O'REILLY®

Third Edition

Head First

Android Development

A Learner's Guide to Building Android Apps with Kotlin

Dawn Griffiths
& David Griffiths

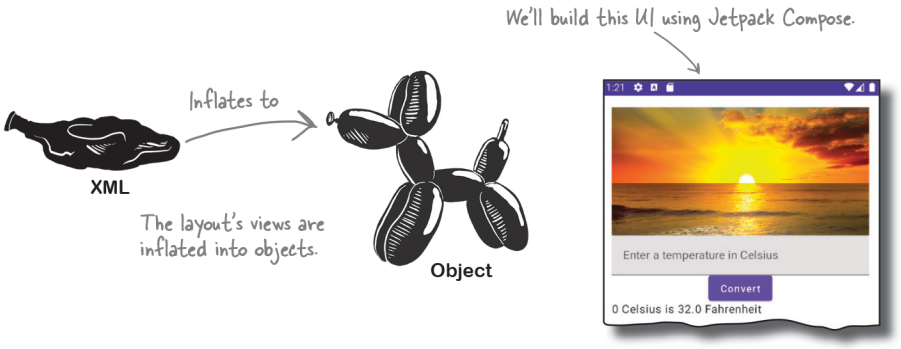
A Brain-Friendly Guide

Head First

Android Development

What will you learn from this book?

If you have an idea for a killer Android app, this fully revised and updated book will get you up and running in a jiffy. You'll learn hands-on how to structure your app, design flexible and interactive user interfaces, persist data in a database, and use the latest features of Android Jetpack, including Jetpack Compose. It's like having an experienced Android developer sitting right next to you! All you need to get started is a little Kotlin know-how.



What's so special about this book?

If you've read a Head First book, you know what to expect—a visually rich format designed for the way your brain works. If you haven't, you're in for a treat. With this book, you'll learn Android development through a multisensory experience that engages your mind rather than a text-heavy approach that puts you to sleep.

"Android development changes completely every few years, which makes writing a book like this extremely difficult. In this third edition of their classic, the Griffiths have done their usual excellent job showing how modern Android development is done by almost completely rewriting their book. The result is, once again, the best available book in the field. If you want to learn the right way to build Android apps, buy this book."

—Ken Kousen
President of Kousen IT, Inc.

PROGRAMMING/ANDROID

US $79.99 **CAN $105.99**

ISBN: 978-1-492-07652-0

O'REILLY®

Advance Praise for *Head First Android Development*

"Android development changes completely every few years, which makes writing a book like this extremely difficult. In this third edition of their classic, the Griffiths have done their usual excellent job showing how modern Android development is done by almost completely rewriting their book. The result is, once again, the best available book in the field. If you want to learn the right way to build Android apps, buy this book."

— Ken Kousen, President of Kousen IT, Inc.

"Creating material that is both accessible and accurate is incredibly difficult, but the Griffiths have nailed it. They explain the complexity of Android in such a clear, precise way that even novice developers can understand. I'm very impressed."

—G. Blake Meike, Android developer and coauthor of *Programming Android 2e*

"Dawn and David have enthusiastically updated their Java examples to Kotlin, making this 3rd edition an excellent resource for developers who want to learn modern Android development."

—Duncan McGregor and Nat Pryce, authors of *Java to Kotlin: A Refactoring Guidebook*

"Thank you for this book. I am new to Android development, and this was the book that explained each and every thing in a very clear way for me."

—Ambreen Khan, Android developer

"Android development has evolved. A lot! Let this book be your friendly, accurate and able mentor on your path to learn swimming the right way with the right composure in only the right places. And avoid the battery-piranha-, async-snake- and performance-leech-infested waters of the past, while also having a lot of fun!"

—Ingo Krotzky, Android learner

A new Beginning

Machine is Luring

So many Questions

This Book

To answer them All

—Susan B. Brenner, Machine Poet

Praise for *Head First Kotlin*

"Clear, intuitive, and easy to understand. If you're new to Kotlin, this is an excellent introduction."

— Ken Kousen, Official Kotlin Trainer, certified by JetBrains

"*Head First Kotlin* will definitely help you come to grips fast, build a solid foundation, and (re)gain your joy in writing code."

— Ingo Krotzky, Kotlin learner

"At last! Learn Kotlin without knowing Java. Simple, concise and fun, this is the book I've been waiting for."

— Dr. Matt Wenham, Data scientist and Python coder

"Kotlin is a hot, up-and-coming language with no signs of slowing down in the tech industry. Head First Kotlin is a fun, non-intimidating way of learning to code Kotlin from scratch. This book is friendly for folks learning to code, complete with screenshots for using the IntelliJ IDEA, notes around the code, and visuals."

— Amanda Hinchman-Dominguez, Android Engineer, Groupon and Kotlin GDE, Google

Head First
Android Development

Dawn Griffiths
David Griffiths

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Head First Android Development

by Dawn Griffiths and David Griffiths

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Series Creators:	Kathy Sierra, Bert Bates
Series Advisors:	Eric Freeman, Elisabeth Robson
Editor:	Virginia Wilson
Cover Designer:	Karen Montgomery
Production Editor:	Katherine Tozer
Production Services:	Charles Roumeliotis
Indexer:	Potomac Indexing, LLC
Page Viewers:	Mum and Dad, Rob and Lorraine

Printing History:

June 2015: First Edition.
August 2017: Second Edition.
November 2021: Third Edition.

No kittens were harmed in the making of this book, but several pizzas were eaten.

ISBN: 978-1-492-07652-0
[LSI] [2022-01-21]

To all the key workers who kept us and our loved ones safe over the past two years. We're immensely grateful.

Authors of Head First Android Development

Dawn Griffiths started life as a mathematician at a top UK university, where she was awarded a first-class honors degree in mathematics. She went on to pursue a career in software development and has over 20 years' experience working in the IT industry as a senior software developer.

When Dawn's not working, you'll find her honing her Tai Chi skills, reading, running, making bobbin lace, or cooking. She particularly enjoys spending time with her wonderful husband, David.

David Griffiths began programming at age 12, when he saw a documentary on the work of Seymour Papert. At age 15, he wrote an implementation of Papert's computer language LOGO. Since then, he has worked as an agile coach, a software developer, and a garage attendant, but not in that order.

When David's not working, he spends much of his spare time traveling with his lovely wife, Dawn.

Together, Dawn and David have written a whole host of books, including Head First Android Development, *Head First Kotlin*, *Head First C*, *Head First Rails*, *Head First Statistics,* and the *React Cookbook*. They contributed to *97 Things Every Java Programmer Should Know*, and they created the video course *The Agile Sketchpad* as a way of teaching key concepts and techniques in a way that keeps your brain active and engaged. They also deliver live, online training through the O'Reilly learning platform at *https://www.oreilly.com/live-events.*

You can follow Dawn and David on Twitter at *https://twitter.com/HeadFirstDroid*.

Table of Contents (summary)

Table of Contents (the real thing)

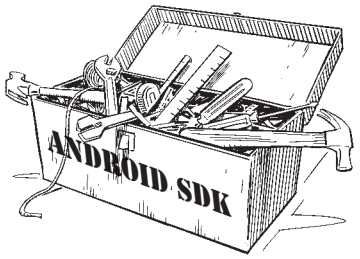
Intro

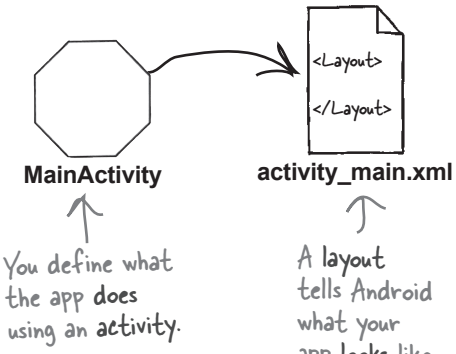
Your brain on Android. Here you are trying to learn something, while here your brain is, doing you a favor by making sure the learning doesn't stick. Your brain's thinking, "Better leave room for more important things, like which wild animals to avoid and whether naked snowboarding is a bad idea." So how do you trick your brain into thinking that your life depends on knowing how to develop Android apps?

getting started

1 Diving In

Android is the world's most popular mobile operating system. And there are billions of Android users worldwide, all waiting to download your next great idea. In this chapter, you'll find out how to start turning your ideas into reality by **building a basic Android app**, and updating it. You'll learn how to run it on physical and virtual devices. Along the way, you'll meet two of the core components of all Android apps: **activities** and **layouts**.

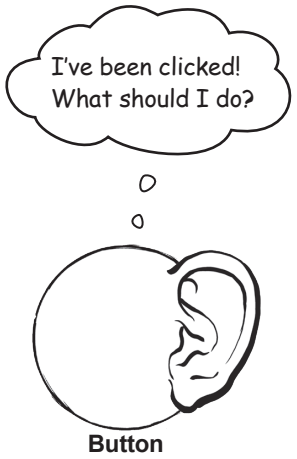


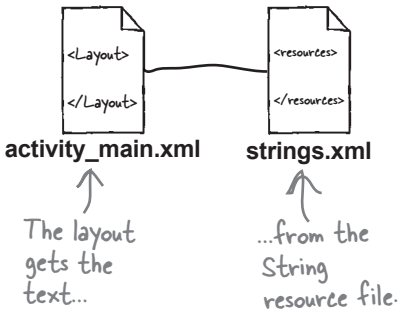


building interactive apps

2 Apps That Do Something

Most apps need to respond to the user in some way. And in this chapter, you'll see how you can make your apps **more interactive**. You'll discover how to add an ***OnClickListener*** to your activity code so that your app can **listen to what the user's doing**, and make an appropriate response. You'll find out more about **how to design layouts**, and you'll learn how each UI component you add to your layout is derived from **a common View ancestor**. Along the way, you'll discover **why String resources are so important** for flexible, well-designed apps.

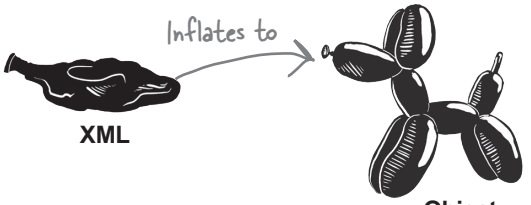


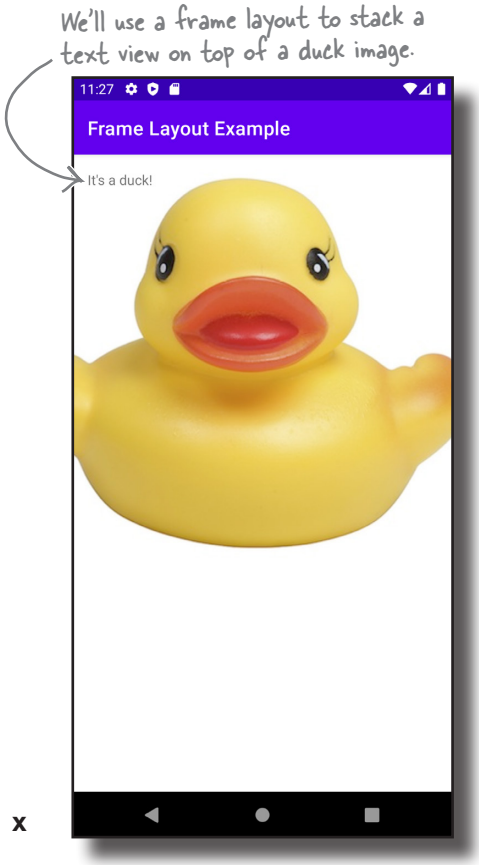


layouts

3 Being a Layout

We've only scratched the surface of using layouts. So far, you've seen how to arrange views in a simple linear layout, but there's so much more that layouts can do. In this chapter we'll **go a little deeper** and show you how layouts really work. You'll learn **how to fine-tune your linear layouts**. You'll discover how to use **frame layouts** and **scroll views**. And by the end of the chapter, you'll learn that even though they might look a little different, all layouts—and the views you add to them—have **more in common than you might think**.



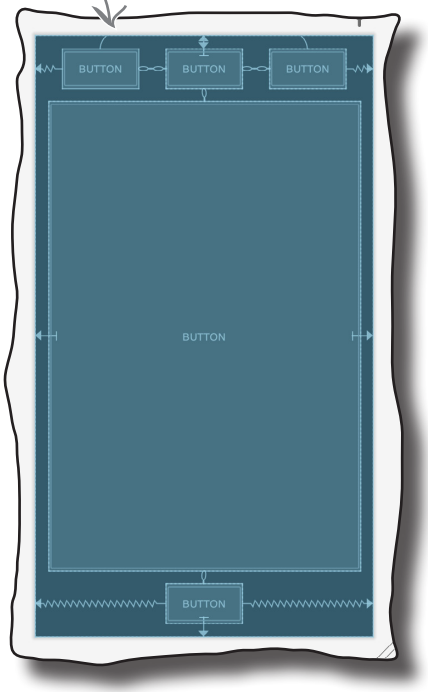


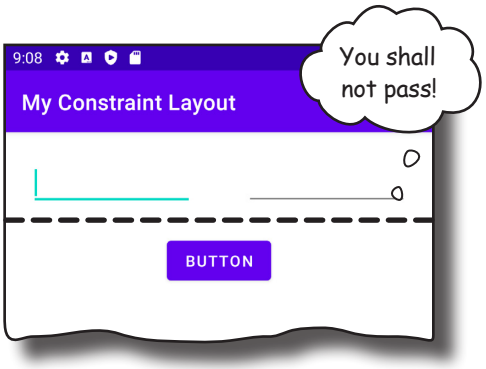
constraint layouts

4 Draw Up a Blueprint

You don't build a house without a blueprint. And some layouts use **blueprints** to make sure they **look exactly the way you want**. In this chapter, we'll introduce you to Android's **constraint layout**: a **flexible way of designing more complex UIs**. You'll discover how **constraints** and **bias** let you position and size your views, **irrespective of screen size and orientation**. You'll find out how to keep views in their place with **guidelines** and **barriers.** Finally, you'll learn how to pack or spread views with **chains** and **flows**.

This is a constraint layout's blueprint. It contains all the information that the constraint layout needs to arrange views on the screen.

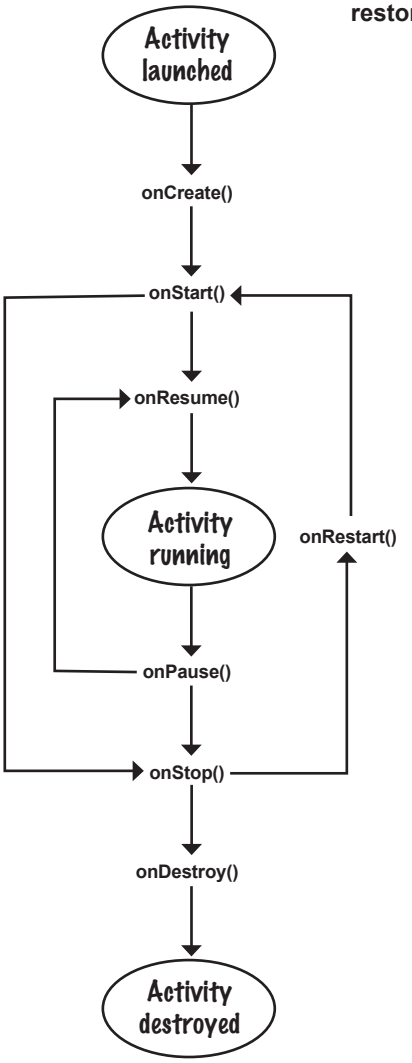




5 the activity lifecycle

Being an Activity

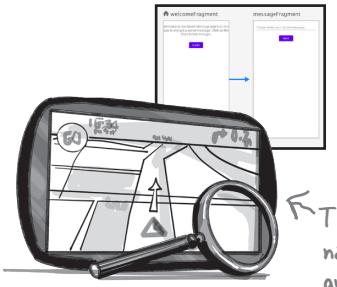
Activities form the foundation of every Android app. So far you've seen how to create an activity, and use it to interact with the user. But if you don't know about **the activity lifecycle**, some of its behavior **might take you by surprise**. In this chapter, you'll learn what happens when an activity is **created** and **destroyed**, and how this can lead to **unexpected consequences**. You'll find out how to control its behavior when it's made **visible**, or **hidden**. You'll even discover ways of **saving and restoring your activity's state**, just when you need it.

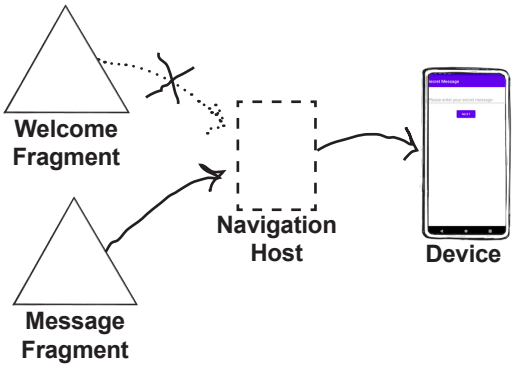


6 Fragments and navigation

Finding Your Way

Most apps require more than one screen. So far, we've just looked at how to create single-screen apps, which is fine for simple applications. But what if you have **more complex requirements**? In this chapter, you'll learn how to use **fragments** and the **Navigation component** to **build multi-screen apps**. You'll learn how **fragments are like subactivities** with their own methods. You'll find out how to **design effective navigation graphs**. Finally, you'll meet the **navigation host** and **navigation controller**, and learn how they help you navigate from place to place.

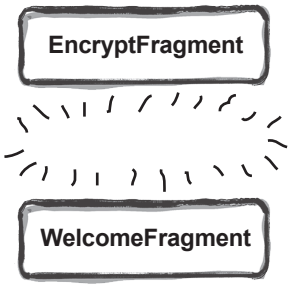




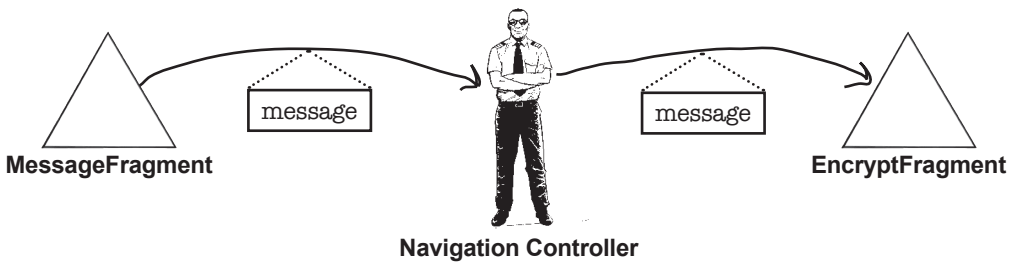
7 safe args

Passing Information

Sometimes fragments need extra information to work properly. If a fragment shows details of a contact, for example, it needs to know which contact to display. But what if this information **comes from another fragment**? In this chapter, you'll **build on your navigation know-how** by learning **how to pass data between fragments**. You'll find out **how to add arguments** to navigation destinations so they can receive the information they need. You'll meet **the Safe Args plug-in**, and learn how to use it to **write type-safe code**. Finally, you'll discover **how to manipulate the back stack**, and take control of back button behavior.



Removing MessageFragment from the back stack means that when you click on the Back button from EncryptFragment, WelcomeFragment gets displayed instead of MessageFragment.

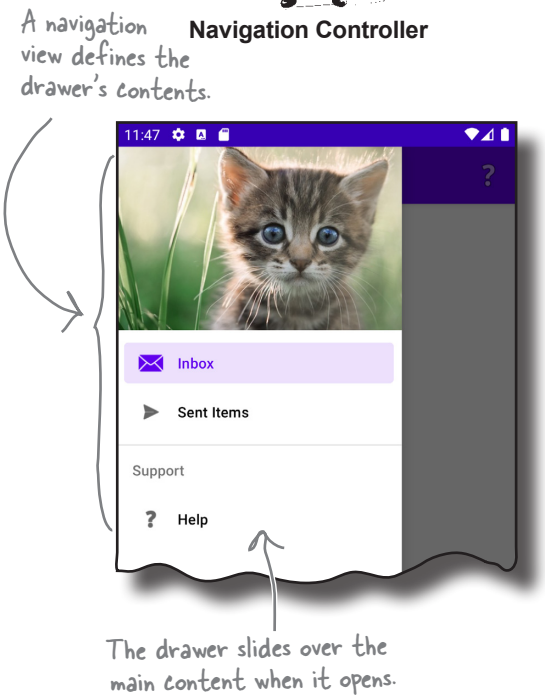


navigation ui

8 Going Places

Most apps need to be able to navigate between destinations.

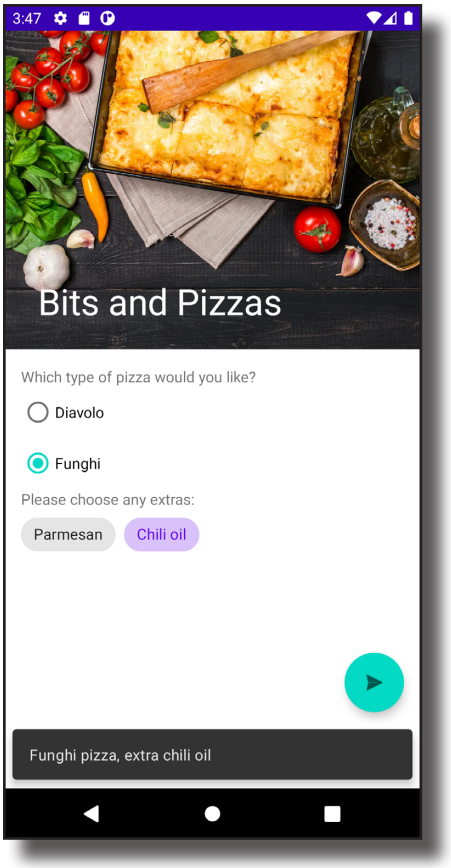
And with the Android the Navigation component, building this UI became much simpler. Here, you'll learn how to use some of Android's navigation UI components so that **your users can navigate your app more easily**. You'll see how to use **themes**, and replace your app's default app bar with a **toolbar**. You'll learn how to add **menu items** you can **use for navigation**. You'll discover how to implement **bottom bar navigation**. Finally, you'll create a swish **navigation drawer**: a panel that slides out from the side of your activity.



material views

9 A Material World

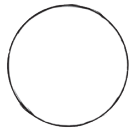
Most apps need a slick UI that responds to the user. You've so far learned how to use views such as **text views, buttons, and spinners**, and applied **Material themes** to make sweeping changes to your app's look and feel. But there's so much more you can do. Here, you'll learn how to make your UI more responsive with the **coordinator layout**. You'll create **toolbars** that can **collapse or scroll** on a whim. You'll discover **exciting new views** such as **checkboxes**, **radio buttons**, **chips**, and **floating action buttons**. Finally, you'll find out how to display friendly pop-up messages using **toasts** and **snackbars**.

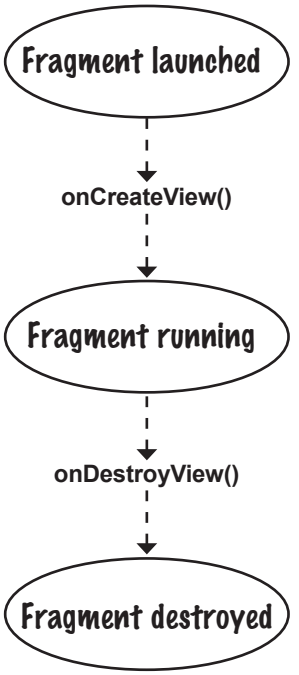


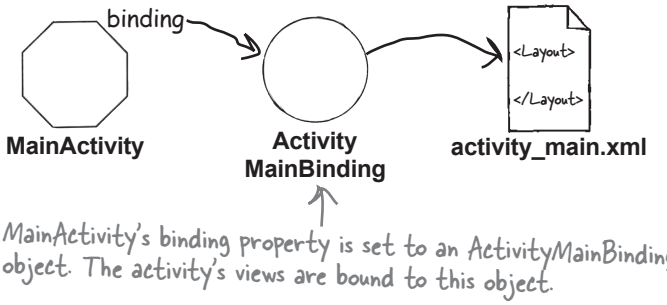
view binding

10 Bound Together

It's time to wave farewell to findViewById(). As you've probably noticed by now, the more views you have and the more interactive your apps become, **the more calls you need to make to *findViewById()***. And if you're getting tired of typing the code for this method every time you want to work with a view, *y*ou're not alone. In this chapter, you'll discover **how to make *findViewById()* a thing of the past** by implementing **view binding**. You'll find out how to apply this technique to both **activity and fragment code**, and you'll learn why this approach is a **safer, more efficient** way of accessing your layout's views.





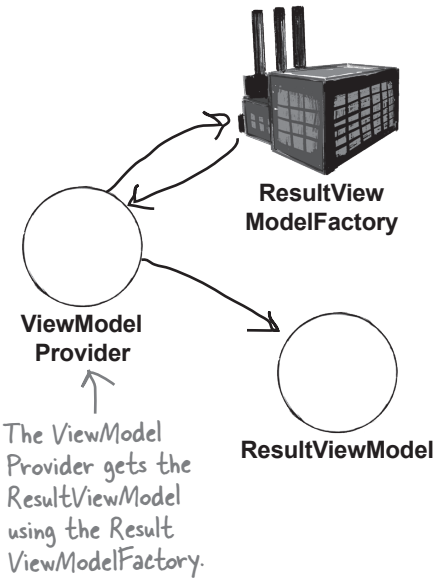


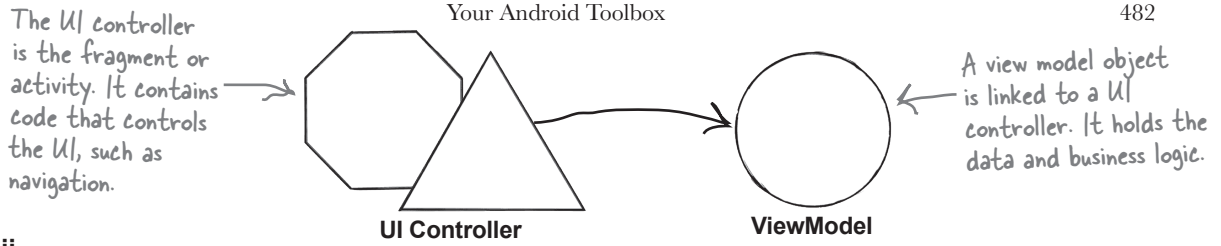
view models

11 Model Behavior

As apps grow more complex, fragments have more to juggle.

And if you're not careful, this can lead to **bloated code** that tries to do everything. Business logic, navigation, controlling the UI, dealing with configuration changes…you name it, it's in there. In this chapter, you'll learn how to deal with this kind of situation using **view models**. You'll discover **how they simplify your activity and fragment code**. You'll find out **how they survive configuration changes**, keeping your app's state safe and sound. Finally, we'll show you how to build a **view model factory**, and when this might be needed.



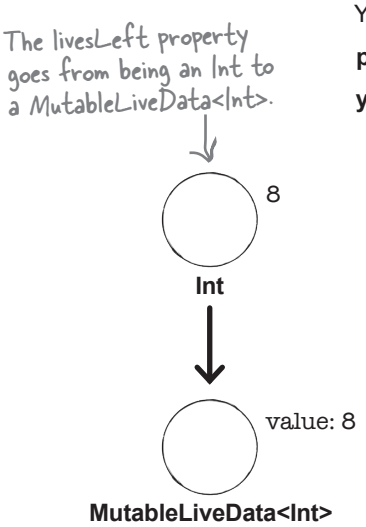


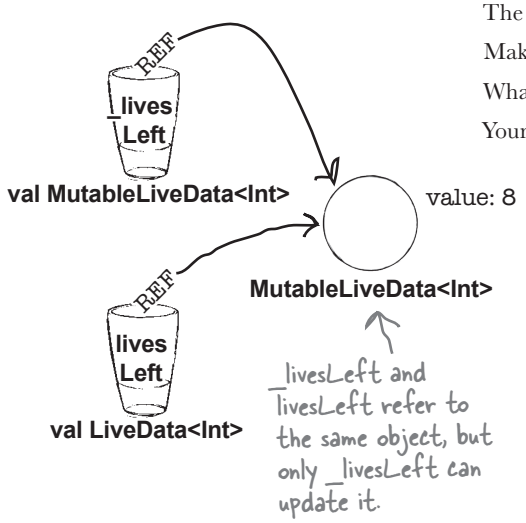
live data

12 Leaping into Action

Your code often needs to react to property value changes.

If a view model property changes value, for example, **the fragment might need to respond** by updating its views or navigating elsewhere. But **how does a fragment get to hear when a property's been updated**? Here, we'll introduce you to **live data**: a way of **telling interested parties when something's changed**. You'll learn all about ***MutableLiveData***, and how to **make your fragment observe properties** of this type. You'll discover **how the *LiveData* type helps maintain your app's integrity**.



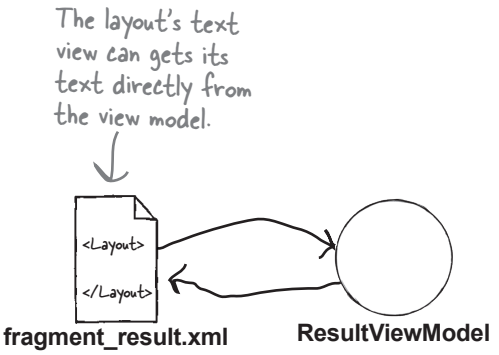


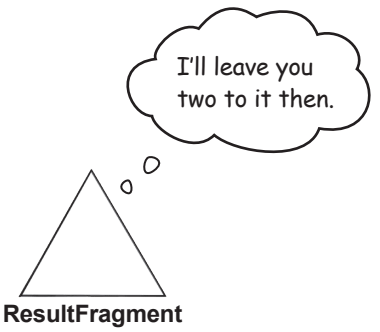
data binding

13 Building Smart Layouts

Layouts can do more than control your app's appearance.

All of the layouts you've written so far have been told how they should behave by activity or fragment code. But just imagine if **the layouts could think for themselves**, and **make their own decisions**. In this chapter, we'll introduce you to **data binding**: a way of **boosting your layout's IQ**. You'll find out **how to make views get values** directly from the view model. You'll use **listener binding** to make buttons call their methods. You'll even discover how **one simple line of code lets views respond to live data updates**.

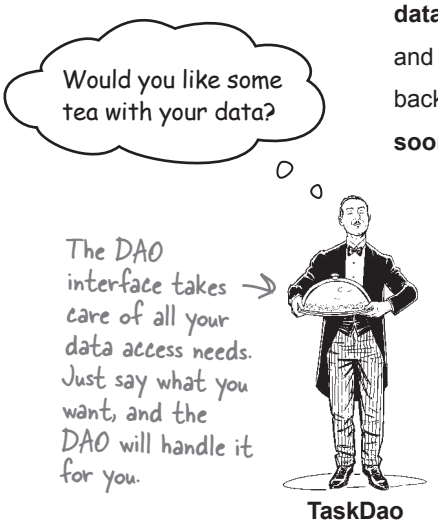


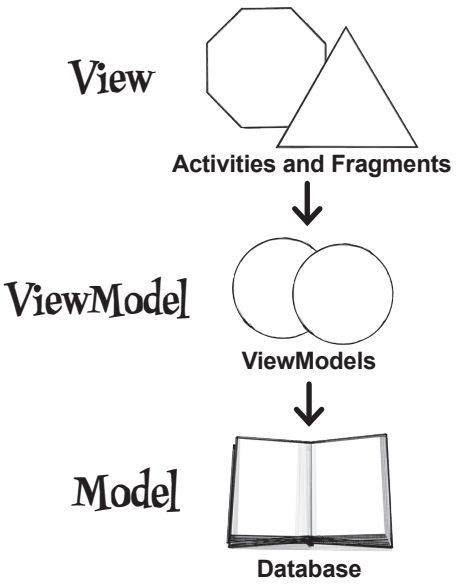


room databases

14 Room with a View

Most apps need data that persists. But if you don't take steps to store this data somewhere, **it will be lost forever** as soon as the app is closed down. You usually keep data safe in Androidville by **storing it in a database**, so in this chapter, we'll introduce you to the **Room persistence library**. You'll learn how to **build databases**, **create tables,** and **define data access methods** using annotated classes and interfaces. You'll find out how to **use coroutines** to run database code in the background. And along the way, you'll discover how to **transform your live data as soon as it changes** with a little help from ***Transformations.map()***.

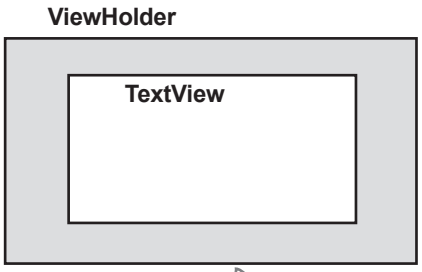




recycler views

15 Reduce, Reuse, Recycle

Lists of data are a key part of most apps. And in this chapter, we'll show you how to create one using a **recycler view**: a **super-flexible** way of building a **scrollable list**. You'll learn how to create **flexible layouts** for your list, including text views, checkboxes, and more. You'll find out **how to create adapters** that **squish your data** into the recycler view in whatever way you choose. You'll discover how to use **card views** to give your data a **3D material look**. Finally, we'll show you how **layout managers** can completely change the look of your list with **just one or two lines of code**.

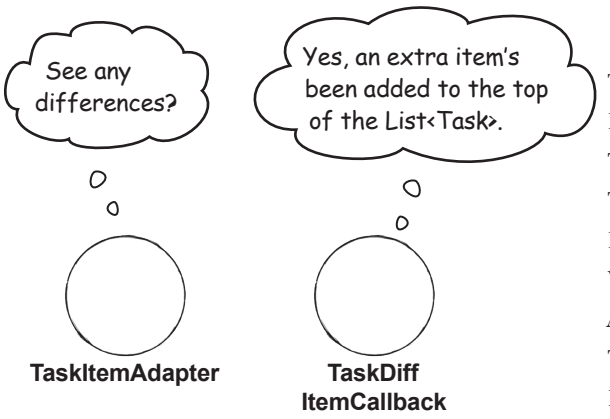


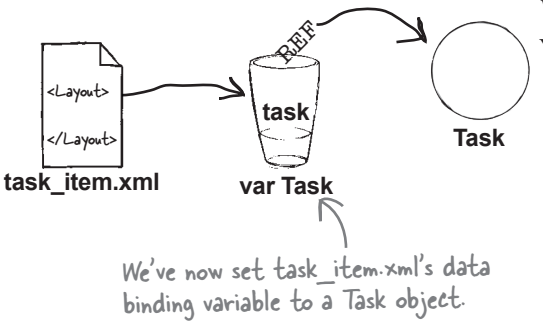
Each view holder holds the root view of the layout for each item. Here the root view is a text view.

diffutil and data binding

16 Life in the Fast Lane

Your app needs to run as smoothly and efficiently as possible. But if you're not careful, large or complex data sets can cause your recycler view to glitch. In this chapter, we'll introduce you to ***DiffUtil***: a utility class that **adds extra smarts to your recycler view**. You'll find out how to use it to **make efficient updates** to your recycler view. You'll discover how ***ListAdapters* make using *DiffUtil* a breeze**. And along the way, you'll learn **how to get rid of *findViewById()* for good** by implementing **data binding in your recycler view code**.

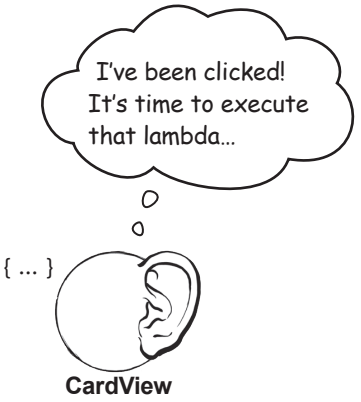


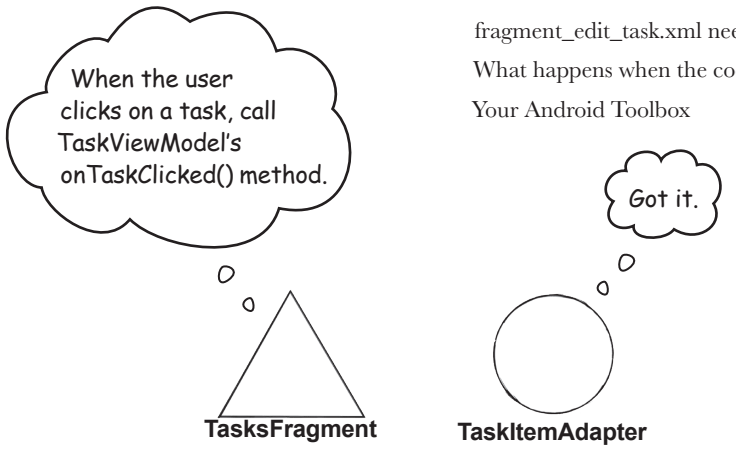


recycler view navigation

17 Pick a Card

Some apps rely on the user selecting an item from a list. And in this chapter, you'll learn **how to make recycler views a core part of your app design** by **making their items clickable**. You'll discover how to **implement recycler view navigation** by making the app navigate to a new screen each time the user clicks on a record. You'll find out **how to show the user extra information** about their chosen record, and update it in the database.

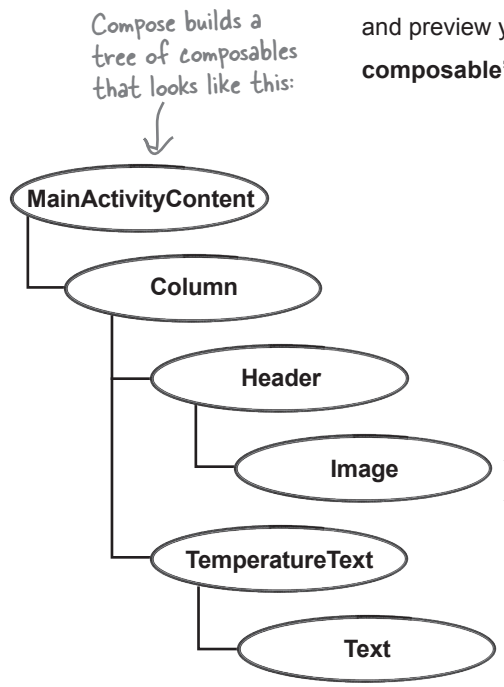


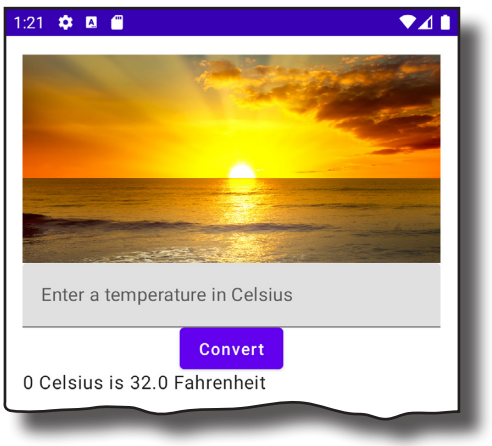


jetpack compose

18 Compose Yourself

All the UIs you've built so far have used views and layout files. But with **Jetpack Compose**, **that's not the only option**. In this chapter, we're going to take **a road trip to Composeville**, and find out how to build UIs using Compose components called **composables** instead of views. You'll learn how to use built-in composables such as **Text**, **Image**, **TextField,** and **Button**. You'll discover how to arrange them in **Rows** and **Columns**, and style them using **themes**. You'll write and preview your own **composable functions**. You'll even find out **how to manage a composable's state** using ***MutableState*** objects.

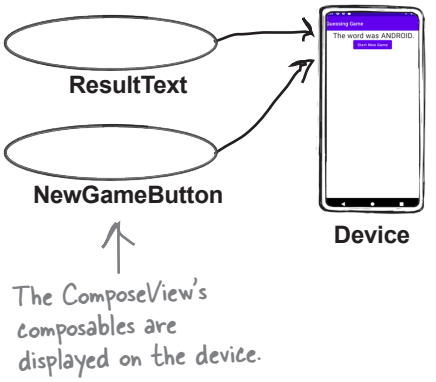


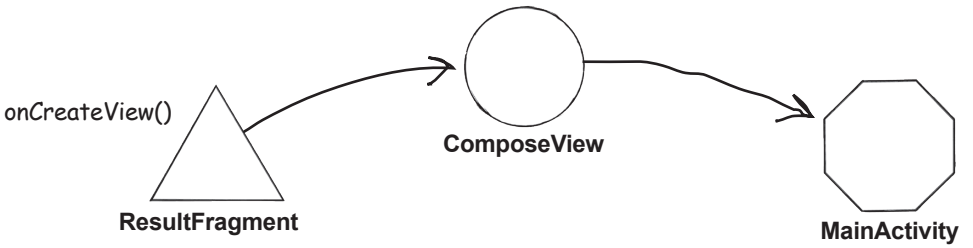


integrating compose with views

19 Perfect Harmony

You get the best results when things work together. So far, you've learned how to build a UI using views or composables. But what if you want to use **both**? In this chapter, you'll find out how to get **the best of both worlds** by **adding composables to a View-based UI**. You'll discover techniques for making **composables work with view models**. You'll even find out how to make them **respond to *LiveData* updates**. By the end of the chapter, you'll have all the tools you need to **use composables with views**, or even **migrate to a pure Compose UI.**

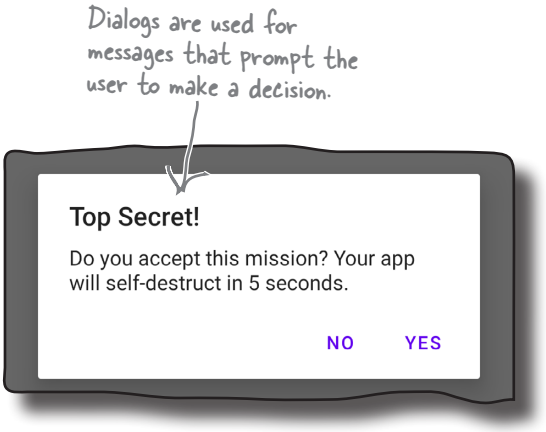


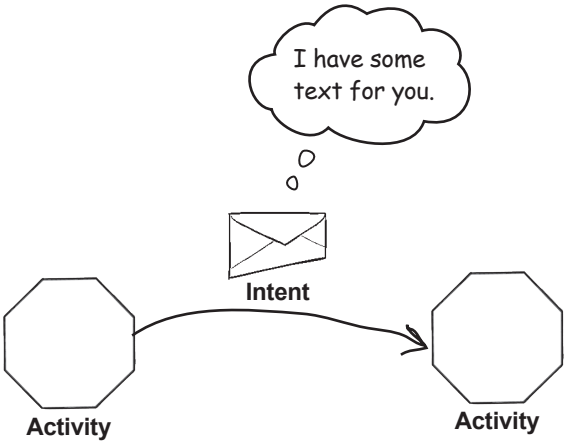


leftovers

The Top Ten Things (we didn't cover)

Even after all that, there's still a little more. There are just a few more things we think you need to know. We wouldn't feel right about ignoring them, and we really wanted to give you a book you'd be able to lift without extensive training at the local gym. Before you put down the book, **read through these tidbits**.





how to use this book

Intro

In this section, we answer the burning question:
"So why DID they put that in a book on Android?"

Who is this book for?

If you can answer "yes" to all of these:

1. Do you already know how to program in Kotlin, Java, or another object-oriented language?
2. Do you want to master Android app development, create the next big thing in software, make a small fortune, and retire to your own private island?
3. Do you prefer actually doing things and applying the stuff you learn over listening to someone in a lecture rattle on for hours on end?

OK, maybe that one's a little far-fetched. But, you gotta start somewhere, right?

this book is for you.

Who should probably back away from this book?

If you can answer "yes" to any of these:

1. Are you looking for a quick introduction or reference book to developing Android apps?
2. Would you rather have your toenails pulled out by 15 screaming monkeys than learn something new? Do you believe an Android book should cover *everything*, especially all the obscure stuff you'll never use, and if it bores the reader to tears in the process, then so much the better?

this book is ***not*** for you.

[Note from Marketing: this book is for anyone with a credit card or a PayPal account]

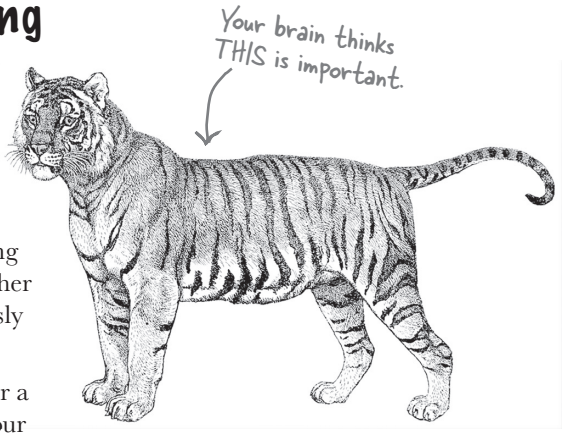
We know what you're thinking

"How can *this* be a serious book on developing Android apps?"

"What's with all the graphics?"

"Can I actually *learn* it this way?"

We know what your *brain* is thinking



Your brain craves novelty. It's always searching, scanning, *waiting* for something unusual. It was built that way, and it helps you stay alive.

So what does your brain do with all the routine, ordinary, normal things you encounter? Everything it *can* to stop them from interfering with the brain's *real* job—recording things that *matter*. It doesn't bother saving the boring things; they never make it past the "this is obviously not important" filter.

How does your brain *know* what's important? Suppose you're out for a day hike and a tiger jumps in front of you—what happens inside your head and body?

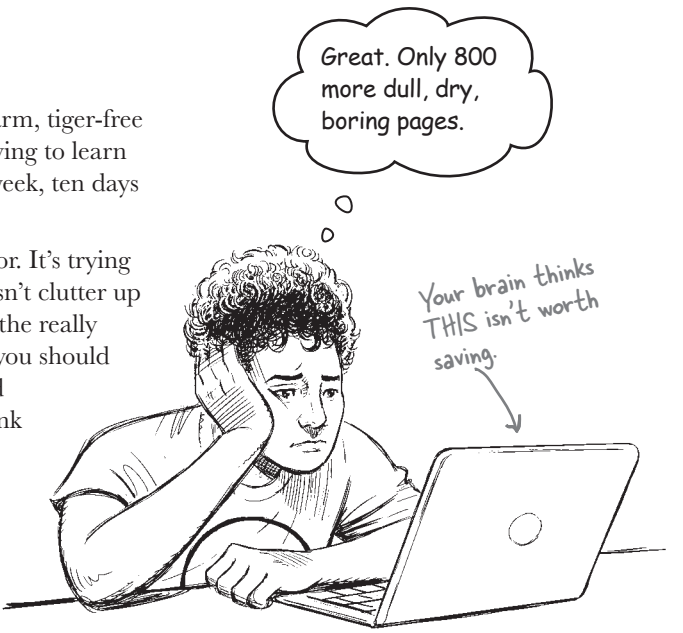
Neurons fire. Emotions crank up. *Chemicals surge*.

And that's how your brain knows…

This must be important! Don't forget it!

But imagine you're at home or in a library. It's a safe, warm, tiger-free zone. You're studying. Getting ready for an exam. Or trying to learn some tough technical topic your boss thinks will take a week, ten days at the most.

Just one problem. Your brain's trying to do you a big favor. It's trying to make sure that this *obviously* unimportant content doesn't clutter up scarce resources. Resources that are better spent storing the really *big* things. Like tigers. Like the danger of fire. Like how you should never have posted those party photos on Instagram. And there's no simple way to tell your brain, "Hey brain, thank you very much, but no matter how dull this book is, and how little I'm registering on the emotional Richter scale right now, I really *do* want you to keep this stuff around."



We think of a "Head First" reader as a learner.

So what does it take to *learn* something? First, you have to *get* it, then make sure you don't *forget* it. It's not about pushing facts into your head. Based on the latest research in cognitive science, neurobiology, and educational psychology, *learning* takes a lot more than text on a page. We know what turns your brain on.

Some of the Head First learning principles:

Make it visual. Images are far more memorable than words alone, and make learning much more effective (up to 89% improvement in recall and transfer studies). It also makes things more understandable. **Put the words within or near the graphics** they relate to, rather than on the bottom or on another page, and learners will be up to *twice* as likely to solve problems related to the content.

Use a conversational and personalized style. In recent studies, students performed up to 40% better on post-learning tests if the content spoke directly to the reader, using a first-person, conversational style rather than taking a formal tone. Tell stories instead of lecturing. Use casual language. Don't take yourself too seriously. Which would *you* pay more attention to: a stimulating dinner-party companion, or a lecture?

Get the learner to think more deeply. In other words, unless you actively flex your neurons, nothing much happens in your head. A reader has to be motivated, engaged, curious, and inspired to solve problems, draw conclusions, and generate new knowledge. And for that, you need challenges, exercises, and thought-provoking questions, and activities that involve both sides of the brain and multiple senses.

Get—and keep—the reader's attention. We've all had the "I really want to learn this, but I can't stay awake past page one" experience. Your brain pays attention to things that are out of the ordinary, interesting, strange, eye-catching, unexpected. Learning a new, tough, technical topic doesn't have to be boring. Your brain will learn much more quickly if it's not.

Touch their emotions. We now know that your ability to remember something is largely dependent on its emotional content. You remember what you care about. You remember when you *feel* something. No, we're not talking heart-wrenching stories about a boy and his dog. We're talking emotions like surprise, curiosity, fun, "what the…?", and the feeling of "I rule!" that comes when you solve a puzzle, learn something everybody else thinks is hard, or realize you know something that "I'm more technical than thou" Bob from Engineering *doesn't*.

Metacognition: thinking about thinking

If you really want to learn, and you want to learn more quickly and more deeply, pay attention to how you pay attention. Think about how you think. Learn how you learn.

Most of us did not take courses on metacognition or learning theory when we were growing up. We were *expected* to learn, but rarely *taught* to learn.

But we assume that if you're holding this book, you really want to learn how to develop Android apps. And you probably don't want to spend a lot of time. If you want to use what you read in this book, you need to *remember* what you read. And for that, you've got to *understand* it. To get the most from this book, or *any* book or learning experience, take responsibility for your brain. Your brain on *this* content.

The trick is to get your brain to see the new material you're learning as Really Important. Crucial to your well-being. As important as a tiger. Otherwise, you're in for a constant battle, with your brain doing its best to keep the new content from sticking.

So just how *DO* you get your brain to treat Android development like it was a hungry tiger?

There's the slow, tedious way, or the faster, more effective way. The slow way is about sheer repetition. You obviously know that you *are* able to learn and remember even the dullest of topics if you keep pounding the same thing into your brain. With enough repetition, your brain says, "This doesn't *feel* important to him, but he keeps looking at the same thing *over* and *over* and *over*, so I suppose it must be."

The faster way is to do ***anything that increases brain activity,*** especially different *types* of brain activity. The things on the previous page are a big part of the solution, and they're all things that have been proven to help your brain work in your favor. For example, studies show that putting words *within* the pictures they describe (as opposed to somewhere else in the page, like a caption or in the body text) causes your brain to try to makes sense of how the words and picture relate, and this causes more neurons to fire. More neurons firing = more chances for your brain to *get* that this is something worth paying attention to, and possibly recording.

A conversational style helps because people tend to pay more attention when they perceive that they're in a conversation, since they're expected to follow along and hold up their end. The amazing thing is, your brain doesn't necessarily *care* that the "conversation" is between you and a book! On the other hand, if the writing style is formal and dry, your brain perceives it the same way you experience being lectured to while sitting in a roomful of passive attendees. No need to stay awake.

But pictures and conversational style are just the beginning…

Here's what WE did

We used ***pictures***, because your brain is tuned for visuals, not text. As far as your brain's concerned, a picture really *is* worth a thousand words. And when text and pictures work together, we embedded the text *in* the pictures because your brain works more effectively when the text is *within* the thing it refers to, as opposed to in a caption or buried in the body text somewhere.

We used ***redundancy***, saying the same thing in *different* ways and with different media types, and *multiple senses*, to increase the chance that the content gets coded into more than one area of your brain.

We used concepts and pictures in ***unexpected*** ways because your brain is tuned for novelty, and we used pictures and ideas with at least *some* ***emotional*** *content*, because your brain is tuned to pay attention to the biochemistry of emotions. That which causes you to *feel* something is more likely to be remembered, even if that feeling is nothing more than a little ***humor***, ***surprise***, or ***interest.***

We used a personalized, ***conversational style***, because your brain is tuned to pay more attention when it believes you're in a conversation than if it thinks you're passively listening to a presentation. Your brain does this even when you're *reading*.

We included ***activities***, because your brain is tuned to learn and remember more when you ***do*** things than when you *read* about things. And we made the exercises challenging-yet-doable, because that's what most people prefer.

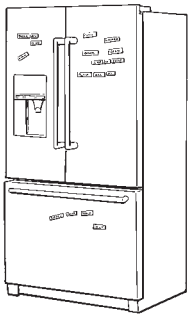
We used ***multiple learning styles***, because *you* might prefer step-by-step procedures, while someone else wants to understand the big picture first, and someone else just wants to see an example. But regardless of your own learning preference, *everyone* benefits from seeing the same content represented in multiple ways.

We include content for ***both sides of your brain***, because the more of your brain you engage, the more likely you are to learn and remember, and the longer you can stay focused. Since working one side of the brain often means giving the other side a chance to rest, you can be more productive at learning for a longer period of time.

And we included ***stories*** and exercises that present ***more than one point of view,*** because your brain is tuned to learn more deeply when it's forced to make evaluations and judgments.

We included ***challenges***, with exercises, and by asking ***questions*** that don't always have a straight answer, because your brain is tuned to learn and remember when it has to *work* at something. Think about it—you can't get your *body* in shape just by *watching* people at the gym. But we did our best to make sure that when you're working hard, it's on the *right* things. That ***you're not spending one extra dendrite*** processing a hard-to-understand example, or parsing difficult, jargon-laden, or overly terse text.

We used ***people***. In stories, examples, pictures, etc., because, well, *you're* a person. And your brain pays more attention to *people* than it does to *things*.



Here's what YOU can do to bend your brain into submission

So, we did our part. The rest is up to you. These tips are a starting point; listen to your brain and figure out what works for you and what doesn't. Try new things.

Cut this out and stick it on your refrigerator.

1 Slow down. The more you understand, the less you have to memorize.

Don't just *read*. Stop and think. When the book asks you a question, don't just skip to the answer. Imagine that someone really *is* asking the question. The more deeply you force your brain to think, the better chance you have of learning and remembering.

2 Do the exercises. Write your own notes.

We put them in, but if we did them for you, that would be like having someone else do your workouts for you. And don't just *look* at the exercises. **Use a pencil.** There's plenty of evidence that physical activity *while* learning can increase the learning.

3 Read "There Are No Dumb Questions."

That means all of them. They're not optional sidebars, ***they're part of the core content!*** Don't skip them.

4 Make this the last thing you read before bed. Or at least the last challenging thing.

Part of the learning (especially the transfer to long-term memory) happens *after* you put the book down. Your brain needs time on its own, to do more processing. If you put in something new during that processing time, some of what you just learned will be lost.

5 Talk about it. Out loud.

Speaking activates a different part of the brain. If you're trying to understand something, or increase your chance of remembering it later, say it out loud. Better still, try to explain it out loud to someone else. You'll learn more quickly, and you might uncover ideas you hadn't known were there when you were reading about it.

6 Drink water. Lots of it.

Your brain works best in a nice bath of fluid. Dehydration (which can happen before you ever feel thirsty) decreases cognitive function.

7 Listen to your brain.

Pay attention to whether your brain is getting overloaded. If you find yourself starting to skim the surface or forget what you just read, it's time for a break. Once you go past a certain point, you won't learn faster by trying to shove more in, and you might even hurt the process.

8 Feel something.

Your brain needs to know that this *matters*. Get involved with the stories. Make up your own captions for the photos. Groaning over a bad joke is *still* better than feeling nothing at all.

9 Write a lot of code!

There's only one way to learn to develop Android apps: **write a lot of code**. And that's what you're going to do throughout this book. Coding is a skill, and the only way to get good at it is to practice. We're going to give you a lot of practice: every chapter has exercises that pose a problem for you to solve. Don't just skip over them—a lot of the learning happens when you solve the exercises. We included a solution to each exercise—don't be afraid to **peek at the solution** if you get stuck! (It's easy to get snagged on something small.) But try to solve the problem before you look at the solution. And definitely get it working before you move on to the next part of the book.

Read me

This is a learning experience, not a reference book. We deliberately stripped out everything that might get in the way of learning whatever it is we're working on at that point in the book. And the first time through, you need to begin at the beginning, because the book makes assumptions about what you've already seen and learned.

We assume you're new to Android, but not to Kotlin.

You're going to learn how to build Android apps using a combination of Kotlin and XML. We assume that you're familiar with the Kotlin programming language, or another object-oriented language such as Java. If you've never done any Kotlin programming *at all,* then you might want to read *Head First Kotlin* before you start on this one.

We start off by building an app in the very first chapter.

Believe it or not, even if you've never developed for Android before, you can jump right in and start building apps. Along the way, you'll learn your way around Android Studio, the official IDE for Android development.

The examples are designed for learning.

As you work through the book, you'll build a number of different apps. Some of these are very small so you can focus on a specific part of Android. Other apps are larger so you can see how different components fit together. We won't complete every part of every app, but feel free to experiment and finish them yourself. It's all part of the learning experience.

We show you the code in context.

We know how frustrating it can be to be shown a piece of code out of context, with no explanation of how it works, or how to use it in your own projects. In this book, we'll show you small snippets of code and explain what each bit does and why. We'll then show you how the code works in the context of a complete project. You can even examine all the book's source code by downloading it from here: *https://tinyurl.com/hfad3*.

The activities are NOT optional.

The exercises and activities are not add-ons; they're part of the core content of the book. Some of them are to help with memory, some are for understanding, and some will help you apply what you've learned. ***Don't skip the exercises.***

The redundancy is intentional and important.

One distinct difference in a Head First book is that we want you to *really* get it. And we want you to finish the book remembering what you've learned. Most reference books don't have retention and recall as a goal, but this book is about *learning*, so you'll see some of the same concepts come up more than once and in different ways.

The Brain Power exercises don't have answers.

For some of them, there is no right answer, and for others, part of the learning experience of the Brain Power activities is for you to decide if and when your answers are right. In some of the Brain Power exercises, you will find hints to point you in the right direction.

The truly awesome technical review team

Technical reviewers:

Jacqui Cope started coding to avoid school netball practice. Since then, she has gathered considerable experience working with a variety of financial and education systems, from coding in COBOL to test management. She has since gained her MSc in Computer Security and has moved into Quality Assurance in the higher education sector. In her spare time, Jacqui likes to cook, walk in the countryside, and watch *Doctor Who* from behind the sofa.

Ingo Krotzky has been working in a variety of roles in the healthcare industry, mostly for Contract Research Organizations conducting clinical trials—a systems architect, DBA and database programmer, software developer and data engineer. In his spare time, he enjoys living amidst the (mostly not-so-wild) wildlife in the countryside; pair programming with squirrels, jaybirds, and finches; and exploring the hottest new mobile frameworks.

Ken Kousen is a Java Champion, Oracle Groundbreaker Ambassador, and Grails Rock Star. He is the author of the Pragmatic Library book *Help Your Boss Help You*, the O'Reilly books *Kotlin Cookbook*, *Modern Java Recipes*, and *Gradle Recipes for Android*, and the Manning book *Making Java Groovy*. He has recorded over a dozen video courses for the O'Reilly Learning Platform, covering topics related to Android, Spring, Java, Groovy, Grails, and Gradle. He is also a multiple-time winner of the JavaOne Rockstar award. His academic background includes BS degrees in Mechanical Engineering and Mathematics from MIT, an MA and PhD in Aerospace Engineering from Princeton, and an MS in Computer Science from RPI He is currently President of Kousen IT, Inc., based in Connecticut.

Ash Tappin is a polyglot software developer who works primarily on web applications. He gets very enthusiastic about building new things that make people's lives easier. When not coding, he enjoys practicing guitar, listening to music, running, cycling, exploring the outdoors, and gardening.

Acknowledgments

Our editor:

We are immensely grateful to our wonderful editor **Virginia Wilson** for picking up the reins on the third edition of this book. She's been a delight to work with, and gave us invaluable feedback and insights. Her amazing organization and herding skills helped keep this book on track, and she worked hard to make sure we had everything we needed exactly when we needed it. We've truly appreciated her hard work and support.

The O'Reilly team:

A big thank you goes to **Zan McQuade** for asking us to write the third edition of this book, and for all her support; **Shira Evans** and **Nicole Taché** for giving us extra editorial feedback; the **design team** for their amazing new artwork, and for helping us to replace the old elements; and **Katie Tozer** and **Kristen Brown** for making early releases of the book available. Finally, thanks go to the **production team** for expertly steering the book through the production process, and for working so hard behind the scenes.

Family, friends, and colleagues:

Writing a Head First book is always a rollercoaster of a ride, and the third edition of this book has been no exception. We've truly valued the kindness and support of our family and friends along the way. Special thanks go to **Mum**, **Dad**, **Rob**, **Lorraine**, **Mark**, **Laura**, **Andy**, **Aisha**, **Andy**, **Matti**, **Ian**, **Vanessa**, **Dawn**, **William,** and **Simon**.

The without-whom list:

Our awesome technical review team worked hard to give us their thoughts on the book, keep us straight, and make sure that what we covered was spot on. We're also grateful to everyone who gave us feedback on the early release of this book, and the previous two editions. We think the book's much better as a result.

Finally, our thanks to **Kathy Sierra** and **Bert Bates** for creating this extraordinary series of books, teaching us to throw away the old rulebook, and letting us into their brains.

O'Reilly Online Learning

For more than 40 years, O'Reilly Media has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

1 getting started

Diving In

Android is the world's most popular mobile operating system.

And there are billions of Android users worldwide, all waiting to download your next great idea. In this chapter, you'll find out how to start turning your ideas into reality by **building a basic Android app**, and updating it. You'll learn how to run it on physical and virtual devices. Along the way, you'll meet two of the core components of all Android apps: **activities** and **layouts**. Let's dive in…

Welcome to Androidville

Android is the world's most popular mobile platform. At last count, there were over *three billion* active Android devices worldwide, and that number is still growing.

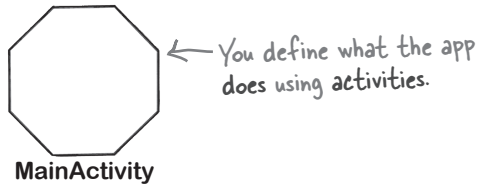
Android is a comprehensive open source platform based on Linux and championed by Google. It's a powerful development framework that includes everything you need to build great apps. What's more, it enables you to deploy those apps to a wide variety of devices—phones, tablets, and more.

So what makes up a typical Android app?

We're going to build Android apps using Kotlin and XML. We'll explain things along the way, but you'll need some understanding of Kotlin to get the most out of this book.

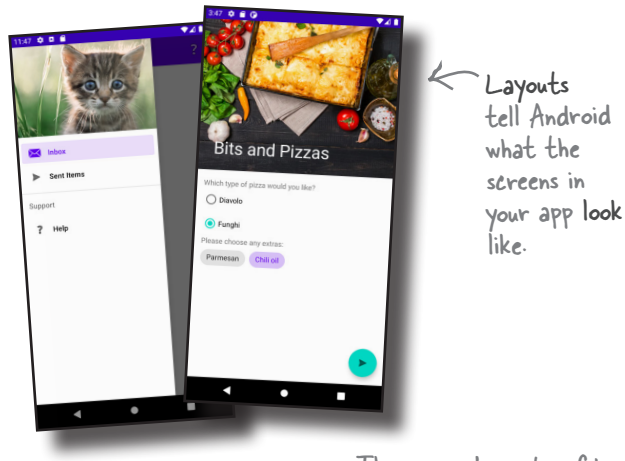
Activities define what the app does

Each Android app includes one or more **activities**. An activity is a special class —usually written in Kotlin—that controls the app's behavior, and decides how to respond to the user. If the app includes a button, for example, you add code to the activity that says what the button should do when the user taps or clicks on it.



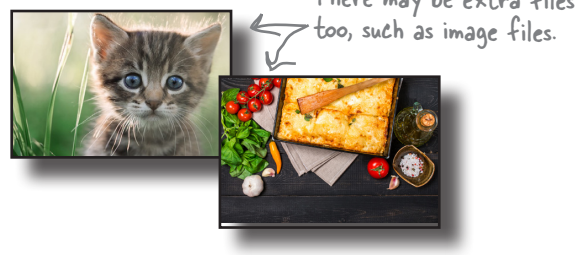
Layouts define what each screen looks like

A typical Android app is composed of one or more screens. You define what each screen looks like using a **layout** file, or more activity code. Layouts are usually defined using XML, and each screen can include components such as buttons, text, and images.



There may be extra files too

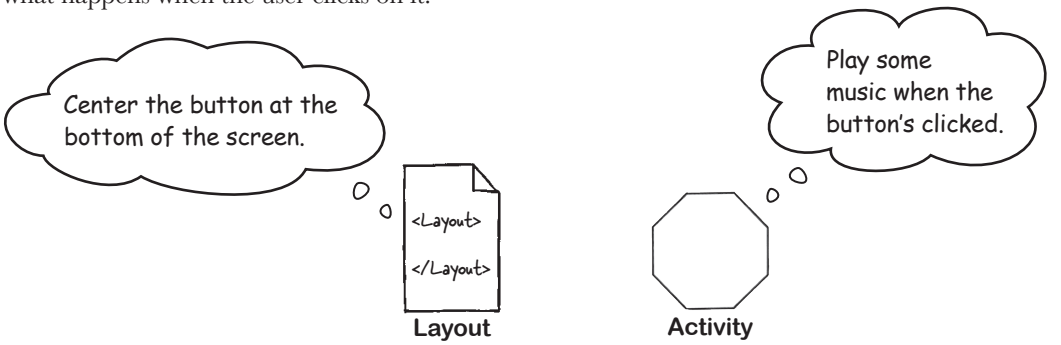
In addition to activities and layouts, Android apps often need extra resources such as image files and application data. You can add any extra files you need to the app.



Android apps are really just a bunch of files in particular directories. When you build your app, these files get bundled together, giving you an app you can run on your device.

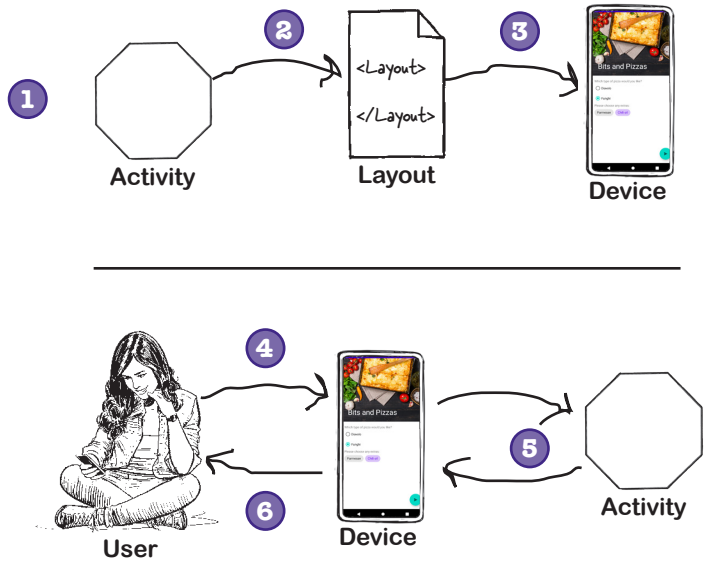
Activities and layouts form the backbone of your app

In a typical app, activities and layouts work together to define the app's user interface. The layouts tell Android how the different screen elements should be arranged, and the activities control the app's behavior. If the app features a button, for example, the layout specifies its position, and the activity controls what happens when the user clicks on it.



Here's how activities and layouts work together when you run an app on your device:

1. Android starts the app's main activity.
2. The activity tells Android to use a specific layout.
3. The layout is displayed on the device.
4. The user interacts with the layout.
5. The activity responds to these interactions, and updates the display…
6. …which the user sees on the device.



Now that you have an idea about how Android apps are put together, let's go ahead and build a basic Android app.

Here's what we're going to do

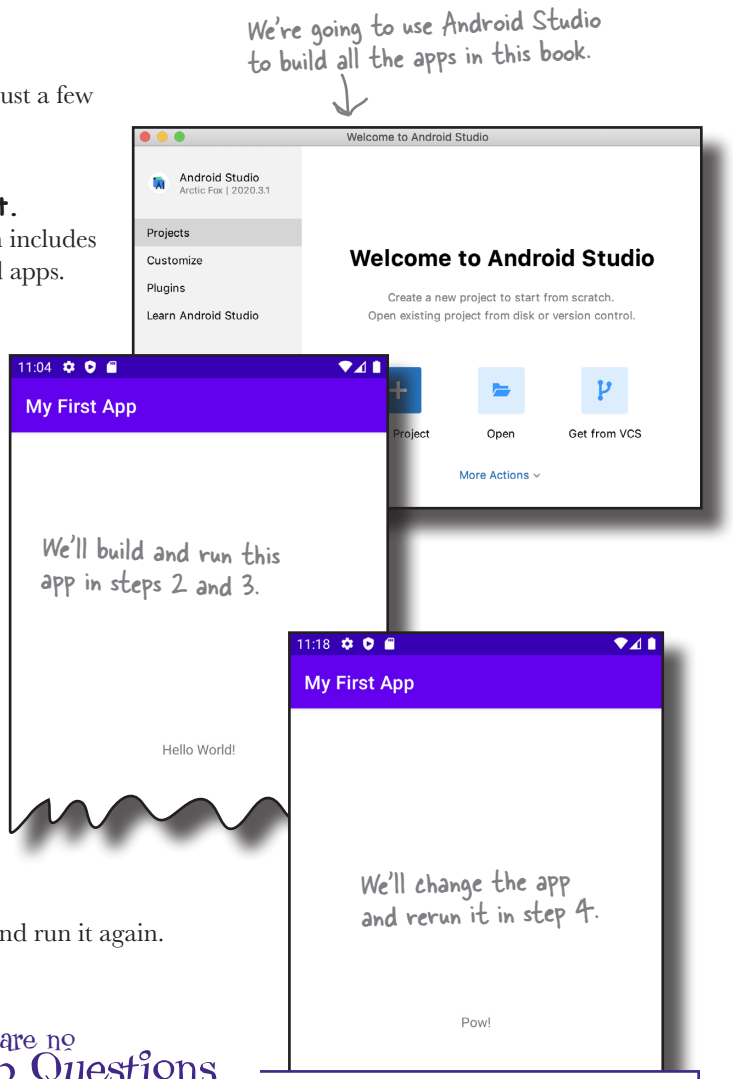
So let's dive in and create an Android app. There are just a few things we need to do:

1. **Set up a development environment.**
We need to install Android Studio, which includes all the tools you need to develop Android apps.

2. **Build a basic app.**
We'll build a simple app using Android Studio that will display some sample text on the screen.

3. **Run the app.**
We'll run the app on a physical device, and a virtual one, to see it up and running.

4. **Change the app.**
Finally, we'll tweak the app we created, and run it again.



there are no Dumb Questions

Q: Are all Android apps developed in Kotlin?

A: You can develop Android apps in other languages too, such as Java, but nowadays it's best if you use Kotlin. This is because there are some features that are only available in Kotlin.

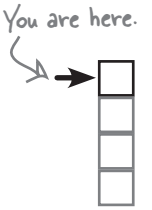
Q: How do I learn Kotlin?

A: We may be biased, but we think that the best way of learning Kotlin is to get a copy of our book *Head First Kotlin*. It will teach you everything you need to know about Kotlin in order to get the most out of this book.

Q: Did you say you can use activity code to define the app's appearance instead of using layout files?

A: Yes, using a toolkit called Jetpack Compose.

You'll learn how to use Compose later in the book. For now, we're going to focus on defining the app's UI using layout files.



Android Studio: your development environment

The best way of developing Android apps is to use **Android Studio**, the official IDE for Android app development.

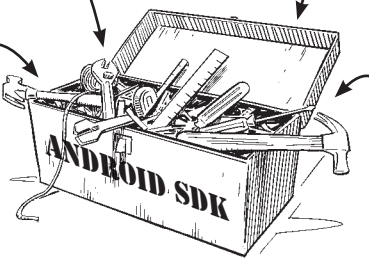
Android Studio is based on IntelliJ IDEA, which you may already be familiar with. It includes a set of code editors, UI tools and templates, which are all designed to make your life in Androidville easier.

It also includes the **Android SDK**—or Android Software Development Kit—which is required for all Android app development. The Android SDK includes Android source files, and a compiler that's used to compile your code into an Android format.

Here are some of the main components of the Android SDK:

SDK Tools
This includes the complete set of tools used to develop and debug Android apps.

SDK Build Tools
This contains the tools that are required for building Android apps.



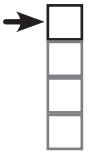
SDK Platform
Each version of Android has its own SDK Platform package. It lets you compile your app for that version of Android, and includes the source files for that version.

SDK Platform Tools
This includes tools that interface with the Android platform. The tools are backward compatible, but new features may only be available for new versions of Android.

You need to install Android Studio

As Android Studio includes all the tools and features you need in order to develop Android apps, we're going to use it to build all of the apps featured in this book.

Before we go any further, **you need to install Android Studio on your machine**. There are more details about how to do this on the next page.



Set up environment
Build app
Run app
Change app

Install Android Studio

To get the most out of this book, you need to install Android Studio. We're not including the full installation instructions here as they can get out of date pretty quickly, but you'll be fine if you follow the online instructions.

First, check the Android Studio system requirements here:

https://developer.android.com/studio#Requirements

Then download Android Studio from here:

https://developer.android.com/studio

These URLs sometimes change. If they don't work, search for Android Studio and you should find the appropriate pages.

and follow the installation instructions

Once you've installed Android Studio, open it and follow the instructions to add the latest SDK tools and Support Libraries.

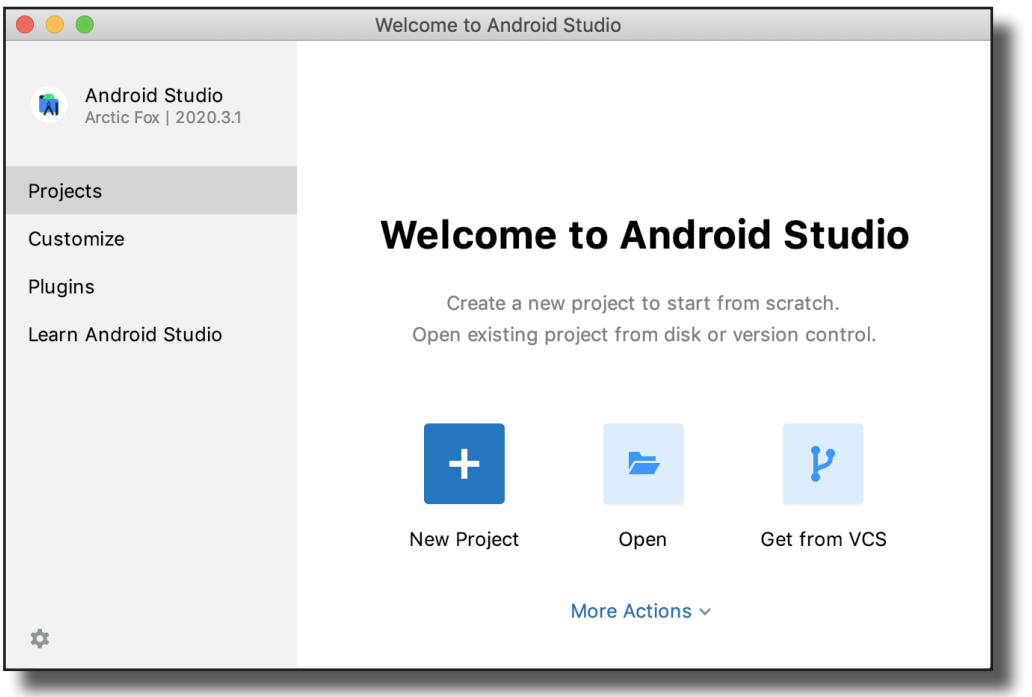
If you've previously installed an earlier version of Android Studio, we recommend that you restore the IDE's default settings. To do this, go to the File menu, choose Manage IDE Settings, and then select the Restore Default Settings option.

This resets any old settings Android Studio might be holding on to that could stop your code from running.

When you're done, you should see the Android Studio welcome screen like this:

In this book, we're using Android Studio 2020.3.1 (known as Arctic Fox). Make sure you install at least this version.

This is the Android Studio welcome screen. It includes a set of options for things you can do.



You've completed this step now, so we've checked it off.

Let's build a basic app

Now that you've set up your development environment, you're ready to create your first Android app. Here's what it will look like:

This is the app you're going to build. It's very simple, but it's all you need for your very first Android app.

Let's go ahead and build the app.

there are no Dumb Questions

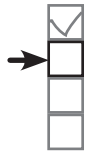
Q: Do I have to use Android Studio to build Android apps? Can I use another IDE instead?

A: Strictly speaking, all you need is a tool that will let you write and compile Kotlin code, plus a few other tools to convert the compiled code into a form that Android devices can run. Saying that, Android Studio is the official IDE for Android, and the Android team recommends using it. We think it's a great IDE for Android app development, so that's why we've decided to use it.

Q: Can I write Android apps without using an IDE?

A: It's possible, but it's a lot more work.

It's far quicker and easier to use an IDE, so we recommend that you use Android Studio.



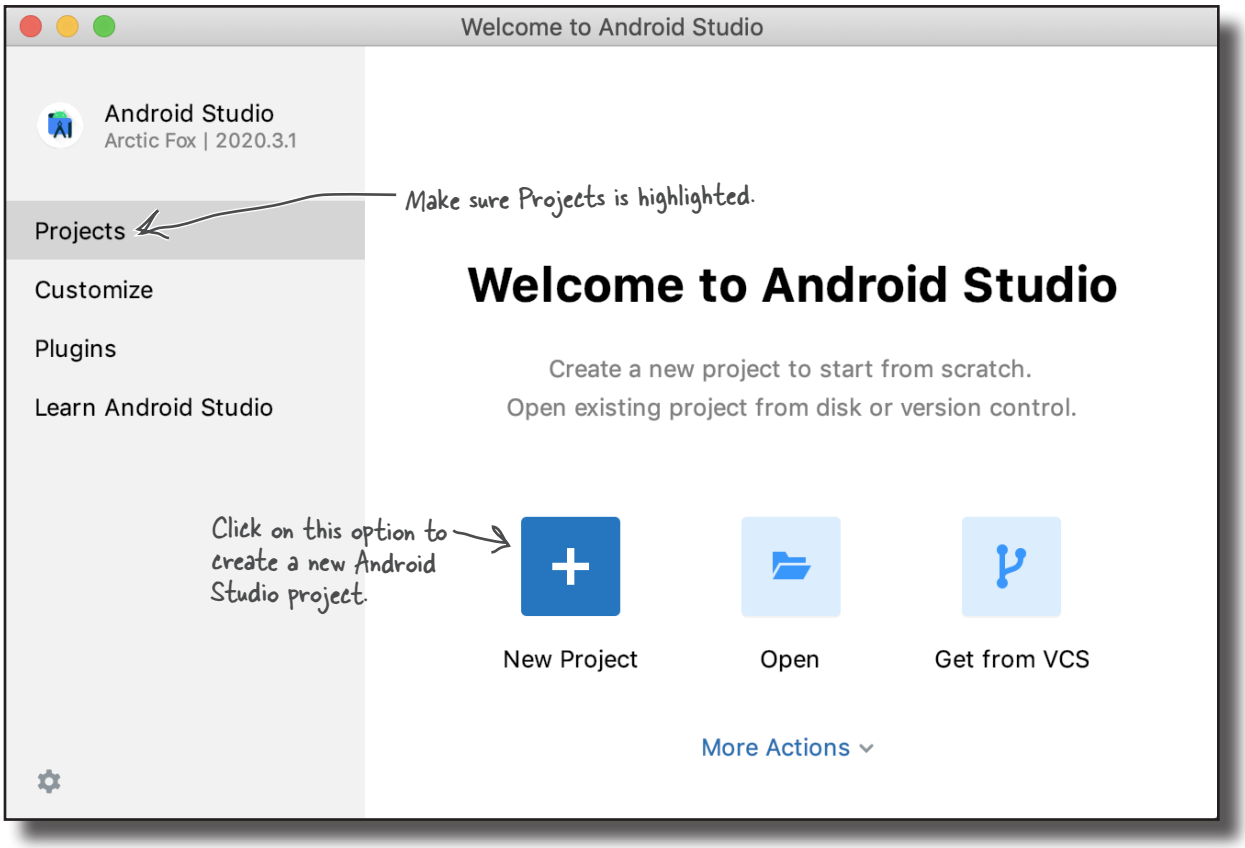
Set up environment
Build app
Run app
Change app

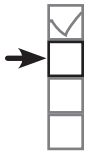
How to build the app

Whenever you create a new app, you need to create a new project for it. Make sure you have Android Studio open, and follow along with us.

1. Create a new project

The Android Studio welcome screen gives you a number of options. We want to create a new project, so make sure the Projects option is selected, then click on "New Project."

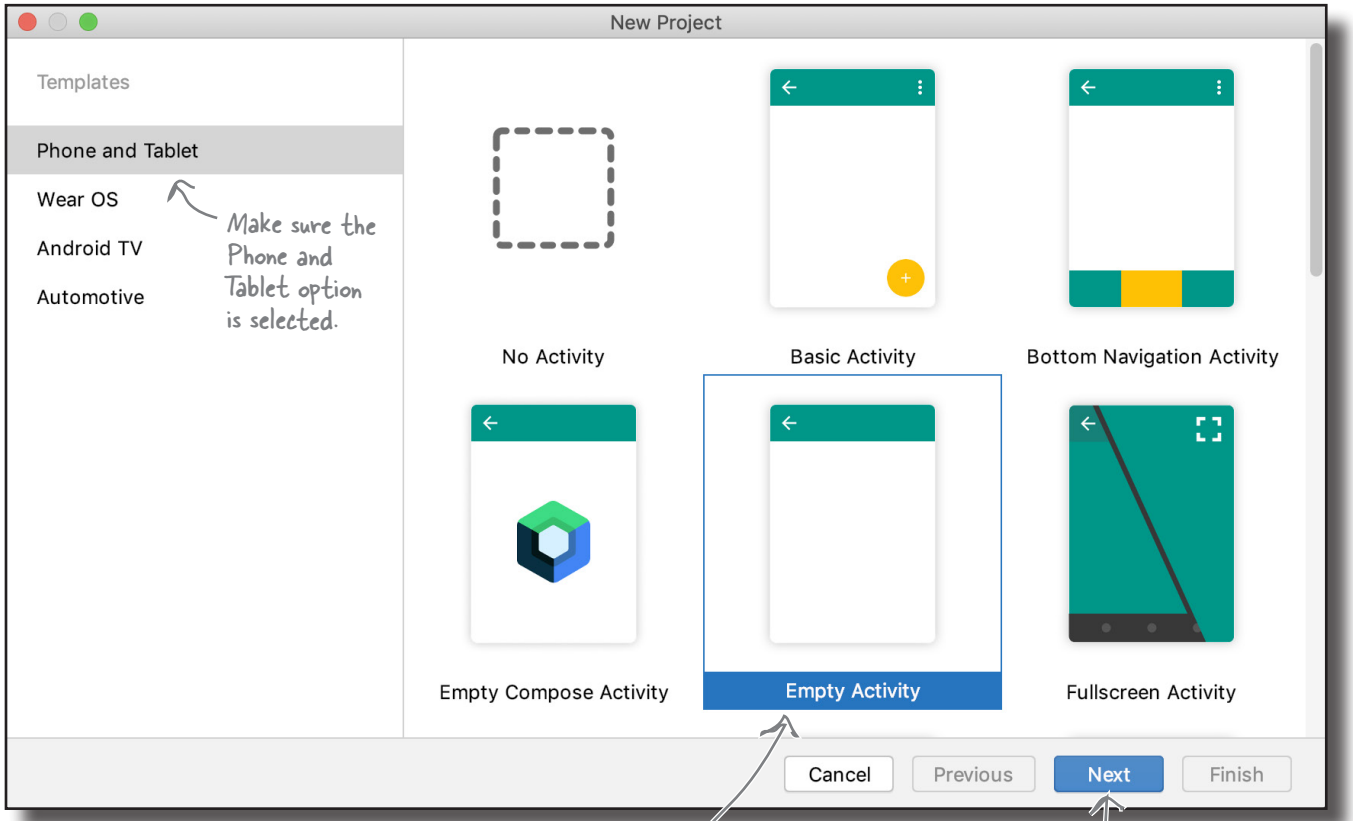


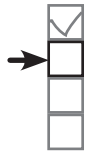


Set up environment
Build app
Run app
Change app

2. Select a project template

You next need to specify the type of Android Studio project you want to create. We're going to create an app with an empty activity that runs on a phone or tablet, so make sure the Phone and Tablet option is selected, and choose the Empty Activity option. You'll find out more about what the Empty Activity option gives you a few pages ahead, but for now, click the Next button to move to the next step.





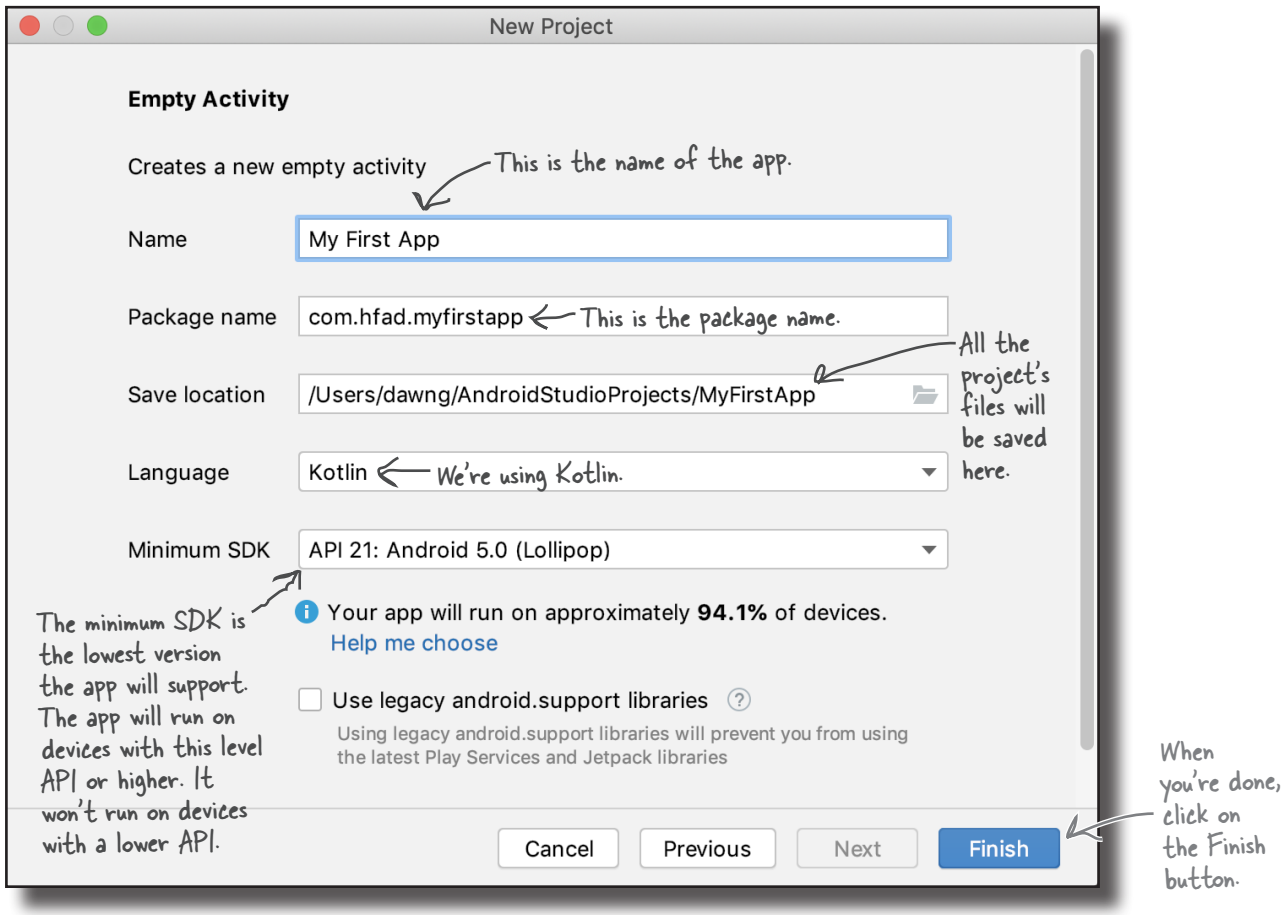
Set up environment
Build app
Run app
Change app

3. Configure your project

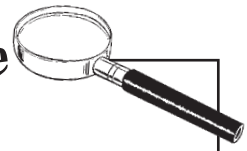
You now need to configure the project by specifying an app name, package name, and save location. Enter a name of "My First App" and a package name of "com.hfad.myfirstapp" and accept the default save location.

You also need to tell Android Studio which programming language you want to use, and specify a minimum SDK. This refers to the lowest version of Android the app will support: there's more about SDK levels on the next page.

Select Kotlin for the language and choose a minimum SDK of API 21 so that the app will run on most devices. When you click on the Finish button, Android Studio will create the project. We'll look at what this involves a couple of pages ahead.



Android Versions Up Close



You've probably heard a lot of tasty things about Android versions like Nougat, Oreo, and Pie. So what's with all the food?

Android versions have a version number and a code name. The version number gives the precise version of Android (e.g., 9.0), while the code name is a more generic name that may cover several versions of Android (e.g., Pie). The API level refers to the version of the APIs used by apps. As an example, the equivalent API level for Android version 9.0 is 28.

After Pie, Android code names stopped sounding quite so tasty. Android version 10.0, for example, is simply known as Android 10.

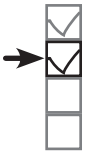
Version	Code name	API level
1.0		1
1.1	Petit Four	2
1.5	Cupcake	3
1.6	Donut	4
2.0–2.1	Eclair	5–7
2.2–2.2.3	Froyo	8
2.3–2.3.7	Gingerbread	9–10
3.0–3.2.6	Honeycomb	11–13
4.0–4.0.4	Ice Cream Sandwich	14–15
4.1–4.3.1	Jelly Bean	16–18
4.4–4.4.4	KitKat	19–20
5.0–5.1.1	Lollipop	21–22
6.0–6.0.1	Marshmallow	23
7.0–7.1.2	Nougat	24–25
8.0–8.1	Oreo	26–27
9.0	Pie	28
10.0	10	29
11.0	11	30
12.0	12	31

Hardly anyone uses these versions anymore.

Most devices use one of these APIs.

When you develop Android apps, you need to consider which versions of Android you want the app to be compatible with. If you specify that the app is only compatible with the very latest version of the SDK, you might find that it can't be run on many devices. You can find out more about the percentage of devices running particular versions by clicking on the "Help me choose" option when you create a new project.

You've created your first Android project



Set up environment
Build app
Run app
Change app

Once you've been through the New Project wizard, it takes Android Studio a minute or so to create the project. During this time, it does the following:

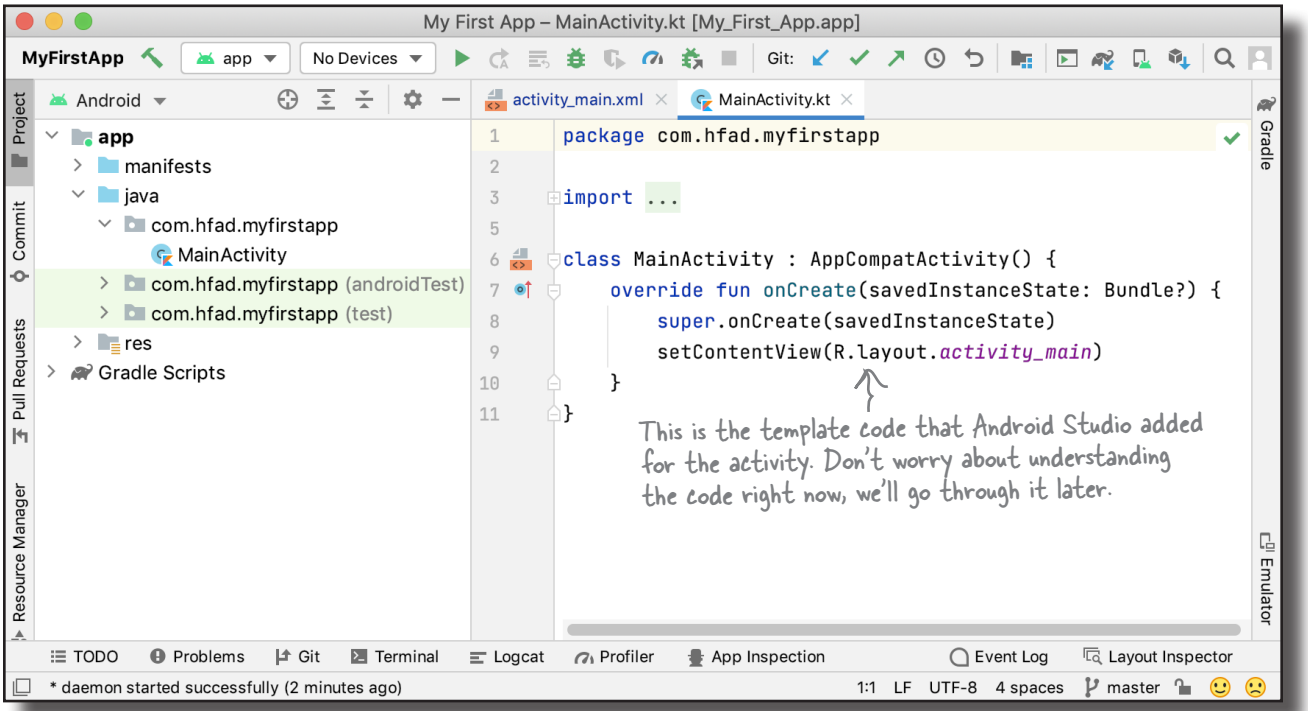
It configures the project to your specifications.
Android Studio looks at the minimum SDK you want the app to support, and includes all of the files and folders needed for a basic valid app. It also sets up the package structure and names the app.

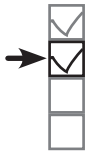
It adds some template code.
The template code is comprised of a layout written in XML and an activity written in Kotlin. You'll find out more about these later in the chapter.

When Android Studio has finished creating the project, it automatically opens it for you.

Here's what our project looks like (don't worry if it looks complicated—we'll break it down over the next few pages):

This is the project in Android Studio.





Set up environment
Build app
Run app
Change app

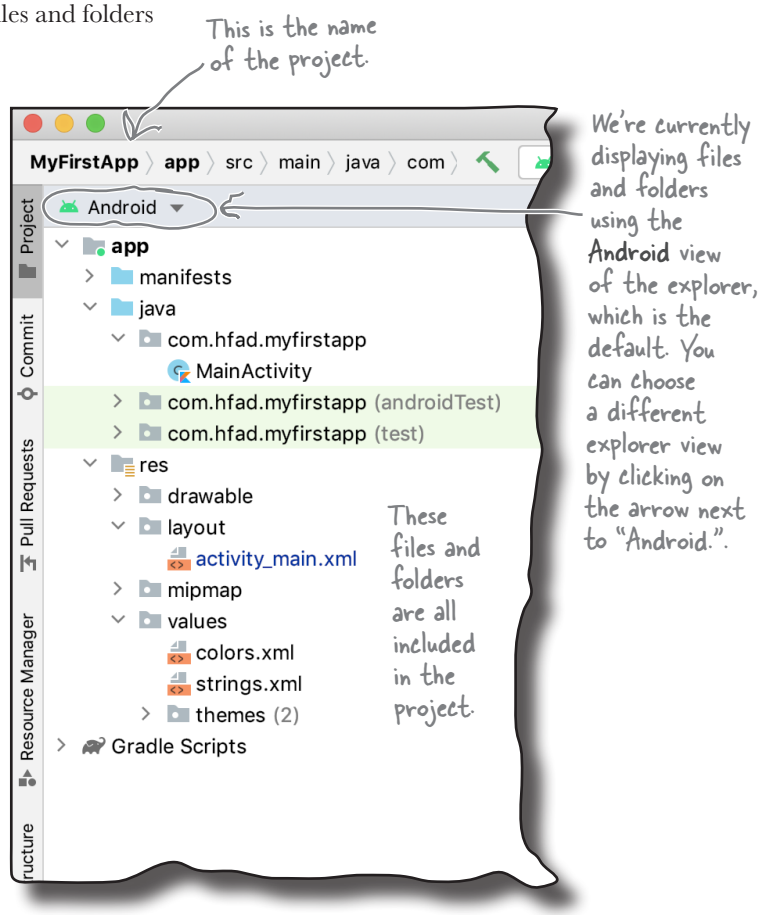
Dissecting your new project

An Android app is really just a bunch of valid files in a particular folder structure, and Android Studio sets all of this up for you when you create a new app. The easiest way of looking at this folder structure is to use the explorer in the leftmost column of Android Studio.

The folder structure includes different types of files

The explorer contains all of the projects that you currently have open. Here, we have a single project named MyFirstApp, which is the one we've just created.

If you browse through the various folders in the explorer, you'll see that the wizard has created various types of files and folders for you like this:



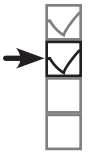
Kotlin and XML source files
Android Studio automatically created an activity file named *MainActivity.kt*, and a layout named *activity_main.xml*.

Resource files
These include default image files, themes the app might use, and any common `String` values used by the app.

Android libraries
In the wizard, you specified the minimum SDK you want the app to be compatible with. Android Studio makes sure the app includes the relevant Android libraries for that version.

Configuration files
The configuration files tell Android what's included in the app, and how the app should run.

Let's take a closer look at some of the key files and folders in your project.



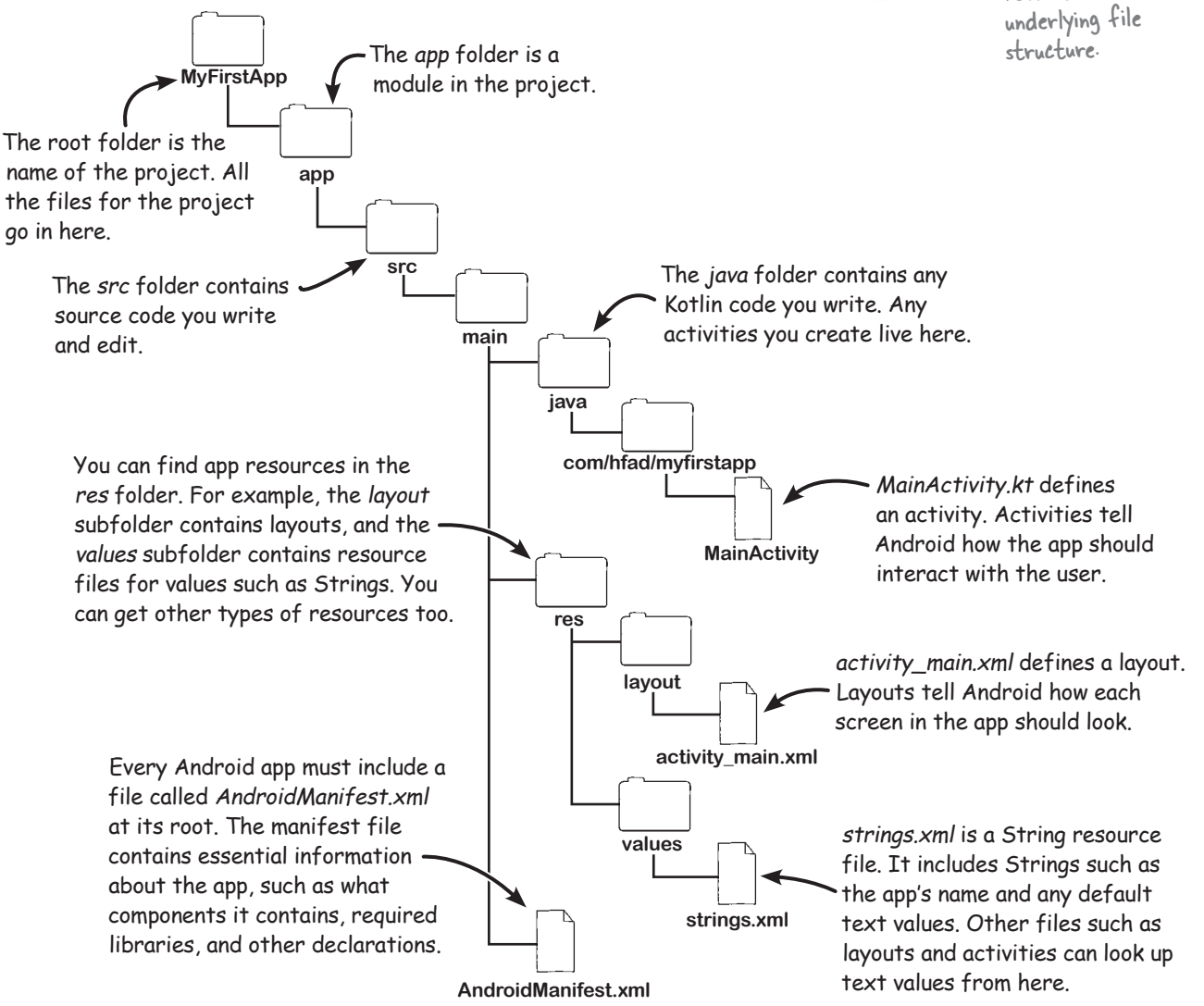
Set up environment
Build app
Run app
Change app

Introducing the key files in your project

Android Studio projects use the Gradle build system to compile and deploy apps, and Gradle projects have a standard structure. Below are some of the key files and folders in that structure you'll be working with.

To see this view of the folder structure, change the explorer view in Android Studio from Android to Project. You do this by clicking on the arrow at the top of the explorer pane, and selecting the Project option.

Use this arrow to change the explorer view. We usually use the Project view as this reflects the underlying file structure.



Set up environment
Build app
Run app
Change app

Edit code with the Android Studio editors

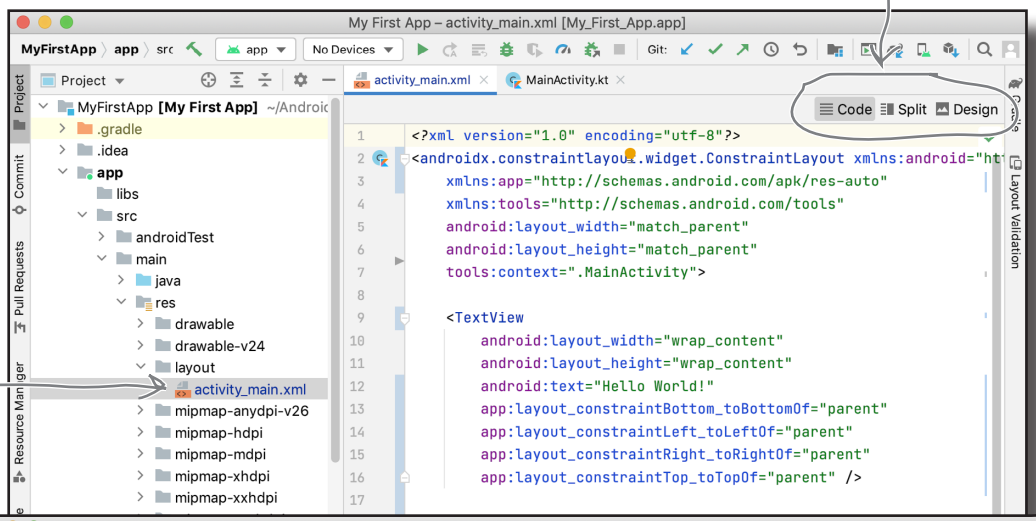
You view and edit files using the Android Studio editors. Double-click on the file you want to work with, and the file's contents will appear in the middle of the Android Studio window.

You dictate which editor you're using with these.

The code editor

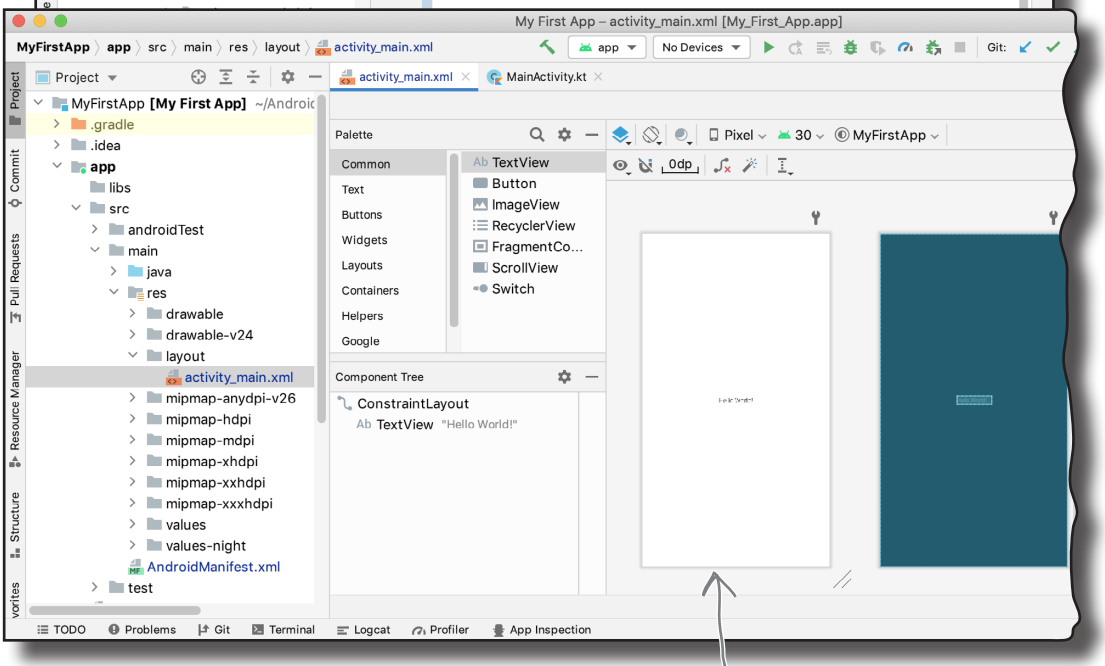
Most files get displayed in the code editor, which is just like a text editor, but with extra features such as syntax highlighting and code checking.

Double-click on the file in the explorer and the file contents appear in the editor panel.

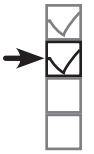


The design editor

If you're editing a layout (for example *activity_main.xml*), you have an extra option. Rather than edit the code, you can use the design editor, which allows you to drag GUI components onto the layout, and arrange them how you want. The code editor and design editor give different views of the same file, so you can switch back and forth between the two.



You can edit layouts using the visual editor by dragging and dropping components.



Set up environment
Build app
Run app
Change app

The story so far

In the chapter so far, we've done two things:

1. **We set up the development environment.**
   We're using Android Studio to develop Android apps, so you needed to install it on a machine.

2. **We've built a basic app.**
   We used Android Studio to create a new Android project.

You've had a glimpse of what the app looks like in Android Studio, and got a feel for how it hangs together. But what you *really* want is to see it running, right?

Android Studio lets you run an app in two ways: on a physical Android device, and on a virtual one. We'll show you each approach a few pages ahead.

there are no Dumb Questions

Q: Why does Android Studio put Kotlin code in a folder named *java*?

A: Before the Android team adopted Kotlin as their preferred language, most Android apps were developed using Java. The *java* folder name is left over from this time, but it may change in the future.

Q: You said that the project includes an activity named *MainActivity.kt* and a layout named *activity_main.xml*. Where did these come from?

A: When we created the project, we chose the option to create a project using an empty activity. Android Studio automatically named the activity file *MainActivity.kt*, and added a corresponding layout file named *activity_main.xml*. You'll find out more about these files later in the chapter.

Q: You said that Android Studio projects use Gradle. Remind me, what's that?

A: Gradle is a tool that lets you compile, build, and deliver code. Most IDEs use Gradle, and it can be used for more than just Android app development.

All Gradle projects have a standard folder structure, and Android Studio uses this folder structure for all its projects.

Q: Why does Android Studio use Gradle?

A: When you develop Android apps, you often need to include extra libraries that aren't included in the Android SDK. Gradle can download these libraries for you, which makes your coding life much easier.

Gradle also uses Groovy and Kotlin as a scripting language, which means you can easily create quite complex builds with Gradle.

Q: How much do I need to know about Gradle?

A: You don't need any Gradle experience to get the most out of this book. We'll explain everything you need to know when it's needed.

What's My Purpose?

Here's the code from an example activity file named *MainActivity.kt*. We know you've not seen activity code before, but see if you can match each of the descriptions at the bottom of the page to the correct lines of code. We've done one to get you started.

MainActivity.kt

```
package com.hfad.myfirstapp

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {

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

}
```

This is the package name.

These are Android classes used in `MainActivity`.

Specify which layout to use.

Implement the `onCreate()` method from the `AppCompatActivity` class. This method is called when the activity is first created.

`MainActivity` extends the class `AppCompatActivity`.

What's My Purpose? Solution

Here's the code from an example activity file named *MainActivity.kt*. We know you've not seen activity code before, but see if you can match each of the descriptions at the bottom of the page to the correct lines of code. We've done one to get you started.

MainActivity.kt

```
package com.hfad.myfirstapp

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {

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

}
```

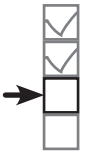
This is the package name.

These are Android classes used in `MainActivity`.

Specify which layout to use.

Implement the `onCreate()` method from the `AppCompatActivity` class. This method is called when the activity is first created.

`MainActivity` extends the class `AppCompatActivity`.



Set up environment
Build app
Run app
Change app

How to run the app on a physical device

If you have an Android device that's running Lollipop or above, you can use it to run the app we've just created.

You can check this by looking at the Android version in the device settings. Lollipop is version 5.0, so it needs to 5.0 or higher.

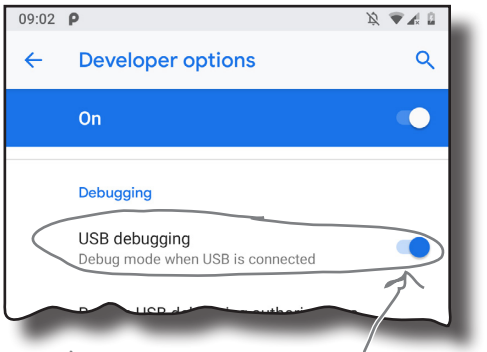
Here are the steps to run the app on a physical device:

1. Enable USB debugging on your device

To allow Android Studio to run apps on your device, you need to enable USB debugging. This feature is available in the "Developer options" setting, which is disabled by default.

Yep, seriously.

On your device, go to Settings → About Phone and tap the build number seven times. This enables the developer options. Then, go to Settings → System → Advanced → Developer options, and turn on USB debugging.



You need to enable USB debugging.

2. Set up your computer to detect the device

If you're using a Mac, you can skip this step.

If you're using Windows, you'll need to install a USB driver if one isn't already there. The latest instructions are here:

https://developer.android.com/studio/run/oem-usb

If you're using Ubuntu Linux, you need to create a `udev` rules file. The latest instructions on how to do this are here:

https://developer.android.com/studio/run/device#setting-up

3. Use a USB cable to plug your device into your computer

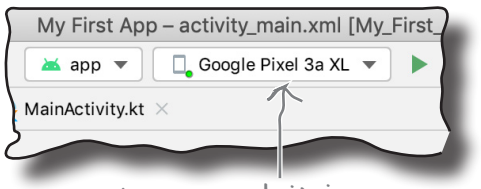
You will probably be asked if want to allow USB debugging. If so, check the "Always allow from this computer" option and choose OK.

4. Run the app

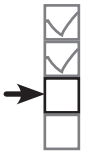
Finally, select the device from the list of devices in Android Studio's top toolbar (if it's not there, on your device go to Settings → Connected devices, select USB, and choose the "File transfer" option). Then run the app by choosing "Run 'app'" from the Run menu. Android Studio will build the project, install the app on your device, and launch it.

We'll look at the app a few pages ahead, after we've seen how to run it on a virtual device.

When we created the app, we specified a minimum API level of 21 (Lollipop). Your device needs this version of Android or above for the app to run.



Make sure your device is selected before running the app. Here, we're using a Pixel 3a XL.



Set up environment
Build app
Run app
Change app

How to run the app on a virtual device

If you don't have an Android device on hand, or it doesn't have the right version of Android, you can run the app on a virtual device instead. Running the app on a virtual device is useful if you want to see how the app looks on a type of device you don't own, or test how it behaves on a different version of Android.

The Android SDK features a built-in emulator that you can use to set up one or more **Android Virtual Devices** (AVDs). Once you've set up an AVD, you can run the app on it as though it's running on a physical device.

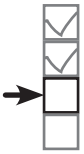
You can find system requirements for using the emulator here: https://developer.android.com/studio/run/emulator#requirements

The emulator recreates the exact hardware environment of an Android device: from its CPU and memory through to the sound chips and the video display. The emulator is built on an existing emulator called QEMU (pronounced "queue em you"), which is similar to other virtual machine applications you may have used, like VirtualBox or VMWare.

The exact appearance and behavior of the AVD depends on how you set it up. If, say, you create an AVD based on a Pixel 3 running Android 11, it will look and behave just like a Pixel 3 running this version of Android on your computer.

This is an AVD that's running in the Android emulator. It's based on a Pixel 3 device running Android 11. It has the same device specifications, and runs just like the physical device.

Let's set up an AVD so that you can see the app running in the emulator.



Set up environment
Build app
Run app
Change app

Create an Android Virtual Device (AVD)

There are a few steps you need to go through in order to set up an AVD within Android Studio. We'll set up a Pixel 3 AVD running API level 30 (Android 11) so that you can see how the app looks and behaves running on this type of device. The steps are pretty much identical no matter what type of virtual device you want to set up.

Open the Android Virtual Device Manager

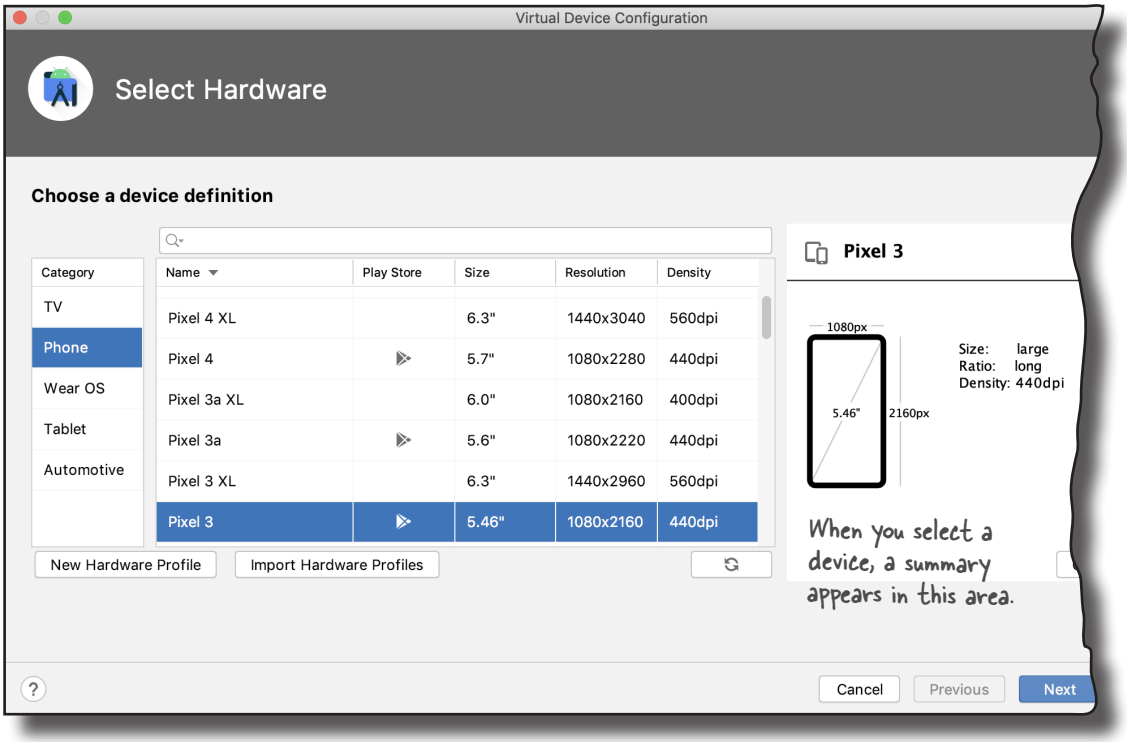
The AVD Manager allows you to set up new AVDs, and view and edit ones you've already created. Open it by selecting AVD Manager on the Tools menu.

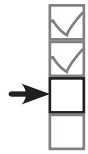
If you have no AVDs set up already, you'll be presented with a screen prompting you to create one. Click on the Create Virtual Device button.

Select the hardware

On the next screen, you'll be prompted to choose a device definition. This is the type of device your AVD will emulate. You can choose a variety of phone, tablet, wear, or TV devices.

We're going to see what the app looks like running on a Pixel 3 phone. Choose Phone from the Category menu and Pixel 3 from the list. Then click on the Next button.





Set up environment
Build app
Run app
Change app

Select a system image

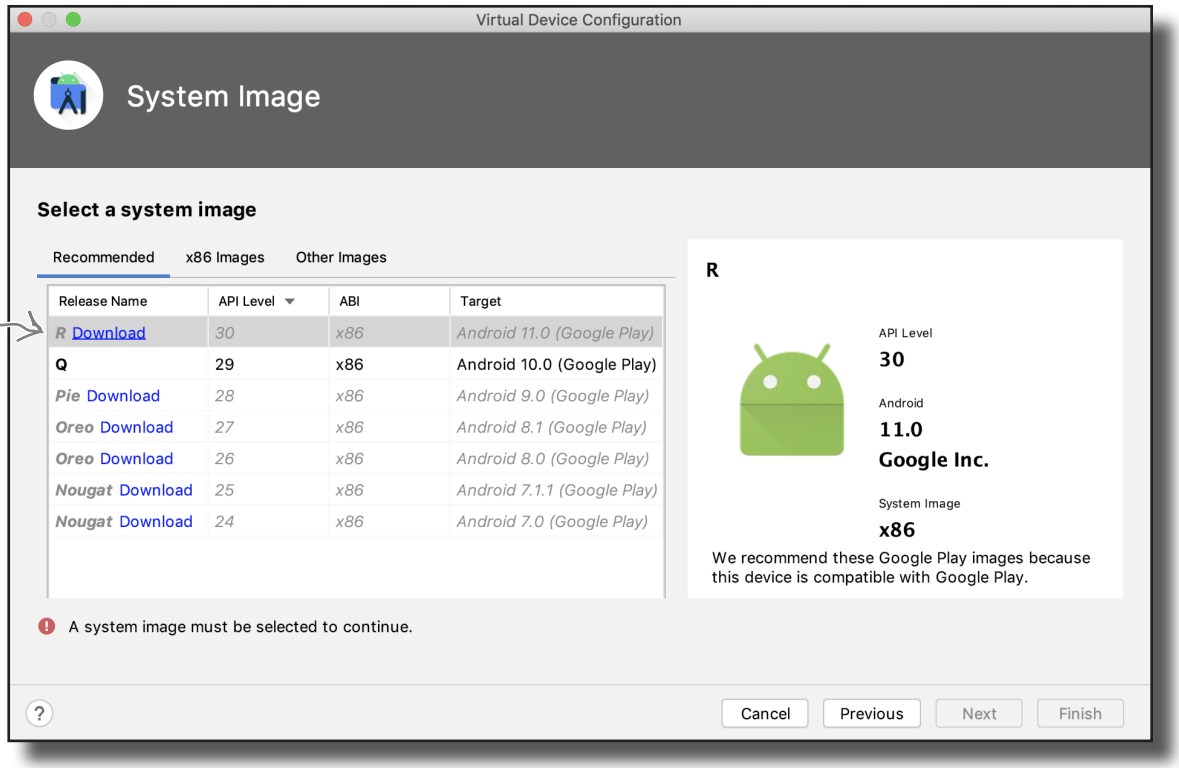
Next, you need to select a system image. This specifies which version of Android you want to be on the AVD.

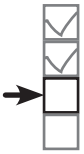
You need to choose a version of Android that's compatible with the app you're building. It must be *at least* the minimum SDK that the app supports.

When you created your Android project, you specified that the minimum SDK is API level 21. This means that you need to choose a system image that's for API level 21 (Lollipop) or above. If you choose an older version of Android than this, the app won't be able to run on the device.

Here, we're going to see what the app looks like on a relatively new version of Android, so choose the system image with a release name of R and a target of Android 11.0 (API level 30). Then click Next.

If you don't have this system image installed, you'll be presented with a Download link. Click on this link to download and install the system image.

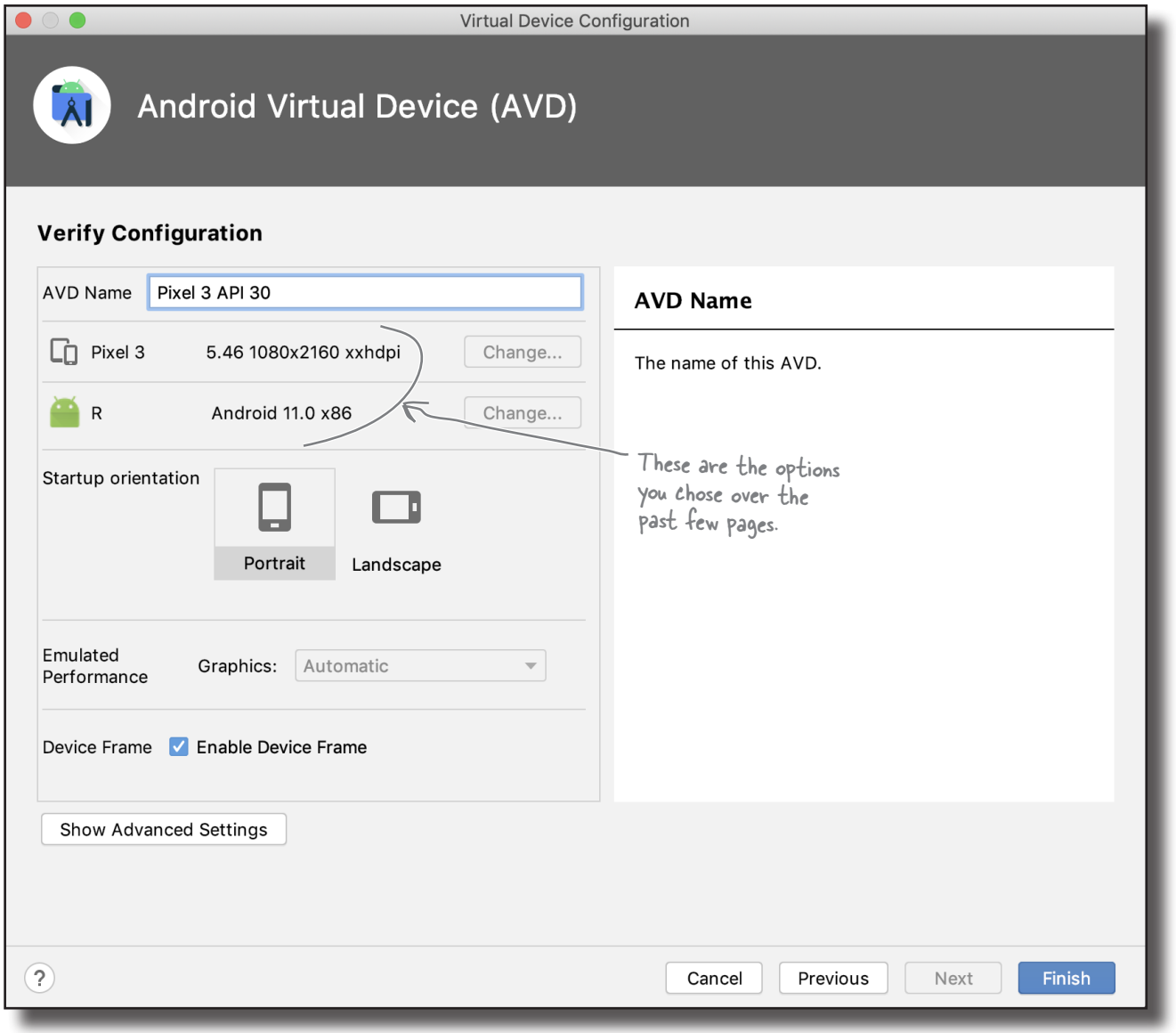


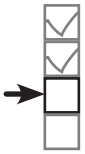


Set up environment
Build app
Run app
Change app

Verify the AVD configuration

On the next screen, you'll be asked to verify the configuration. This screen summarizes the options you chose over the last few screens, and gives you the option of changing them. Accept the options, and click on the Finish button.

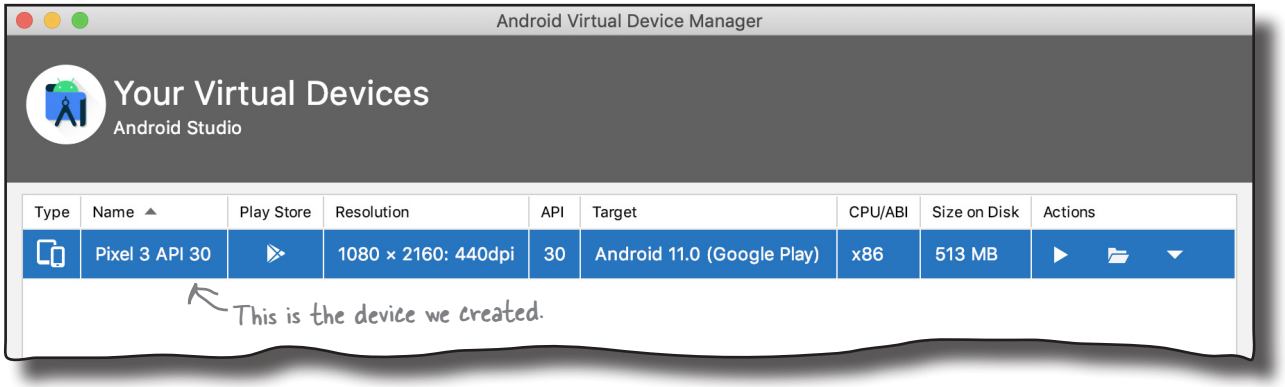




Set up environment
Build app
Run app
Change app

The virtual device gets created

When you click on the Finish button, the Device Manager creates the virtual device for you, and displays it in the AVD Manager's list of virtual devices like this:



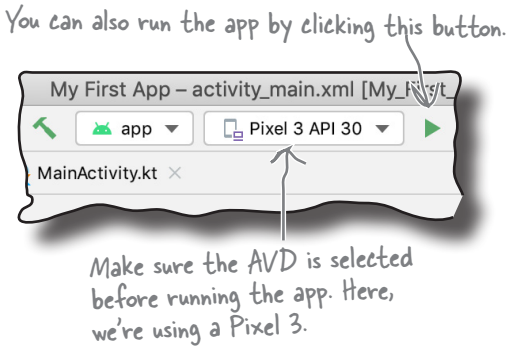
Check the new AVD is listed, then close the AVD Manager.

Run the app on the AVD

Once you've created the AVD, you can run the app on it.

To run the app, make sure that the virtual device is selected in the list of devices in Android Studio's top toolbar, then run the app by choosing the "Run 'app'" command from the Run menu.

The AVD can take a while to load, so while we wait, let's take a look at what happens behind the scenes when you use the Run command.



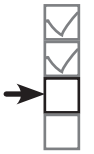
there are no Dumb Questions

Q: Do I need to create a new AVD each time I create a new app?

A: No, once you've created the AVD you can use it to run any app that can run that version of Android.

Q: Can I create multiple AVDs?

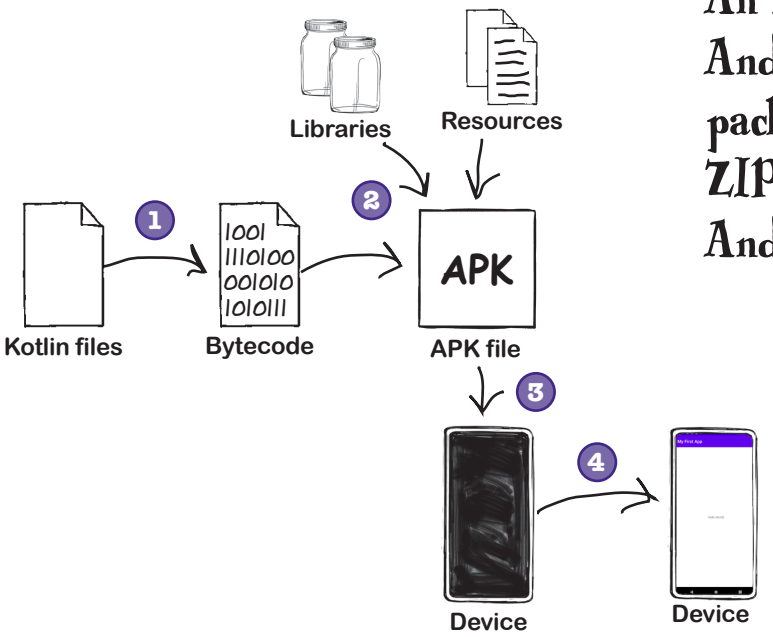
A: Yes! You can set up AVDs to emulate different devices or Android versions, including Android beta releases. This lets you test how the app runs on each one. You might, say, want to create a tablet AVD so you can see how the app looks and behaves on larger devices.



Compile, package, deploy, run

The Run command doesn't just run your app. It also handles all the preliminary tasks that are needed for the app to run.

Here's an overview of what happens:



An APK file is an Android application package. It's like a ZIP or JAR file for Android applications.

1. **The Kotlin source files get compiled to bytecode.**

2. **An Android application package, or APK file, gets created.**
   The APK file includes the compiled Kotlin files, along with any libraries and resources needed by the app.

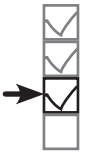
3. **The APK is installed on the device.**
   If the device is virtual, Android Studio launches the emulator and waits until the AVD is active before installing the APK.

   If the device is physical, it just installs the APK.

4. **The device starts the app's main activity.**
   The app is displayed on the device screen, and it's all ready for you to play with.

Now that you know what happens when you use the Run command, let's see what the app we've built looks like.

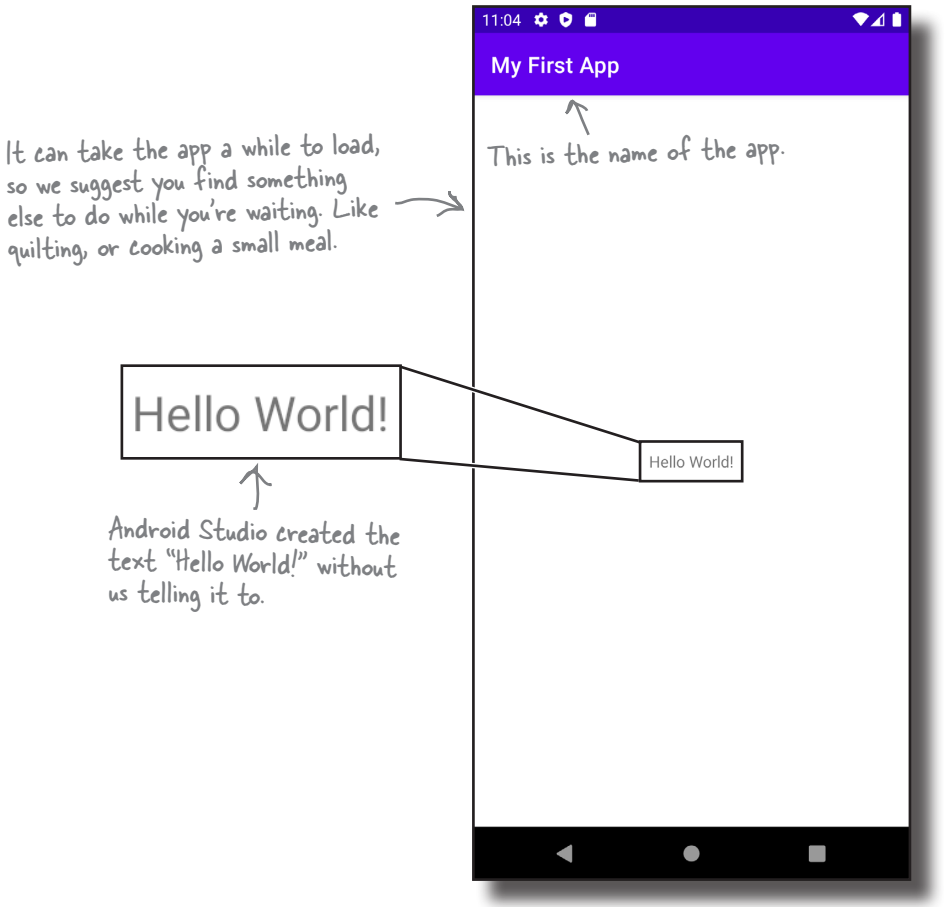
Test Drive



Set up environment
Build app
Run app
Change app

Make sure you've run the app on a physical or virtual device by choosing the "Run 'app'" command from the Run menu.

Android Studio loads the app onto the device and starts it. The app name "My First App" appears at the top of the screen, and the text "Hello World!" is displayed in the center.

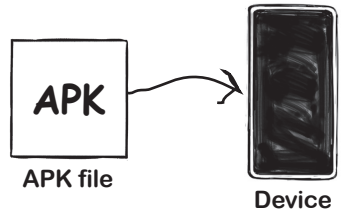


Let's run through what just happened.

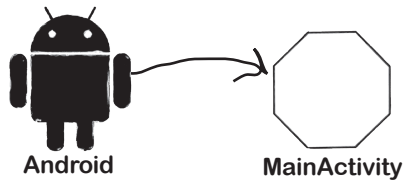
What just happened?

Let's break down what happens when you run the app:

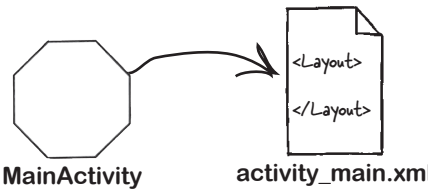
1. **Android Studio installs the app on the device.**
If the device is virtual, it waits for the emulator to start before installing the app.



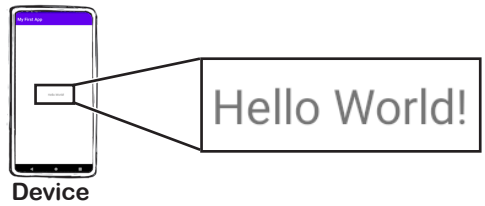
2. **Android starts the app's main activity.**
It uses the code in *MainActivity.kt* (which Android Studio automatically included in the project) to create a `MainActivity` object.



3. **MainActivity specifies that it uses the layout activity_main.xml.**



4. **The layout is displayed on the screen.**
The text "Hello World!" appears in the center of the screen.



there are no Dumb Questions

Q: Kotlin and Java are both Java VM languages. Does this mean that Android apps run in a JVM?

A: No. Each app runs in its own process using the Android runtime (ART). This means that apps run a lot faster and more efficiently.

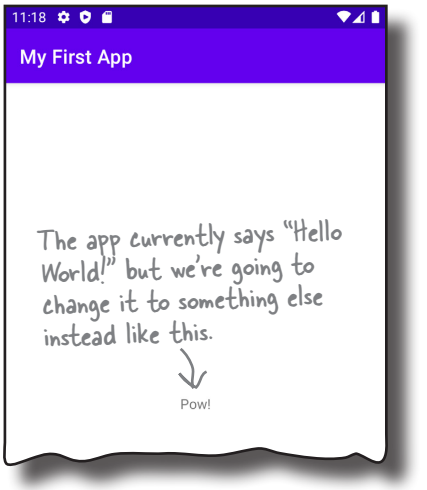
Set up environment
Build app
Run app
Change app

Let's refine the app

So far in this chapter, you've built a basic Android app and seen it running on a physical or virtual device. Next, we're going to refine the app.

At the moment, the app displays the sample text "Hello World!" that the wizard put in as a placeholder. You're going to change that text to say something else instead. So what do we need to change in order to achieve that?

To answer that question, let's take a step back and see how the app is currently built.



The app has one activity and one layout

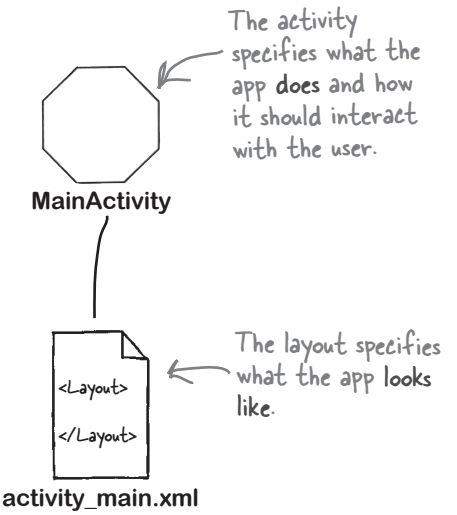
When we built the app, we told Android Studio how to configure it, and the wizard did the rest. The wizard created an activity for us, and also a default layout.

The activity controls what the app does

Android Studio created an activity for us named *MainActivity.kt*. The activity specifies what the app **does** and how it should respond to the user.

The layout controls the app's appearance

MainActivity.kt uses the layout Android Studio created for us named *activity_main.xml*. The layout specifies what the app **looks like**.



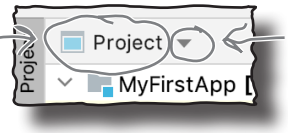
We want to change the appearance of the app by updating the text that's displayed. This means that we need to update the file that controls what the app looks like, so we need to take a closer look at the *layout*.

Set up environment
Build app
Run app
Change app

What's in the layout?

We want to change the sample "Hello World!" text that Android Studio created for us, so let's start with the layout file *activity_main.xml*. Open it now (if it's not already open) by finding the file in the *app/src/main/res/layout* folder in the explorer and double-clicking on it.

If you can't see the folder structure in the explorer, try switching to Project view.



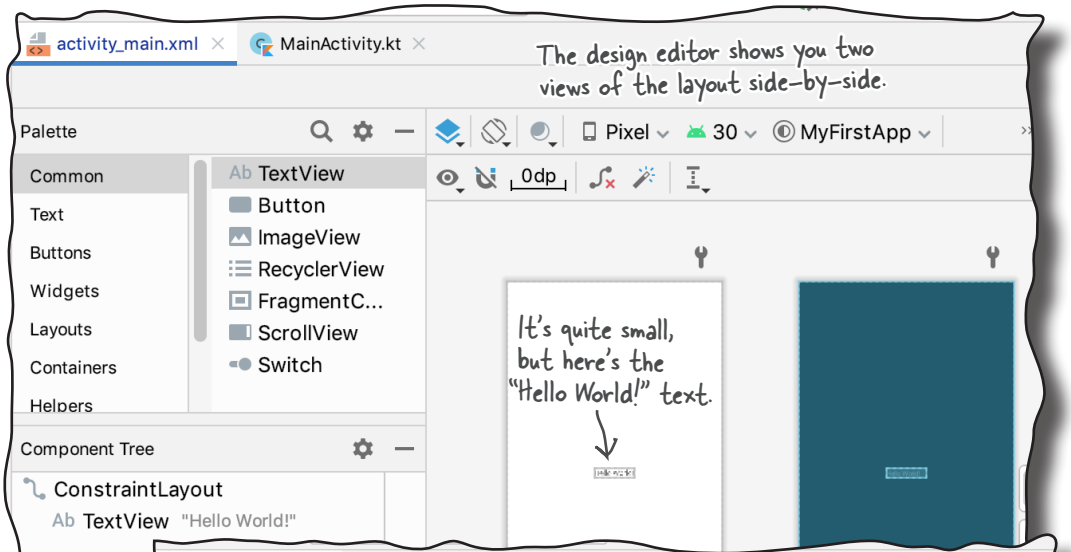
Click on this arrow to change how the files and folders are shown.

The design editor

As you learned earlier, there are two ways of viewing and editing layout files in Android Studio: through the **design editor** and through the **code editor**.

When you choose the design option, you can see that the sample text "Hello World!" appears in the layout as you might expect. But what's in the underlying XML?

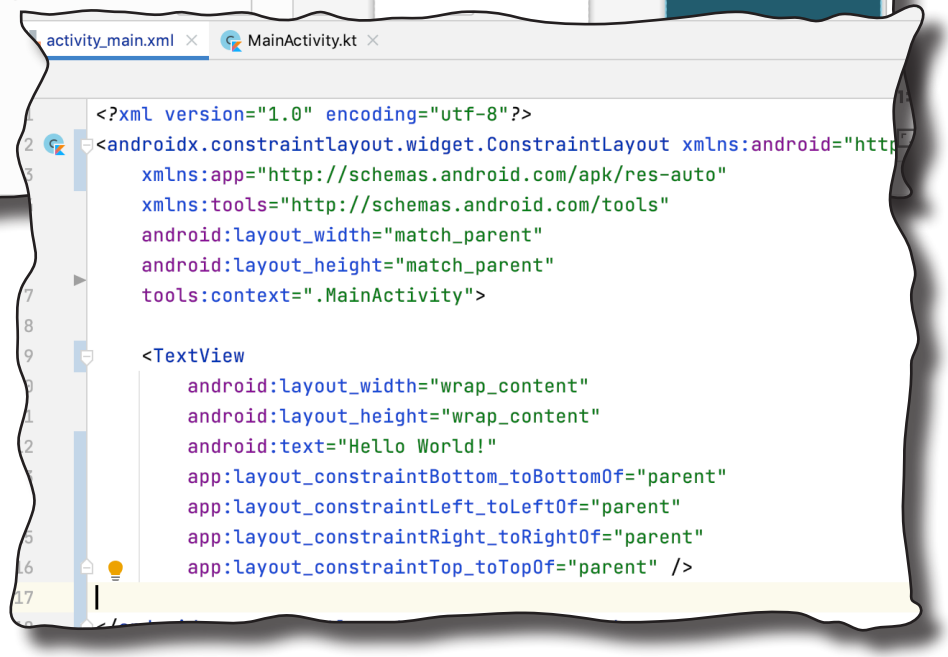
Let's see by switching to the code editor.

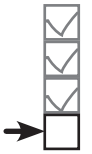


The code editor

Switch to the code editor by clicking on the Code option at the top of the editor. This shows you the layout's underlying XML.

Let's take a closer look at the code.





Set up environment
Build app
Run app
Change app

activity_main.xml has two elements

Below is the code from *activity_main.xml* that Android Studio generated for us. We've left out some of the details you don't need to think about just yet; we'll cover them in more detail through the rest of the book.

Here's the code:

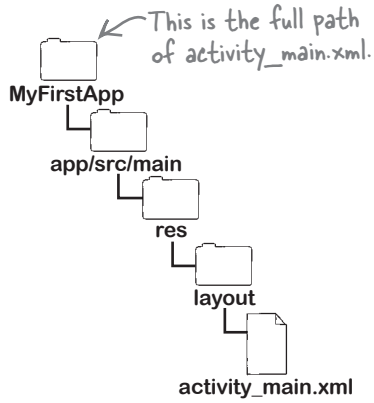
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    ... >
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        ... />
</androidx.constraintlayout.widget.ConstraintLayout>
```

This element determines how components should be displayed, in this case the "Hello World!" text.

Android Studio gave us more XML here, but you don't need to think about that yet.

This is a <TextView> element.

We've left out some of the <TextView> XML too.



As you can see, the code contains two elements.

The first is a <...ConstraintLayout> element. This is a type of layout element that tells Android how to display components on the device screen. There are various types of layout available for you to use, and you'll find out more about these later in the book.

The most important element for now is the second element, the <TextView>. This element is used to display text to the user, in this case the text "Hello World!"

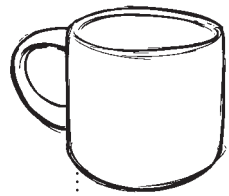
The key part of the code within the <TextView> element is the line starting with android:text. This is a text attribute that describes the text that should be displayed:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!"
    ... />
```

The <TextView> element describes the text in the layout.

This is the text that's being displayed.

We'll change the text to something else after you've had a go at the following exercise.



Relax

Don't worry if your layout code looks different from ours.

Android Studio may give you slightly different XML depending on which version you're using. You don't need to worry about this, because from the next chapter onward you'll learn how to roll your own layout code, and replace a lot of what Android Studio gives you.

Sharpen your pencil

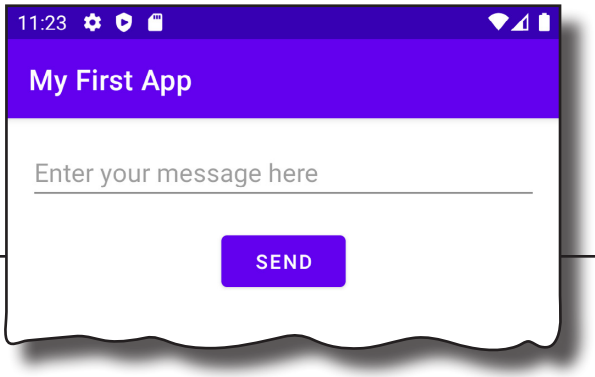
We know we've not shown you much Android code yet, but see if you can guess what each line does in the layout XML below. We've completed the first few to get you started.

```
<?xml version="1.0" encoding="utf-8"?>   XML declaration at the top of the file.
<LinearLayout   Arranges components linearly, one after another.
    xmlns:android="http://schemas.android.com/apk/res/android"   Defines a namespace.
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <EditText
        android:id="@+id/message"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        android:hint="Enter your message here"
        android:inputType="text" />

    <Button
        android:id="@+id/send"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:text="SEND" />
</LinearLayout>
```

This is what appears on the screen when you run an app with the above layout.



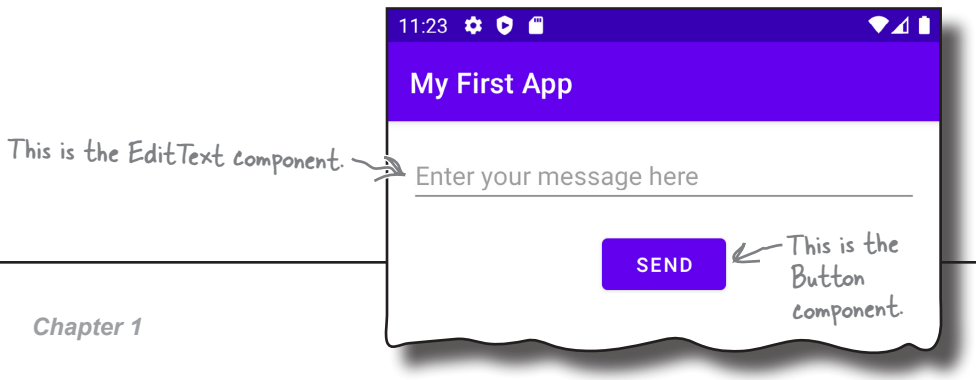
Sharpen your pencil Solution

We know we've not shown you much Android code yet, but see if you can guess what each line does in the layout XML below. We've completed the first few to get you started.

```
<?xml version="1.0" encoding="utf-8"?> ..XML declaration at the top of the file.....
<LinearLayout ..Arranges components linearly, one after another.....
    xmlns:android="http://schemas.android.com/apk/res/android" ..Defines a namespace.
    xmlns:tools="http://schemas.android.com/tools" Defines a second namespace.....
    android:layout_width="match_parent" ..Make the width match the device screen size.....
    android:layout_height="match_parent" .Make the height match the device screen size.....
    android:orientation="vertical" ..Arrange components vertically.....
    tools:context=".MainActivity"> ..Indicates to tools that the layout is used by MainActivity.

    <EditText ..Defines an EditText component (an editable text view).....
        android:id="@+id/message" .Assigns an id named "message".....
        android:layout_width="match_parent" .Make it as wide as the layout.....
        android:layout_height="wrap_content" ..Make it high enough for the content.....
        android:layout_margin="16dp" ..Set the component's margin to 16dp.....
        android:hint="Enter your message here" Include some hint text.....
        android:inputType="text" /> ..Use a text keyboard input type.....

    <Button ..Defines a Button component.....
        android:id="@+id/send" ..Assigns an id named "send".....
        android:layout_width="wrap_content" .Make it wide enough for the content.....
        android:layout_height="wrap_content" ..Make it high enough for the content.....
        android:layout_gravity="center_horizontal" .Center the button horizontally.....
        android:text="SEND" /> ..Add the text "SEND" to the button.....
</LinearLayout> ..Closes the LinearLayout element.....
```



Set up environment
Build app
Run app
Change app

Update the text displayed in the layout

We want to change the text in *activity_main.xml* so that when we run the app, it displays something other than "Hello World!" We can do this by changing the `text` attribute in the layout's `<TextView>` element:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!"
    ... />
```

We can update the text by updating this attribute.

The `text` attribute is defined inside the `<TextView>` element using the code `android:text`. It specifies what text should be displayed, in this case "Hello World!"

Display the text...

```
android:text="Hello World!" />
```

...."Hello World!"

To update the text that's displayed in the layout, simply change the value of the `text` attribute from `"Hello World!"` to some other text, such as `"Pow!"`. The new code for the `<TextView>` element should look like this:

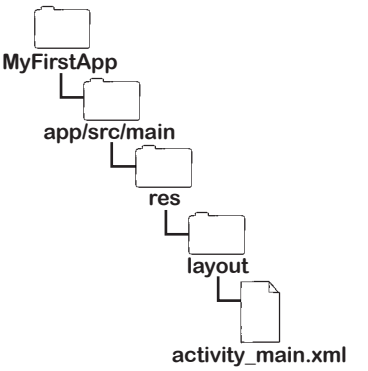
We've left out some of the code, since all we're doing for now is changing the text that's displayed.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    ... >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="~~Hello World!~~ Pow!"
        ... />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Change the text here from "Hello World!" to "Pow!"



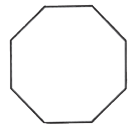
That's the only change you need to make in order to update the text. Let's see what happens when the code runs.

What the code does

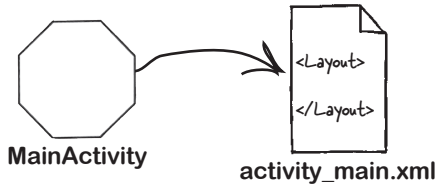
Set up environment
Build app
Run app
Change app

Before we take the app for a test drive, let's go through what the code does.

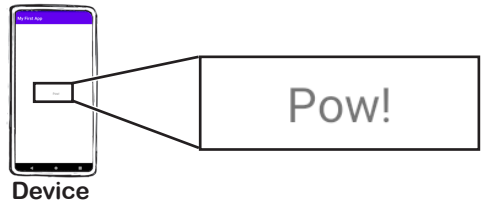
1. Android uses MainActivity.kt to create the activity object MainActivity.



2. MainActivity specifies that it uses the layout activity_main.xml.



3. The layout displays the text "Pow!" in the center of the app on the device.



there are no Dumb Questions

Q: My layout code looks different from yours. Is that OK?

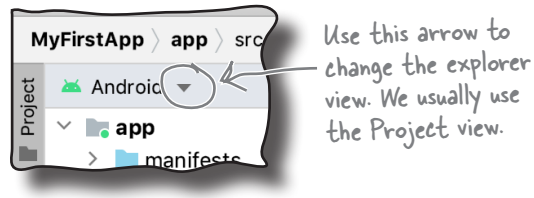
A: Yes, that's fine. Android Studio may generate slightly different code if you're using a different version than us, but that doesn't really matter. From now on you'll be learning how to create your own layout code, and you'll replace a lot of what Android Studio gives you.

Q: Am I right in thinking we're hardcoding the text that's displayed?

A: Yes, purely so that you can see how to update text in the layout. There's a better way of displaying text values than hardcoding them in the layouts, but you'll have to wait for the next chapter to learn what this is.

Q: The folders in my explorer pane look different from yours. Why is that?

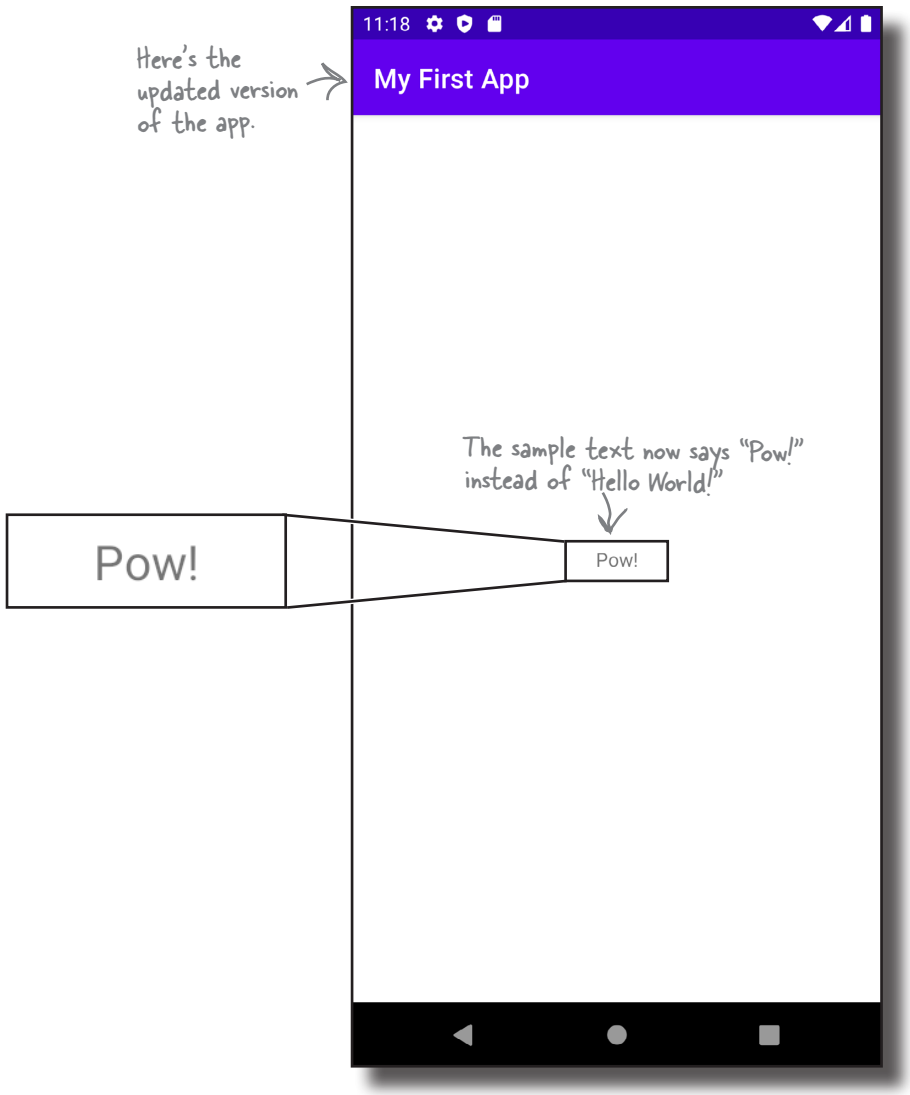
A: Android Studio lets you choose alternate views for how to display the folder hierarchy, and it defaults to the Android view. We prefer the Project view, as it reflects the underlying folder structure. You can change the explorer to the Project view by clicking on the arrow at the top of the explorer pane, and selecting the Project option.



Test Drive

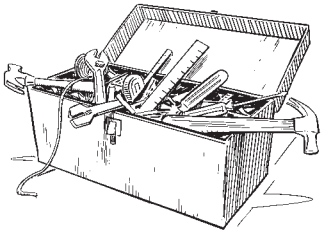
Set up environment
Build app
Run app
Change app

Once you've edited the file, try running the app in the emulator again by choosing the "Run 'app'" command from the Run menu, or clicking on the Run button. You should see that the app now says "Pow!" instead of "Hello World!"



Congratulations! You've now built and updated your first app, and learned how Android apps hang together in the process. We'll build on this further in the next chapter by creating an app you can interact with.

Your Android Toolbox



You've got Chapter 1 under your belt and now you've added Android basic concepts to your toolbox.

You can download the full code for the chapter from tinyurl.com/hfad3.

Bullet Points

- Versions of Android have a version number, API level, and code name.
- Android Studio is based on IntelliJ IDEA. It interfaces with the Android Software Development Kit (SDK), and the Gradle build system.
- A typical Android app is composed of activities, layouts, and resource files.
- Layouts describe what the app looks like. They're held in the *app/src/main/res/layout* folder.
- Activities describe what the app does, and how it interacts with the user. The activities you write are held in the *app/src/main/java* folder.
- *AndroidManifest.xml* contains information about the app itself. It lives in the *app/src/main* folder.
- An AVD is an Android Virtual Device. It runs in the Android emulator and mimics a physical Android device.
- An APK is an Android application package. It contains the app's bytecode, libraries, and resources. You install an app on a device by installing the APK.
- Android apps run in separate processes using the Android runtime (ART).
- The `<TextView>` element is used for displaying text.
- The `<TextView>` element's `text` attribute specifies what text it should display.

Apps That Do Something

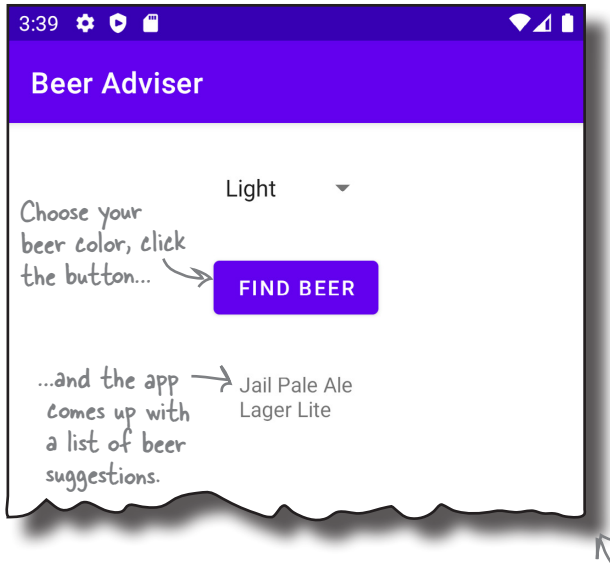
Most apps need to respond to the user in some way.

And in this chapter, you'll see how you can make your apps **more interactive**. You'll discover how to add an ***OnClickListener*** to your activity code so that your app can **listen to what the user's doing**, and make an appropriate response. You'll find out more about **how to design layouts**, and you'll learn how each UI component you add to your layout is derived from **a common View ancestor**. Along the way, you'll discover **why String resources are so important** for flexible, well-designed apps.

Let's build a Beer Adviser app

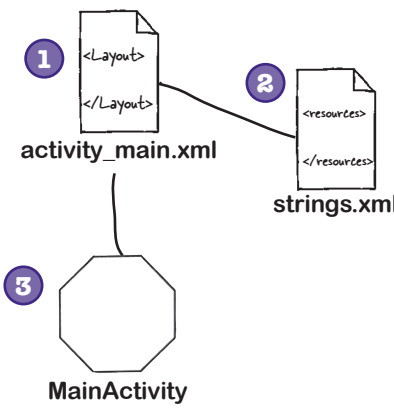
When you create an Android app, you're usually going to want it to *do* something.

In this chapter, we're going to show you how to create an app that the user can interact with. We'll create a Beer Adviser app where users can select the color of beer they enjoy most, click a button, and get back a list of tasty beers to try out.



This is what the layout looks like.

Here's how the app will be structured:



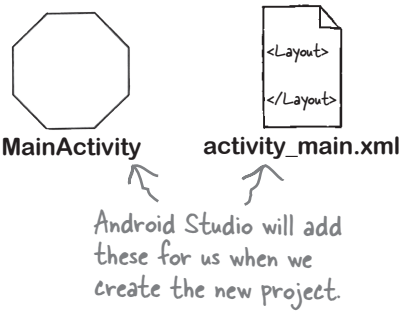
1. **The layout activity_main.xml specifies what the app will look like.**
   It includes three UI components:
   - A drop-down list of values called a spinner, which allows the user to choose which color of beer they want
   - A button that, when clicked, will return a selection of beers
   - A text view that displays the beers

2. **The file strings.xml includes any String resources needed by the layout.**
   The button's label, for example, and the beer colors.

3. **The activity MainActivity specifies how the app should interact with the user.**
   It takes the beer color the user chooses, and uses this to display a list of beers the user might be interested in.

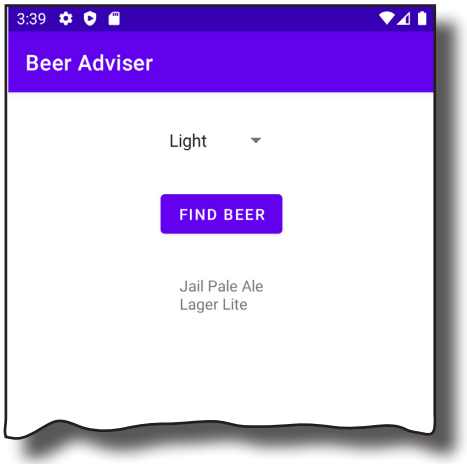
Here's what we're going to do

So let's get to work. There are a few steps you need to go through to build the Beer Adviser app (we'll tackle these throughout the rest of the chapter):

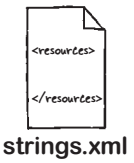


1. **Create a project.**
You're creating a brand-new app, so you'll need to create a new project that includes an empty activity and a layout.

2. **Update the layout.**
Once you've set up the project, you need to amend the layout so that it includes all the UI components the app needs.

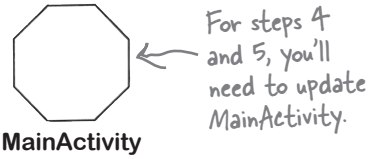


3. **Add String resources.**
We'll replace any hardcoded text with `String` resources so that all the text that's used by the app is held in a single file.

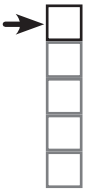


4. **Make the button respond to clicks.**
The layout only specifies the visuals. To make the button *do* something when it's clicked, you need to write some activity code.

5. **Write the application logic.**
You'll add a new method to the activity, and use it to make sure users get the right beer based on their selection.



Let's start by creating the project.



Create project
Update layout
Add resources
Respond to clicks
Write logic

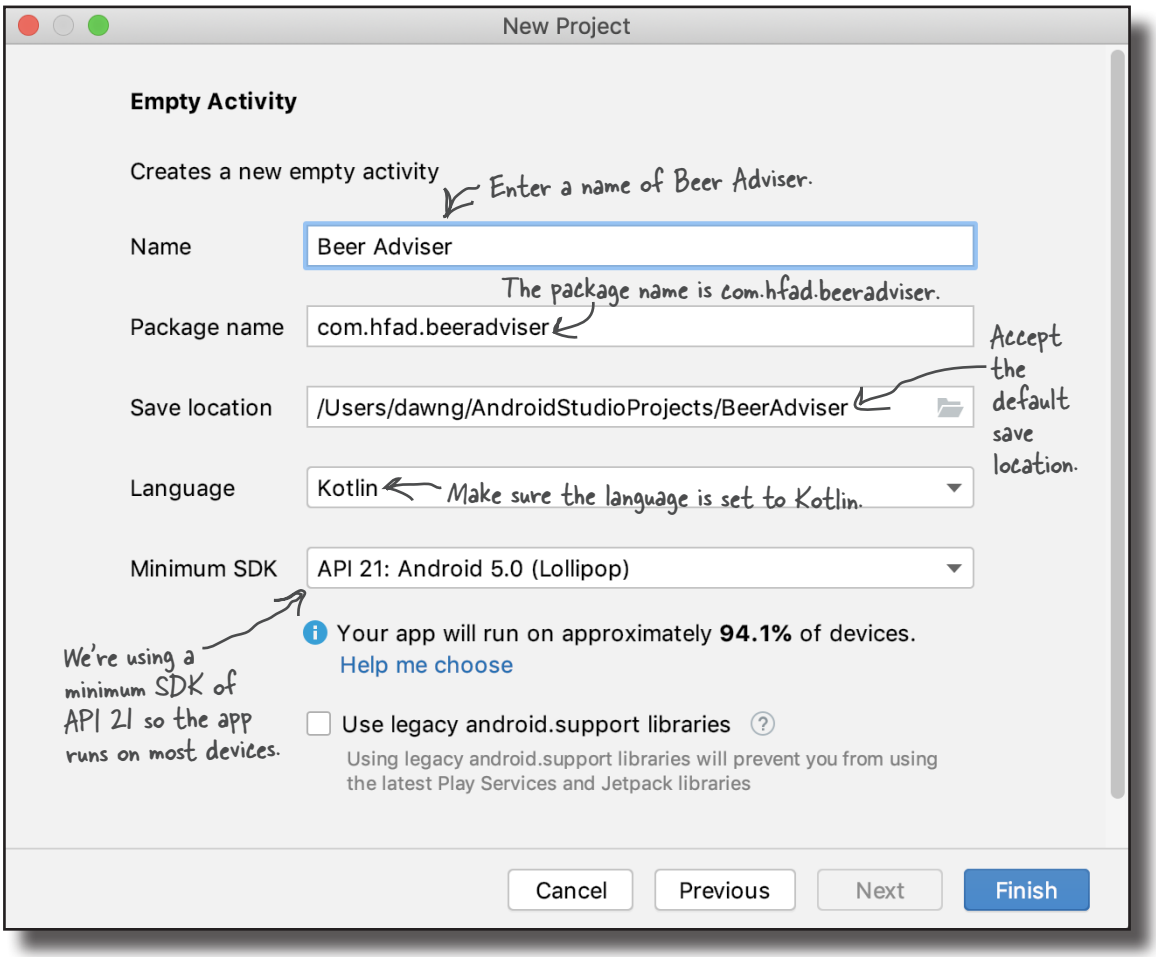
Create the project

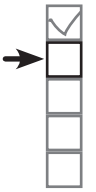
The steps for creating the new project are nearly identical to the ones we used in the previous chapter:

1. Open Android Studio, close any open projects, and choose "New Project" from the welcome screen. This starts the wizard you saw in Chapter 1.

2. Make sure the Phone and Tablet option is selected, and choose the Empty Activity option.

3. Enter a name of "Beer Adviser" and a package name of "com.hfad.beeradviser" and accept the default save location. Make sure that the language is set to Kotlin, and that the minimum SDK is API 21 so that it will run on most Android devices. Then click on the Finish button.





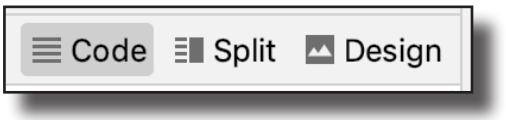
Create project
Update layout
Add resources
Respond to clicks
Write logic

We've created a default activity and layout

When you click on the Finish button, Android Studio creates a new project containing an activity named *MainActivity.kt* and a layout named *activity_main.xml*, just as it did for the project we created in Chapter 1. We need to modify these files to make the app look and behave the way we want.

We'll start by updating the layout file *activity_main.xml* to modify the app's appearance. We'll build up the layout over the next few pages, but for now, switch to the Project view of Android Studio's explorer, go to the *app/src/main/res/layout* folder, and open the file *activity_main.xml*. Then switch to the code editor, and **replace the entire code** in *activity_main.xml* with the following:

Click on the Code option to open the code editor.



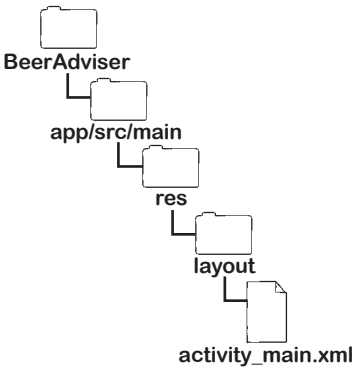
We're replacing the code Android Studio generated for us.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/brands"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Beer types" />
</LinearLayout>
```

These elements relate to the layout as a whole. They determine the layout width and height, any padding in the layout margins, and whether components should be laid out vertically or horizontally.

This is used to display text.

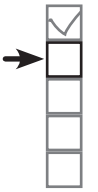


The above code features a linear layout (denoted by the `<LinearLayout>` element) and a text view (denoted by the `<TextView>` element). You'll find out more about these elements later in the chapter, but for now, all you need to know is that the linear layout is used to arrange UI components in a vertical column, and the text view displays the text "Beer types".

Any changes you make to a layout's XML are reflected in the design editor. Switch to this now by clicking on the Design option at the top of the editor pane.

Click on the Design tab to open the design editor.

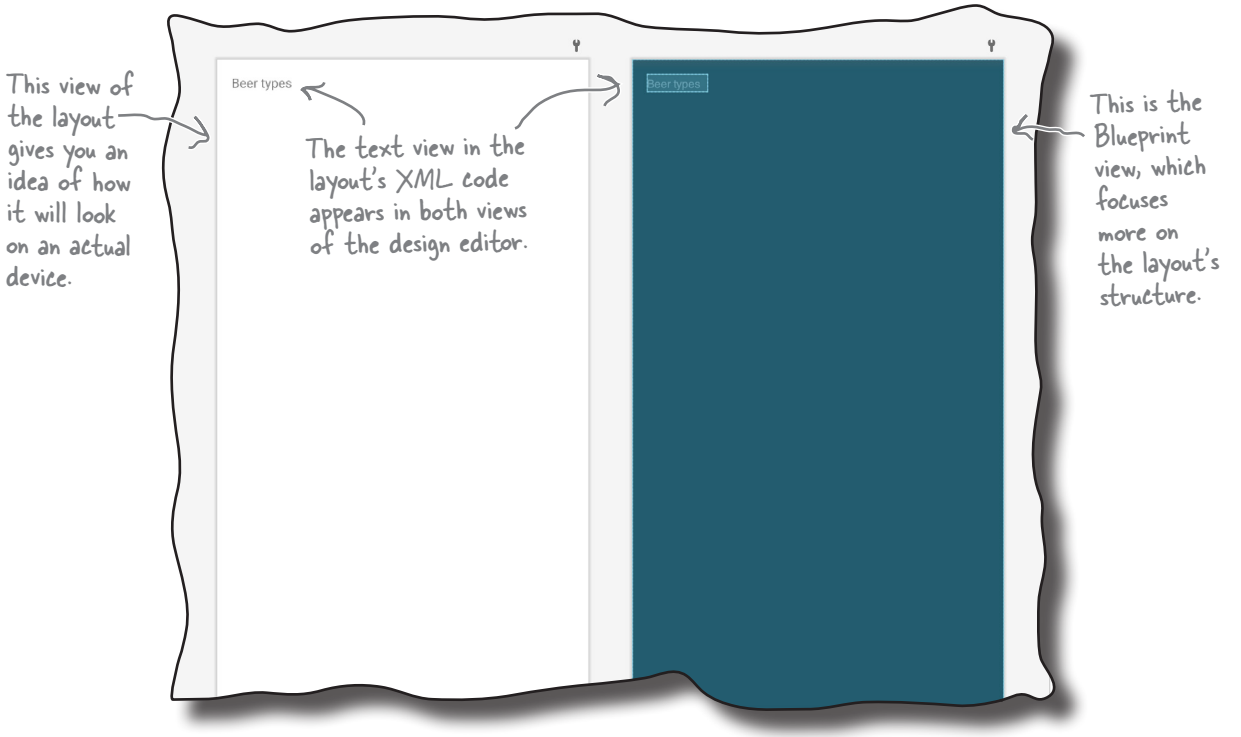
A closer look at the design editor



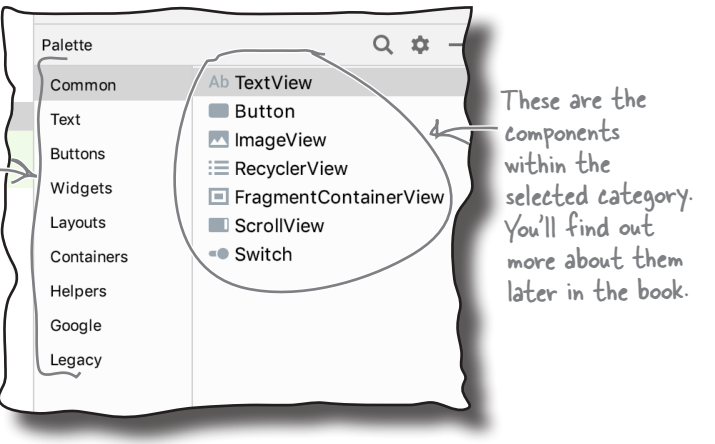
- Create project
- **Update layout**
- Add resources
- Respond to clicks
- Write logic

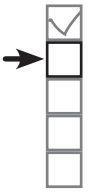
As you learned in Chapter 1, the design editor presents you with a more visual way of editing layout code than editing XML. It features two different views of the layouts design. One shows you how the layout will look on an actual device, and the other shows you a blueprint of its structure:

If Android Studio doesn't show you both views of the layout, click on this button in the design editor's toolbar and choose the "Design and Blueprint" option.



To the left of the design editor is a palette that contains components you can drag to the layout. You'll use this on the next page to add a button to the layout, which later in the chapter will be used to update the text that's displayed in the app.

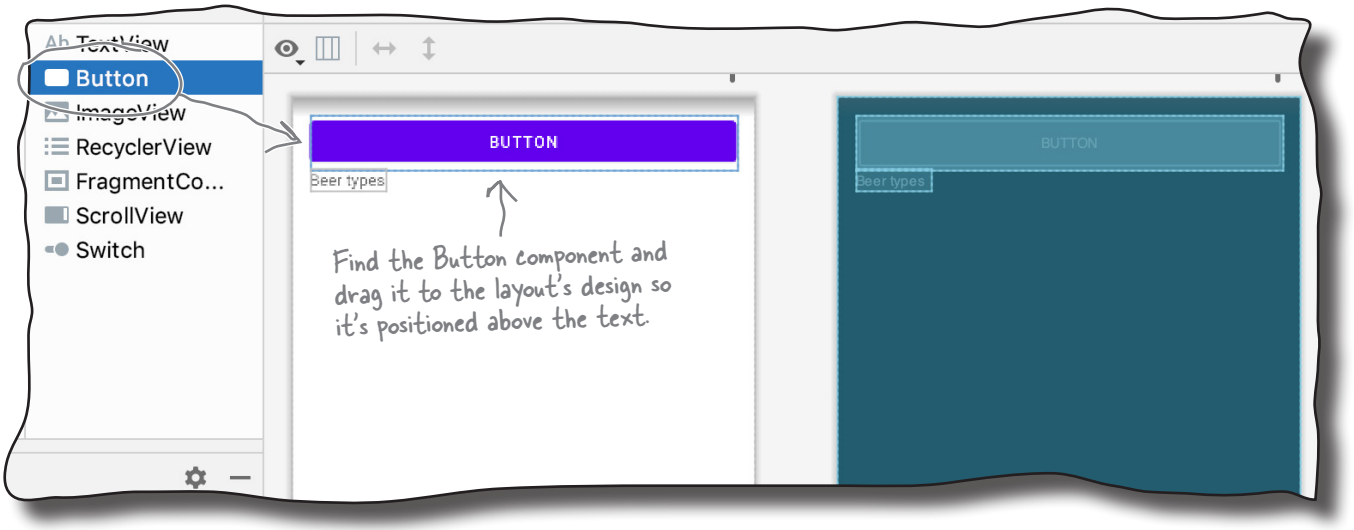




- Create project
- **Update layout**
- Add resources
- Respond to clicks
- Write logic

Add a button using the design editor

To add a button to the layout, find the Button component in the palette, click on it, and then drag it into the design editor so it's positioned above the text view. The button appears in the layout's design:



Changes in the design editor are reflected in the XML

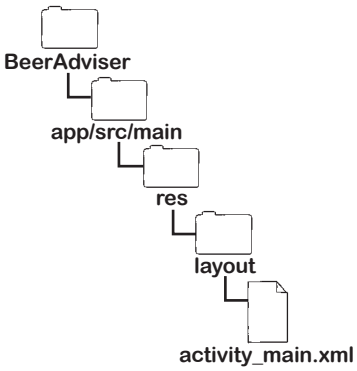
Dragging UI components to the layout like this is a convenient way of updating the layout. If you switch to the code editor, you'll see that adding the button via the design editor has added some more code to the underlying XML:

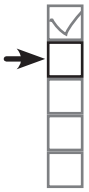
There's a new <Button> element that describes the new button you've dragged to the layout. We'll look at this in more detail over the next few pages.

```
...
<Button
    android:id="@+id/button"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Button" />

<TextView
    android:id="@+id/brands"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Beer types" />
...
```

The code the design editor adds depends on where you place the button, so don't worry if your code looks different from ours.





- Create project
- **Update layout**
- Add resources
- Respond to clicks
- Write logic

activity_main.xml has a new button

As you've just seen, the design editor has added a new `<Button>` element to *activity_main.xml*. Its code looks like this:

```
<Button
    android:id="@+id/button"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Button" />
```

A button in Androidville is a UI component the user can click to trigger an action. The `<Button>` element includes attributes controlling its size and appearance. These attributes aren't unique to buttons—other UI components such as text views have them too.

Buttons and text views are subclasses of the same Android View class

There's a very good reason why buttons and text views have attributes in common—they both inherit from the same Android **`View`** class. You'll find out more about this as we go through the book, but for now, here are some of the most common attributes:

android:id

This gives the component an identifying name so that the activity code can access it and control its behavior:

```
android:id="@+id/button"
```

android:layout_width, android:layout_height

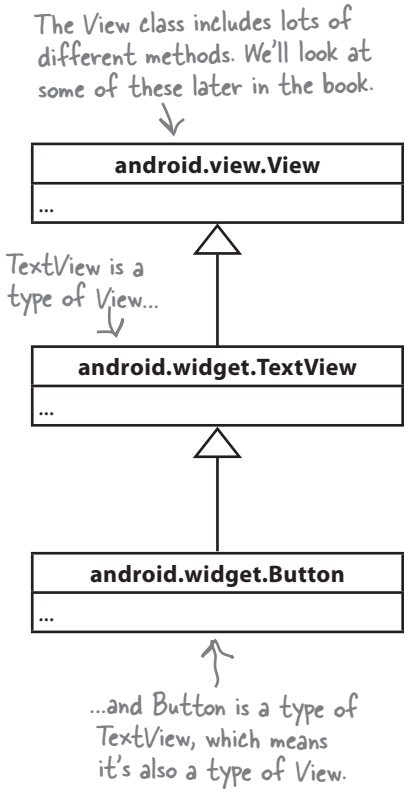
These attributes specify the width and height of the component. `"wrap_content"` means it should be just big enough for the content, and `"match_parent"` means it should be as wide as the layout containing it:

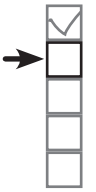
```
android:layout_width="match_parent"
android:layout_height="wrap_content"
```

android:text

This tells Android what text the component should display, such as the text that appears on a button:

```
android:text="Button"
```





Create project
Update layout
Add resources
Respond to clicks
Write logic

A closer look at the layout code

Let's take a closer look at the layout code, and break it down so that you can see what it's actually doing (don't worry if your code looks a little different, just follow along):

This is the <LinearLayout> element

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:orientation="vertical"
    tools:context=".MainActivity">
```

This is the button.

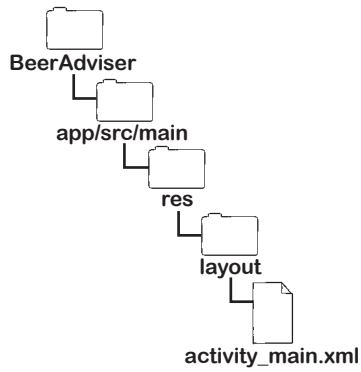
```
    <Button
        android:id="@+id/button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Button" />
```

This is the text view.

```
    <TextView
        android:id="@+id/brands"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Beer types" />
```

```
</LinearLayout>
```

This closes the <LinearLayout> element.



The <LinearLayout> element

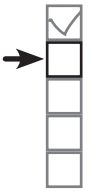
The first element in the layout code is the `<LinearLayout>`. This element tells Android that the different UI components in the layout should be displayed one after another in a single row or column.

You specify the orientation using the `android:orientation` attribute. In this example we're using:

```
android:orientation="vertical"
```

so the UI components are displayed in a single vertical column.

A closer look at the layout code (continued)



- Create project
- **Update layout**
- Add resources
- Respond to clicks
- Write logic

The <LinearLayout> element (on the previous page) contains two further elements: a <Button> and a <TextView>.

The <Button> element

The first element is the <Button>:

```
...
<Button
    android:id="@+id/button"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Button" />
...
```

As this is the first element inside the <LinearLayout>, it appears first in the layout (at the top of the screen). It has a layout_width of "match_parent", which makes it as wide as its parent element, the <LinearLayout>. Its layout_height is "wrap_content", which means it should be just tall enough to display its text.

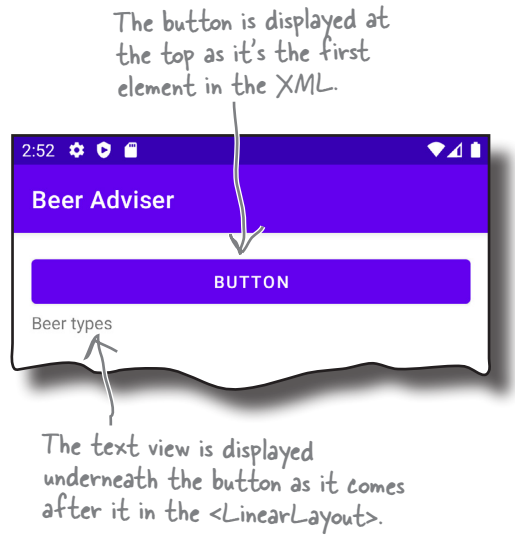
Using a linear layout means that UI components are displayed in a vertical column or a horizontal row.

The <TextView> element

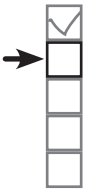
The final element inside the <LinearLayout> is the <TextView>:

```
...
<TextView
    android:id="@+id/brands"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Beer types" />
...
```

As this is the second element and we've set the <LinearLayout> element's orientation to "vertical", it's displayed underneath the button (the first element). Its layout_width and layout_height attributes are both set to "wrap_content" so that it takes up just enough space to contain its text.



You've seen how adding components to the design editor adds them to the layout XML. The opposite applies too—any changes you make to the layout XML are applied to the design. Let's see this in action.



- Create project
- **Update layout**
- Add resources
- Respond to clicks
- Write logic

Let's update the layout XML

We'll update the layout by adding a new **spinner** component, and tweaking the button and text view components that are already there. A spinner is a drop-down list of values. When you click it, it expands to show you the list so that you can pick a single value.

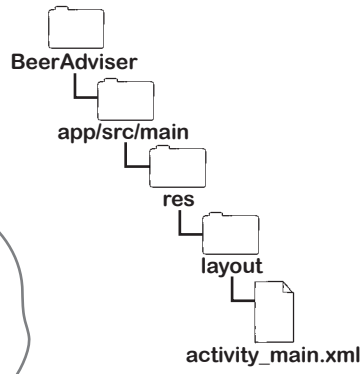
Update the *activity_main.xml* code with the following changes (highlighted in bold):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <Spinner
        android:id="@+id/beer_color"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="16dp" />

    <Button
        android:id="@+id/button_find_beer"
        android:layout_width="match_parent wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="16dp"
        android:text="Button Find Beer" />

    <TextView
        android:id="@+id/brands"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="16dp"
        android:text="Beer types" />
</LinearLayout>
```



Here is the new spinner component. It allows you to choose a single value from a selection.

Change the button's ID to "find_beer". We'll use this later in the chapter.

Change the button's width so it's as wide as its content.

Center the button horizontally and give it a margin.

Change the button's text.

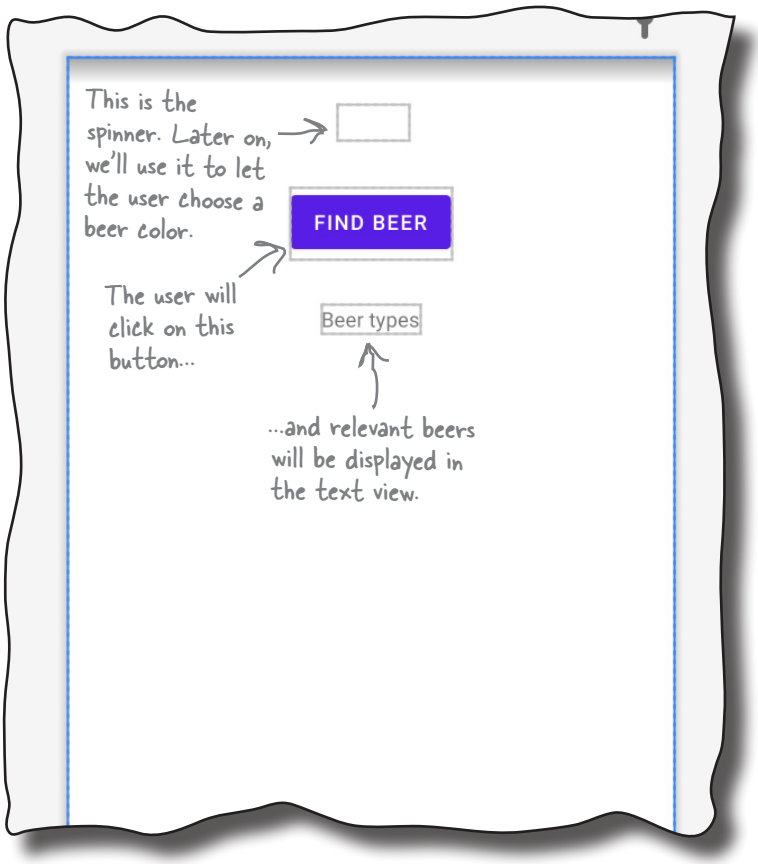
Center the text view and apply a margin.

Don't forget!

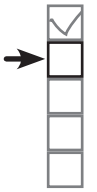
Make sure you update the contents of activity_main.xml with the changes shown here.

The XML changes are reflected in the design editor

Once you've changed the layout XML, switch to the design editor. Instead of displaying a layout containing a button with a text view underneath it, the design editor now shows a spinner, button, and text view centered horizontally in a single column like this:



We've now added all the components to *activity_main.xml* that we need for the Beer Adviser app's layout. We still have more work to do, but let's take the app for a test drive so that we can see how it looks on a device.

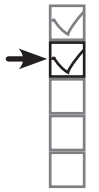


- Create project
- **Update layout**
- Add resources
- Respond to clicks
- Write logic

A spinner provides a drop-down list of values. It allows you to choose a single value from a set of values.

All the UI components you add to layout files—such as buttons, spinners, and text views—use the same or similar attributes because they are all types of View. Behind the scenes, they all inherit from the same Android View class.

These UI components are often referred to as views because they inherit from the same View class.

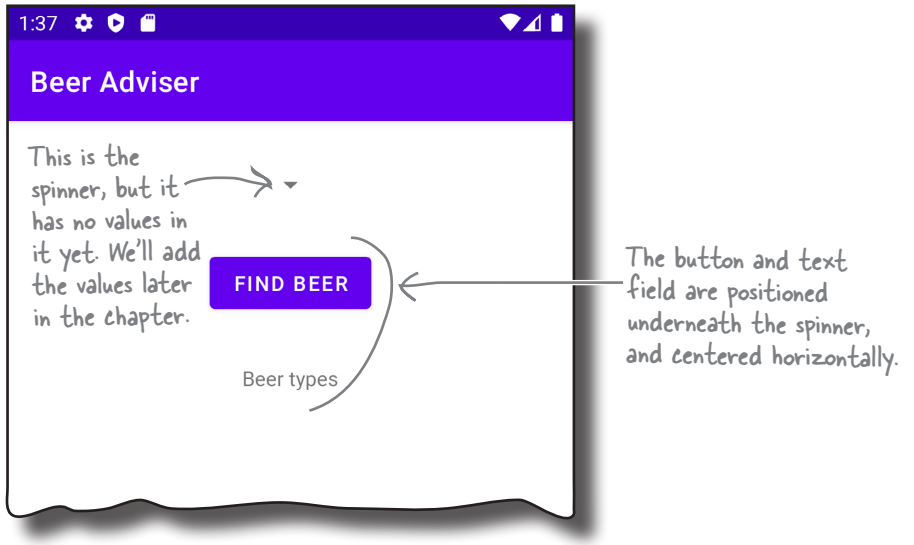


- Create project
- **Update layout**
- Add resources
- Respond to clicks
- Write logic

Test Drive

Run the app by choosing the "Run 'app'" command from the Run menu or clicking on the Run button, and wait patiently for the app to load.

When the app appears on your device, notice that it displays an empty spinner, button, and text view in a single column.



there are no Dumb Questions

Q: You said you can add views through the design editor or by editing the XML. Which approach is best?

A: It's mostly a matter of personal preference. For simple layouts, we prefer to update the underlying XML, but you'll find out in Chapter 4 that some types of layout are easier to update if you use the design editor.

Most of the time, we're going to ask you to update the underlying XML as this is the easiest way to make sure your code matches ours.

Q: Why did we replace the default layout code that Android Studio provided for us?

A: There are a couple of reasons.

First, a linear layout is the most appropriate type of layout to use if you want to arrange the components in a single row or column. This makes it a good fit for the app we're building in this chapter.

The second reason is that a linear layout is the simplest type of layout to learn how to use. You'll discover more advanced layouts later in the book, but for now, we're focusing on the basics.

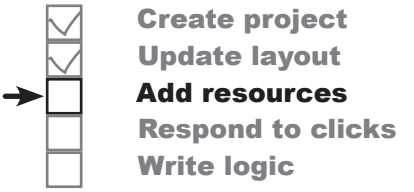
Q: Why does the button show its text as uppercase?

A: It's because the app's default theme changes the button text to uppercase by default. You'll find out more about themes—and how to override their behavior—in Chapter 8.

For now, you can stop the button text from being displayed as uppercase by adding the following code to each `<Button>` element in your layout:

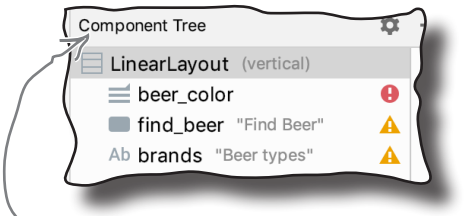
```
android:textAllCaps="false"
```

There are warnings in the layout...



When you develop layouts in Android Studio, the IDE automatically checks your code for errors, and alerts you to potential improvements. An easy way of viewing any warnings or suggestions is to switch to the design editor view of the layout, and check out the component tree panel. This panel is usually located beneath the palette, and it displays a hierarchical tree of the components in the layout.

If Android Studio has any suggestions for how to improve your code, you'll see a badge or icon to the right of the relevant component. We have warning badges, for example, next to the `find_beer` and `brands` components. If we hover the mouse cursor over each one, we can see messages warning us about hardcoded text:



The component tree is located beneath the palette in the layout's design editor.

There's a warning about the spinner too, which we'll address later on.

Here's one of the warning messages.

...because there's hardcoded text

When we defined the layout, we hardcoded the text that needs to be displayed in the text view and button components using code like this:

```
android:text="Find Beer"

android:text="Beer types"
```

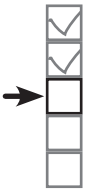
Two of the components in the layout use hardcoded text values.

While this approach is fine when you're just learning, hardcoding the text in the layout isn't the best approach.

Suppose you've created an app that's a big hit on your local Google Play Store. You don't want to limit yourself to just one country or language—you want to make it available internationally and for different languages. But if you've hardcoded all of the text in your layout files, sending the app global will be difficult.

It also makes it much harder to make app-wide changes to the text. Imagine your boss asks you to change the wording in an app because the company's changed its name. If you've hardcoded all of the text, this means that you might need to edit a whole host of files in order to change the text.

So what's the alternative?



- Create project
- Update layout
- **Add resources**
- Respond to clicks
- Write logic

Put text in a String resource file

A better approach is to put the text values into a **`String` resource file**. This makes it much easier to make app-wide changes to the text used in the application. Instead of having to change hardcoded text values in a whole host of different activity and layout files, you only need to edit the text in the resource file.

This approach also makes it much easier to localize the app. Instead of hardcoding the text in one language, you can provide separate `String` resource files for each language you want to support. This enables the app to switch the language that's used in the app so that it matches the device locale.

Android Studio helps you extract String resources

If you have a layout that contains hardcoded text, Android Studio provides you with an easy way of extracting the text and adding it to a `String` resource file. Simply click (or double-click) on each badge warning you about the hardcoded text, then click on the Fix button to fix the problem.

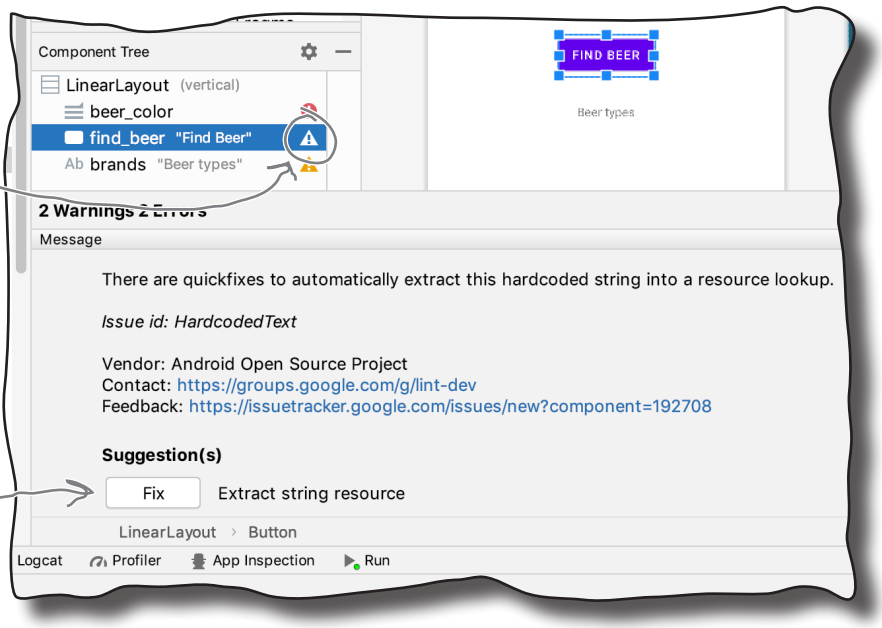
You might need to scroll down to see this button.

Let's try this with one of the components in the layout. Make sure that you're using the design view of *activity_main.xml*, and click on the warning badge next to the `find_beer` component.

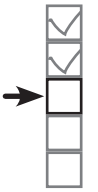
You'll be shown an explanation about why hardcoded text is a problem. Scroll to the end of this explanation, then click on the Fix button:

Click on the warning badge next to the find_beer component...

Then click on the Fix button at the end of the explanatory message.



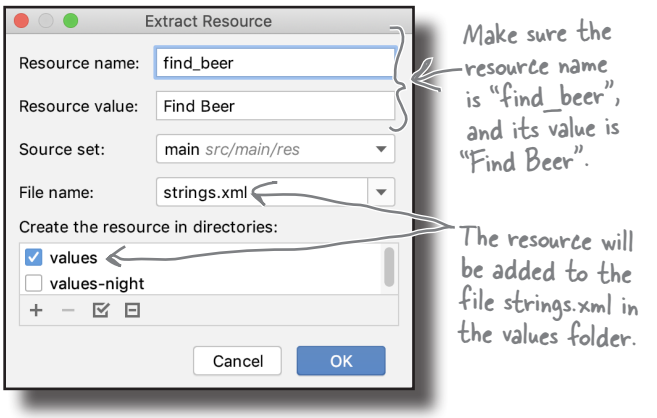
Extract the String resource



Create project
Update layout
Add resources
Respond to clicks
Write logic

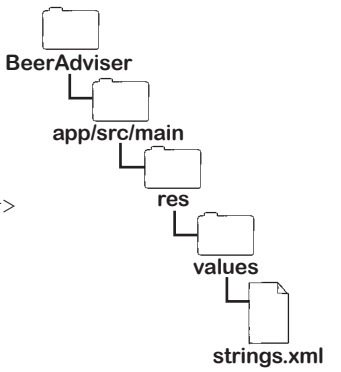
When you click on the Fix button, the Extract Resource window appears. This allows you to specify the name of the `String` resource, its value, and the name of the `String` resource file. Make sure that the resource name is "find_beer", the file name is "strings.xml", the source set is "main", and the values directory is checked. Then click on the OK button.

When you click on the OK button, Android Studio adds the `find_beer` component's hardcoded text to a `String` resource file named *strings.xml*, and changes the layout's XML so that it uses the `String` resource. We'll look at both of these changes, starting with the `String` resource file.



A String resource has been added to strings.xml

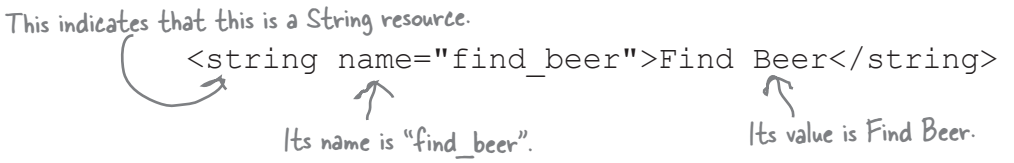
strings.xml is the app's default `String` resource file, and Android Studio automatically creates this file for you when you create a new project. Open *strings.xml* now by opening it in Android Studio's explorer: you'll find *strings.xml* in the *app/src/main/res/values* folder.



The file contents should look something like this:

```
<resources>
    <string name="app_name">Beer Adviser</string>
    <string name="find_beer">Find Beer</string>
</resources>
```

The above code describes two `String` resources, where each resource is a name/value pair. The first resource is named `app_name` and has a value of "Beer Adviser", while the second is named `find_beer` and has a value of "Find Beer". The second resource was added when we extracted the hardcoded text for the `find_beer` component:



We'll look at `String` resources in more detail a few pages ahead, but for now, let's see what change was made to *activity_main.xml*.