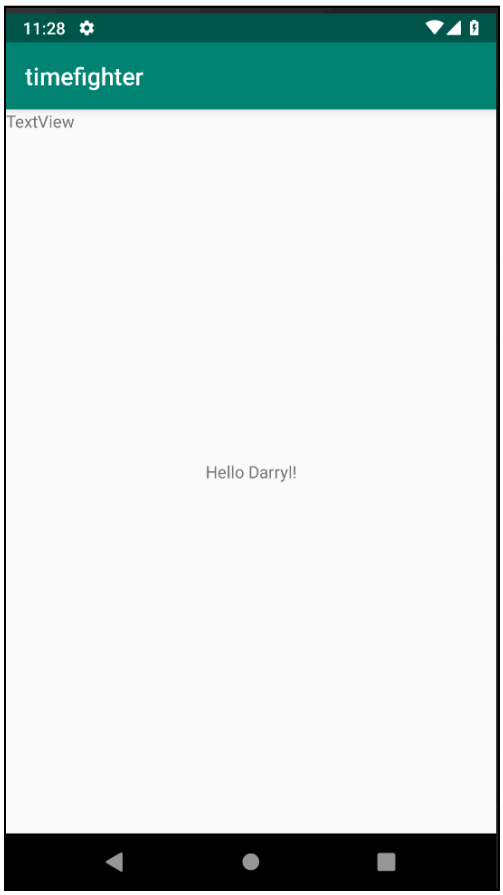
At this point, you have the start of your app, with your TextView sitting in the top left-hand corner. Come to think of it, how does the device know where to position the TextView? What happens if someone rotates the device?

As it stands now, the app doesn't know where to place the TextView — and you can prove it!

In the Visual editor, drag the newly placed TextView somewhere in the middle of the screen, like so:

Layout as seen in the Visual Editor

Click **Run 'app'** in the top-right of Android Studio and launch the emulator.



Layout as seen on device

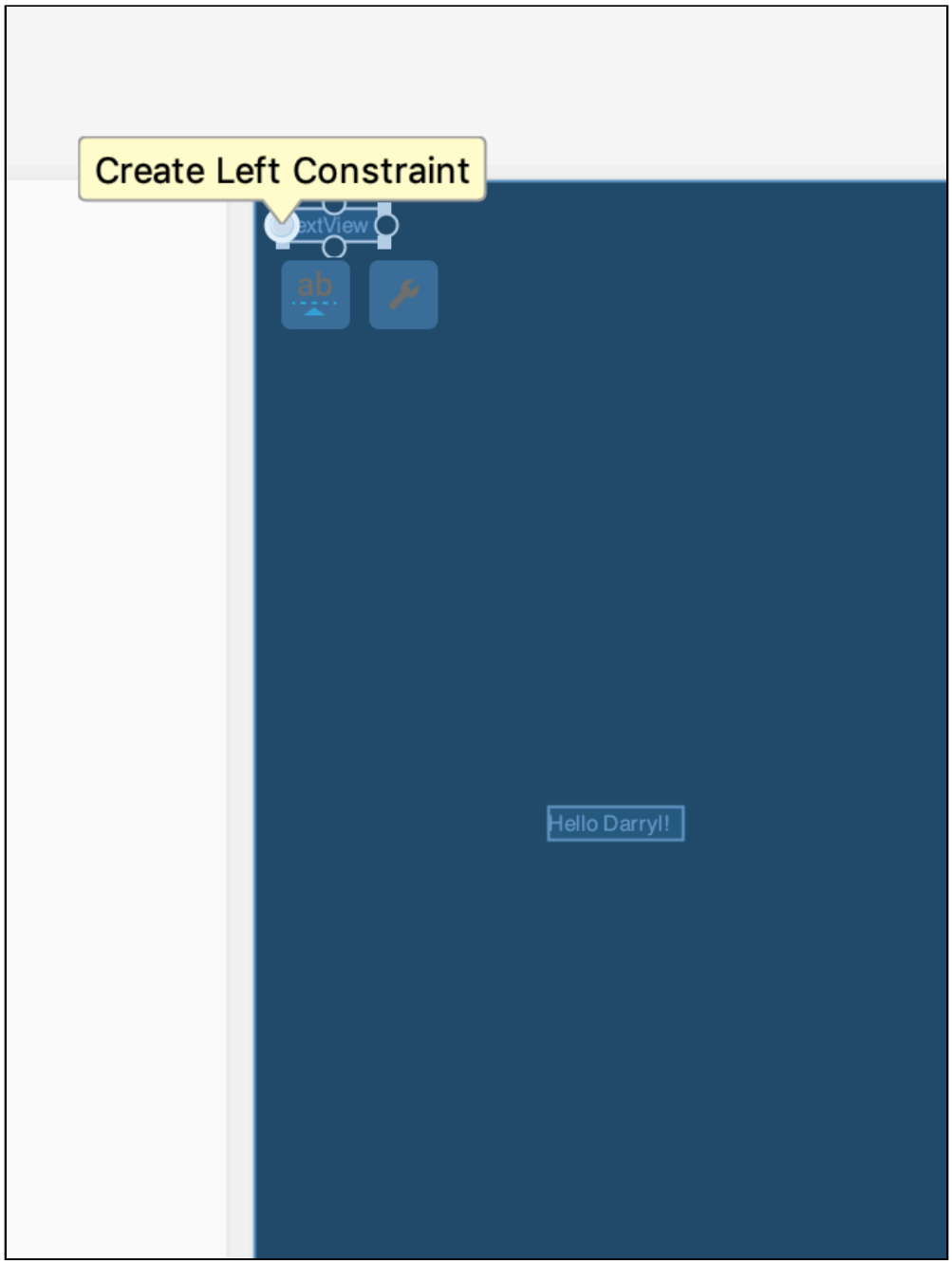
Hey, that's not where you placed the TextView! But don't worry; in the next section, you'll ensure the TextView stays put.

Adding rules to your position

There are millions of Android devices out there that come in all shapes and sizes. To ensure your app looks great on all of those different screens, you need to do a little Layout work and give the TextView some *rules* on where it should show up on the screen.

The Blueprint screen to the right of the preview gives you a visual representation of the rules that exist within your Layout. You'll use this tool to create new rules for your TextView.

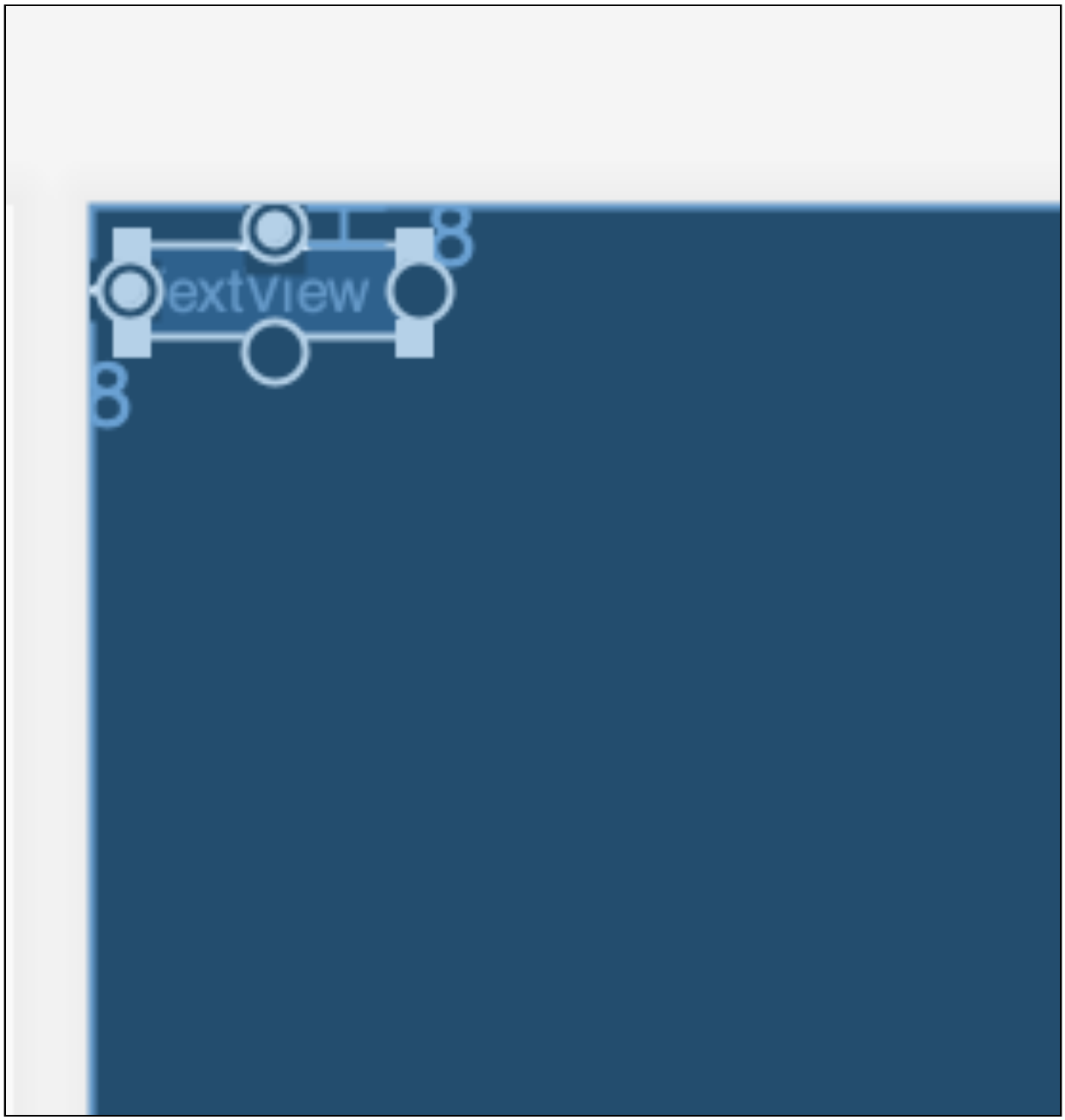
In the Preview screen, click and drag the TextView to the top-left corner of the screen. Now, hover your mouse over the left side of the newly placed TextView in the Blueprint screen. A circle with a white outline appears and a **Create Left Constraint** bubble pops up:



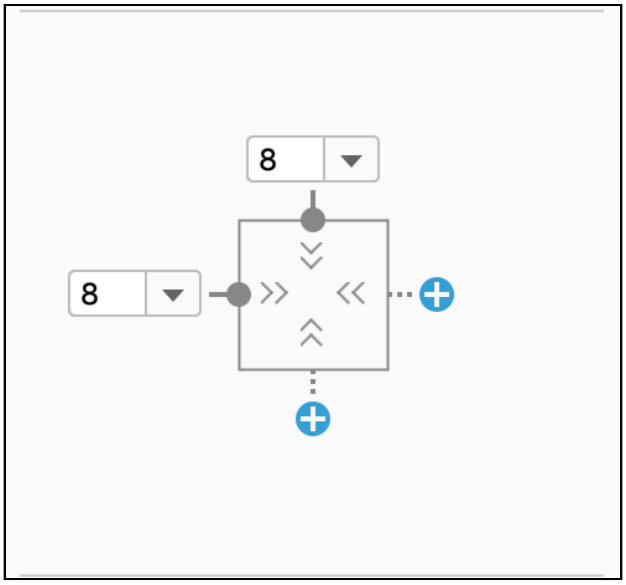
Click and drag toward the left edge of the Blueprint screen, and you'll see the TextView move slightly to the right. At this point, release the mouse button.

Congratulations, you just created your first layout rule!

Next, you need to create the top layout rule. Move your mouse to the top of the TextView until the outlined circle appears, and drag to the top edge of the screen until the TextView moves down slightly. Release the mouse button again to create the second layout rule.



To see what's happening, select the TextView and look for a panel on the right side of Android Studio in the **Properties** window. Look at the top of the Properties window, and you'll see a square with some chevrons inside:



Layout rule for your TextView

If you look closely, you'll see two solid lines running from the left and the top of the rectangle, pushing against two grey rectangles with a number **8** floating beside them. These are the rules, or **constraints**, you just created that hold your TextView against the edges of the screen, and they instruct the TextView how to position itself relative to the screen's edge.

If you want to position this TextView with greater control, you can adjust the margins of the constraint by clicking the number beside the constraint line and selecting one of the preset numbers in the drop-down or entering your own.

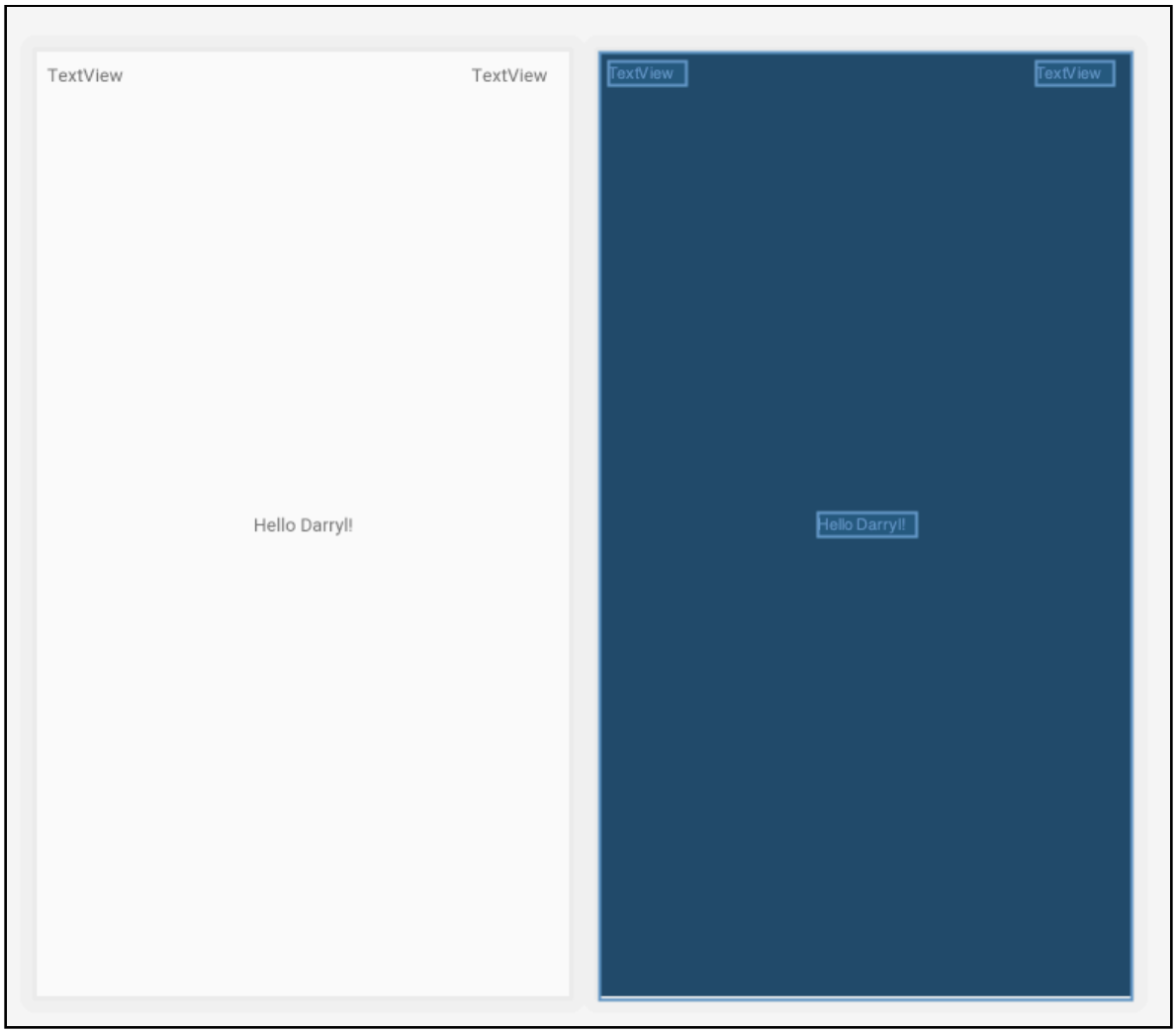
It's time to finish off the screen!

Finishing the screen

Go back to the Palette window and drag another TextView into the Preview window, this time putting it in the top-right corner of the screen to serve as your time remaining label.

In the Blueprint window, select the new TextView and hover over the right edge of it until the **Create Right Constraint** bubble appears. Create a new constraint against the right side of the screen. Now, do the same for the top of the TextView against the top of the screen.

You'll end up with something like this:



That takes care of the two TextViews. All that's left is the **Tap Me!** button.

First, remove that TextView floating in the middle of the screen. Select the **Hello User! TextView** and press the **delete** key — the TextView disappears.

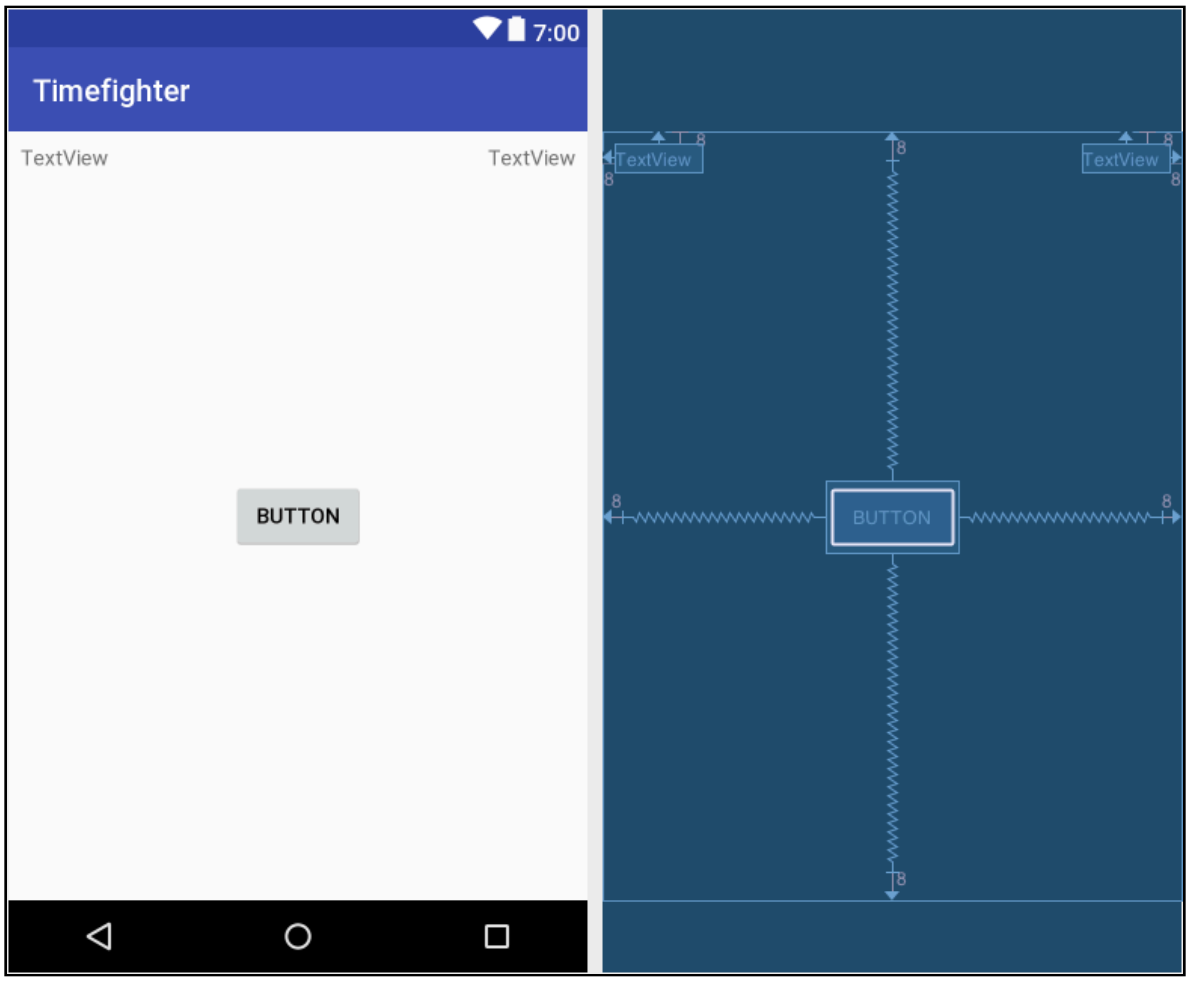
With the TextView removed, you can add the button.

Once again, in the Palette window, click the **Common** tab. When you see the **Button** in the Palette, click and drag it to the center of the screen. You may even see some helpful dotted guidelines to help position your Button right in the center of the screen.

Now, you need to create constraints for the Button, just like you did for the TextViews. This Button needs to stay in the center of the screen, so you need four constraints, one for each side.

In the Blueprint screen, hover over each side of the Button and pull the connector toward its respective edge of the screen. The Button will move around quite a bit as you do this but don't panic, it's just trying to respect the constraints as you add them.

Keep dragging each constraint to the edge of the screen.



Finally, click **Run 'app'** from the top menu in Android Studio. Your emulator or device loads the latest changes to your app, and all of your hard work is rewarded with an app that contains two correctly placed TextViews and one Button.

Where to go from here?

Although you learned a lot, you only used a fraction of the power that Constraints offer. There's a dedicated component for Constraints — **ConstraintLayout** — that provides all of this functionality.

Other Layouts provide other structures your Views can leverage, such as **LinearLayout** and **FrameLayout** among others. It's recommended to use a **ConstraintLayout** where possible. However, there are times where it might be awkward or not practical.

This book uses ConstraintLayout as its go-to Layout of choice. If you want to learn more about it, review the documentation on ConstraintLayout on the Android Developer: https://developer.android.com/training/constraint-layout/index.html.

Pat yourself on the back for making it this far! You've taken your first step into the world of Android development.

If you had any problems following along with the starter app, review the completed solution in the **final** folder for this chapter's materials.

In the next chapter, you'll attach some logic to your Button and make those TextViews display something more interesting than the words "TextView". You'll also get your first taste of writing code. See you there!

Chapter 3: Activities

By Darryl Bayliss

A lifestyle of various activity — like cardio, strength training and endurance — can keep you healthy. Although they're all different, they each have a specific purpose or goal in mind.

Android apps are similar — they're built around a set of screens. Each screen is known as an **Activity** and is built around a single task. For example, you might have a Settings screen where users can adjust the app's settings, or a Sign-in screen where users can log in with a username and password.

In this chapter, you'll start building an Activity focused around the gameplay for **TimeFighter** — and you'll finally get to lay down some code!

Getting started

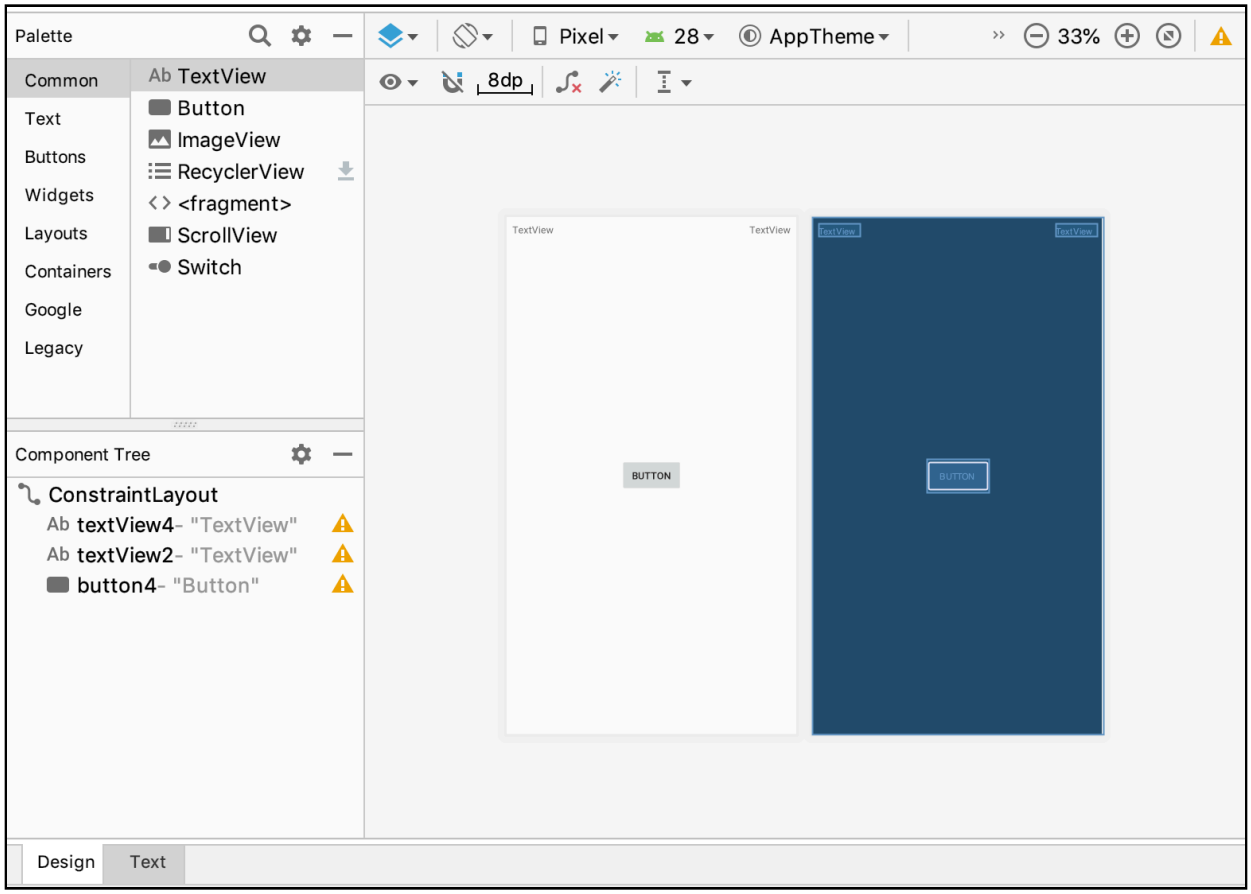
Before you jump head first into writing code, you first need to understand how IDs work. In Android, IDs play a fundamental role in connecting things, for example, connecting Views to your code.

In the previous chapter, you positioned Views and established that the top-left TextView will show the score, the top-right TextView will show the time and the Button, when pressed, will increment the score. Connecting your code to these Views will require each to have its own ID.

If you were following along making your app, open it and keep using it for this chapter. If not, don't worry — locate the **projects** folder for this chapter and open the **TimeFighter** app inside the **starter** folder.

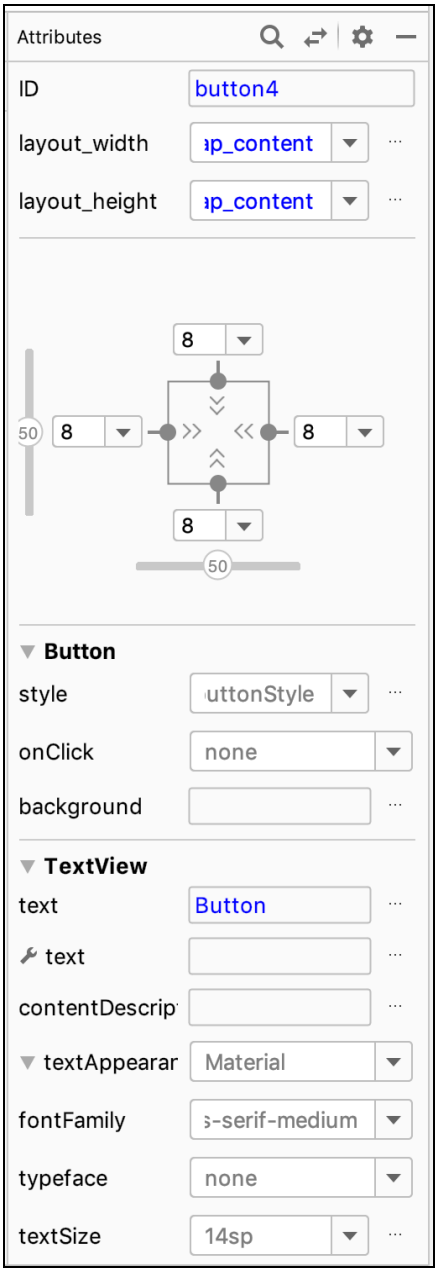
The first time you open the project, Android Studio takes a few minutes to set up your environment and update its dependencies.

Open **activity_main.xml** where you built your Layout and make sure you've selected **Visual editor**. Next to the Palette tab, you'll see a window called the **Component Tree**:



This window provides you with an overview of the Views available in your Layout and their relationship relative to one another.

In the Component Tree, click on the row labeled **button**, or **buttonX**, where **X** is a number. This action highlights the Button in the middle of the screen and updates the Properties window on the right with details about the Button.



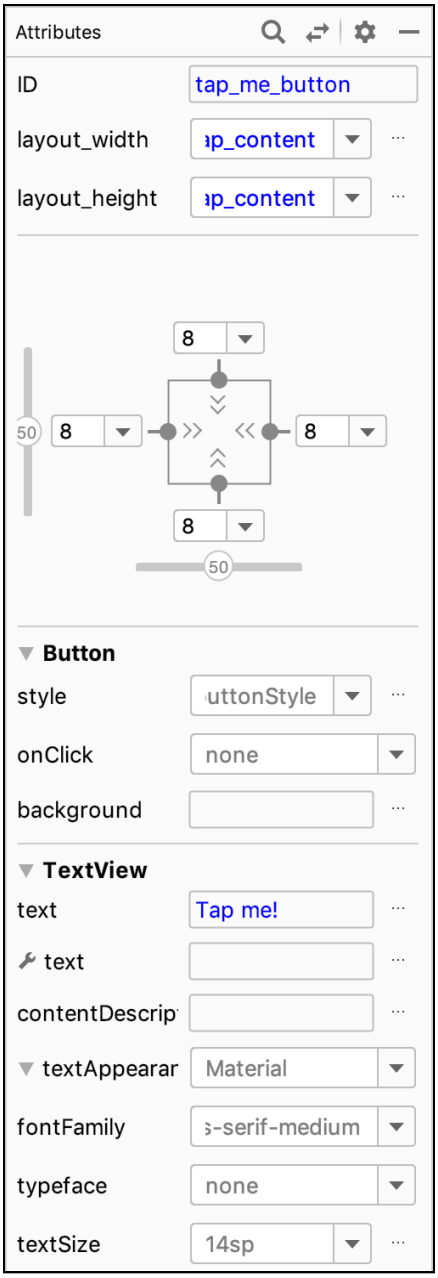
The Button in the screen above already has an ID of **button4**, but this isn't very descriptive.

> **Note**: In your project, it might have a different string value.

Theoretically, you could leave the ID as **button4**, but it's unlikely that in a year that'll mean anything to you. Using descriptive IDs makes it easier to know which IDs refers to which Views.

Change the value of the **ID** field from **button** to **tap_me_button**.

It's also a good idea to give the Button a more descriptive name too. Change the value of **text** in the TextView section of the Properties window to **Tap me!**



Select the TextView on the top-left from the Component Tree. Set its ID to **game_score_text_view** and change the text to **Your Score: %1$d**. Finally, select the

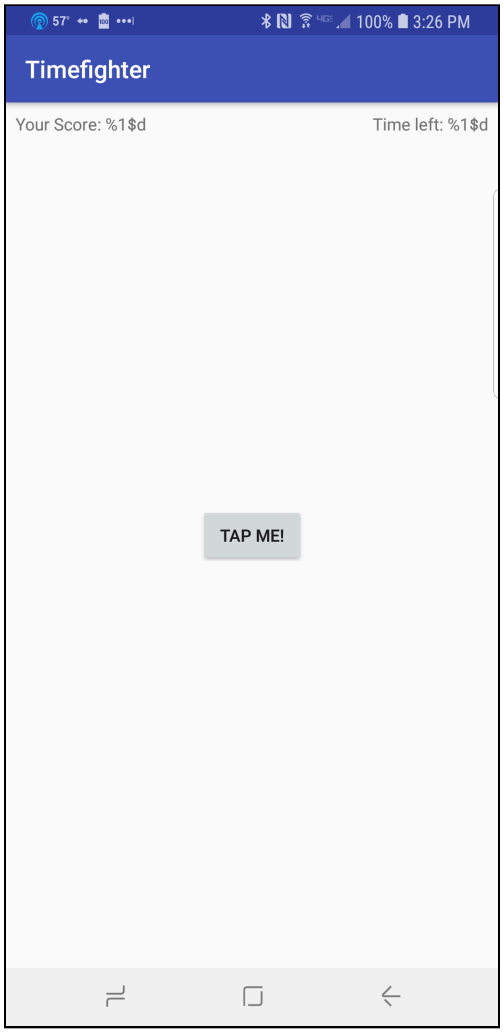
TextView you added to the top-right and change its ID to **time_left_text_view** and its text to **Time left: %1$d**.

So, what's the deal with the "%1$d"? That's a placeholder for any integer you want to inject into your text values. You'll fill in those placeholders later.

At build time, Android Studio takes these IDs and turns them into constants that your code can access through what's known as the **R** file.

You'll see more about R files in the upcoming sections, but for now, know that Android takes an ID such as **game_score_text_view** that you assigned to your View in your Layout and creates a constant named `R.id.game_score_text_view`, which you can then access in your code.

Run your app now in the emulator or on a device, and you'll see these text changes reflected on the screen:



Now that all of the Views in the project have IDs, you can finally start exploring and understanding your first Activity.

Exploring Activities

In the Project navigator on the left, ensure that the app folder is expanded. Navigate to **MainActivity.kt**, which is located in **src/main/java/com.raywenderlich.timefighter**. Open the file, and you'll see the following contents:

```
package com.raywenderlich.timefighter

import android.support.v7.app.AppCompatActivity
import android.os.Bundle

// 1
class MainActivity : AppCompatActivity() {
  // 2
  override fun onCreate(savedInstanceState: Bundle?) {
    // 3
    super.onCreate(savedInstanceState)
    // 4
    setContentView(R.layout.activity_main)
  }
}
```

This is where the logic for your game screen goes. Take a moment to explore what it does:

1. `MainActivity` is declared as extending `AppCompatActivity`. It's your first and only Activity in this app. What `AppCompatActivity` does isn't important right now; all you need to know is that subclassing it is required to deal with content on the screen.

2. `onCreate()` is the entry point to this Activity. It starts with the keyword `override`, meaning you'll have to provide a custom implementation from the base `AppCompatActivity` class.

3. Calling the base's implementation of `onCreate()` is not only important — it's required. You do this by calling `super.onCreate`. Android needs to set up a few things itself before your own implementation executes, so you notify the base class that it can do so at this point.

4. This line takes the Layout you created and puts it on your device screen by passing in the identifier for the Layout. Android Studio generates the identifier in the R file at build time using the Layout file name created in the previous chapter.

So, if you had a Layout named **really_good_looking_screen**, then the identifier generated would be `R.layout.really_good_looking_screen`.

These four lines are key ingredients in creating Activities in Android. You'll see them in every Activity you create. In the most general sense, any logic you add must come after calling `setContentView`.

> **Note**: `onCreate()` isn't the only entry point available for Activities, but it is the one you should be most familiar with. `onCreate()` also works in conjunction with other methods you can override that make up an Activity's *lifecycle*.
>
> This book covers a number of those lifecycle methods, but if you're curious, you can find out more at https://developer.android.com/guide/components/activities/activity-lifecycle.html.

Replace the entire contents of **MainActivity.kt** with the following skeleton:

```
package com.raywenderlich.timefighter

import android.os.Bundle
import android.os.CountDownTimer
import android.support.v7.app.AppCompatActivity
import android.widget.Button
import android.widget.TextView
import android.widget.Toast

class MainActivity : AppCompatActivity() {
  private lateinit var gameScoreTextView: TextView
  private lateinit var timeLeftTextView: TextView
  private lateinit var tapMeButton: Button

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    // connect views to variables
  }

  private fun incrementScore() {
    // increment score logic
  }

  private fun resetGame() {
    // reset game logic
  }

  private fun startGame() {
    // start game logic
  }

  private fun endGame() {
    // end game logic
```

```
    }
}
```

This contains a number of placeholder functions. You'll explore the purpose of each one in this chapter; however, this skeleton gives you an overview of the things you'll need to complete this app.

> **Note**: Sometimes, when using new objects in your classes, Android Studio won't recognize them until you import the class definition. This is shown by Android Studio highlighting the object in red.
>
> To import the class definition:
>
> • macOS: Click the object and press Alt-Return
>
> • Windows: Click the object and press Alt-Enter.
>
> You can also choose to let Android Studio handle imports automatically for you when pasting code.
>
> On macOS, select **Android Studio ▸ Preferences ▸ Editor ▸ General ▸ Auto Import** from the top menu. Set **Insert imports on paste** to **All**. Finally, tick the **Add unambiguous imports on the fly** checkbox.
>
> To do this on Windows or Linux, select **File ▸ Settings ▸ Editor ▸ Auto Import** from the top menu. Set **Insert imports on paste** to **All**. Finally, tick the **Add unambiguous imports on the fly** checkbox.

Hooking up Views

As an Android developer, one of the most common things your app will do is react to button clicks, and then convert those clicks into a change reflected in the app.

In **MainActivity**, you added three variables: `gameScoreTextView`, `timeLeftTextView` and `tapMeButton`. The first thing you need to do is attach these variables to the Views you added to the Layout.

In `onCreate(savedInstanceState: Bundle?)`, add the following code immediately after `setContentView`:

```
// 1
gameScoreTextView = findViewById(R.id.game_score_text_view)

timeLeftTextView = findViewById(R.id.time_left_text_view)
```

```
tapMeButton = findViewById(R.id.tap_me_button)

// 2
tapMeButton.setOnClickListener { incrementScore() }
```

Going through the code:

1. `findViewById` searches through the `activity_game` Layout to find the View with the corresponding ID and provides a reference to it you can store as a variable.
2. `setOnClickListener` attaches a click (or tap) listener to the Button which calls `incrementScore()`. You're instructing the Button to listen for a click; then whenever it's clicked, you increment the score.

You're nearly there now. Add a new variable to the top of `MainActivity` and initialize it to `0`:

```
private var score = 0
```

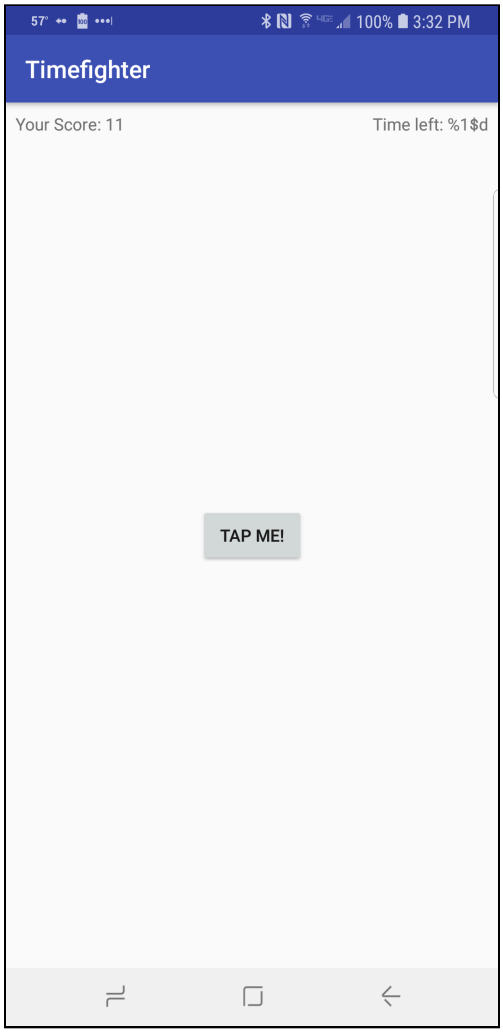
Next, replace the contents of `incrementScore()` with the following:

```
private fun incrementScore() {
  score++

  val newScore = "Your Score: $score"
  gameScoreTextView.text = newScore
}
```

You increment the new score variable to the next number and use that number in a string to use with your score text view.

Finally, you use `newScore` to set the text of `gameScoreTextView`.

Ready to see things in action? Run the app and tap the button a few times. The score in the top-left corner of the screen increments with each tap.



You just hit a milestone in your Android app development: You created a View, gave it an ID, accessed it in code and reacted to user input. These are the fundamental tasks of app development, and you'll repeat this cycle many times in your career. Take a moment to appreciate this significant accomplishment.

Managing strings in your app

You've gotten your first taste of writing code, you have something up and running resembling a game, and you undoubtedly want to take things further.

One of the most important elements of any app is the text, or strings, displayed on the screen. As you move ahead in your Android development career, you'll do well to master the ins and outs of using strings.

For instance, you're using English labels in your app, but that doesn't mean it's the only language your app can support. Supporting multiple languages in your app can often lead to broader markets, and it's a feature you should seriously consider when putting your app on the Google Play store.

In the previous section, you set `gameScoreTextView` to use the string `"Your Score: $score")`. This works well if you're only targeting English-speaking users. But how would you support one, two or even a dozen other languages?

The answer to this is **String resources**.

In the Project navigator, expand **res/values** and open **strings.xml**. You'll see a file with the following content:

```
<resources>
    <string name="app_name">Timefighter</string>
</resources>
```

strings.xml gives you a place to store all of the strings used in your app. This helps to keep strings from being sprinkled throughout your code. This also makes it easy to add support for another language. Rather than hunting through the entire project to change all of the strings, you copy the file and change it to hold the language translations of your choice.

You'll use this file to keep your English text in a separate location. Add the following lines under the `app_name` string:

```
<resources>
  <string name="app_name">Timefighter</string>
  <string name="tap_me">Tap me!</string>
  <string name="your_score">Your Score: %1$d</string>
  <string name="time_left">Time left: %1$d</string>
  <string name="game_over_message">Times up! Your score was: %1$d</
string>
</resources>
```

Now, go back to `incrementScore()` in **MainActivity.kt** and replace the contents of that method with the following:

```
private fun incrementScore() {
  score++

  val newScore = getString(R.string.your_score, score)
  gameScoreTextView.text = newScore
}
```

`getString` is an Activity-provided method that allows you to reference strings from the R file name or ID. In this case, you're retrieving the strings you added earlier to **strings.xml**. You also pass in an `int` for the placeholder `%1$d` you added way back at the beginning of this chapter.

> **Note**: To learn more about String Resources in Android, review the Android developer documentation at https://developer.android.com/guide/topics/resources/string-resource.html where you can also learn about string arrays and plurals.

Besides following the best practices for strings, your app is also ready for porting to another language. Sprinkling strings throughout your app is one of the worst types of technical debt to incur (Technical debt reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution).

With that out of the way, you can get back to developing your game.

Progressing the game

Currently, the game lets you increment the score infinitely. However, for a game named **TimeFighter**, there isn't much time fighting going on. In this section, you'll add a countdown timer that limits the amount of time you have to increase your score. `CountDownTimer` is an Android class that starts with a value in milliseconds and counts down until finished.

At the top of `MainActivity`, add the following new properties underneath the View properties:

```
private var gameStarted = false

private lateinit var countDownTimer: CountDownTimer
private var initialCountDown: Long = 60000
```

```
  private var countDownInterval: Long = 1000
  private var timeLeft = 60
```

Here, you declare new properties for your game: a `Boolean` property to indicate when the game has started, a countdown object named `countDownTimer` for you to race against, a count down interval variable to set the rate at which the countdown decrements and finally a variable to hold how many seconds are left in the countdown.

Finally, replace `resetGame()` with the following method:

```
  private fun resetGame() {
    // 1
    score = 0

    val initialScore = getString(R.string.your_score, score)
    gameScoreTextView.text = initialScore

    val initialTimeLeft = getString(R.string.time_left, 60)
    timeLeftTextView.text = initialTimeLeft

    // 2
    countDownTimer = object : CountDownTimer(initialCountDown,
countDownInterval) {
      // 3
      override fun onTick(millisUntilFinished: Long) {
        timeLeft = millisUntilFinished.toInt() / 1000

        val timeLeftString = getString(R.string.time_left, timeLeft)
        timeLeftTextView.text = timeLeftString
      }

      override fun onFinish() {
        // To Be Implemented Later
      }
    }

    // 4
    gameStarted = false
  }
```

Here, you initialize your game with a default state. You may have noticed when you first ran your game before there were oddities like symbols appearing next to the time left TextView or the score TextView before you started the game. This method ensures that your game always has a default state to begin.

Take a closer look:

1. You first set the score to 0, then create a variable to store the score as a string, using the `getString` method to insert the score value into your string stored in **strings.xml**. You then initialize `gameScoreTextView` with this value. You repeat the process for the amount of time left and assign it to `timeLeftTextView`.
2. You create a new **CountDownTimer** object and pass it into `initialCountDown` and `countDownInterval`, set to 60000 and 1000. The CountDownTimer object will count from 60000 milliseconds, or 60 seconds, in 1000 milliseconds, or 1 second, increments, until it hits zero.
3. Inside the **CountDownTimer** you have two overridden methods: `onTick` and `onFinish`. `onTick` is called at every interval you passed into the timer; in this case, once a second. Each interval, the `timeLeft` property is updated with the time remaining by converting the millisecond representation into seconds. You then update `timeLeftTextView` with this new time. You call `onFinish` when `CountDownTimer` has finished counting down. You'll add some code to this later.
4. You inform your `gameStarted` property that the game has not started by setting it to `false`.

The next step is to hook up `resetGame()` to run when you first create the Activity. You can do this in `onCreate()`.

Add the following line to the bottom of `onCreate()`:

```
resetGame()
```

Starting the game

Run your app. Things should look a little less jarring. The score TextView and time left TextView now show numbers instead of placeholders. Nice!

Click the Tap me button, and — no countdown! What is this madness?

Ah — you haven't told your countdown timer to begin counting down once the button has been clicked. Let's do that now. Replace `startGame()` with the following:

```
private fun startGame() {
  countDownTimer.start()
  gameStarted = true
}
```

You inform the countdown timer to start. You also set `gameStarted` to `true` to say the game has indeed started.

Finally, add the following lines to the top of `incrementScore()`:

```
if (!gameStarted) {
  startGame()
}
```

This code snippet checks to make sure the game has started when you tap the button. If not, then it starts the game for you.

Run the app to see what's changed.

Nice! Your countdown timer is now ticking merrily away.

Ending the game

Huzzah! T-Minus 60 seconds and counting to do — what exactly? The answer is "nothing" because the game doesn't know what to do after 60 seconds. Time to fix that!

In `MainActivity`, replace `endGame()` with the following code:

```
private fun endGame() {
  Toast.makeText(this, getString(R.string.game_over_message, score),
Toast.LENGTH_LONG).show()
  resetGame()
}
```

You make use of a **Toast** to notify something to the user. A Toast is a small alert that pops up briefly to inform you of some event that's occurred — in this case, the end of the game.

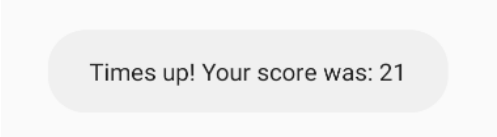
You pass into the Toast the Activity you want the Toast to appear on and the message to display. The end game state is a good time to display the score along with the game over message you put into **strings.xml**.

You also inform the Toast to display for a long time with `Toast.LENGTH_LONG`, which is a few seconds, and then show the Toast. Once that's done, you reset the game. You need to call `endGame()` from somewhere. The best time to call this is when `countDownTimer` finishes counting.

Head over to `resetGame()` and add the following line to the `onFinish` callback:

```
endGame()
```

Run your app again, and keep clicking the button. The countdown will continue to decrement until it hits 0. Once it does, you'll see the Toast with your score and game over message, at which point the game resets.



Where to go from here?

With a small amount of code, you created a functional game while learning some of the foundational elements of building an Android app. Although this Activity is small, they can get complicated as you add more Views. However, no matter its size, creating an Activity has the same flow:

1. Create a Layout for the Activity.
2. Give the Views in your Layout valid IDs.
3. Create properties in the Activity code and reference those IDs.
4. Manipulate your Views as needed or required.

If you find using `findViewById` cumbersome, you can leverage Kotlin to find your Views for you using the **Kotlin Android Extensions** (KAE) library. This library binds your Views to your code automatically, and provides many more benefits. You can learn how to use KAE over at: https://www.raywenderlich.com/84-kotlin-android-extensions.

Next, you'll learn how to fix potential problems in your app using Android debugging techniques.

Chapter 4: Debugging

By Darryl Bayliss

In the previous two chapters, you developed TimeFighter into a full-fledged app. In this chapter, you'll focus on debugging it.

All apps have bugs. Some are subtle, such as glitches within the UI, while others are obvious, such as outright crashes. As a developer, it's your job to keep your app bug-free.

Android Studio provides developers with some tools to help track down and fix bugs. In this chapter, you'll learn how to:

1. Debug your app using Android Studio's debug tools.
2. Add landscape support to TimeFighter.

Getting started

If you've been following along, open your project in Android Studio and keep using it for this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **TimeFighter** app inside the **starter** folder.

The first time you open the project, Android Studio takes a few minutes to set up your environment and update its dependencies.

You might not have noticed, but TimeFighter has a bug. Start the app in the emulator or on your device. Push **TAP ME** a few times, and then change the orientation of the device to landscape.

> Note: For Android Pie devices. You may need to enable auto-rotate on your device

or emulator if the screen doesn't rotate automatically.

To do this, swipe the notification drawer down to reveal the quick settings and ensure the auto-rotate button is colored green to signify it's enabled.

Notice anything strange? TimeFighter resets the game when you rotate the device. Whoops! To understand why this happens, you need to put on your debugging hat and analyze the code.

Add some logging

The first debugging approach is to add logging to your app. With logging, you can find out what's happening at certain potins within your code. You can even log and check the values of your variables at runtime.

In **MainActivity.kt**, add the following property to the top of the existing properties:

```
  // 1
private val TAG = MainActivity::class.java.simpleName
```

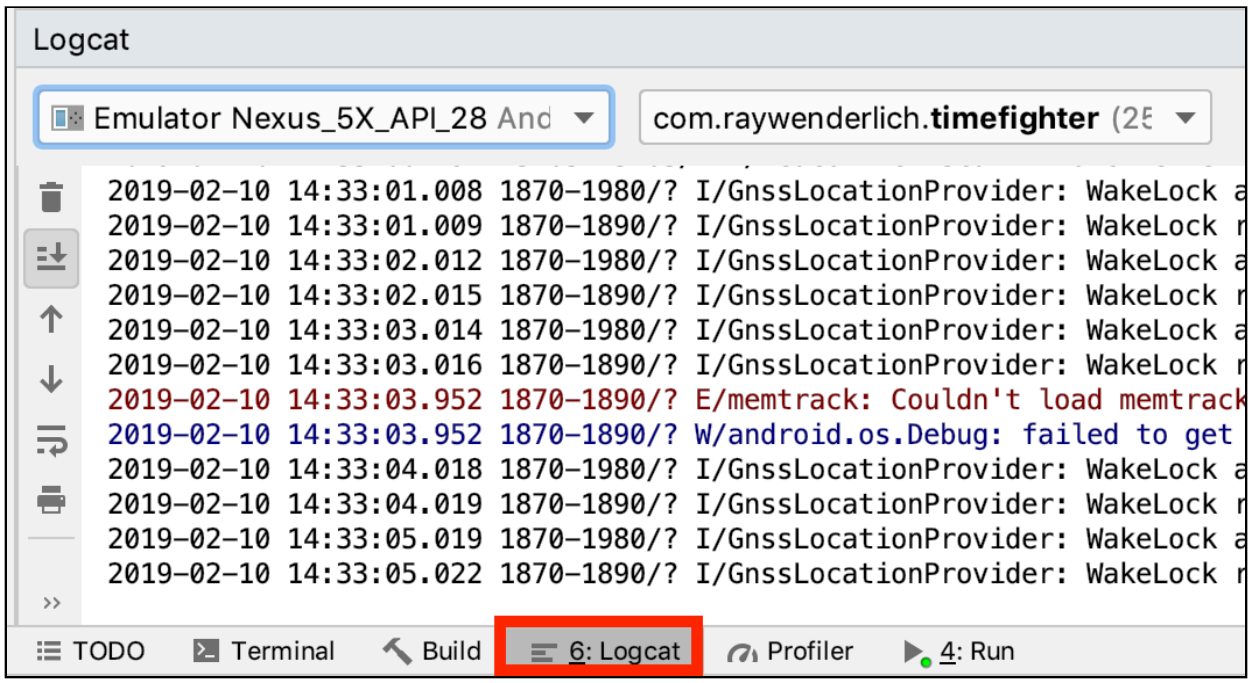
Then, add the following line below the call to `setContentView` in `onCreate()`:

```
  // 2
Log.d(TAG, "onCreate called. Score is: $score")
```

Having a look at the code:

1. You assign the name of your class to `TAG`. The convention is to use the class name in log messages. This makes it easier to see which class the message is coming from.
2. You `Log` a message when the Activity is created. Your app informs you when `onCreate()` is called and informs you of the current value in `score`. Injecting `$score` into the message is an example of **string interpolation** in Kotlin. At runtime, Kotlin looks for `score` and replaces it in the log message.

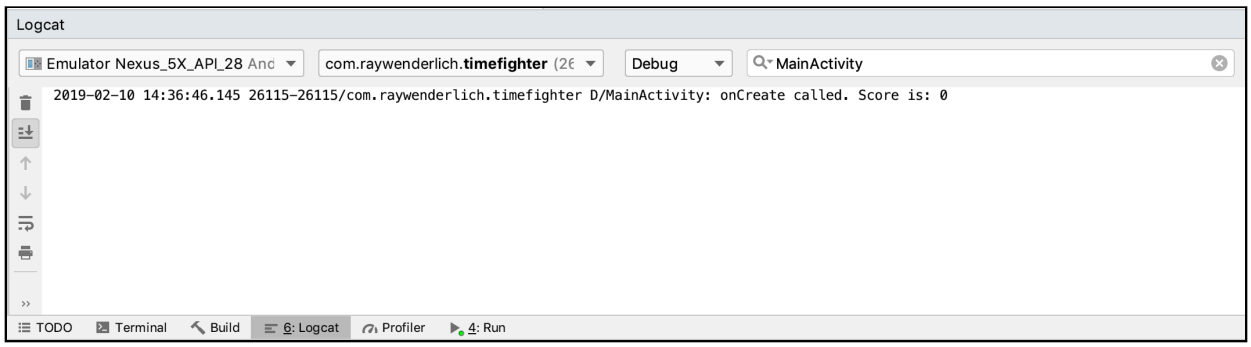
Run the app again. After it's loaded, go to Android Studio. At the bottom of the window there's a button labeled **Logcat**. Click that button, and Android Studio displays a console-like window at the bottom:



With Logcat, you can see everything your emulator or device is doing via log messages, including messages coming from outside of your app. For now, you can ignore most messages and filter down to only the ones you've added yourself.

In the Logcat window, there's a search bar with a magnifying glass. The text you enter here filters the log messages so that you'll only see log messages that match that text.

In the Logcat search bar, type the name of your Activity — **MainActivity** — and watch as the filter gets applied.



Excellent, you can now see the log messages you added earlier. The score is currently 0 because you haven't yet started the game.

Try to reproduce the bug by rotating the screen as you play the game.

That's strange! Why is the score reset to 0? You'll work that out in the next section.

> **Note**: You'll only scratch the surface of Logcat in this chapter. For more information about Logcat and everything it can do, read the Android developer documentation: https://developer.android.com/studio/command-line/logcat.html.

Orientation changes

From your log messages, you can establish that `score` is reset to 0 whenever you rotate the device. But why? The reason for this relates to how Android handles device orientation changes.

When Android detects a change in orientation, it does three things:

1. Attempts to save any properties for the Activity specified by the developer.
2. Destroys the Activity.
3. Recreates the Activity for the new orientation by calling `onCreate()`, which resets any properties specified by the developer.

But it's more than just orientation changes. Android performs these steps any time there's a change to the **configuration** of a device. A configuration change can happen for many reasons, including changes to the orientation or the selected device language. In fact, your Activity can get destroyed and recreated several times while the user is using the app, so it's incredibly important that you develop your app so it can recover from these changes.

Back in **MainActivity.kt**, add the following companion object at the bottom of the properties you declared earlier:

```
// 1
companion object {

  private const val SCORE_KEY = "SCORE_KEY"

  private const val TIME_LEFT_KEY = "TIME_LEFT_KEY"
}
```

Next, add the following methods below `onCreate()`:

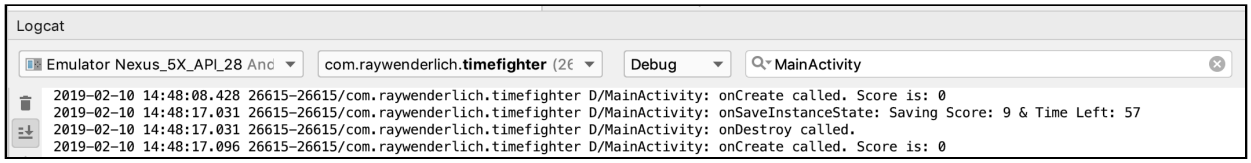
```
// 2
override fun onSaveInstanceState(outState: Bundle) {

  super.onSaveInstanceState(outState)

  outState.putInt(SCORE_KEY, score)
  outState.putInt(TIME_LEFT_KEY, timeLeft)
  countDownTimer.cancel()

  Log.d(TAG, "onSaveInstanceState: Saving Score: $score & Time Left:
$timeLeft")
}

// 3
override fun onDestroy() {
  super.onDestroy()

  Log.d(TAG, "onDestroy called.")
}
```

Here's what's happening:

1. You create a companion object containing two string constants, `SCORE_KEY` and `TIME_LEFT_KEY`, to track the variables you want to save when the orientation changes. You'll use these constants as keys into a dictionary of saved properties.
2. You override `onSaveInstanceState` and insert the values of `score` and `timeLeft` into the passed-in `Bundle`, which is a hashmap Android uses to pass values across different screens. You also cancel the game timer and add a log to track when the method is called.

3. You override `onDestroy()`, a method used by the Activity to clean itself up when it is being destroyed. You call `super` so your Activity can perform any essential cleanup, and you add a final log to track when `onDestroy()` is called.

Run your app again. Play the game for a few seconds, change the orientation, and then look at the Logcat output:



The Activity is still resetting the score back to 0. However, the log statement in `onSaveInstanceState()` is informing you that the score and the amount of time left are saved. How can you verify this? Breakpoints!

Breakpoints

Logging is an effective way of understanding what your app is doing, but it can be tedious to write a log message, recompile, rerun your app and attempt to reproduce the bug. But don't worry, there's another way!

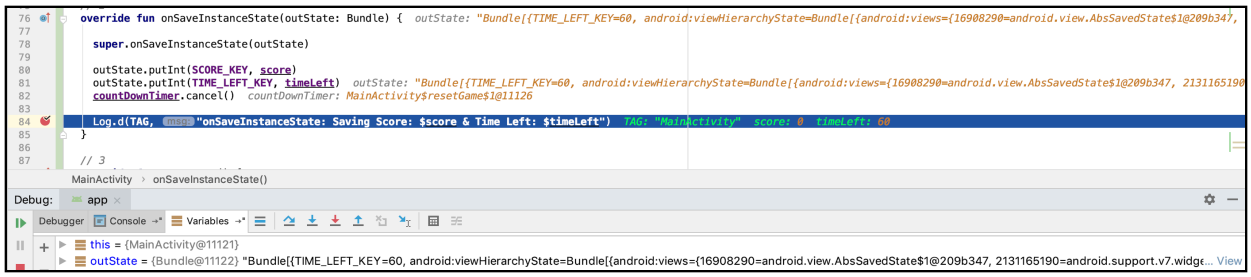
Android Studio provides **breakpoints**. With breakpoints, you can pause the execution of your app to inspect its current state.

In **MainActivity.kt**, scroll to `onSaveInstanceState()` and find the log line at the bottom of the function. Click on the grey border (also known as the gutter) to the left of the line.

This adds a red dot to the gutter to indicate where the breakpoint will trigger. Click the **Debug** button at the top of the window, it looks like a green bug.

The app loads in the same way it did when using the run button, except this time, it attaches the debugger.

Once the app reloads, rotate the screen. Android Studio changes windows and highlights the breakpoint.



Your app is paused at the line that has the breakpoint. In this case, it's the log message you added earlier where you save the game variables to a Bundle.

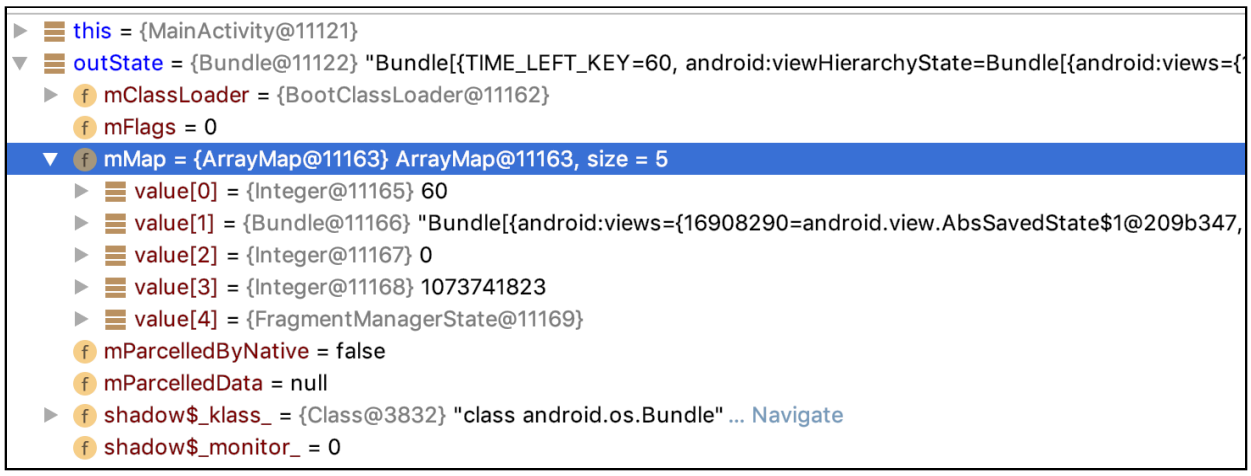
When Android Studio hits a breakpoint, it gives you the opportunity to inspect your app's state at that exact moment in time. You can see this information in the **Debug** window below your code. Move to the debugger view and click the arrow next to `this = {MainActivity}`.

The number postfixing your MainActivity is likely different since this number indicates where your Activity is allocated in memory.

You might recognize some of the values as your own. However, there are also other values that may be unfamiliar to you. These are values specific to an Activity and give you an appreciation of how much work the `Activity` class does behind the scenes.

Also, when Android Studio hits a breakpoint, it inlines some debugging information within your code, which makes it even easier to inspect things.

Time to put this knowledge to use. Close `this` in the debugger, expand `outState`, and then expand `mMap`. You'll see some familiar values.



Compare those numbers with the values of `score` and `timeLeft` — they should match. This informs you that those values are now safely stored in the Bundle. In the next section, you'll see how to restore those numbers when the device orientation changes.

Restarting the game

So far, you've only used `onCreate()` to set up your Activity. You've yet to use the `savedInstanceState` object passed in as a parameter. Are you ready?

Inside `onCreate()`, replace the call to `resetGame()` with the following:

```
if (savedInstanceState != null) {
  score = savedInstanceState.getInt(SCORE_KEY)
  timeLeft = savedInstanceState.getInt(TIME_LEFT_KEY)
  restoreGame()
} else {
  resetGame()
}
```

Here, you check to see if `savedInstanceState` contains a value. If it does, you attempt to get the values of `score` and `timeLeft` from the Bundle that you passed in earlier from `onSaveInstanceState`. You then assign those values to the properties and restore the game. If, however, `savedInstanceState` does not contain a value, you reset the game.

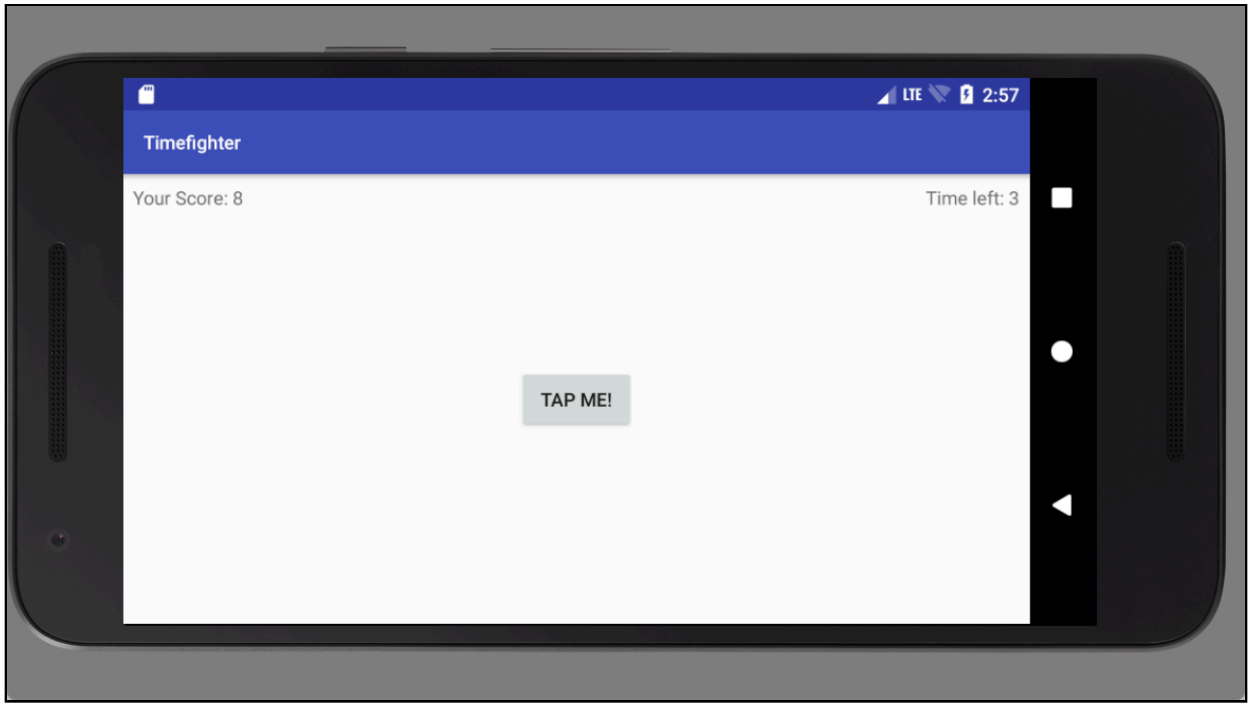
Next, implement the following method below `resetGame()`:

```
  private fun restoreGame() {

    val restoredScore = getString(R.string.your_score,
Integer.toString(score))
```

```
    gameScoreTextView.text = restoredScore

    val restoredTime = getString(R.string.time_left,
Integer.toString(timeLeft))
    timeLeftTextView.text = restoredTime

    countDownTimer = object : CountDownTimer((timeLeft * 1000).toLong(),
countDownInterval) {
      override fun onTick(millisUntilFinished: Long) {

        timeLeft = millisUntilFinished.toInt() / 1000

        val timeLeftString = getString(R.string.time_left,
Integer.toString(timeLeft))
        timeLeftTextView.text = timeLeftString
      }

      override fun onFinish() {
        endGame()
      }
    }

  countDownTimer.start()
  gameStarted = true
}
```

`restoreGame()` sets up the TextViews and `countDownTimer` properties using the values inserted into the Bundle before the change in orientation.

Run the app and play the game for a few seconds. Then, rotate the device to see what happens:



Woohoo! The score and time remaining stayed the same — bug fixed.

Where to go from here?

You only scratched the surface of debugging in Android Studio. Finding and fixing bugs is an important part of software development, so it's essential that you get comfortable with the tools.

Android Studio contains many debugging tools that are beyond the scope of this chapter. To find out more, read the Android developer documentation: https://developer.android.com/studio/debug/index.html.

> **Note**: Sometimes you aren't able to fix bugs due to factors beyond your control. There may be bugs in a third-party library you're using, or maybe even within Android itself. If you find yourself in this situation, inform the developers who maintain that code via their bug reporting channels.

For now, you're armed with enough tools and techniques to debug potential problems in your own apps. In the next chapter, you'll finish up TimeFighter so that it looks and feels more in place in the Android ecosystem.

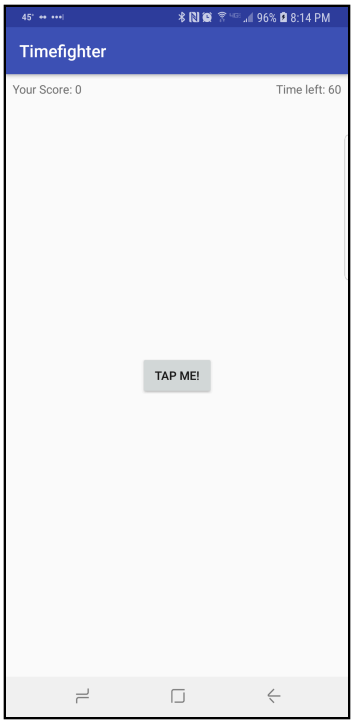
Chapter 5: Prettifying the App

By Darryl Bayliss

Take a moment to congratulate yourself and recognize what you've accomplished so far: You have a working Android app that lets users fight the clock and score as many points as possible.

You also fixed a few undiscovered bugs and added support for portrait and landscape mode, regardless of their device. By all accounts, your app is ready to entertain people for years to come!

There's one problem though: It's not *visually exciting*.



Nothing special here...

An app that looks visually appealing tends to stick out when compared to similar apps. While it's not integral to the functionality of your app, it *does* give it that "wow!" factor.

In this final chapter for the section, you'll learn how to:

1. Adjust your app to adhere to the Material Design Guidelines.
2. Add small touches to give your app a polished look and feel.
3. Add a simple animation to your app to give it some life.

Getting started

If you've been following along, open your project and keep using it for this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **TimeFighter** app inside the **starter** folder.

The first time you open the project, Android Studio takes a few minutes to set up your environment and update dependencies.

With TimeFighter open, run the app and consider some things you can do to improve the way it looks and feels. Perhaps you can change the color of the app bar or the white screen? Maybe the button feels a little lifeless when tapped? And why is the app so silent — maybe it needs sound effects?

The important thing to remember is that you don't need to do *everything*. You only need to make changes that add to the essential elements on the screen. If you add too much, you run the risk of cluttering the screen and confusing the user.

Changing the app bar color

In the Project navigator, on the left side of Android Studio, open **colors.xml**; it's located in **app > res > values**. You'll see something like this:

```
<resources>
  <color name="colorPrimary">#3F51B5</color>
  <color name="colorPrimaryDark">#303F9F</color>
  <color name="colorAccent">#FF4081</color>
</resources>
```

colors.xml stores the color values used in your app. Like **strings.xml**, it's an excellent place to store the color-related values and keep them in one location, which makes it easier to change things later.

To define colors, you use a `<color>` tag along with a `name` attribute that you can use as a reference when it's compiled into **R.java**. The reference is available for use in your XML Layouts and also at runtime in your code.

Within `<color>`, you assign a hexadecimal representation of the color. You close the tag using `</color>`.

With the theory out of the way, you're ready to update the file. In **colors.xml**, change the values to match the following:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="colorPrimary">#0C572A</color>
  <color name="colorPrimaryDark">#388E3C</color>
  <color name="colorAccent">#8BC34A</color>
  <color name="colorBackground">#D3D3D3</color>
</resources>
```

Are you wondering how this changes the color of the app bar at the top? The answer lies in **styles.xml**. This file is located in **app > res > values**.

Open **styles.xml** and review its contents:

```
<resources>
  <!-- Base application theme. -->
  <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
    <!-- Customize your theme here. -->
    <item name="colorPrimary">@color/colorPrimary</item>
    <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
    <item name="colorAccent">@color/colorAccent</item>
  </style>
 </resources>
```

Notice the `<item>` tags. These tags define specific items within your app which adhere to a particular color. In this case, these colors are the colors you updated in **colors.xml**.

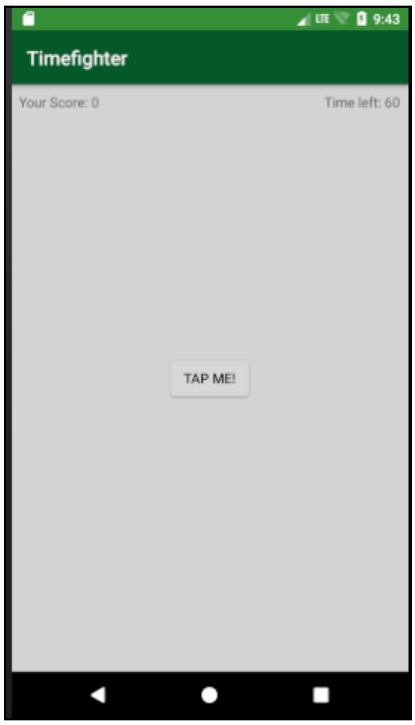
> **Note**: Your app is adhering to a **Style** set within this file. This is used to set the presentation of Views and screens. You can override items that are inherited from other themes provided by Android or other developers. For more information, visit: https://developer.android.com/guide/topics/ui/themes.html.

One final tweak! Open **activity_main.xml**, located in **app > res > layout**. Switch from **Design** to **Text**, and update `ConstraintLayout` to change the color of the background, like so:

```
<android.support.constraint.ConstraintLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:background="@color/colorBackground"
 tools:context="com.raywenderlich.timefighter.MainActivity">
```

You now reference the `colorBackground` color added in **colors.xml**.

With that done, run the app and see if you can still recognize it.



That's more interesting!

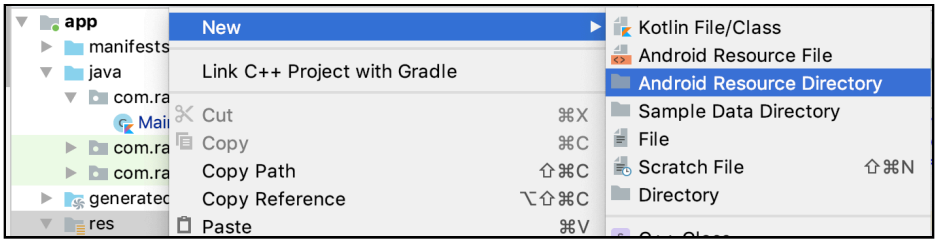
With a few lines of code, you've managed to transform the app and make it more visually appealing.

Animations

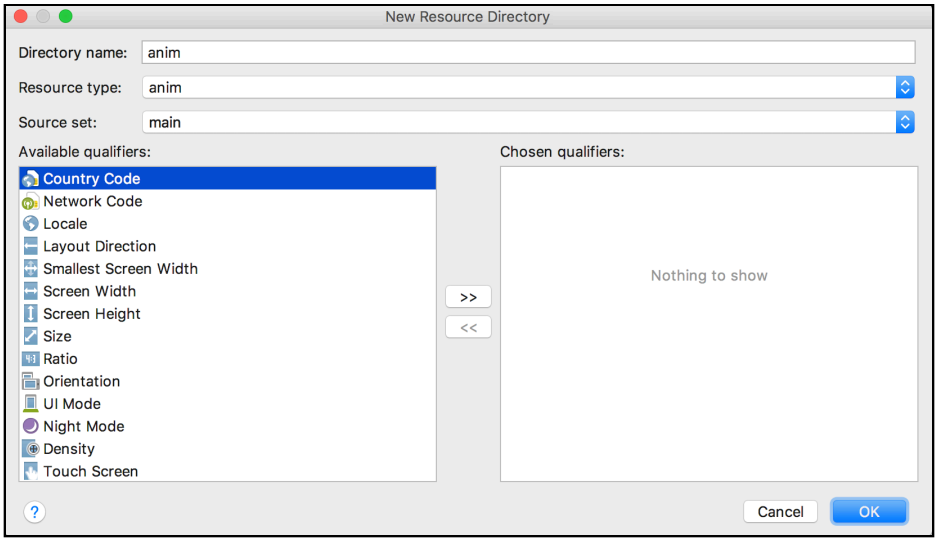
Animations give visual emphasis to elements and help direct the users' attention. When it comes to animation, the most important rule is to use it where and when it matters — not simply because you *can*.

One of the most heavily-used components in TimeFighter is the "Hit Me" button — because that's what earns the user points. So, adding an animation here makes sense!

In the Project navigator, right-click on **res**. In the drop-down window, navigate to **New** and click **Android resource directory**.



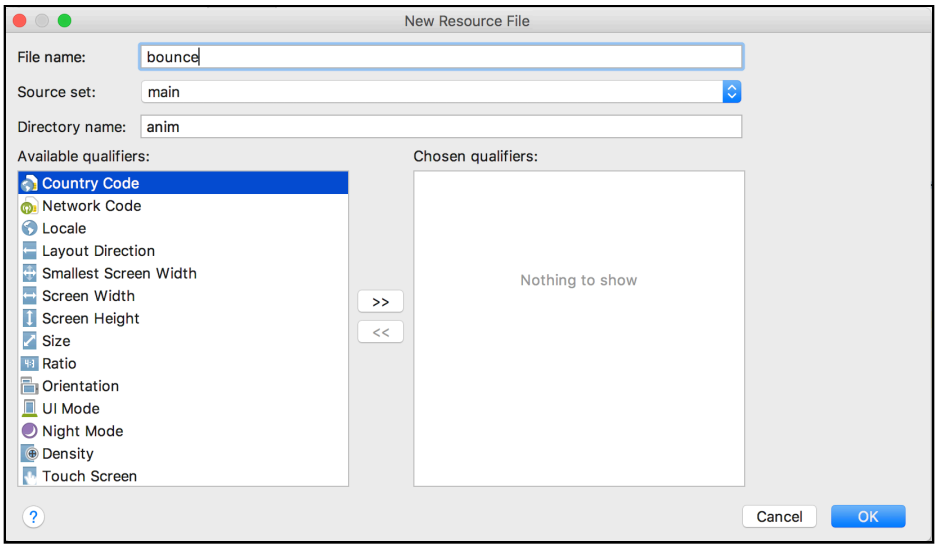
In the **New Resource Directory** window, click the drop-down button next to **Resource type** and change it to **anim** — which automatically changes the name of the directory — and click **OK**.



In the Project navigator, you now have a new folder inside **res** named **anim**.

Next, you need to create the file defining the animation for your button. Right-click on **anim**, navigate to **New**, and click **Animation resource file** on the right-most drop-down.

You're presented with a window similar to the one you saw when creating the **anim** folder. This time though, you need to enter the name of the file. For the **File name**, enter **bounce**, then click **OK**.

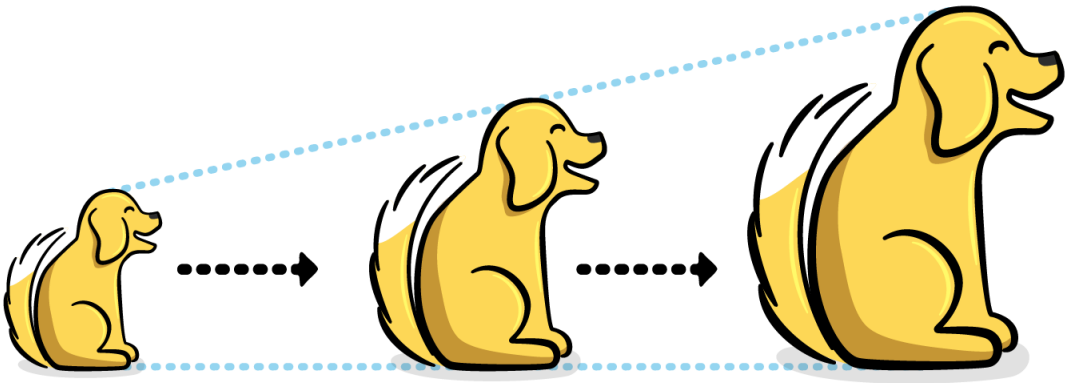


Android Studio creates the file and automatically opens it for you:

```
<?xml version="1.0" encoding="utf-8"?>
  <set xmlns:android="http://schemas.android.com/apk/res/android">

</set>
```

Notice the **set** attribute. This is a container that holds all of the transformations that occur throughout your animation. You can bundle more than one transformation in the same animation and have them all run concurrently.

Think of a transformation as something that happens over time. Imagine a dog moving from the left of the screen to the right: As the dog walks along the screen, his position changes; he may even grow larger as he moves.



This dog is performing two transformations. Moving from left to right and also growing in size!

For this animation, however, you only need one transformation.

Edit **bounce.xml** to match the following:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
  android:fillAfter="true"
  android:interpolator="@android:anim/bounce_interpolator">
  <scale
    android:duration="2000"
    android:fromXScale="2.0"
    android:fromYScale="2.0"
    android:pivotX="50%"
    android:pivotY="50%"
    android:toXScale="1.0"
    android:toYScale="1.0" />
</set>
```

Stepping through the XML to see what's happening:

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
  android:fillAfter="true"
  android:interpolator="@android:anim/bounce_interpolator">
```

The `set` is declared and `fillAfter` is set to `true`, which means the animation won't reset the View to its original position once it's complete. Instead, the View remains wherever it is when the animation ends.

This `set` is also told to use the `bounce_interpolator` from Android. Interpolators affect the rate the entire animation is performed over time, independent of durations set within the transformations.

Android provides many built-in interpolators. It also lets you create your own if you don't find one that suits your needs. For now, the `bounce_interpolator` included with Android works nicely.

Time for the next few lines:

```
<scale
  android:duration="2000"
  android:fromXScale="2.0"
  android:fromYScale="2.0"
  android:pivotX="50%"
  android:pivotY="50%"
  android:toXScale="1.0"
  android:toYScale="1.0" />
```

You declare a `scale` attribute. This informs the animation to resize the View. You also declare that scaling occurs over 2000 milliseconds (2 seconds), via the `duration` attribute.

In addition, you set the width and height of the View as `2.0`, twice the original size when the animation starts via the `fromXScale` and `fromYScale` attributes.

The `pivotX` and `pivotY` attributes specify the center point where the animation occurs. In this case, it occurs from the center of the View, expressed in percentages as `50%`: halfway across the X-axis and halfway across the Y-axis.

Finally, you set the size of the View at the end of the animation as `1.0`, via the `toXScale` and `toYScale` attributes. This sets the View back to its original size.

In summary, the animation you defined will:

- Scale the animated View to twice its size.
- Shrink it back to its original size.

- Do this over the space of two seconds.
- Using a bouncing interpolator.

> **Note**: If you want to know more about animation resources and interpolators on Android, review the Android Developer documentation (https://developer.android.com/guide/topics/resources/animation-resource.html) for an in-depth review.

That's it for the bounce animation!

Open **MainActivity.kt** and modify the `tapMeButton.setOnClickListener` callback in `onCreate()` to use this animation:

```
tapMeButton.setOnClickListener { v ->
  val bounceAnimation = AnimationUtils.loadAnimation(this,
      R.anim.bounce);
  v.startAnimation(bounceAnimation)
  incrementScore()
}
```

Every time you click the button, `tapMeButton.setOnClickListener` loads the bounce animation inside **anim** and instructs the button to use that animation. Run the app and click the button.

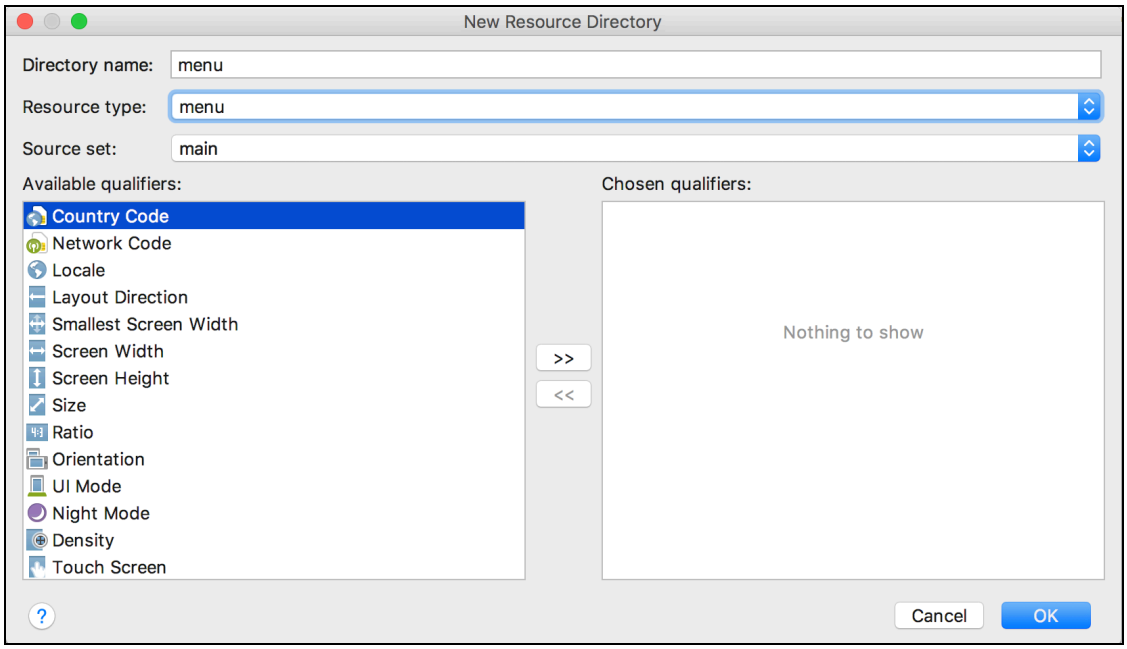
Adding a Dialog

Making an app if fun, and at some point you need to let the users know who created it. But at the same time, you don't want to distract your users while they're playing your game. So what can you do? One option is to use a **Dialog**.

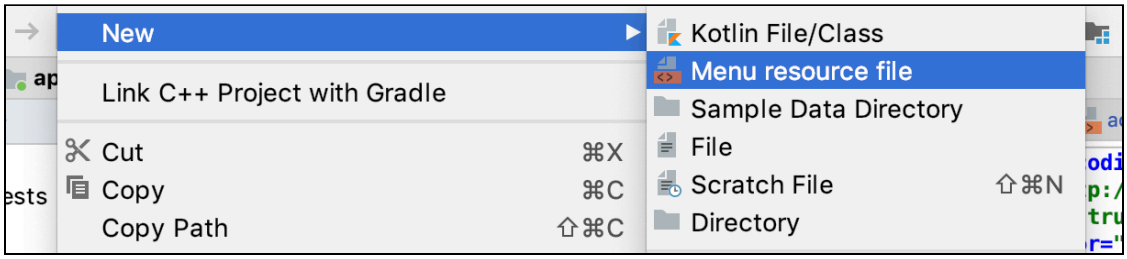
A Dialog is an excellent way to provide a snippet of information without moving away from the main content on the screen. Dialogs come in all shapes and sizes, but in this case, you want to let your users know about the creator of the app and what version of TimeFighter they're running.

An easy way to do that is to set up a button in the top bar. But first, you need to define a menu.

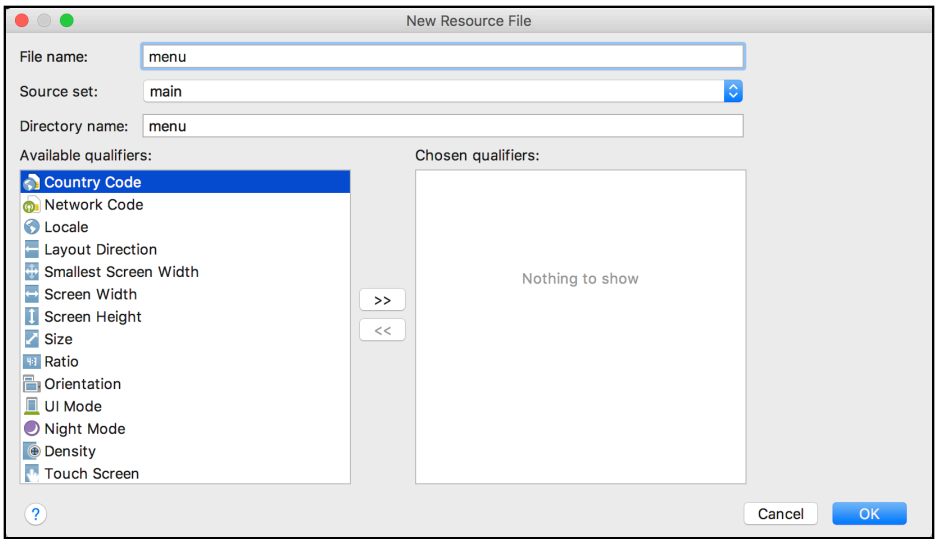
In the Project navigator, locate **res**. Right-click on the folder and select **Android resource directory**. In the new window, click the resource type drop-down and change it to **menu**. Then, click **OK**.



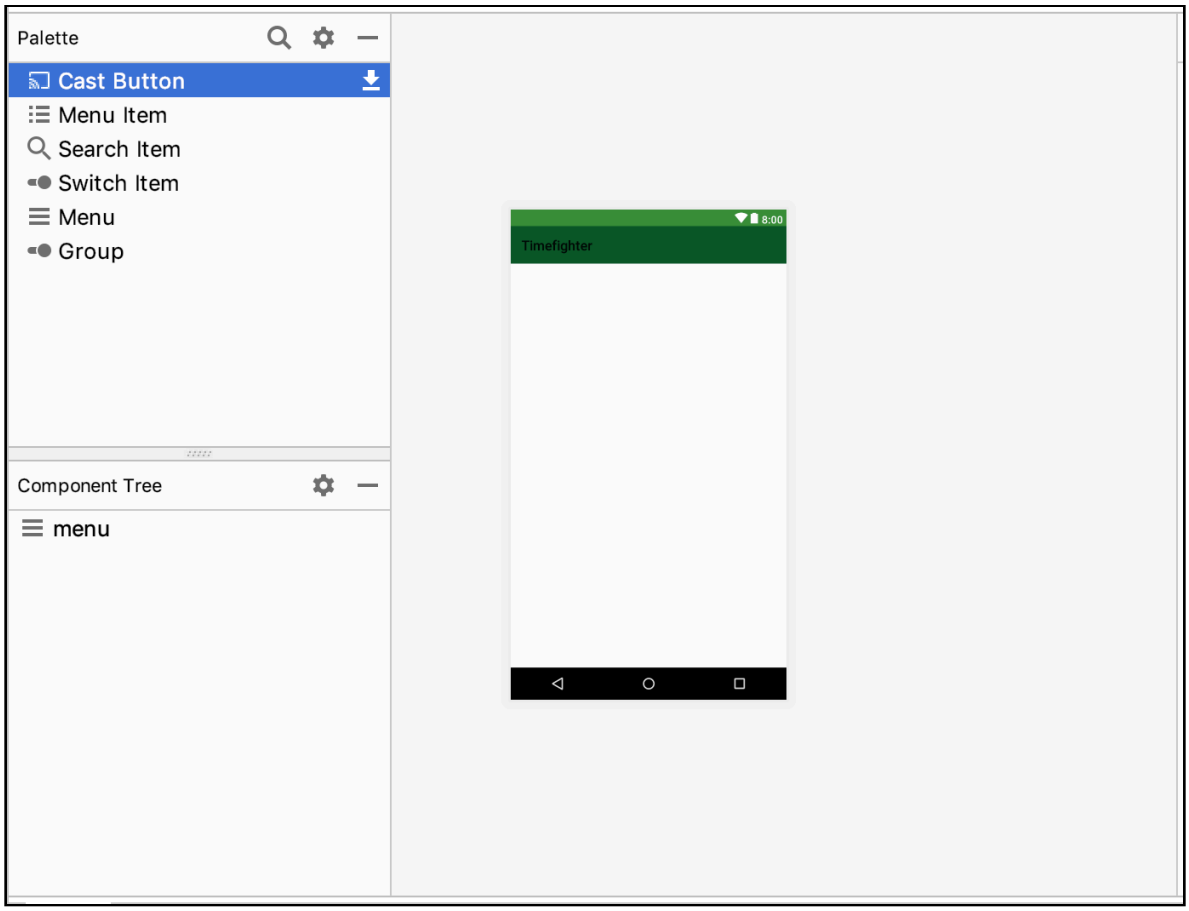
Right-click on the newly created **menu** resource folder. In the pop-up menu, hover over **New**, and then click on **Menu resource file**.



In the **New Resource File** window, enter the file name as **menu** and click **OK**:



Android Studio changes over to the Layout window and shows you a similar setup to what you've seen when editing Layout files:

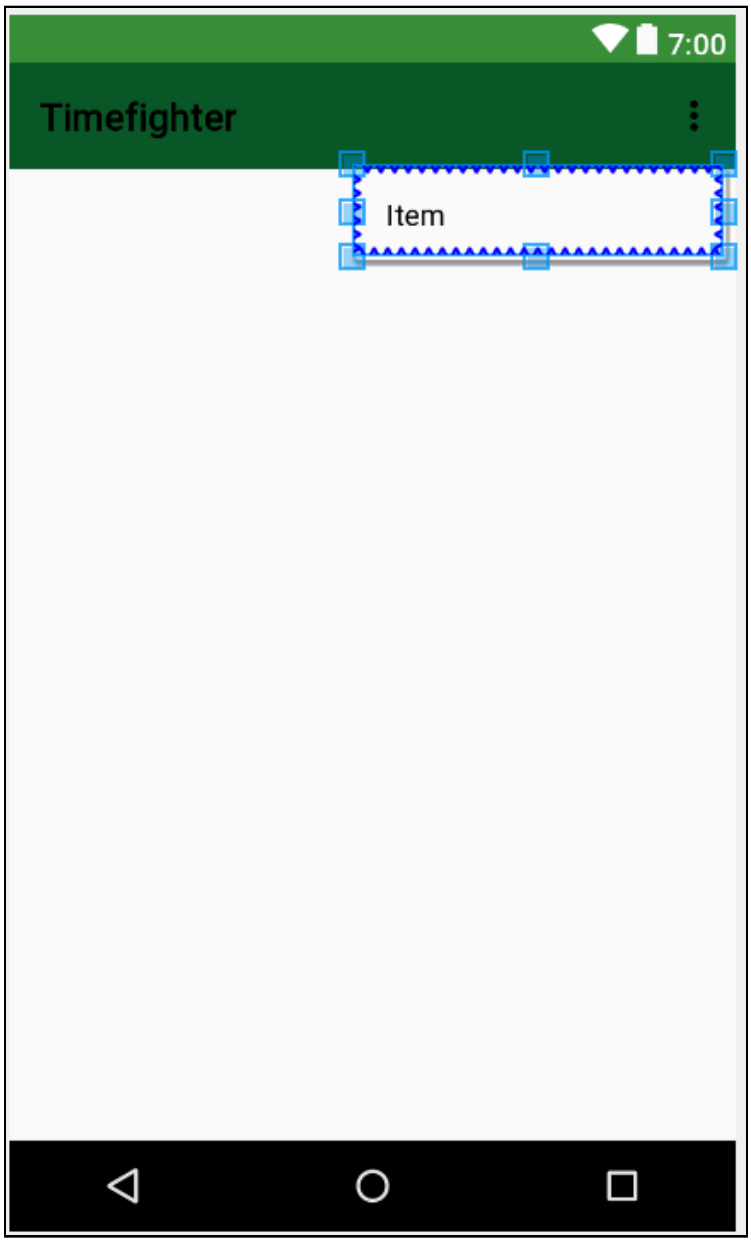


> **Note**: Android Studio may open the Text editor. If this happens, click **Design** at the bottom of the Layout window.

Here, you have the usual windows. The **Palette** in the top-left changes to show only menu-specific items, and the **Component Tree** gives you an overview of the hierarchy for your menu.

You want a single item in your menu. To do that, move your cursor over to the **Menu Item** button in the **Palette** window, and click and drag from the **Menu Item** and onto your Layout.

You'll end up with something like this:

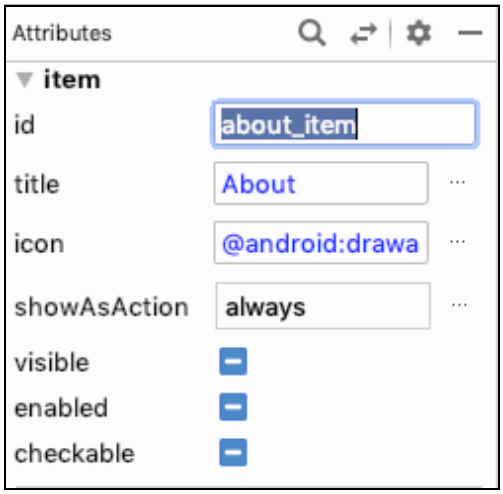


So far, so good. Your newly-placed menu item is now highlighted, and the **Attributes** window is shown on the right side. Time to edit some of these attributes!

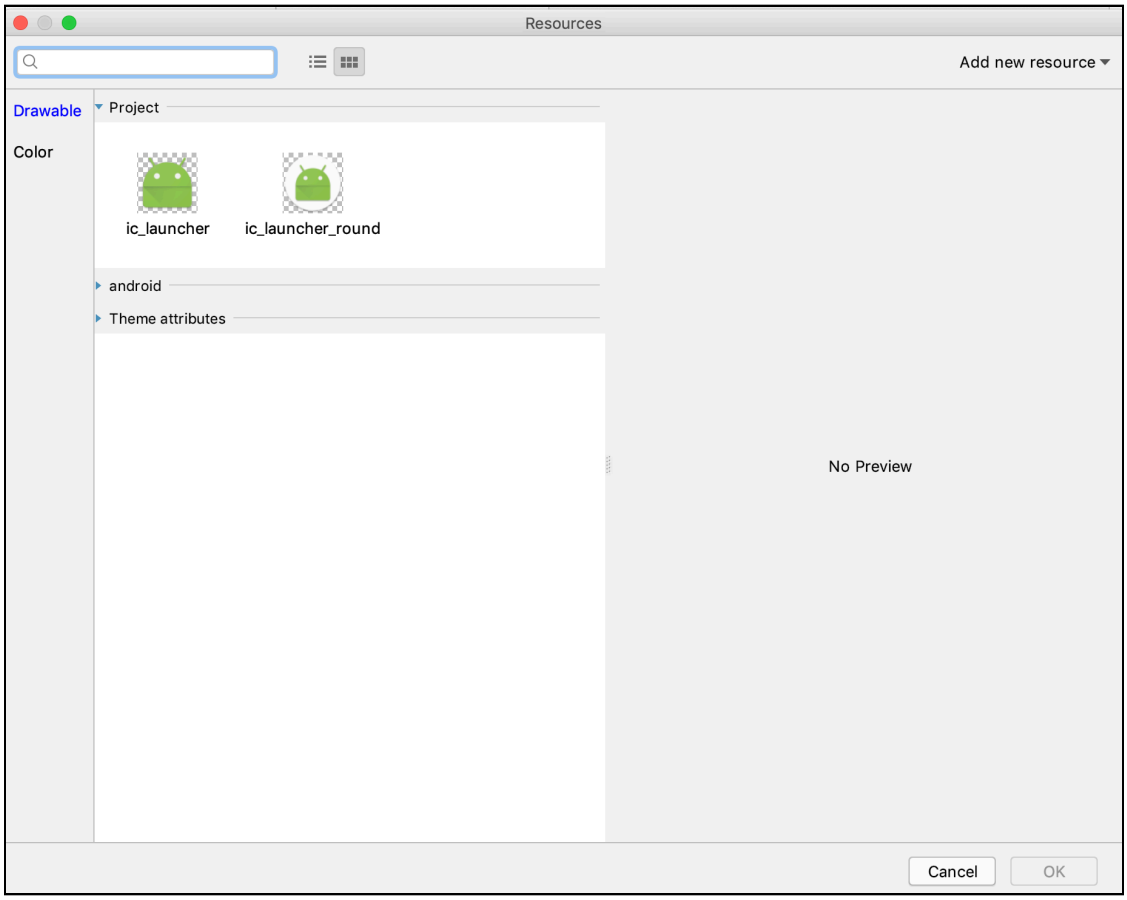
First, set the **id** for the menu item and name it **about_item**. Next, move on to the **title** attribute and name your menu item **About**.

You now need to decide what icon to use for the menu item. Android includes plenty of embedded images from which to choose, so you can use one of those.

Click the small dots next to the **icon** text field. They appear like so:



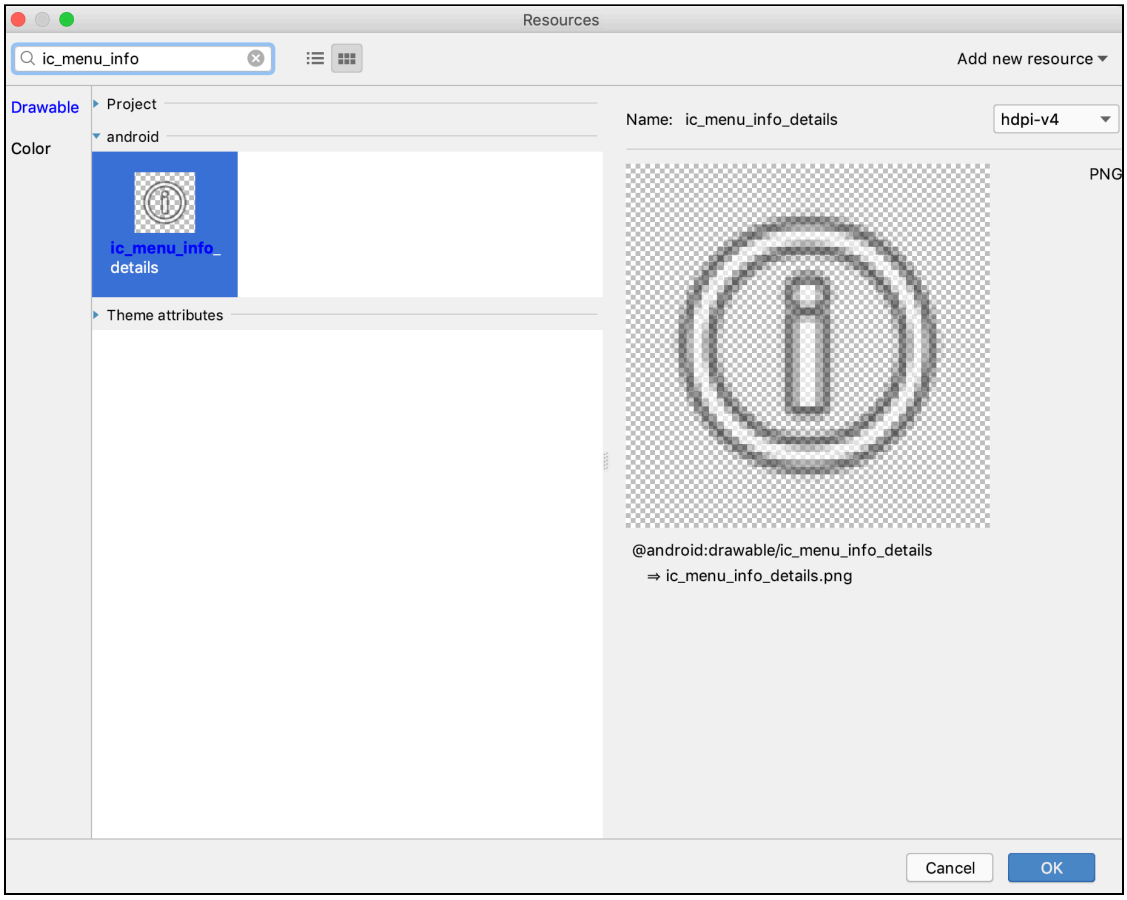
You're presented with the resources window.



This window displays the resources available for use within your app, whether they come from Android or your own custom resources. The window shows both images — or **drawables**, as Android refers to them — or colors.

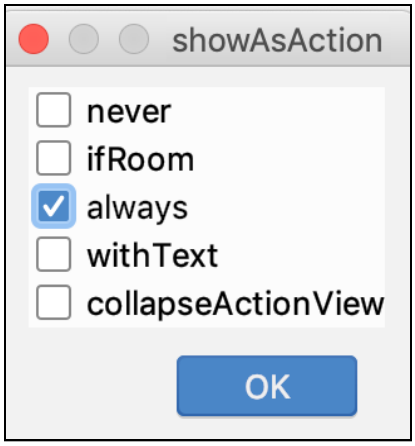
In the top-left of the Resources window is a search bar. Click in the search bar and type **ic_menu_info**.

As you type, the list of resources filters down to match any resources that contain the characters you enetered. In this case, there's only one.



Click the resource under the Android drop-down to select it, and then click **OK**. The Resource window closes and takes you back to the Layout window. The icon text field is populated with the resource you chose.

Finally, to make sure the button is always visible, you need to set the **showAsAction** attribute. Click the small dots next to **showAsAction**. In the dialog that appears, check **Always**, and then click **OK**.



showAsAction affects how your menu item is presented and can have multiple choices depending on the number of items your menu contains and the screen size of your device. You want the menu item to always show up regardless of the circumstances. To do that, you need to set the value to **Always**.

Looking good! Now for some Kotlin code.

In **MainActivity.kt**, add the following method below `onDestroy()`:

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
  // Inflate the menu; this adds items to the action bar if it is
present.
  super.onCreateOptionsMenu(menu)
  menuInflater.inflate(R.menu.menu, menu)
  return true
}
```

This overrides the Activity callback when it attempts to create the menu. You make a call to `super` to give any superclasses of your Activity a chance to set themselves up. You then use the Activity's `menuInflater` to programmatically set up your menu layout for the Activity. Finally, you return `true` to let the Activity know that the menu is set up.

Below `onCreateOptionsMenu(menu: Menu)`, add this method:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
  if (item.itemId == R.id.about_item) {
    showInfo()
  }

  return true
}
```

This method is called whenever a user selects a menu item. You check to see if the ID of the selected menu item is equal to the ID of the item you set up earlier; if so, you call `showInfo()`. Once again, you return `true` to let the Activity know that the event was processed.

Nearly there, just a few more lines!

You still need to create `showInfo()`. Add the following method to **MainActivity.kt**, anywhere inside of the class. aon't worry about editor errors; you'll work on that next.

```
private fun showInfo() {
  val dialogTitle = getString(R.string.about_title,
      BuildConfig.VERSION_NAME)
  val dialogMessage = getString(R.string.about_message)

  val builder = AlertDialog.Builder(this)
  builder.setTitle(dialogTitle)
  builder.setMessage(dialogMessage)
  builder.create().show()
}
```

`showInfo()` handles the setting up of a dialog View for you. It creates two strings to use in the dialog, one for the title and one for the message.

These strings are created using a mixture of the strings stored in **strings.xml** and strings generated when your app is built. In this case, this is the **VERSION_NAME** of your app. The version name is already available, and you'll set up the other strings you need in **strings.xml** in a moment.

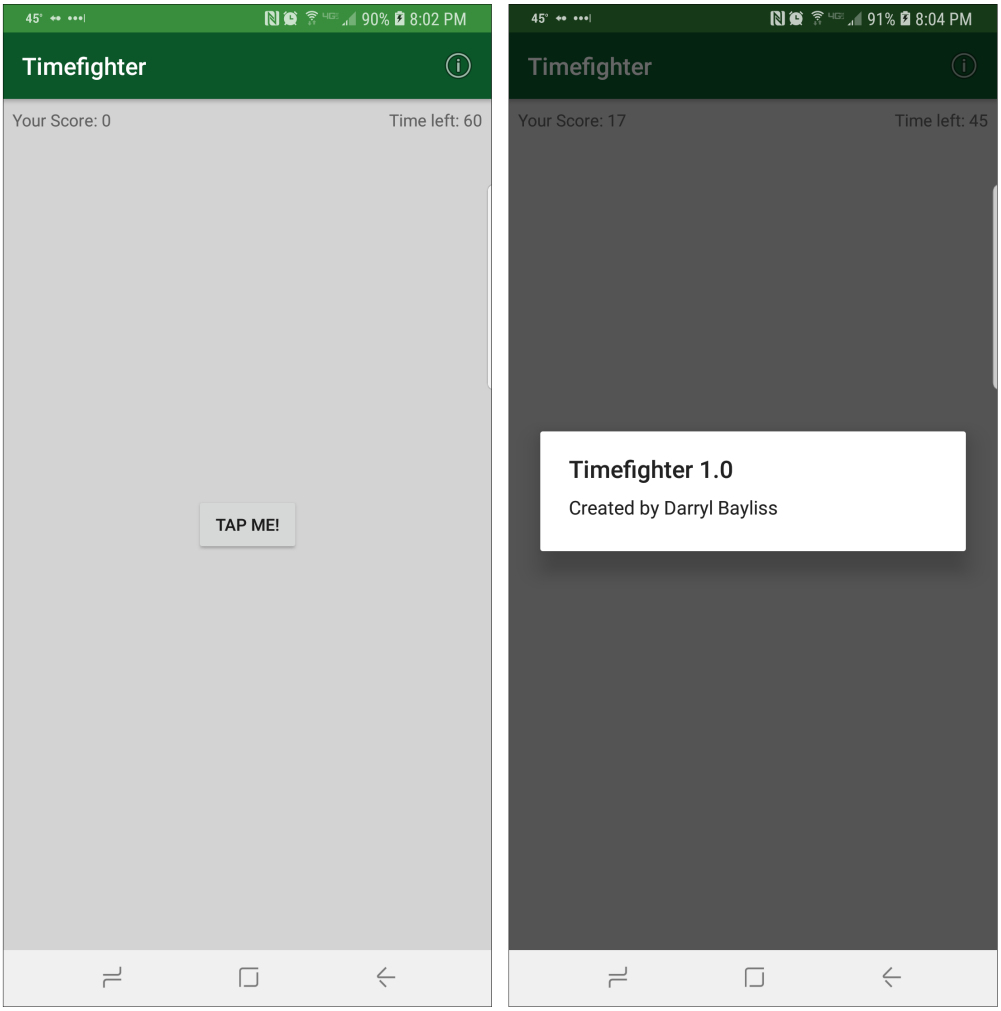
Next, you create an `AlertDialog.Builder` and pass in a Context instance to let the Dialog know where to appear. You pass in the title and message, create the Dialog, and finally display it.

> **Note**: When you add `val builder = AlertDialog.Builder(this)`, Android Studio offers to auto-import a library for you, and it offers several options. Be sure to select the Android Support Library version of AlertDialog: `android.support.v7.app.AlertDialog`.

Open **strings.xml** and add the following strings, substituting your name in `about_message`:

```
<string name="about_title">Timefighter %s</string>
<string name="about_message">Created by YOUR NAME HERE</string>
```

Finally, run the app and check out the new menu sitting in the top-right of the screen. Tap the info button in the menu, and the dialog shows up in the middle of the screen.



Fantastic! You now have a place for people to find out what version of your app they're using and who created it. Well done.

Where to go from here?

Congratulations on completing the first section of the book. You learned a lot over the last few chapters, and you now know how to create a simple game.

In the next section, you'll stop working on TimeFighter and move on to a different app that builds upon the skills and concepts you've learned in this first section.

Where to Go From Here?

We hope you enjoyed this sample of *Android Apprentice*!

If you did, be sure to check out the full book, which contains the following chapters:

1. **Getting Set Up with Android Studio**: To begin creating that killer Android App, you'll need some guidance on how to install the tools you'll need as a young apprentice. Android development takes place inside Android Studio, a customized IDE based on IntelliJ that gives you a powerful set of tools to work with.

2. **Layouts**: If bricks and mortar are the foundation of a sturdy building, then Layouts are the Android equivalent of a sturdy app. Layouts are incredibly flexible and let you define how your user interface is presented on the device to the user.

3. **Activities**: Android apps are built around a set of screens that have a specific purpose. Each screen you see in an app is known as an Activity. An Activity is built around a single task that you want your user to perform.

4. **Debugging**: In the previous two chapters, you focused on developing TimeFighter into a fully-fledged app. In this chapter, you'll focus on how to debug your app when it begins to exhibit bugs.

5. **Prettifying the App**: In this final chapter, you'll learn how to adjust your app to adhere to the Material Design Guidelines, how to add some small touches to give your app that "polished" feeling, and how to add a simple animation to your app to give it some life.

6. **Creating a New Project**: In the previous section, you had a starter project to begin building your app. But in this section, you're going to create your own project from scratch! You'll go through the steps and choices given to you to ensure your project is set up right from the very start.

7. **RecyclerViews**: In Android development, the simplest way to implement lists in your app is to use a class named RecyclerView. You'll how to get started with RecyclerView, how to set up a RecyclerView Adapter to populate your list with data, and how to set up a ViewHolder to handle the layout of each item in your list.

8. **SharedPreferences**: In this chapter, you'll add functionality to ListMaker to create, save, and delete lists. You'll learn what SharedPreferences are and how you can use them to save and retrieve user data.

9. **Communicating Between Activities**: As your app gets more complex, trying to cram more visual elements into a single screen becomes difficult, and can make your app confusing for users. In this chapter, you'll learn how to create separate Activites, how to communicate between Activities using an Intent, and how to pass data between Activities.

10. **Completing the Detail View**: In this chapter, you're going to build up the Activity you created in the previous chapter with familiar components such as a RecyclerView to display the list, and add a FloatingActionButton to add tasks to the list. You'll also learn how to communicate back to the previous Activity using an Intent.

11. **Using Fragments to Expand UI**: When it comes to coding an appealing user interface that adapts across all Android devices with varying screen sizes, things can get tricky! Although you won't be building an app for a fridge just yet (give it time), in this chapter you'll learn what Fragments are and how they work with Activities and how to split up Activities into Fragments.

12. **Material Design**: Material Design is a design language that aims to standardize how a user interacts with an app. This ranges from everything to button clicks, to widget presentation and positioning, even to animation within the app. In this chapter, you'll update ListMaker so it adopts some Material Design principles.

13. **Creating a Map-Based App**: Have you ever been on a road trip and wanted to makes notes about places you've visited? Or needed to warn your future self about some heartburn-inducing greasy food from a roadside diner? If so, then you're in luck! You're about to build PlaceBook, an app that meets all of those needs by letting you bookmark and make notes using a map-based interface.

14. **User Location & Permissions**: In this chapter, you'll tighten up the map experience by automatically centering the map on the user's location at startup, and allowing the user to recenter the map to their current location at any time.

15. **Google Places**: Before you can achieve your ultimate goal of allowing users to bookmark places, you need to let them identify existing places on the map. In this chapter, you'll learn how to identify when a user taps on a place and use the Google Places API to retrieve detailed information about the place.

16. **Saving Bookmarks with Room**: Now that the user can tap on places to get an info window pop-up, it's time to give them a way to bookmark and edit a place. You'll learn about the Room Persistence Library and how it fits into the overall Android Component Architecture.

17. **Detail Activity**: In this chapter you'll add the ability to edit bookmarks. This will involve creating a new activity to display the bookmark details with editable fields.

18. **Navigation & Photos**: Currently, the only way to find an existing bookmark is to locate its pin on the map. In this chapter, you'll add the ability to navigate directly to bookmarks, and you'll replace the photo for a bookmark.

19. **Finishing Touches**: In this chapter you'll add some finishing touches that improve both the look and usability of the PlaceBook app. Even though PlaceBook is perfectly functional as-is, it's often the little touches that make an app go from good to great.

20. **Networking**: In this section, you're going to use many of the skills you've already learned and dive into some more advanced areas of Android development. You'll build a full-featured Podcast manager and player app named PodPlay. This app will allow searching and subscribing to podcasts from iTunes and provide a playback interface with speed controls.

21. **Finding Podcasts**: Creating a search interface can be as simple as adding a text view, responding to the user entering text, and populating a RecyclerView with the results. While this method works fine, the Android SDK provides a built-in search feature that helps future-proof your apps.

22. **Podcast Details**: Now that the user can find their favorite podcasts, you're ready to add a podcast detail screen. In this chapter, you'll design and build the podcast detail fragment, expand on the app architecture, and add a podcast detail fragment.

23. **Podcast Episodes**: Until this point, you've only dealt with the top-level podcast details. Now it's time to dive deeper into the podcast episode details, and that involves loading and parsing the RSS feeds.

24. **Podcast Subscriptions, Part 1**: By giving users the ability to search for podcasts and displaying the podcast episodes, you made great progress in the development of the podcast app. In this section, you'll add the ability to subscribe to favorite podcasts.

25. **Podcast Subscriptions, Part 2**: Now that the user can subscribe to podcasts, it's helpful to notify them when new episodes are available. In this chapter, you'll update the app to periodically check for new episodes in the background and post a notification if any are found.

26. **Podcast Playback**: So far, you've built a decent podcast management app — too bad there's no way to listen to content. In this chapter, you'll learn how to build a media player that plays audio and video podcasts, and integrates into the Android ecosystem.

27. **Episode Player**: In the last chapter, you succeeded in adding audio playback to the app, but you stopped short of adding any built-in playback features. In this final chapter of this section, you'll finish up the PodPlay app by adding a full playback interface and support for videos.

28. **Android Fragmentation & Libraries**: In a perfect world, every Android device would run a single version of Android and app development would be easy. Sadly, the world isn't perfect. As of May 2017, there were two billion active Android devices around the world, all running various versions and flavors of Android. This chapter explores the history of Android versions, and how developers can target as many versions of Android as possible.

29. **Keeping Your App Up to Date**: The more you commit to your app, the more value your users will see in the product. Keeping your app up-to-date is an incentive to growing that important group of users. Publishing an app is an achievement, but supporting an app over the years to come is an even greater achievement.

30. **Preparing for Release**: So you finally built that app you've been dreaming about. Now it's time to share it with the world! But where do you start? Although this chapter will focus primarily on preparing the app for the Google Play store, most of the steps will apply regardless of the publishing platform.

31. **Testing & Publishing**: In this chapter, you'll complete the app publishing process and discover additional ways to distribute your app. You'll also go through the Alpha and Beta testing process to make sure your app is ready to share with the world.

You can find the book on the raywenderlich.com store, here: https://store.raywenderlich.com/products/android-apprentice

We hope you enjoy the book!

— Darryl, Tom, Namrata, Faud and the *Android Apprentice* team

Learn Android programming with Kotlin!

Learning Android programming can be challenging. Sure, there is plenty of documentation, but the tools and libraries available today for Android are easily overwhelming for newcomers to Android and Kotlin.

Android Apprentice takes a different approach. From building a simple first app, all the way to a fully-featured podcast player app, this book walks you step-by-step, building on basic concepts to advanced techniques so you can build amazing apps worthy of the Google Play Store!

Who This Book Is For

This book is for anyone interested in writing mobile apps for Android. Though no previous mobile experience is necessary, this book is also a great resource for iPhone developers transitioning from iOS.

Topics Covered in Android Apprentice:

- **Getting Started:** Learn how to set up Android Studio and the Android Emulator.
- **Layouts:** Create layouts that can be used for both Activities and Fragments
- **Debugging:** No one's perfect! Learn how to dig down and troubleshoot bugs in your apps.
- **Communication:** Design separate Activities and communicate and send data between them using Intents.
- **Scrolling Layouts:** Learn how to use Recycler Views to make efficient, reusable views that scroll fluidly at a touch.
- **Google Places:** Integrate location APIs to bring the magic of maps into your Android apps.
- **Networking:** Learn how to access resources on the internet and handle networked responses.
- **Material Design:** Make sure your apps conform to modern best practices by using Google's standards of Material Design
- **And much, much more!**

One thing you can count on: after reading this book, you'll be prepared to write feature-rich apps from scratch and go all the way to submiting them to the Google Play Store!

About the Tutorial Team

The Tutorial Team is a group of app developers and authors who write tutorials at the popular website **raywenderlich.com.** We take pride in making sure each tutorial we write holds to the highest standards of quality. We want our tutorials to be well written, easy to follow, and fun.

If you've enjoyed the tutorials we've written in the past, you're in for a treat. The tutorials we've written for this book are some of our best yet — and this book contains detailed technical knowledge you simply won't be able to find anywhere else.