O'REILLY®

Third Edition

# Head First Android Development

## A Learner's Guide to Building Android Apps with Kotlin

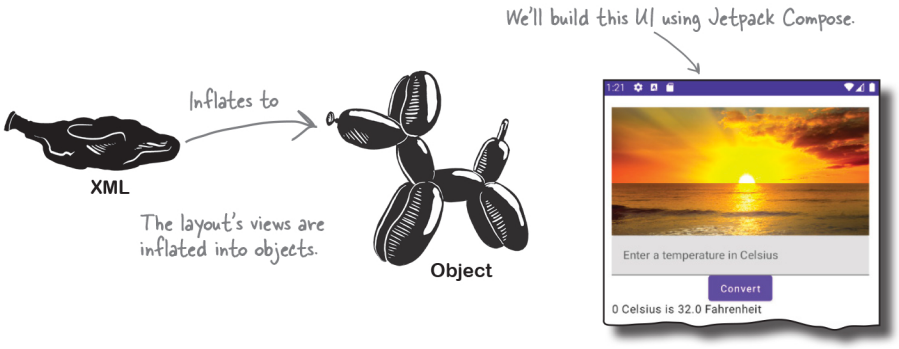
Dawn Griffiths
& David Griffiths

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 $89.99** **CAN $112.99**

**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

Copyright © 2022 David Griffiths and Dawn Griffiths. All rights reserved.

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:** | Susan Thompson, based on a series design by Ellie Volckhausen |
| **Cover/Interior Illustrations:** | José Marzan Jr. |
| **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.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. The *Head First* series designations, *Head First Android Development*, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

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)


| | | |
|---|---|---|
| | Intro | xxix |
| 1 | getting started: *Diving In* | 1 |
| 2 | building interactive apps: *Apps That Do Something* | 37 |
| 3 | layouts: *Being a Layout* | 81 |
| 4 | constraint layouts: *Draw Up a Blueprint* | 121 |
| 5 | the activity lifecycle: *Being an Activity* | 169 |
| 6 | fragments and navigation: *Finding Your Way* | 219 |
| 7 | safe args: *Passing Information* | 257 |
| 8 | navigation ui: *Going Places* | 293 |
| 9 | material views: *A Material World* | 355 |
| 10 | view binding: *Bound Together* | 403 |
| 11 | view models: *Model Behavior* | 435 |
| 12 | live data: *Leaping into Action* | 483 |
| 13 | data binding: *Building Smart Layouts* | 519 |
| 14 | room databases: *Room with a View* | 569 |
| 15 | recycler views: *Reduce, Reuse, Recycle* | 621 |
| 16 | diffutil and data binding: *Life in the Fast Lane* | 671 |
| 17 | recycler view navigation: *Pick a Card* | 705 |
| 18 | jetpack compose: *Compose Yourself* | 755 |
| 19 | integrating compose with views: *Perfect Harmony* | 811 |
| i | leftovers: *The Top 10 Things (we didn't cover)* | 861 |


# Table of Contents (the real thing)

## Intro

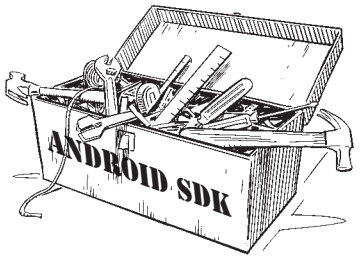
**Your brain on Android.** Here you are trying to learn something, while here your brain is, doing you a favor by making sure the learning doesn't stick. Your brain's thinking, "Better leave room for more important things, like which wild animals to avoid and whether naked snowboarding is a bad idea."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

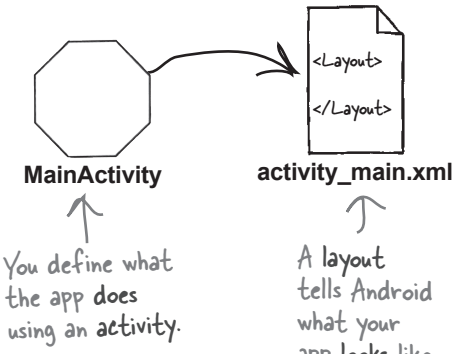

| | |
|---|---|
| Who is this book for? | xxx |
| We know what you're thinking | xxxi |
| We know what your brain is thinking | xxxi |
| Metacognition: thinking about thinking | xxxiii |
| Here's what WE did | xxxiv |
| Read me | xxxvi |
| The technical review team | xxxviii |
| Acknowledgments | xxxix |

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**.



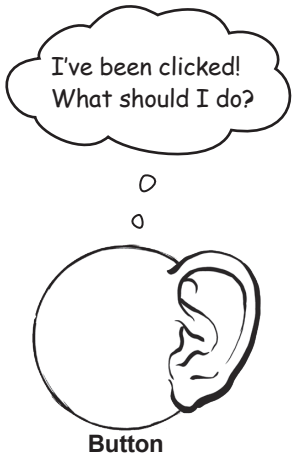


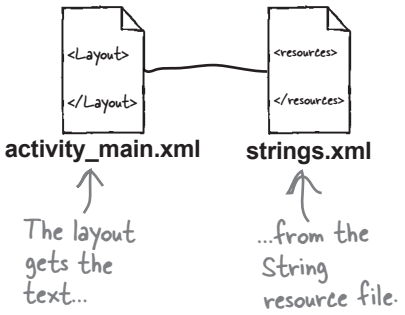
| | |
|---|---|
| Welcome to Androidville | 2 |
| Activities and layouts form the backbone of your app | 3 |
| Here's what we're going to do | 4 |
| Android Studio: your development environment | 5 |
| Install Android Studio | 6 |
| Let's build a basic app | 7 |
| How to build the app | 8 |
| Android versions up close | 11 |
| You've created your first Android project | 12 |
| Dissecting your new project | 13 |
| Introducing the key files in your project | 14 |
| Edit code with the Android Studio editors | 15 |
| The story so far | 16 |
| How to run the app on a physical device | 19 |
| How to run the app on a virtual device | 20 |
| Create an Android Virtual Device (AVD) | 21 |
| Compile, package, deploy, run | 25 |
| What just happened? | 27 |
| Let's refine the app | 28 |
| What's in the layout? | 29 |
| activity_main.xml has two elements | 30 |
| Update the text displayed in the layout | 33 |
| What the code does | 34 |
| Your Android Toolbox | 36 |

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.



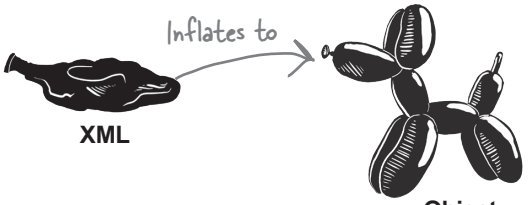


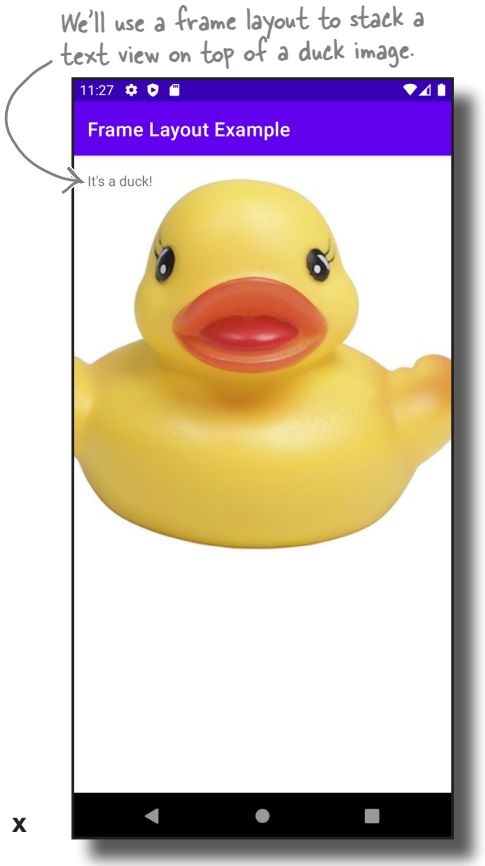
| | |
|---|---|
| Let's build a Beer Adviser app | 38 |
| Create the project | 40 |
| A closer look at the design editor | 42 |
| Add a button using the design editor | 43 |
| A closer look at the layout code | 45 |
| Let's update the layout XML | 47 |
| The XML changes are reflected in the design editor | 48 |
| There are warnings in the layout | 50 |
| Put text in a String resource file | 51 |
| Extract the String resource | 52 |
| Add and use a new String resource | 54 |
| String resources up close | 55 |
| Add values to the spinner | 59 |
| Add the string-array to strings.xml | 60 |
| We need to make the app interactive | 63 |
| What the MainActivity code looks like | 64 |
| A button can listen for on-click events | 65 |
| Get a reference to the button | 66 |
| Pass a lambda to the setOnClickListener method | 67 |
| How to edit a text view's text | 68 |
| The updated code for MainActivity.kt | 71 |
| What happens when you run the code | 72 |
| Add the getBeers() method | 74 |
| Your Android Toolbox | 80 |

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**.





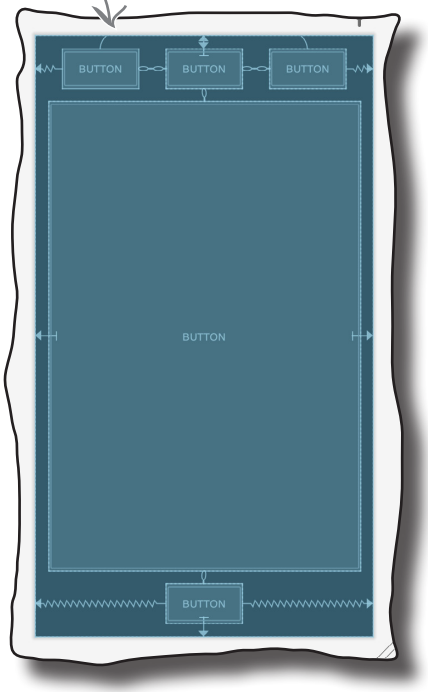
It all starts with a layout 82
Android has different types of layout 83
How to define a linear layout 85
Orientation can be vertical or horizontal 86
Anatomy of AndroidManifest.xml 87
Use padding to add space to the layout's edges 88
The layout code so far 89
An edit text lets you enter text 90
Add views to the layout XML 91
Make a view streeeeetch by adding weight 92
How to add weight to one view 93
How to add weight to multiple views 94
Values you can use with the android:gravity attribute 96
The story so far 99
More values you can use with the android:layout-gravity attribute 101
Use margins to add space between views 102
Your activity code tells Android which layout it uses 107
Layout inflation: an example 108
A frame layout stacks its views 109
Add an image to your project 110
Drawable resources up close 111
A frame layout stacks views in the order they appear in the layout XML 112
All layouts are a type of ViewGroup 113
A scroll view inserts a vertical scrollbar 115
Your Android Toolbox 120

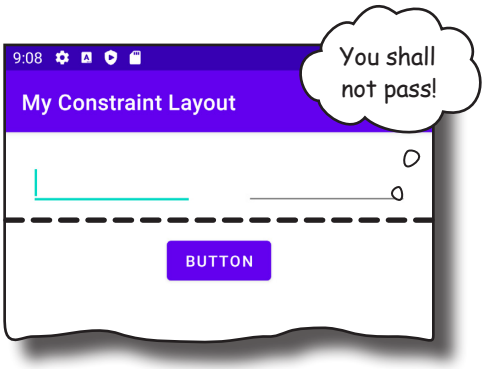
constraint layouts

# 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.



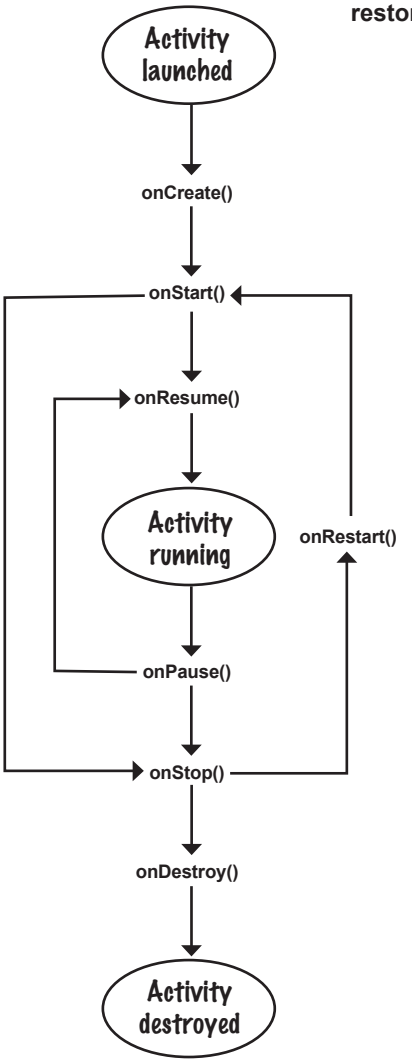


| | |
|---|---|
| Nested layouts revisited | 122 |
| Introducing the constraint layout | 124 |
| Constraint layouts are part of Android Jetpack | 125 |
| Use Gradle to include Jetpack libraries | 127 |
| Let's add a constraint layout to activity_main.xml | 128 |
| Add a button to the blueprint | 129 |
| Position views using constraints | 130 |
| Add a vertical constraint too | 131 |
| Use opposing constraints to center views | 133 |
| Remove constraints with the constraint widget | 135 |
| Changes to the blueprint appear in the XML | 136 |
| Views can have bias | 137 |
| You can change a view's size | 139 |
| Most layouts need multiple views | 144 |
| You can connect views to other views | 145 |
| Align views using guidelines | 147 |
| Guidelines have a fixed position | 148 |
| Create a movable barrier | 149 |
| Add a horizontal barrier | 150 |
| Constrain a button under the barrier | 151 |
| Use a chain to control a linear group of views | 154 |
| Create the horizontal chain | 156 |
| There are different styles of chain | 159 |
| A flow is like a multi-line chain | 162 |
| How to add a flow | 163 |
| You can control the flow's appearance | 164 |
| Your Android Toolbox | 168 |

# 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.



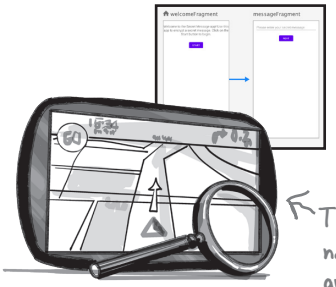


How do activities really work? 170
Create a new project 171
The full code for activity_main.xml 172
The activity code controls the stopwatch 173
The full code for MainActivity.kt 174
What happens when you run the app 176
Rotating the screen changes the device configuration 180
An activity's states 181
The activity lifecycle: from create to destroy 182
Your activity inherits the lifecycle methods 183
Save the current state in a Bundle 184
Save the state using onSaveInstanceState() 185
The updated MainActivity.kt code 186
What happens when you run the app 188
There's more to an activity's life than create and destroy 193
The visible lifecycle 194
We need to implement two more lifecycle methods 195
Restart the stopwatch when the app becomes visible 196
What happens when you run the app 198
What if an activity is only partially visible? 204
The foreground lifecycle 205
Pause the stopwatch if the activity's paused 206
The complete MainActivity.kt code 209
What happens when you run the app 212
Your handy guide to the activity lifecycle methods 214
Your Android Toolbox 217

fragments and navigation

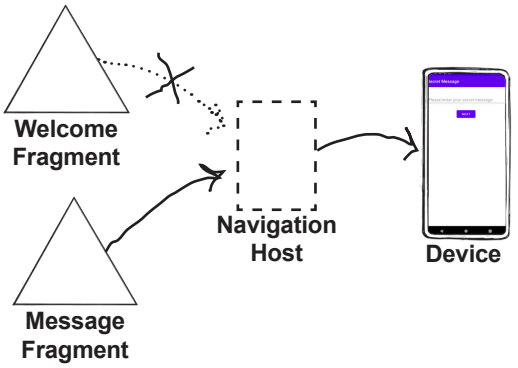
# 6 Finding Your Way

**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.



Navigation Graph

The navigation graph is like a GPS that tells you how to get to different destinations.



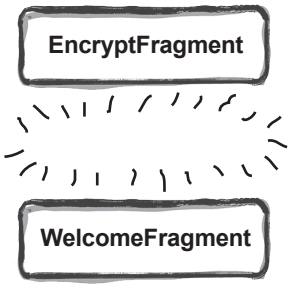


| | |
|---|---|
| Most apps need more than one screen | 220 |
| Each screen is a fragment | 221 |
| Navigate between screens using the Navigation component | 222 |
| Create a new project | 224 |
| Add WelcomeFragment to the project | 225 |
| What fragment code looks like | 226 |
| The fragment's onCreateView() method | 227 |
| Fragment layout code looks like activity layout code | 228 |
| You display a fragment in a FragmentContainerView | 229 |
| Update the activity_main.xml code | 230 |
| What the code does | 231 |
| Create MessageFragment | 235 |
| Use the Navigation component to navigate between fragments | 238 |
| Use Gradle to add the Navigation component to your project | 239 |
| Create a navigation graph | 240 |
| Add fragments to the navigation graph | 241 |
| Connect fragments using an action | 242 |
| Navigation graphs are XML resources | 243 |
| Add a navigation host to the layout using a FragmentContainerView | 244 |
| Add a NavHostFragment to activity_main.xml | 245 |
| Add an OnClickListener to the button | 246 |
| Get a navigation controller | 248 |
| The full code for WelcomeFragment.kt | 249 |
| What happens when the app runs | 250 |
| Your Android Toolbox | 255 |

# 7 safe args

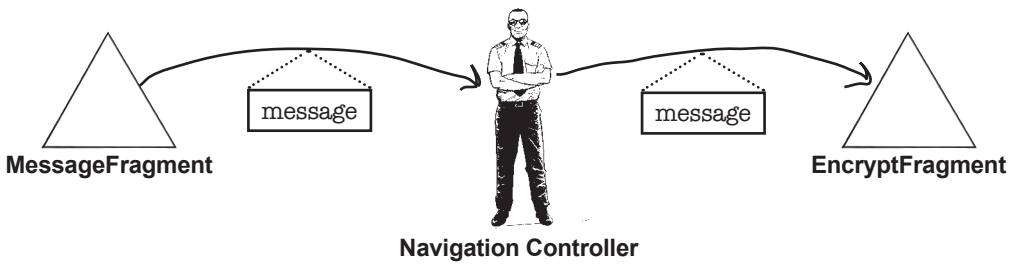
# Passing Information

**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.


| | |
|---|---|
| The Secret Message app navigates between fragments | 258 |
| MessageFragment needs to pass the message to a new fragment | 259 |
| Create EncryptFragment | 261 |
| Add EncryptFragment to the navigation graph | 263 |
| MessageFragment needs to navigate to EncryptFragment | 265 |
| Add Safe Args to the build.gradle files | 267 |
| EncryptFragment needs to accept a String argument | 268 |
| MessageFragment needs to pass a message to EncryptFragment | 270 |
| Safe Args generates Directions classes | 271 |
| Update the MessageFragment.kt code | 272 |
| EncryptFragment needs to get the argument's value | 273 |
| The full code for EncryptFragment.kt | 274 |
| What happens when the app runs | 275 |
| What if the user wants to go back? | 282 |
| Welcome to the back stack | 284 |
| Use the navigation graph to pop fragments off the back stack | 285 |
| Your Android Toolbox | 292 |

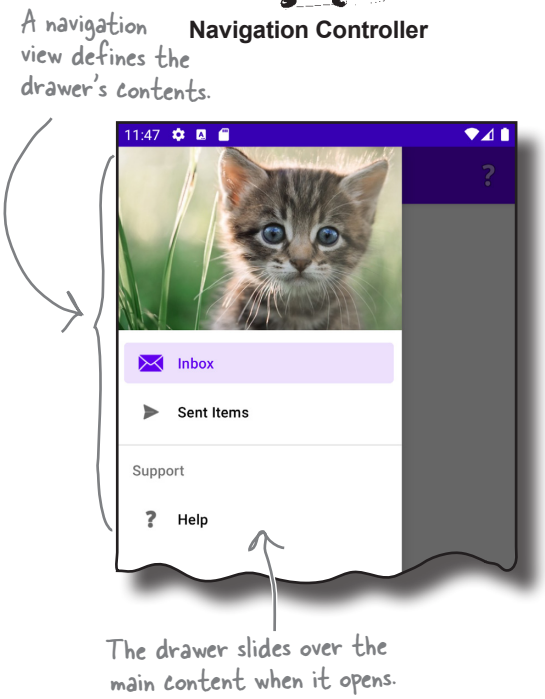



navigation ui

# 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.



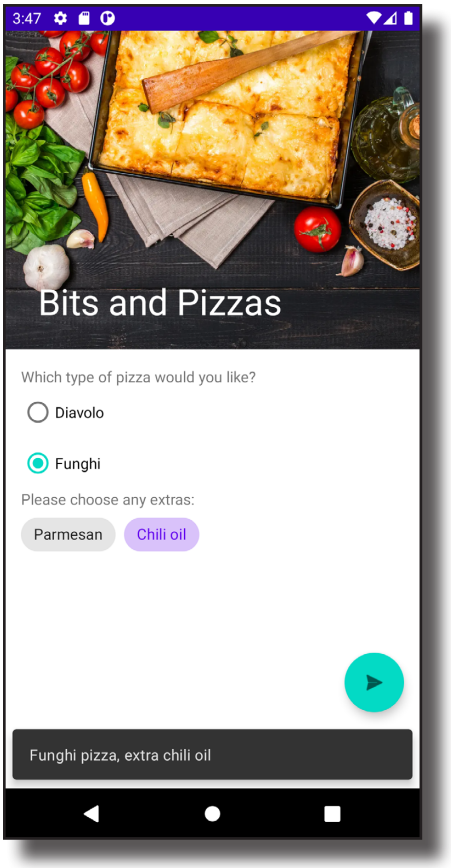


| | |
|---|---|
| Different apps, different structures | 294 |
| Android includes navigation UI components | 295 |
| How the CatChat app will work | 296 |
| Apply a theme in AndroidManifest.xml | 300 |
| Define styles in style resource files | 301 |
| Replace the default app bar with a toolbar | 303 |
| Create InboxFragment and HelpFragment | 308 |
| Specify items in the toolbar with a menu resource file | 315 |
| onCreateOptionsMenu() adds menu items to the toolbar | 317 |
| Respond to menu item clicks with onOptionsItemSelected() | 318 |
| Configure the toolbar using an AppBarConfiguration | 320 |
| What happens when the app runs | 322 |
| Most types of UI navigation work with the Navigation component | 327 |
| Create SentItemsFragment | 328 |
| The bottom navigation bar needs a new menu resource file | 330 |
| Link the bottom navigation bar to the navigation controller | 333 |
| A navigation drawer lets you display many navigation items | 336 |
| Add the support section | 340 |
| Highlight the selected item with groups | 341 |
| Create the navigation drawer's header | 343 |
| How to create a navigation drawer | 344 |
| Configure the toolbar's drawer icon | 347 |
| Your Android Toolbox | 353 |

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**.



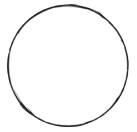


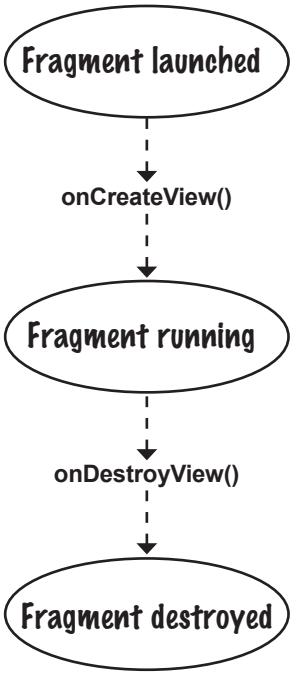
| | |
|---|---|
| Material is used throughout Androidville | 356 |
| The Bits and Pizzas app | 357 |
| Create OrderFragment | 360 |
| Replace the default app bar with a toolbar | 362 |
| Fragments don't have a setSupportActionBar() method | 363 |
| The coordinator layout coordinates animations between views | 365 |
| The app bar layout enables toolbar animation | 366 |
| Tell the toolbar how to respond to scroll events | 367 |
| How to create a plain collapsing toolbar | 374 |
| How to add an image to a collapsing toolbar | 375 |
| We need to build OrderFragment's main content | 380 |
| Choose a pizza type using a radio button | 381 |
| A chip is a type of flexible compound button | 383 |
| Add multiple chips to a chip group | 384 |
| A FAB is a Floating Action Button | 385 |
| You can anchor a FAB to a collapsing toolbar | 386 |
| We need to build OrderFragment's layout | 387 |
| Add an OnClickListener to the FAB | 393 |
| A toast is a simple pop-up message | 394 |
| Display the pizza order in a snackbar | 395 |
| The snackbar code for the pizza order | 396 |
| The full code for OrderFragment.kt | 397 |
| Your Android Toolbox | 402 |

view binding

# 10 Bound Together

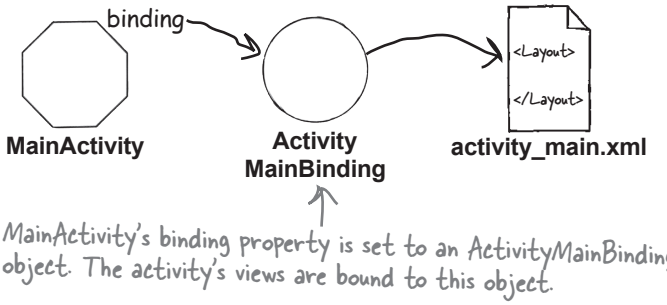
**It's time to wave farewell to findViewById().** As you've probably noticed by now, the more views you have and the more interactive your apps become, **the more calls you need to make to *findViewById()***. And if you're getting tired of typing the code for this method every time you want to work with 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.





| | |
|---|---|
| Behind the scenes of findViewById() | 404 |
| There's a downside to findViewById() | 406 |
| View binding to the rescue | 407 |
| Here's how we'll use view binding | 408 |
| The stopwatch app revisited | 409 |
| Enable view binding in the app build.gradle file | 410 |
| How to add view binding to an activity | 411 |
| Use the binding property to interact with views | 412 |
| The full code for MainActivity.kt | 413 |
| What the code does | 416 |
| Fragments can use view binding too (but the code's a little different) | 419 |
| Enable view binding for Bits and Pizzas | 420 |
| Fragment view binding code is a little different | 421 |
| Fragments can access views from onCreateView() to onDestroyView() | 422 |
| Fragment lifecycle methods up close | 423 |
| What fragment view binding code looks like | 424 |
| _binding refers to the binding object | 425 |
| The full code for OrderFragment.kt | 427 |
| Your Android Toolbox | 433 |

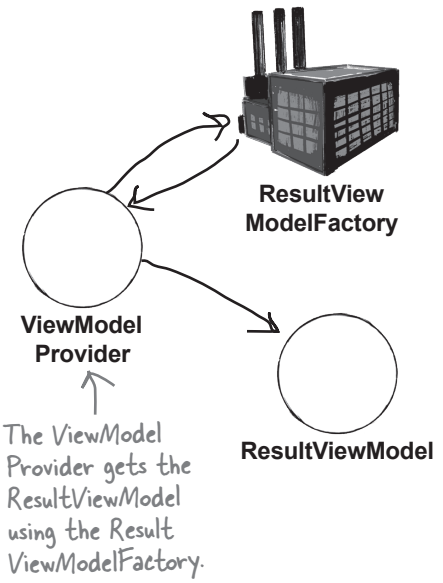



view models

# 11 Model Behavior

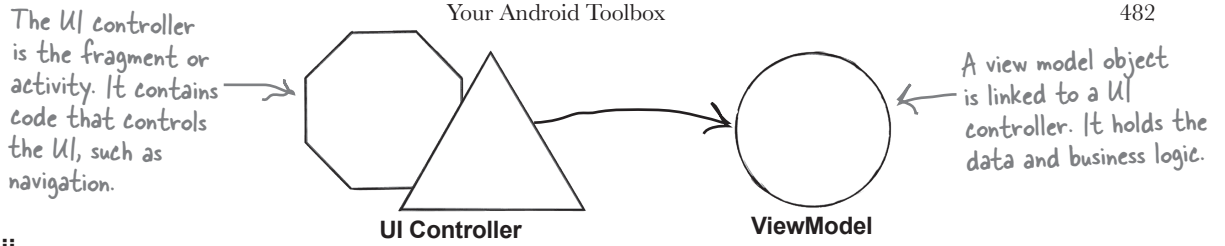
## As apps grow more complex, fragments have more to juggle.

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.





| | |
|---|---|
| Configuration changes revisited | 436 |
| Introducing the view model | 437 |
| What the guessing game will do | 438 |
| How the app will be structured | 439 |
| Update the build.gradle files | 441 |
| The Guessing Game app has two fragments | 442 |
| How navigation should work | 443 |
| Update the navigation graph | 444 |
| What happens when the app runs | 453 |
| The game loses state when the screen rotates | 455 |
| A view model holds business logic | 456 |
| Add a view model dependency to the app build.gradle file | 457 |
| Create a GameViewModel object | 460 |
| What happens when the app runs | 463 |
| View models up close | 466 |
| ResultViewModel needs to hold the result | 470 |
| A view model factory creates view models | 471 |
| Create the ResultViewModelFactory class | 472 |
| Use the factory to create the view model | 473 |
| What happens when the app runs | 476 |
| Your Android Toolbox | 482 |

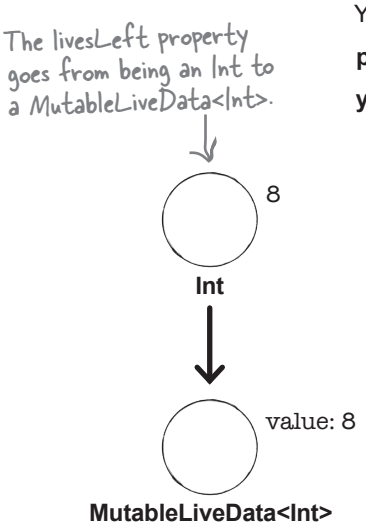



live data

# 12 Leaping into Action

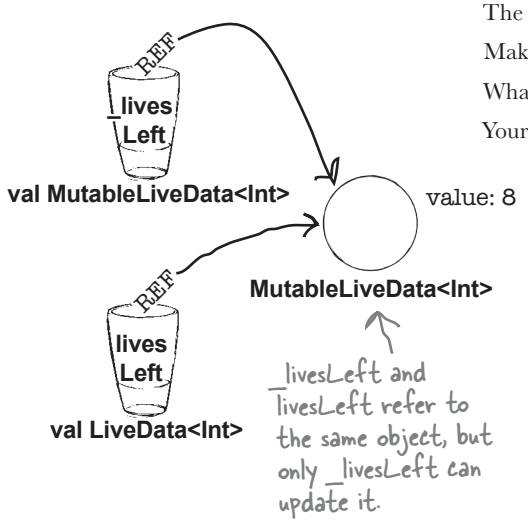
## Your code often needs to react to property value changes.

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**.





| | |
|---|---|
| The Guessing Game app revisited | 484 |
| The fragments decide when to update views | 485 |
| Here's what we're going to do | 486 |
| GameViewModel and GameFragment need to use live data | 487 |
| Live data objects use a value property | 488 |
| The fragment observes the view model properties and reacts to changes. | 492 |
| The full code for GameFragment.kt | 493 |
| What happens when the app runs | 496 |
| Fragments can update GameViewModel's properties | 501 |
| What happens when the app runs | 504 |
| GameFragment still includes game logic | 506 |
| The full code for GameViewModel.kt | 509 |
| Make GameFragment observe the new property | 511 |
| What happens when the app runs | 513 |
| Your Android Toolbox | 518 |

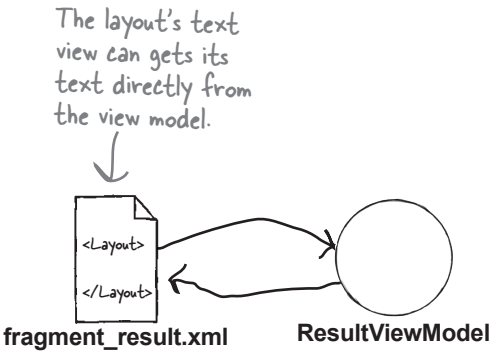



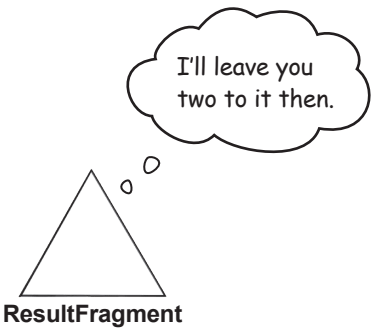
data binding

# 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**.



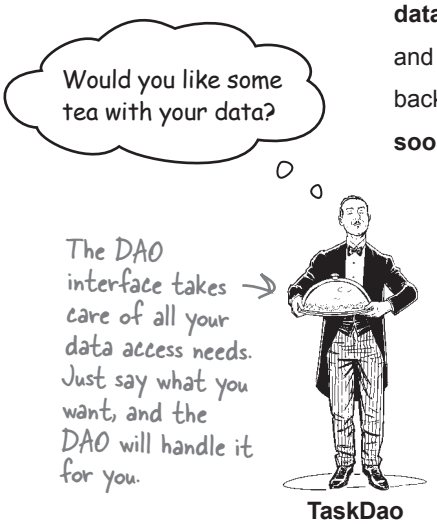


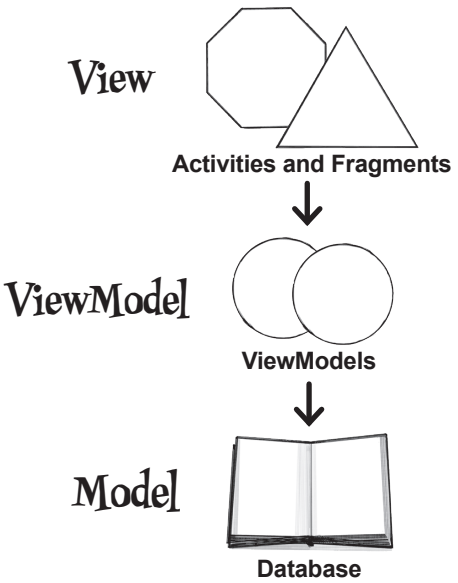
| | |
|---|---|
| Back to the Guessing Game app | 520 |
| The fragments update the views in their layouts | 521 |
| Enable data binding in the app build.gradle file | 523 |
| ResultFragment updates the text in its layout | 524 |
| 1. Add <layout> and <data> elements | 525 |
| 2. Set the layout's data binding variable | 526 |
| 3. Use the layout's data binding variable to access the view model | 527 |
| What happens when the app runs | 530 |
| <layout> up close | 532 |
| GameFragment can use data binding too | 535 |
| Add <layout> and <data> elements to fragment_game.xml | 536 |
| Use the data binding variable to set the layout's text | 537 |
| String resources revisited | 538 |
| The layout can pass parameters to String resources | 539 |
| We need to set the gameViewModel variable | 542 |
| What happens when the app runs | 545 |
| You can use data binding to call methods | 549 |
| Add finishGame() to GameViewModel.kt | 550 |
| Use data binding to make a button call a method when clicked | 552 |
| What happens when the app runs | 555 |
| Fireside chats: view binding vs. data binding | 558 |
| We can switch off view binding | 560 |
| Your Android Toolbox | 567 |

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()***.



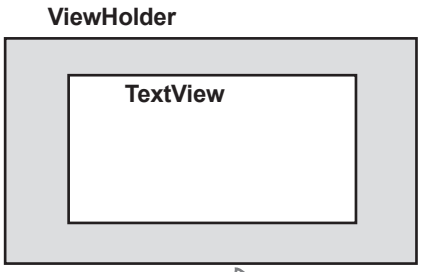


| | |
|---|---|
| Most apps need to store data | 570 |
| Room is a database library that sits on top of SQLite | 572 |
| Create TasksFragment | 575 |
| How Room databases are created | 578 |
| We'll store tasks data in a table | 579 |
| Specify a table name with @Entity | 580 |
| Use an interface to specify data operations | 582 |
| Use @Insert to insert a record | 583 |
| Use @Delete to delete a record | 584 |
| Create a TaskDatabase abstract class | 586 |
| Add properties for any DAO interfaces | 587 |
| MVVM revisited | 591 |
| Database operations can run in slooooow-moooo | 593 |
| 1. Mark TaskDao's methods with suspend | 594 |
| 2. Launch the insert() method in the background | 595 |
| TasksViewModel needs a view model factory | 596 |
| We'll use data binding to insert a record | 600 |
| What happens when the code runs | 603 |
| TasksFragment needs to display records | 607 |
| Use getAll() to get all tasks from the database | 608 |
| A LiveData<List<Task>> is a more complex type | 609 |
| Let's update the TasksViewModel code | 610 |
| We'll bind the tasksString property to the layout's text view | 611 |
| What happens when the code runs | 614 |
| Your Android Toolbox | 620 |

# 15 recycler views

## 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**.



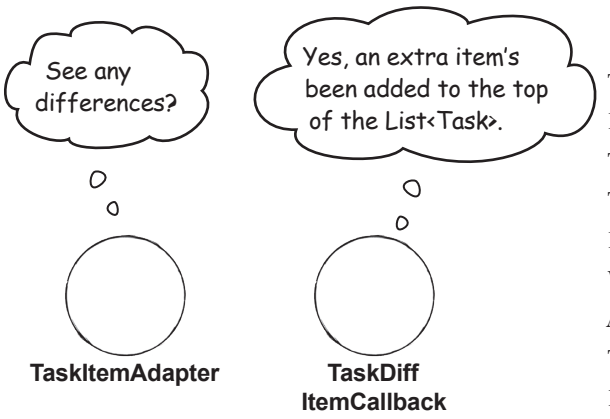

Each view holder holds the root view of the layout for each item. Here the root view is a text view.


| | |
|---|---|
| What the Tasks app currently looks like | 622 |
| We can turn the list into a recycler view | 62 |
| Tell the recycler view how to display each item | 626 |
| The adapter adds data to the recycler view | 627 |
| Define the adapter's view holder | 629 |
| Override the onCreateViewHolder() method | 630 |
| Add data to the layout's view | 631 |
| We need to display the recycler view | 634 |
| We've added a recycler view to TasksFragment's layout | 638 |
| TasksFragment needs to update TaskItemAdapter's data property | 640 |
| What happens when the code runs | 643 |
| Recycler views are very flexible | 650 |
| How to create a card view | 652 |
| The full code for task_item.xml | 653 |
| The adapter's view holder needs to work with the new layout code | 654 |
| The full code for TaskItemAdapter.kt | 655 |
| The layout manager gallery | 658 |
| Update fragment_tasks.xml to arrange items in a grid | 659 |
| What happens when the code runs | 660 |
| Your Android Toolbox | 669 |

diffutil and data binding

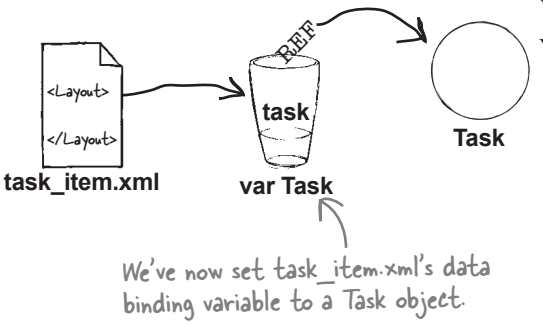
# 16 Life in the Fast Lane

**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**.





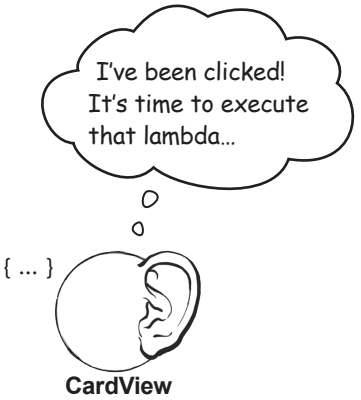
The Tasks app revisited 673
How the recycler view gets its data 674
The data property's setter calls notifyDataSetChanged() 675
Tell the recycler view what needs to change 676
Here's what we're going to do 677
We need to implement DiffUtil.ItemCallback 678
A ListAdapter accepts a DiffUtil.ItemCallback argument 679
The updated code for TaskItemAdapter.kt 680
Populate the ListAdapter's list 681
The updated code for TasksFragment.kt 682
What happens when the code runs 683
Recycler views can use data binding 687
Add a data binding variable to task_item.xml 688
The layout gets inflated in the adapter's view holder code 689
Use the binding class to inflate the layout 690
The full code for TaskItemAdapter.kt 691
The full code for task_item.xml 693
What happens when the code runs 695
Your Android Toolbox 703

recycler view navigation

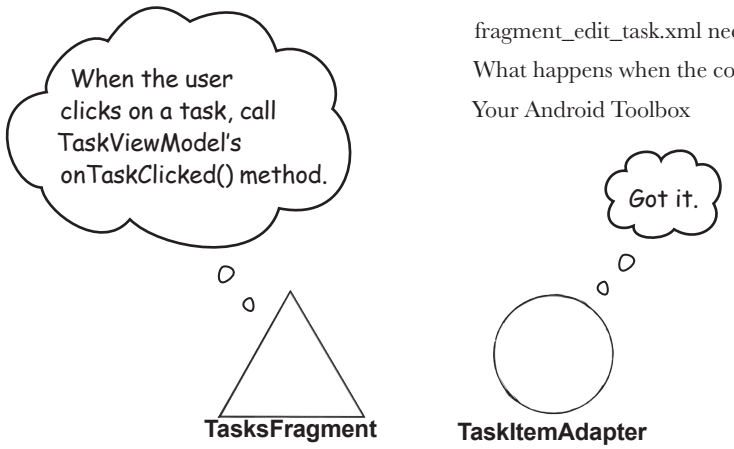
# 17 Pick a Card

**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.





| | |
|---|---|
| Recycler views can be used for navigation | 706 |
| Make each item clickable | 710 |
| Where should we create the Toast? | 711 |
| We'll pass a lambda to TaskItemAdapter | 714 |
| What happens when the code runs | 717 |
| We want to use the recycler view to navigate to a new fragment | 724 |
| Create EditTaskFragment | 726 |
| Update the navigation graph | 727 |
| Make TasksFragment navigate to EditTaskFragment | 729 |
| Make EditTaskFragment display the task ID | 735 |
| What happens when the code runs | 737 |
| We want to use EditTaskFragment to update task records | 740 |
| Use TaskDao to interact with database records | 741 |
| Create EditTaskViewModel | 742 |
| EditTaskViewModel will tell EditTaskFragment when to navigate | 743 |
| EditTaskViewModel needs a view model factory | 745 |
| fragment_edit_task.xml needs to display the Task | 746 |
| What happens when the code runs | 750 |
| Your Android Toolbox | 754 |

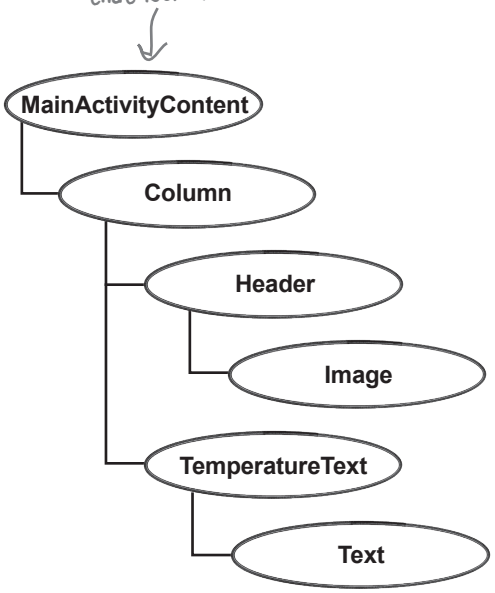



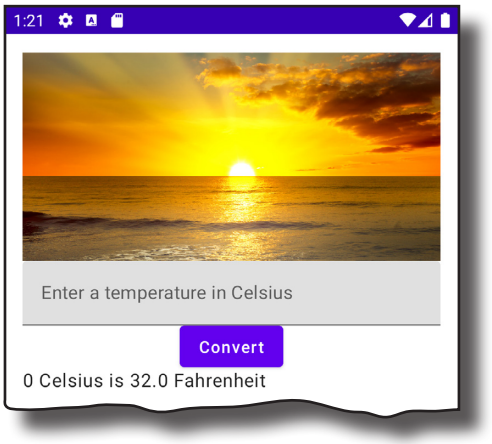
jetpack compose

# 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.

Compose builds a tree of composables that looks like this:



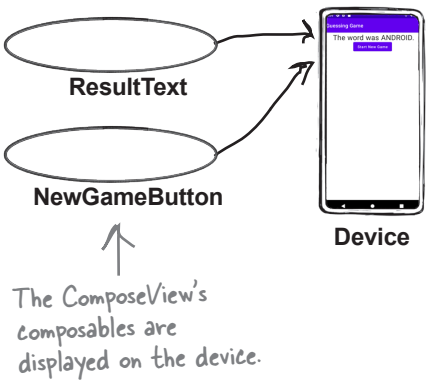


| | |
|---|---|
| UI components don't have to be Views | 756 |
| Create a new Compose project | 760 |
| Compose projects have no layout files | 762 |
| What Compose activity code looks like | 763 |
| Use a Text composable to display text | 764 |
| Use composables in composable functions | 765 |
| Preview composables with the Design or Split option | 770 |
| Let's make the app convert temperatures | 773 |
| Add a MainActivityContent composable function | 774 |
| Add an Image to MainActivity.kt | 776 |
| Let's display the temperature text | 777 |
| Use a Button composable to add a button | 778 |
| We need to pass a lambda to ConvertButton | 779 |
| Composition up close | 780 |
| We need to change the value of TemperatureText's argument | 781 |
| What happens when the app runs | 784 |
| Add a TextField to a composable function | 790 |
| What happens when the app runs | 793 |
| Add padding to the Column composable | 797 |
| You can center composables in Columns or Rows | 798 |
| Applying themes: revisited | 800 |
| Android Studio includes extra theme code | 801 |
| The full code for MainActivity.kt | 803 |
| Your Android Toolbox | 809 |

integrating compose with views

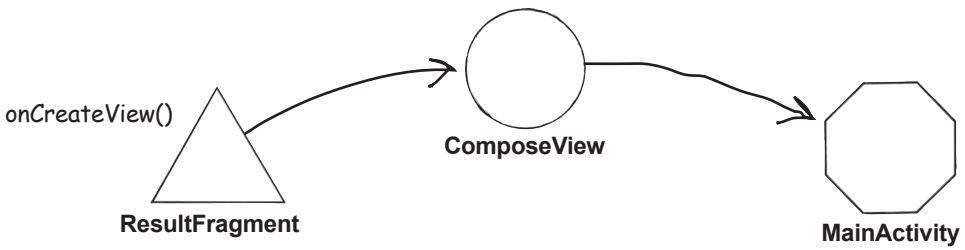
# 19 Perfect Harmony

**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.**





| | |
|---|---|
| You can add composables to View-based UIs | 812 |
| The Guessing Game app structure | 813 |
| We'll replace ResultFragment's views with composables | 816 |
| A ComposeView lets you add composables to a layout | 817 |
| Add composables using Kotlin code | 818 |
| Add a composable function for the fragment's content | 819 |
| Reproduce the Start New Game button | 820 |
| Reproduce ResultFragment's TextView | 821 |
| onCreateView() returns the UI's root View | 826 |
| The full code for ResultFragment.kt | 827 |
| What happens when the app runs | 830 |
| Next we'll make GameFragment use composables, too | 834 |
| We'll add a ComposeView to fragment_game.xml | 835 |
| Add a composable function for GameFragment's content | 836 |
| Reproduce the Finish Game button | 837 |
| Reproduce the EditText with a TextField | 838 |
| Reproduce the Guess button | 839 |
| We'll display the incorrect guesses in a Text composable | 843 |
| Create an IncorrectGuessesText composable function | 844 |
| Your Android Toolbox | 858 |

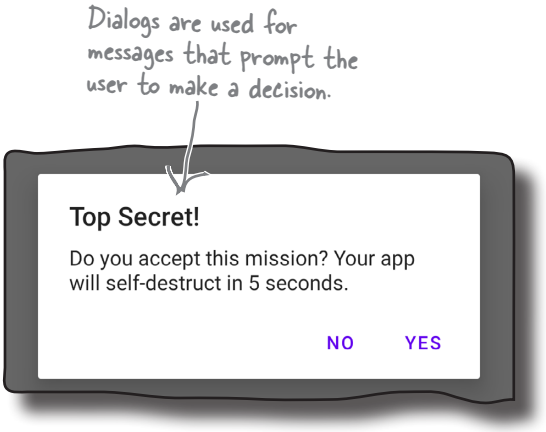



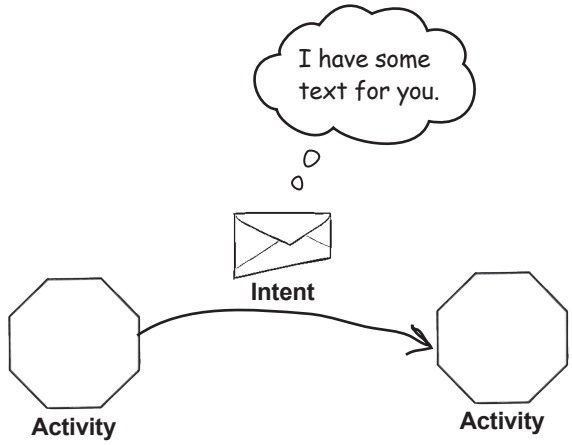
leftovers

# 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**.


1. Sharing data with other apps 862
2. WorkManager 864
3. Dialogs and notifications 865
4. Automated testing 866
5. Supporting different screen sizes 868
6. More Compose features 870
7. Retrofit 873
8. Android Game Development Kit 873
9. CameraX 873
10. Publishing your app 874



# 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]

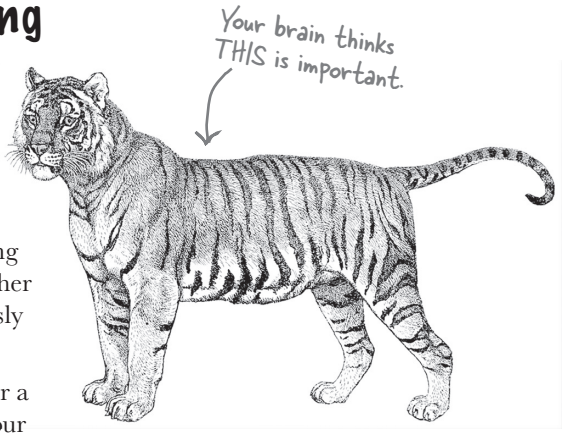
# We know what you're thinking

"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?

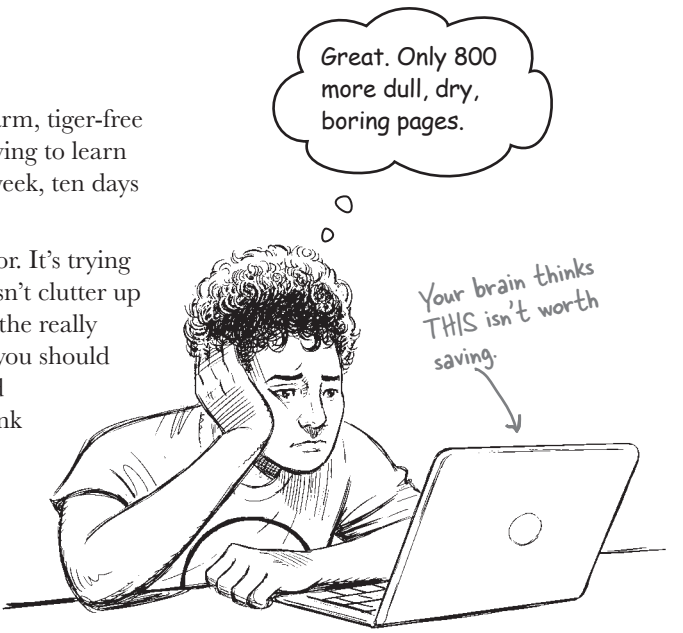
Neurons fire. Emotions crank up. *Chemicals surge*.

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.

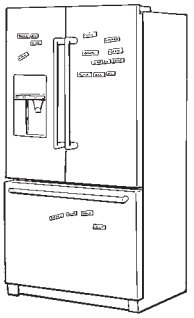
We used ***multiple learning styles***, because *you* might prefer step-by-step procedures, while someone else wants to understand the big picture first, and someone else just wants to see an example. But regardless of your own learning preference, *everyone* benefits from seeing the same content represented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

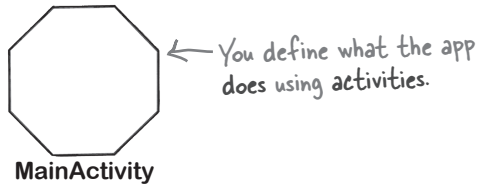
Android is a comprehensive open source platform based on Linux and championed by Google. It's a powerful development framework that includes everything you need to build great apps. What's more, it enables you to deploy those apps to a wide variety of devices—phones, tablets, and more.

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.**

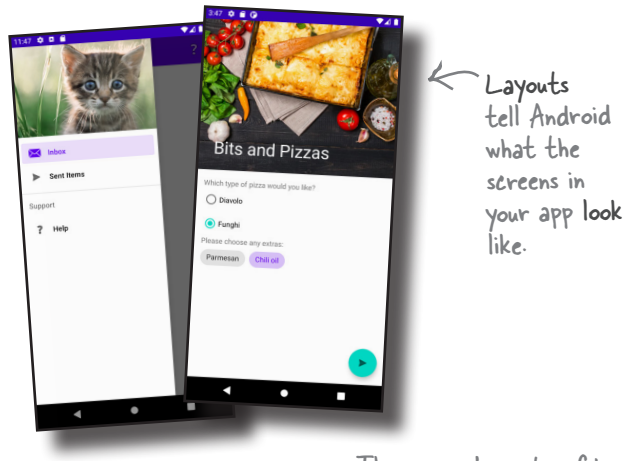
## Activities define what the app does

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.



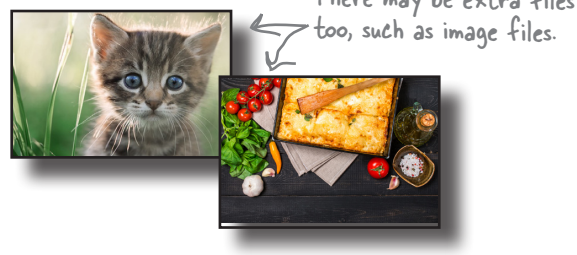

## Layouts define what each screen looks like

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

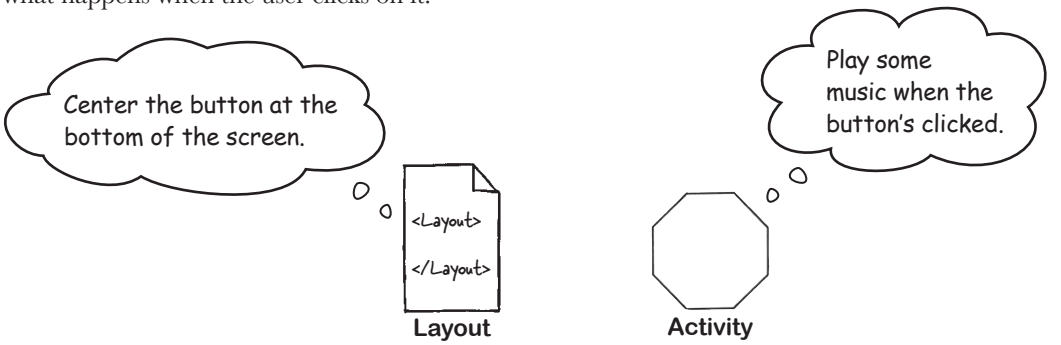
In addition to activities and layouts, Android apps often need extra resources such as image files and application data. You can add any extra files you need to the app.




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.

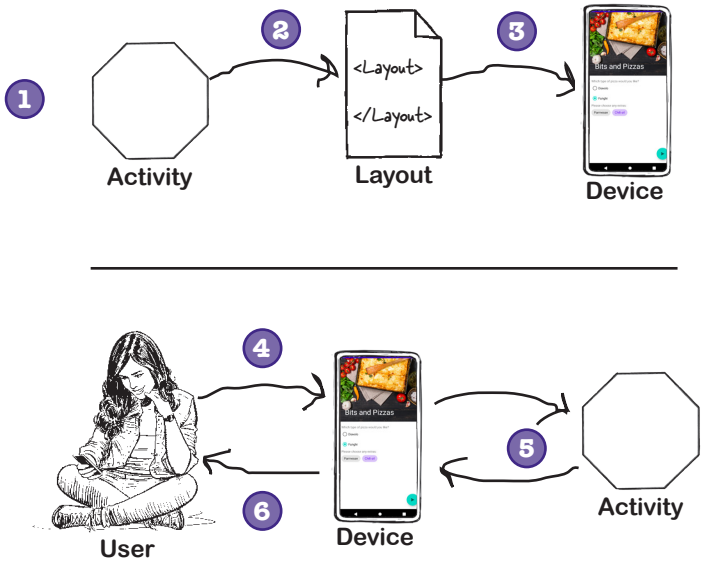
# Activities and layouts form the backbone of your app

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

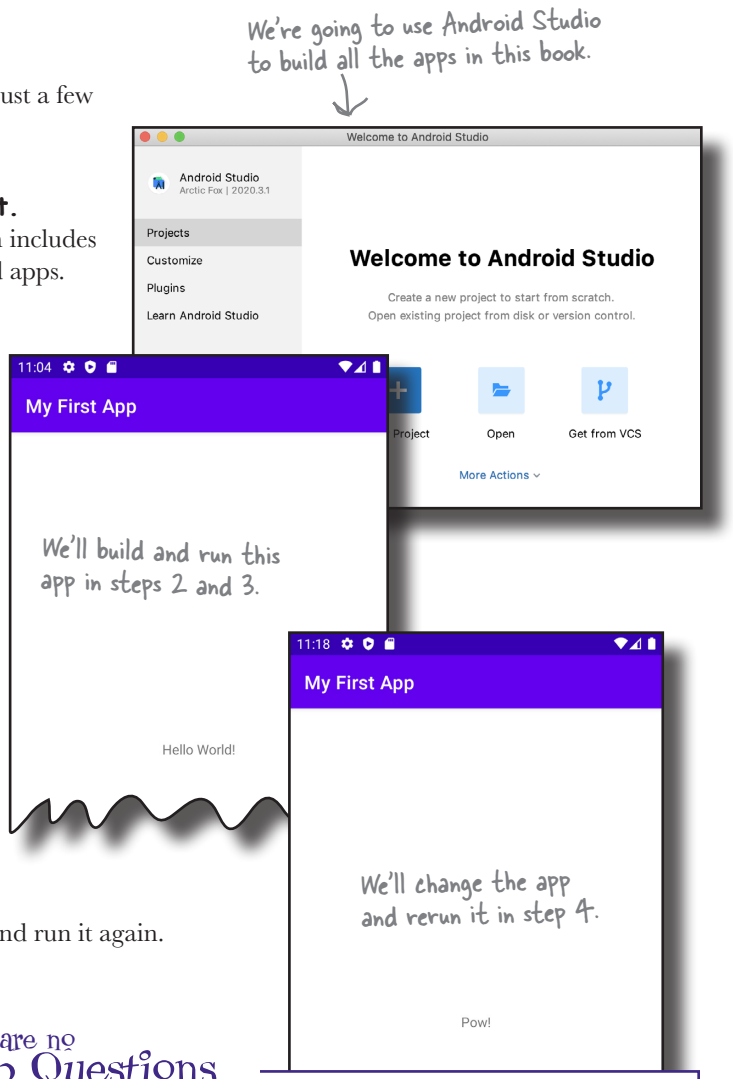
So let's dive in and create an Android app. There are just a few things we need to do:

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.

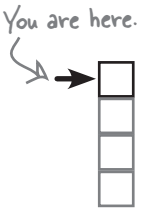
**Q: How do I learn Kotlin?**

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.

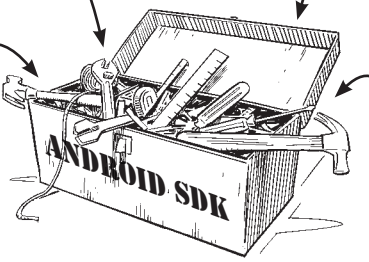
Android Studio is based on IntelliJ IDEA, which you may already be familiar with. It includes a set of code editors, UI tools and templates, which are all designed to make your life in Androidville easier.

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.



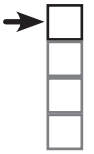

**SDK Platform**
Each version of Android has its own SDK Platform package. It lets you compile your app for that version of Android, and includes the source files for that version.

**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.

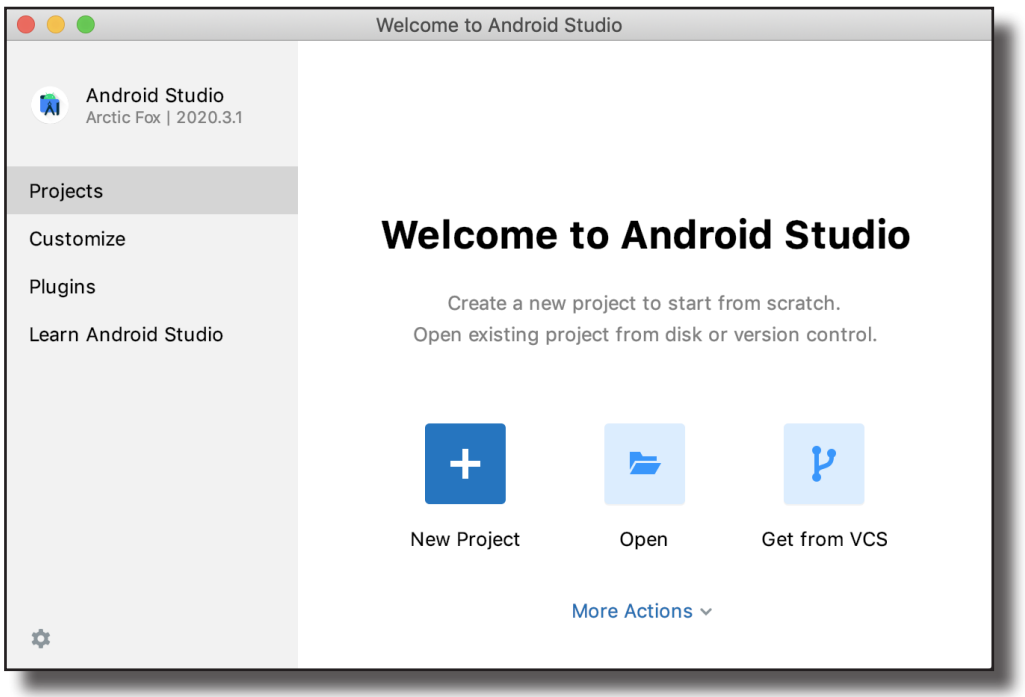
**If you've previously installed an earlier version of Android Studio, we recommend that you restore the IDE's default settings.** To do this, go to the File menu, choose Manage IDE Settings, and then select the Restore Default Settings option.

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

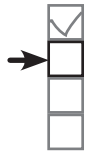
**Q: Do I have to use Android Studio to build Android apps? Can I use another IDE instead?**

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.



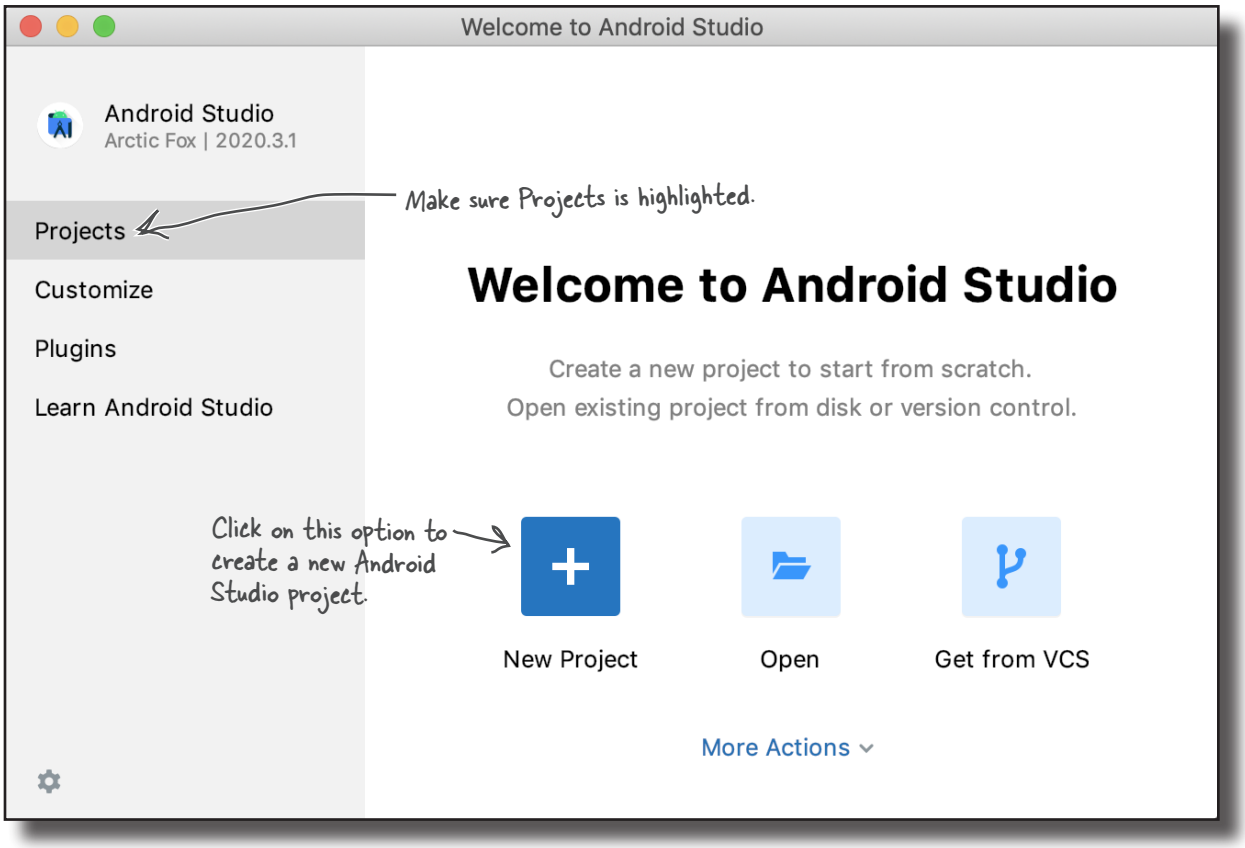
Set up environment
**Build app**
Run app
Change app

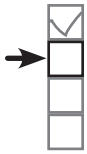
# How to build the app

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."

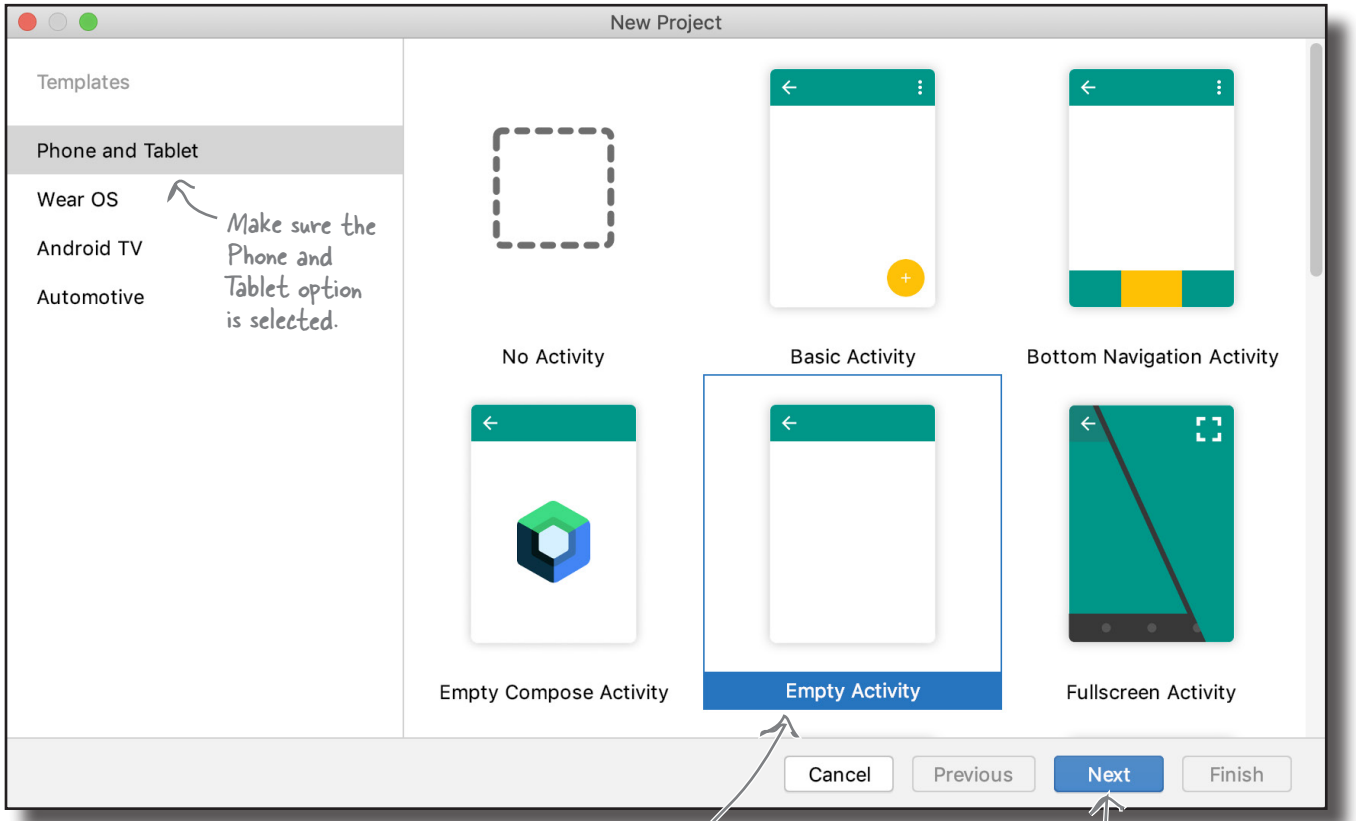


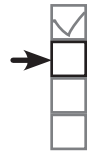


Set up environment
**Build app**
Run app
Change app

## 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.





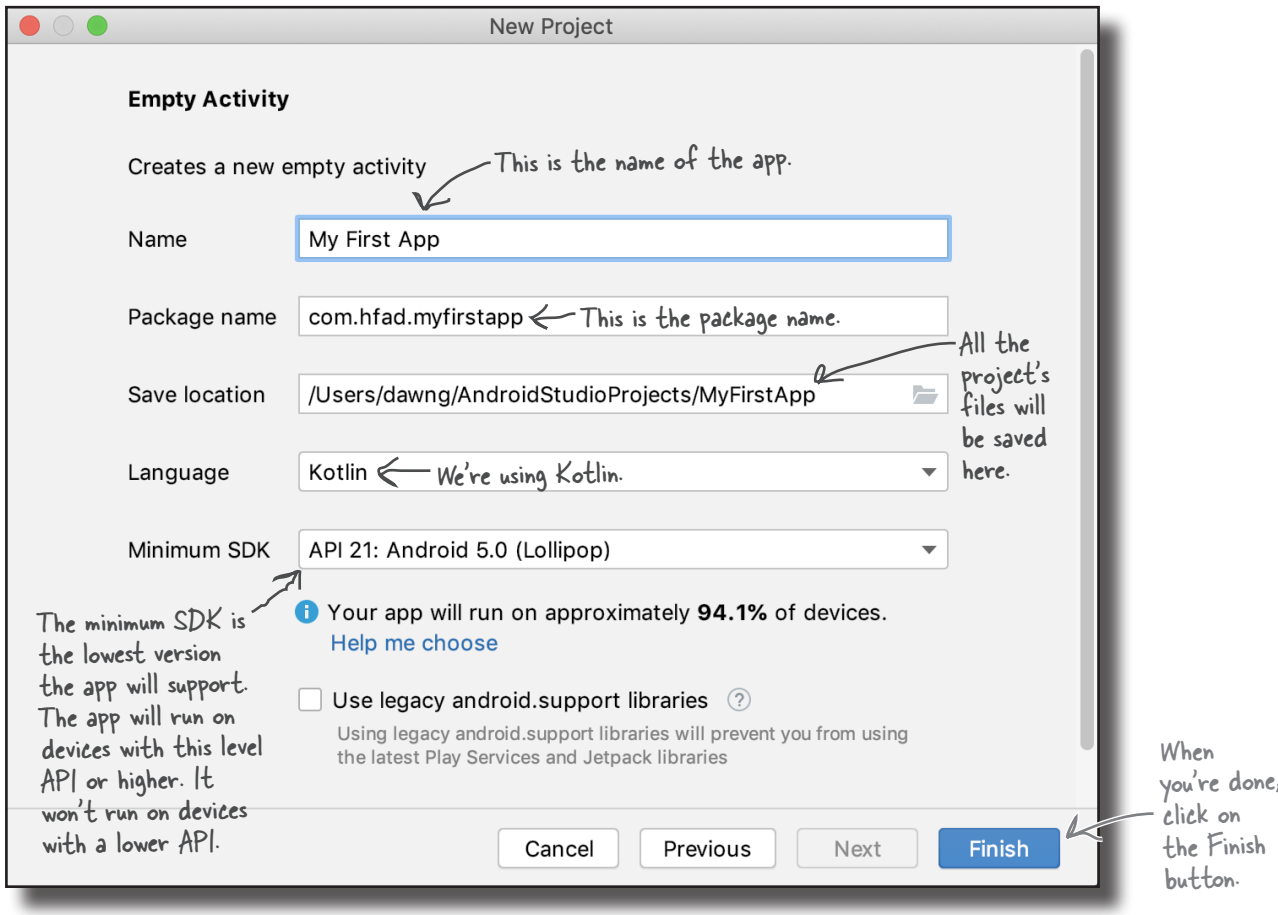
Set up environment
**Build app**
Run app
Change app

## 3. Configure your project

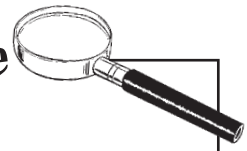
You now need to configure the project by specifying an app name, package name, and save location. Enter a name of "My First App" and a package name of "com.hfad.myfirstapp" and accept the default save location.

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:

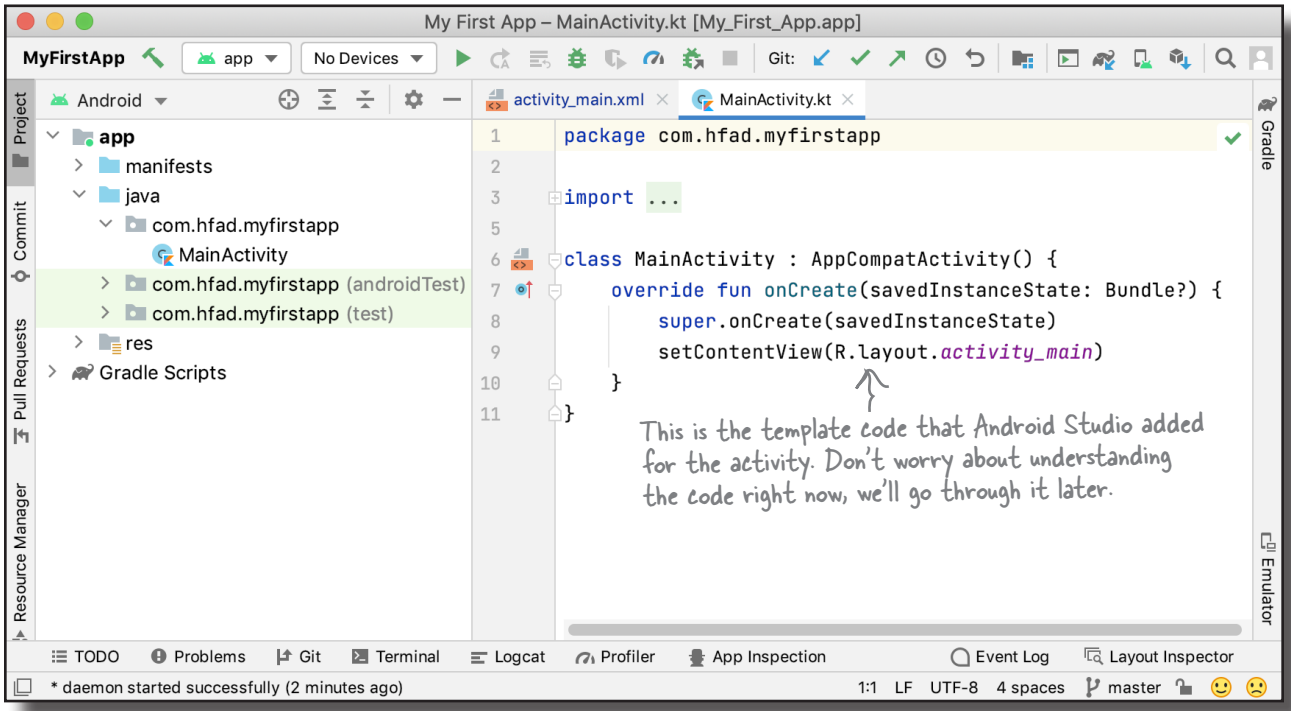
**It configures the project to your specifications.**
Android Studio looks at the minimum SDK you want the app to support, and includes all of the files and folders needed for a basic valid app. It also sets up the package structure and names the app.

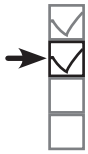
**It adds some template code.**
The template code is comprised of a layout written in XML and an activity written in Kotlin. You'll find out more about these later in the chapter.

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

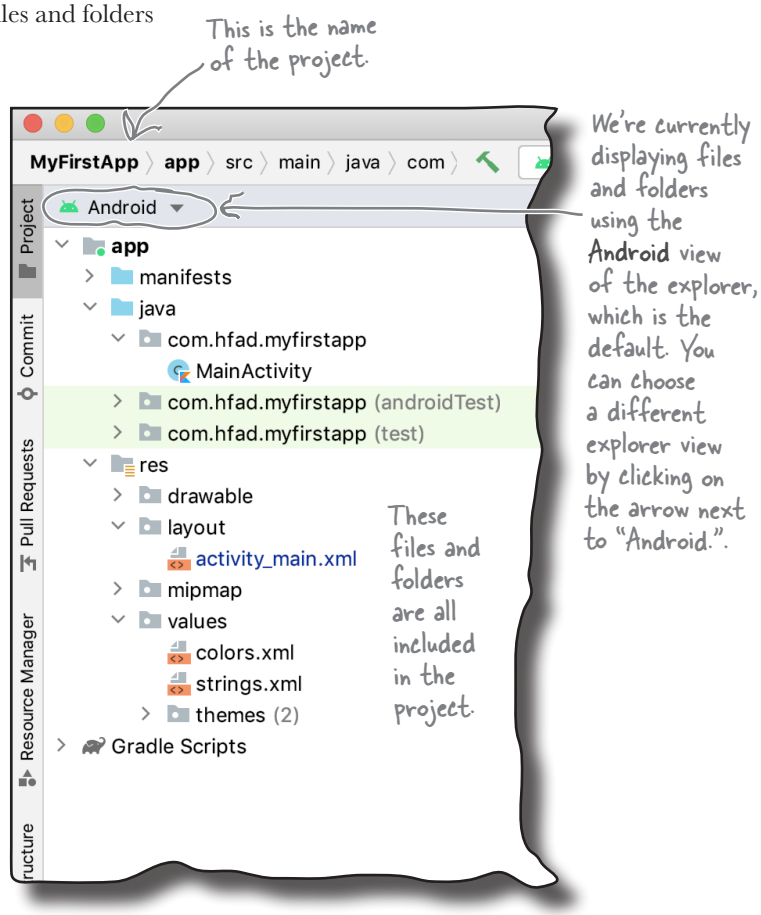
# Dissecting your new project

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.

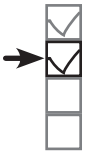
### Android libraries

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.



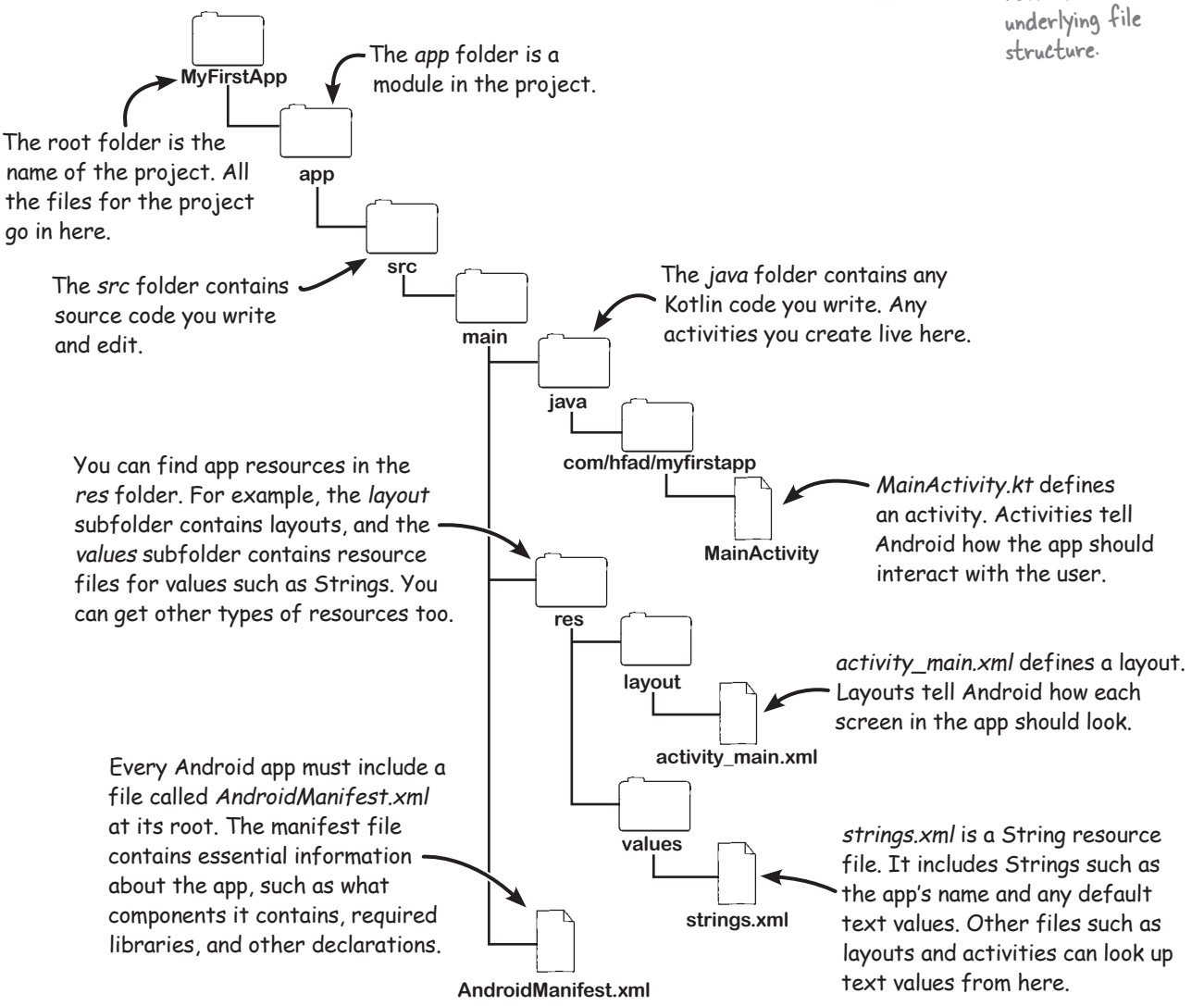
Set up environment
**Build app**
Run app
Change app

# 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

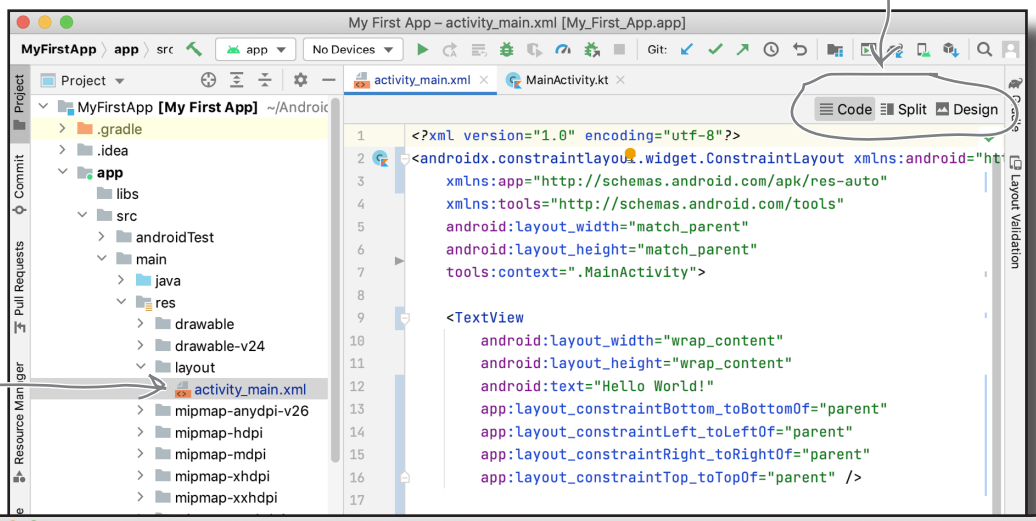
You view and edit files using the Android Studio editors. Double-click on the file you want to work with, and the file's contents will appear in the middle of the Android Studio window.

## 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.

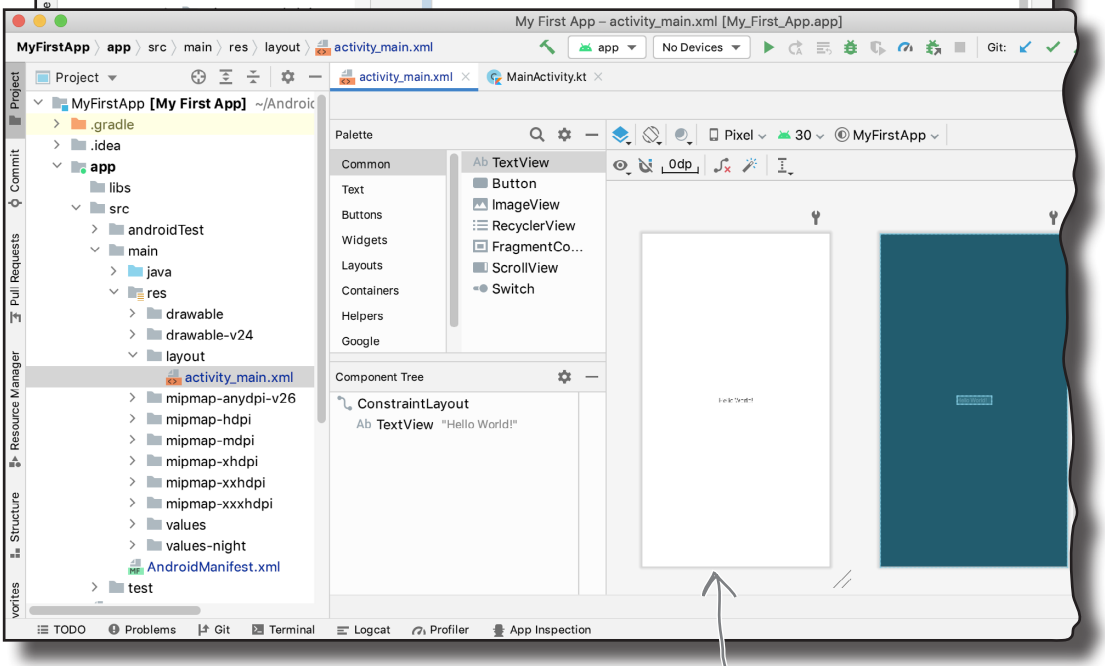
Double-click on the file in the explorer and the file contents appear in the editor panel.

You dictate which editor you're using with these.




## 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)
    }

}
```

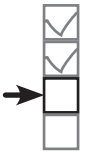
This is the package name.

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.

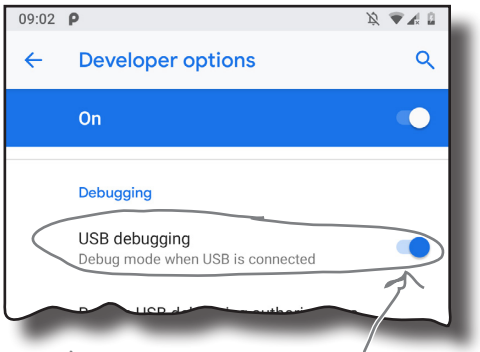
Here are the steps to run the app on a physical device:

## 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

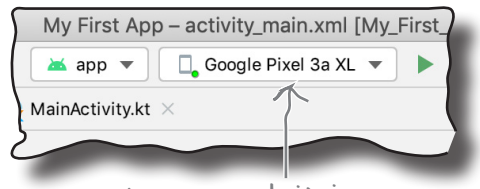
You will probably be asked if want to allow USB debugging. If so, check the "Always allow from this computer" option and choose OK.

## 4. Run the app

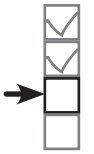
Finally, select the device from the list of devices in Android Studio's top toolbar (if it's not there, on your device go to Settings → Connected devices, select USB, and choose the "File transfer" option). Then run the app by choosing "Run 'app'" from the Run menu. Android Studio will build the project,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

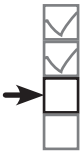
You can find system requirements for using the emulator here: https://developer.android.com/studio/run/emulator#requirements

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

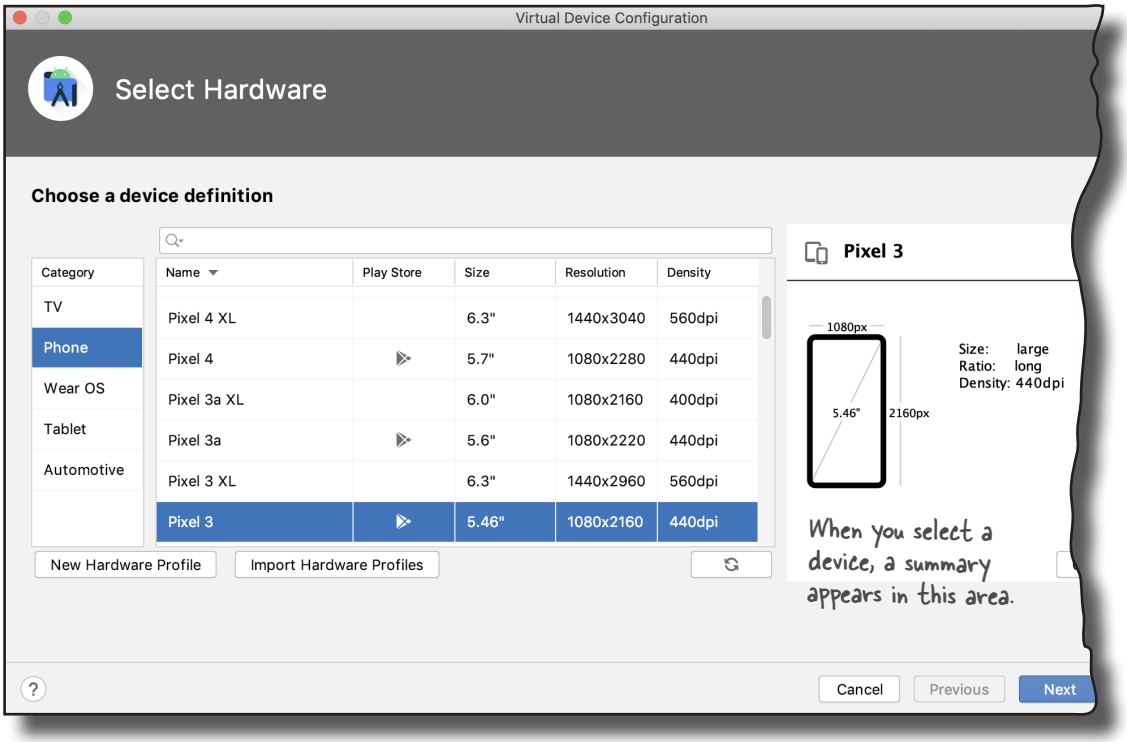
The AVD Manager allows you to set up new AVDs, and view and edit ones you've already created. Open it by selecting AVD Manager on the Tools menu.

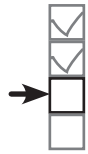
If you have no AVDs set up already, you'll be presented with a screen prompting you to create one. Click on the Create Virtual Device button.

## 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

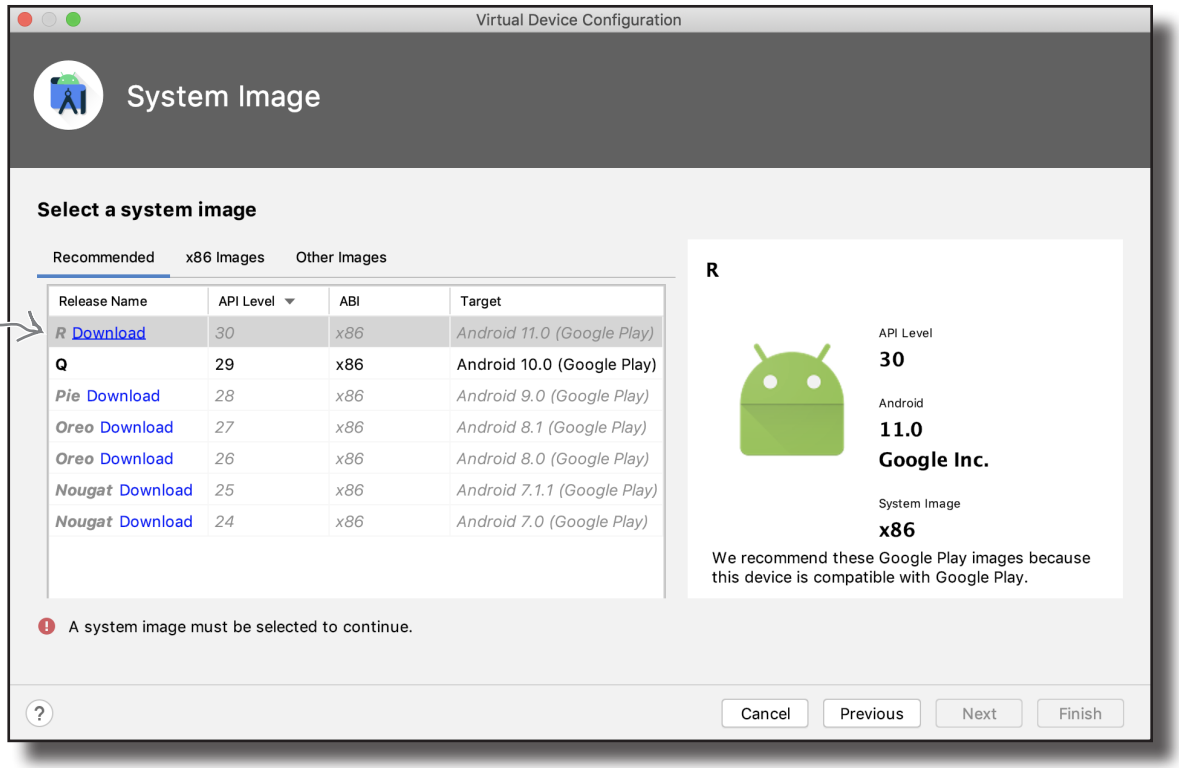
Next, you need to select a system image. This specifies which version of Android you want to be on the AVD.

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.

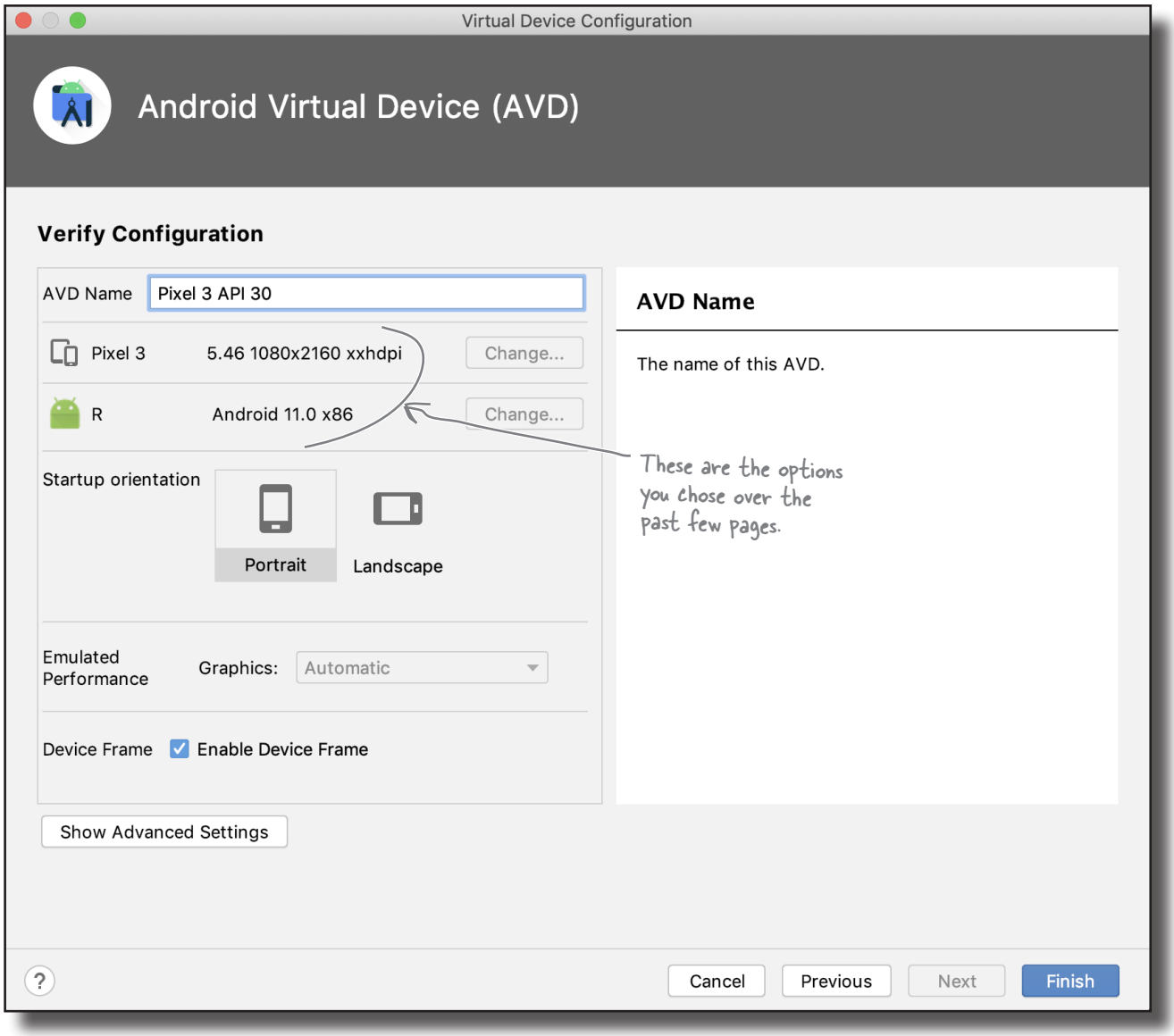
If you don't have this system image installed, you'll be presented with a Download link. Click on this link to download and install the system image.

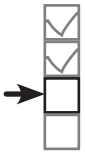


Set up environment
Build app
**Run app**
Change app

## 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.

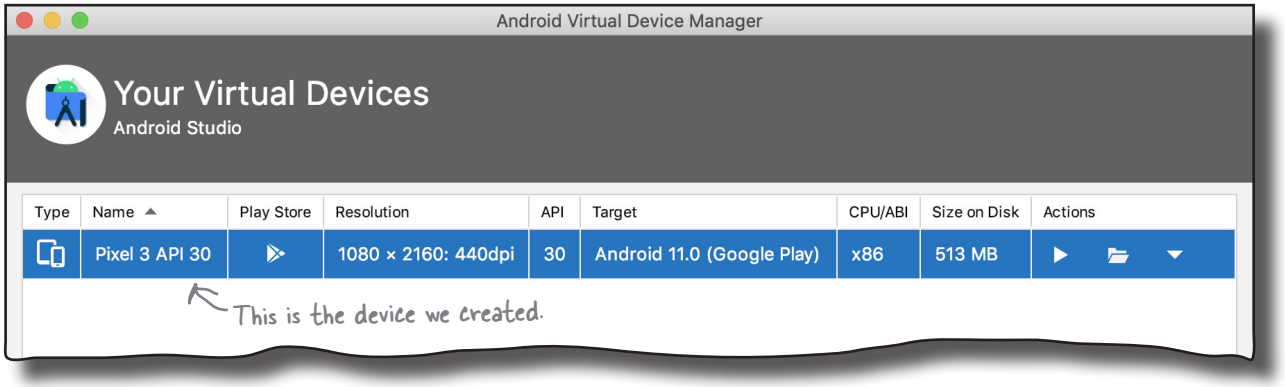




**Set up environment**
**Build app**
**Run app**
**Change app**

## 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:



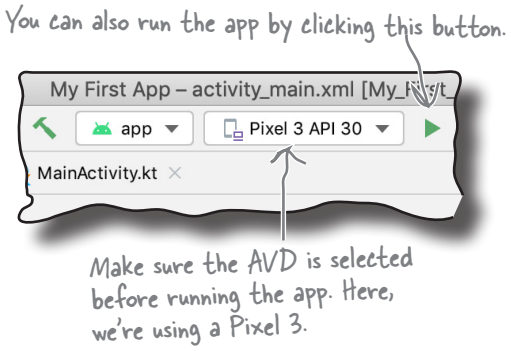

Check the new AVD is listed, then close the AVD Manager.

## 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.



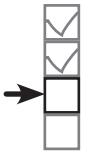

## there are no Dumb Questions

**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.

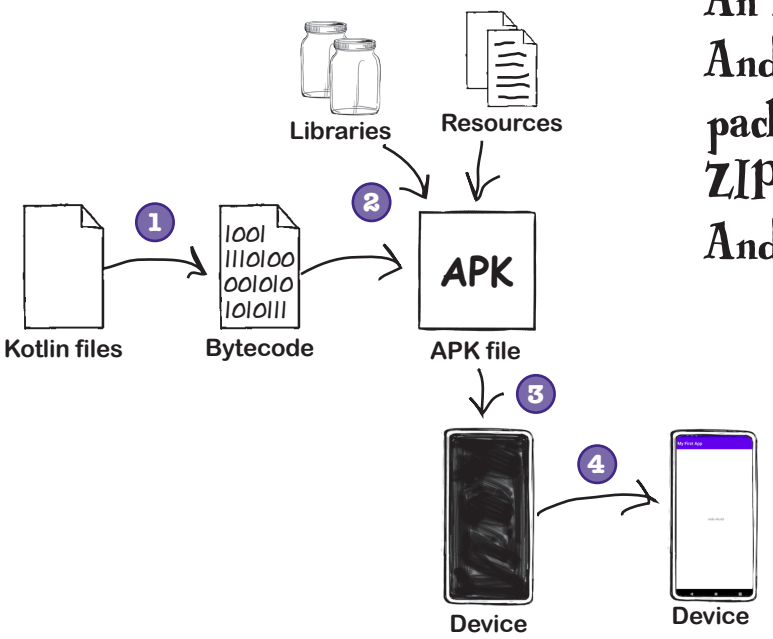




# Compile, package, deploy, run

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.

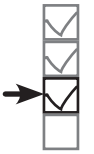
3. **The APK is installed on the device.**
   If the device is virtual, Android Studio launches the emulator and waits until the AVD is active before installing the APK.

   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.

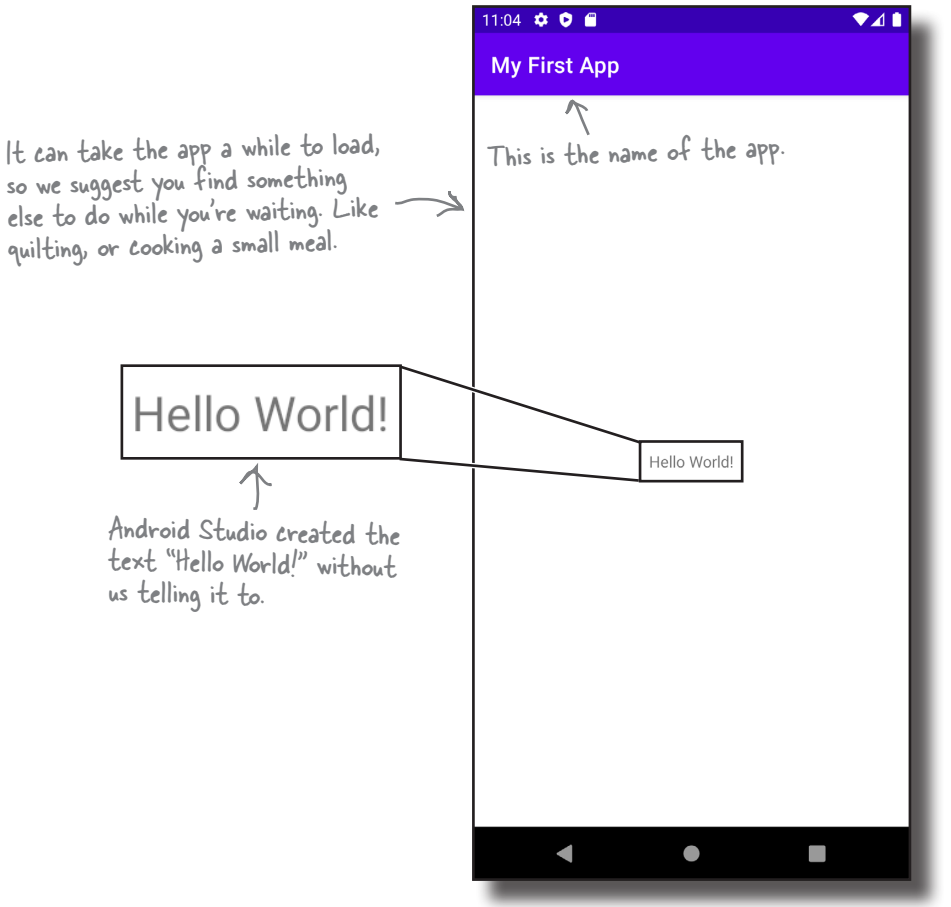
# Test Drive



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.



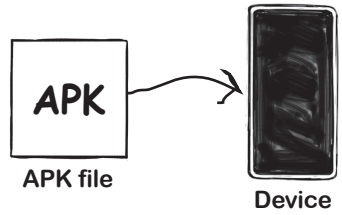

Let's run through what just happened.

Set up environment
Build app
**Run app**
Change app

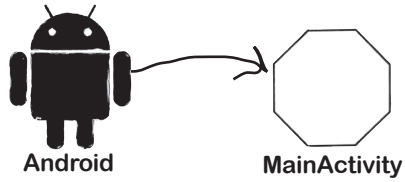
# What just happened?

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

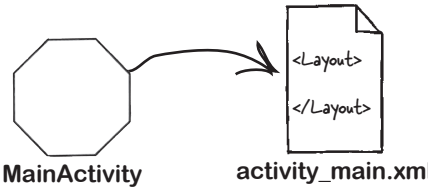
1. **Android Studio installs the app on the device.**
If the device is virtual, it waits for the emulator to start before installing the app.



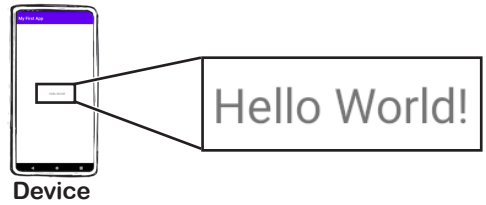

2. **Android starts the app's main activity.**
It uses the code in *MainActivity.kt* (which Android Studio automatically included in the project) to create a `MainActivity` object.




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.

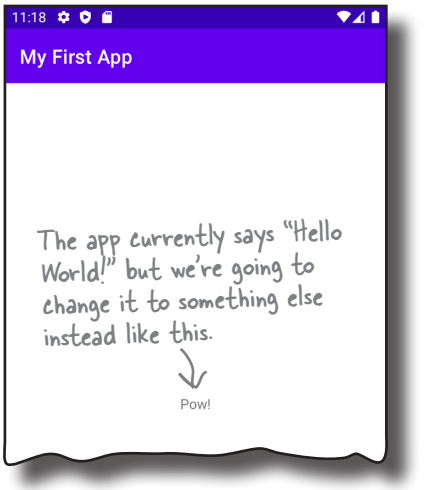
Set up environment
Build app
Run app
**Change app**

# 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

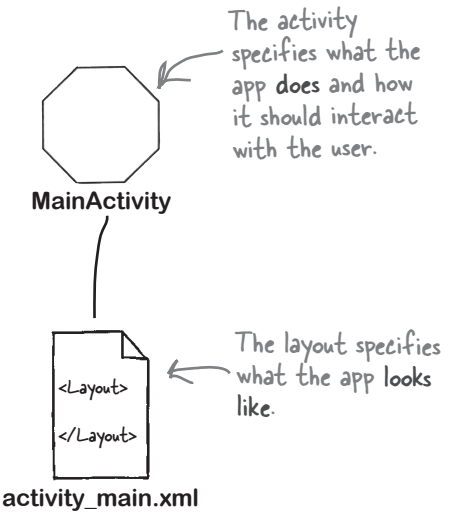
When we built the app, we told Android Studio how to configure it, and the wizard did the rest. The wizard created an activity for us, and also a default layout.

### 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.

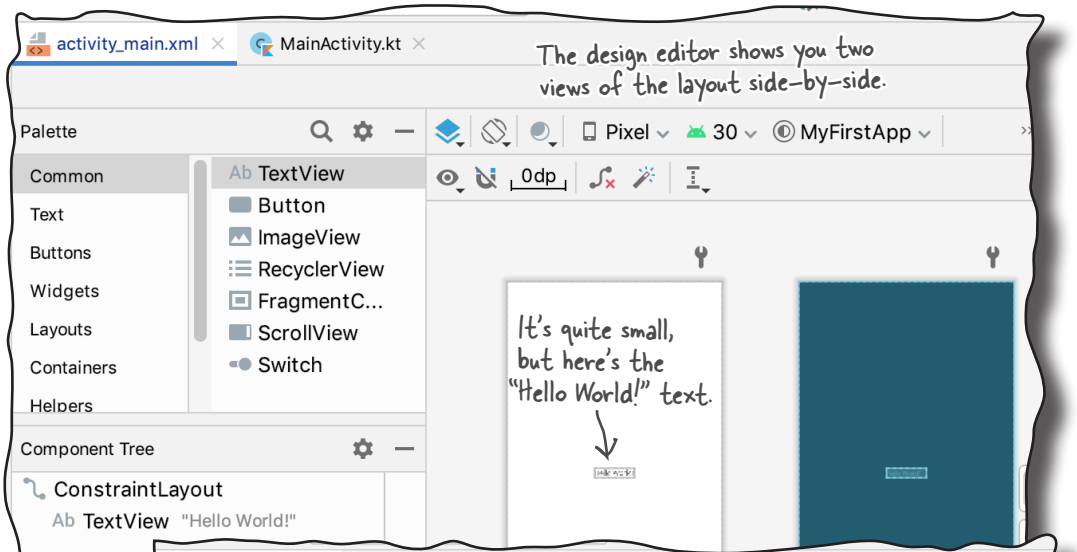
Click on this arrow to change how the files and folders are shown.

## 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?

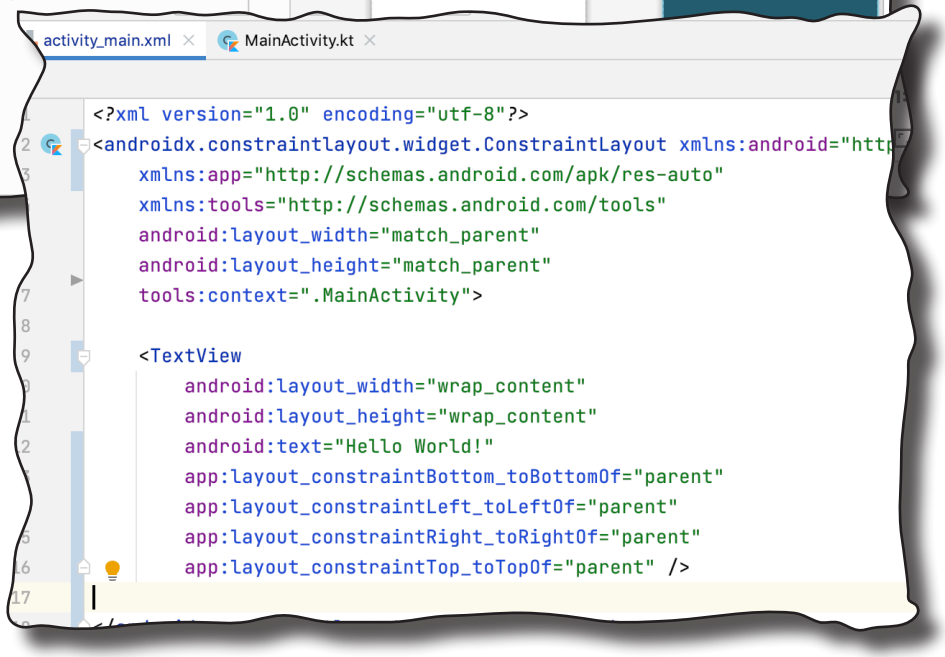
Let's see by switching to the code editor.




## 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:

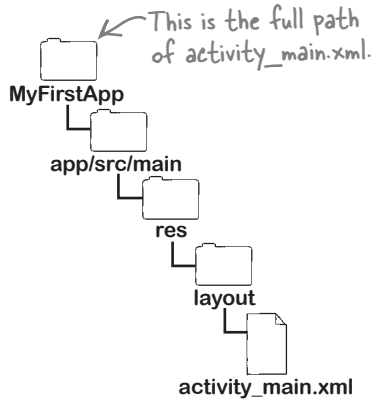
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    ... >
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        ... />
</androidx.constraintlayout.widget.ConstraintLayout>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

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!"

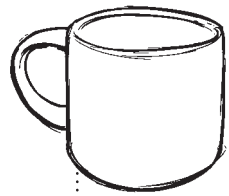
The key part of the code within the <TextView> element is the line starting with android:text. This is a text attribute that describes the text that should be displayed:

```
<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.

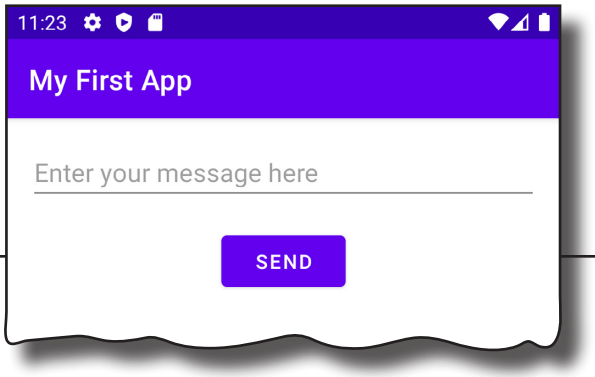
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <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>
```

XML declaration at the top of the file.

Arranges components linearly, one after another.

Defines a namespace.

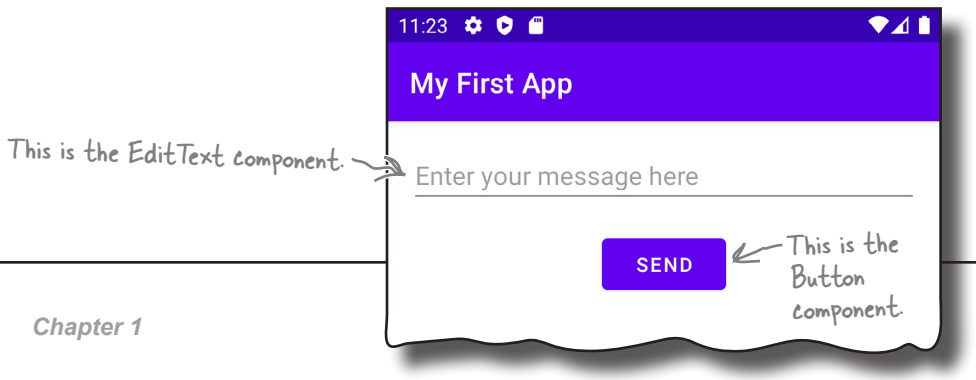
This is what appears on the screen when you run an app with the above layout.



# 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.
```



# 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!" />
```

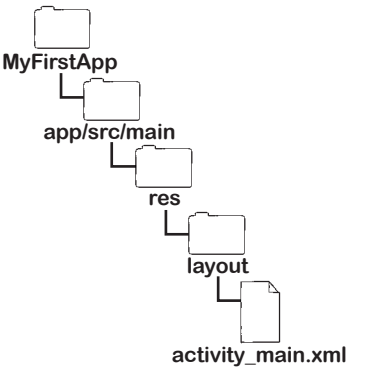
...."Hello World!"

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!"



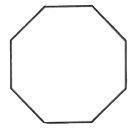

That's the only change you need to make in order to update the text. Let's see what happens when the code runs.

# What the code does

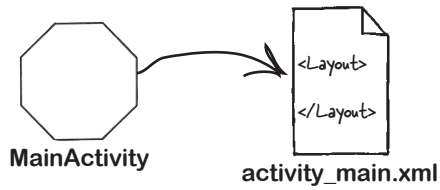
Before we take the app for a test drive, let's go through what the code does.

Set up environment
Build app
Run app
**Change app**

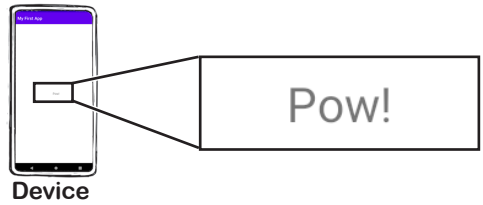
1. Android uses MainActivity.kt to create the activity object MainActivity.




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?**

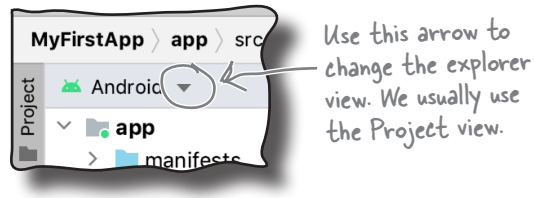
A: Yes, that's fine. Android Studio may generate slightly different code if you're using a different version than us, but that doesn't really matter. From now on you'll be learning how to create your own layout code, and you'll replace a lot of what Android Studio gives you.

**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?**

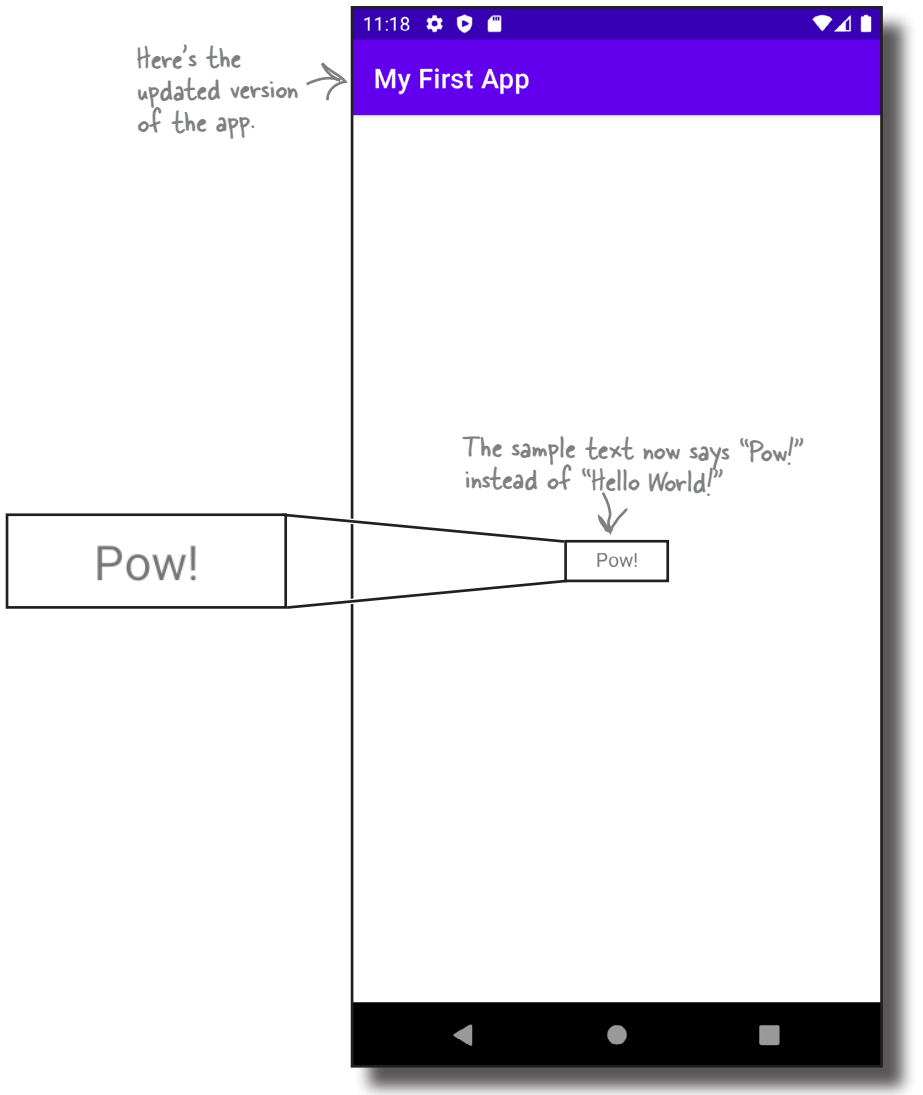
A: Android Studio lets you choose alternate views for how to display the folder hierarchy, and it defaults to the Android view. We prefer the Project view, as it reflects the underlying folder structure. You can change the explorer to the Project view by clicking on the arrow at the top of the explorer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



# Test Drive

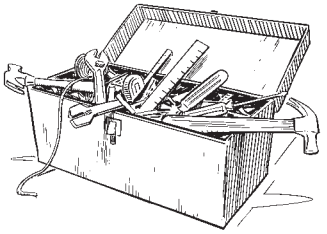
Set up environment
Build app
Run app
**Change app**

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

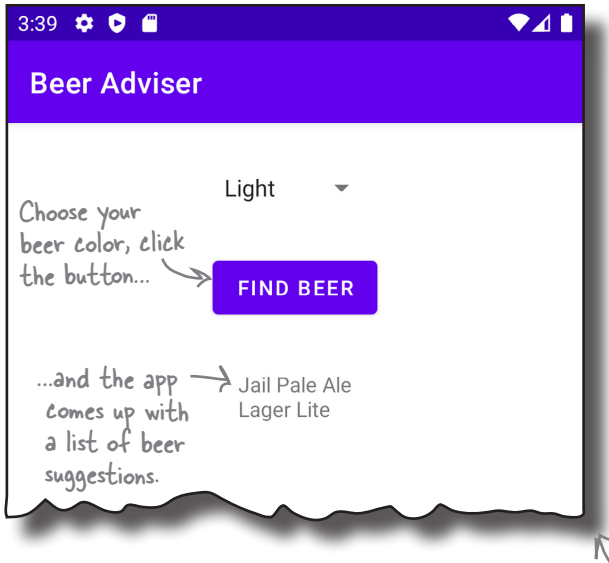
## Most apps need to respond to the user in some way.

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.

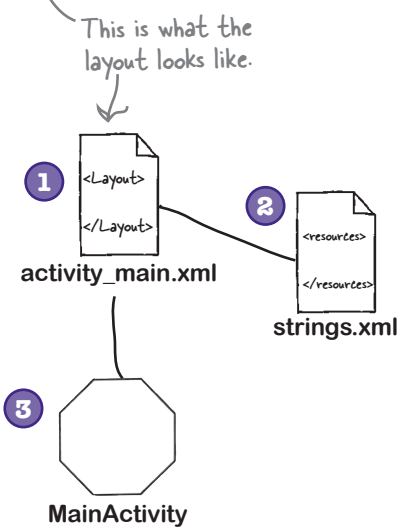
# Let's build a Beer Adviser app

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.




Here's how the app will be structured:



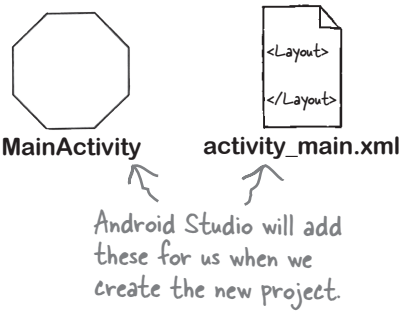

1. **The layout activity_main.xml specifies what the app will look like.**
   It includes three UI components:
   - A drop-down list of values called a spinner, which allows the user to choose which color of beer they want
   - A button that, when clicked, will return a selection of beers
   - A text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

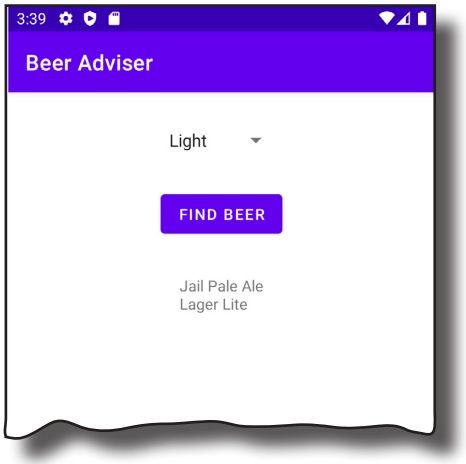
# Here's what we're going to do

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):

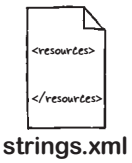



1. **Create a project.**
You're creating a brand-new app, so you'll need to create a new project that includes an empty activity and a layout.

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.

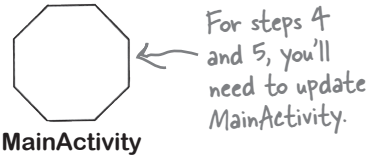



3. **Add String resources.**
We'll replace any hardcoded text with `String` resources so that all the text that's used by the app is held in a single file.

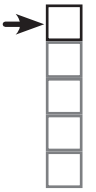



4. **Make the button respond to clicks.**
The layout only specifies the visuals. To make the button *do* something when it's clicked, you need to write some activity code.

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

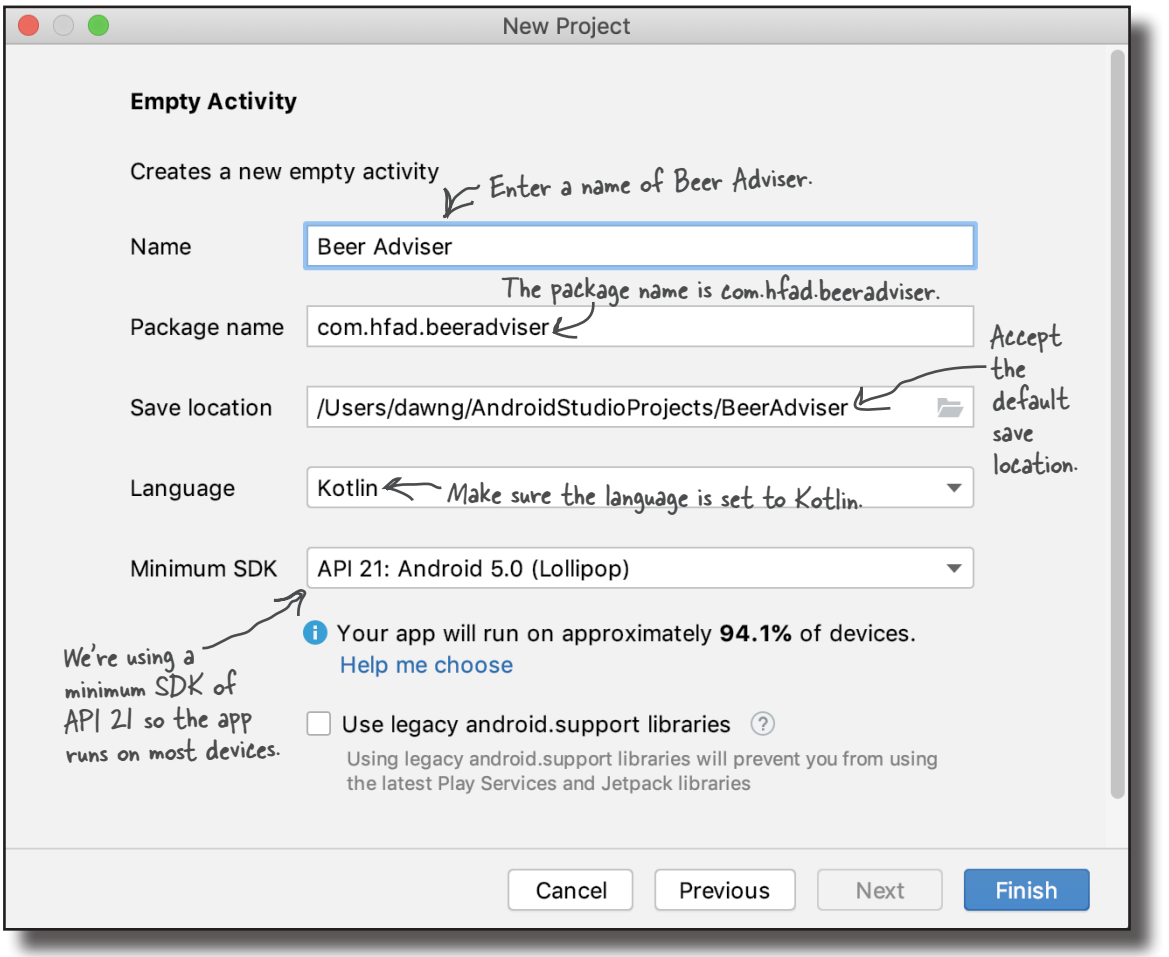
# Create the project

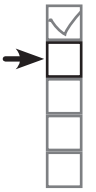
The steps for creating the new project are nearly identical to the ones we used in the previous chapter:

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:

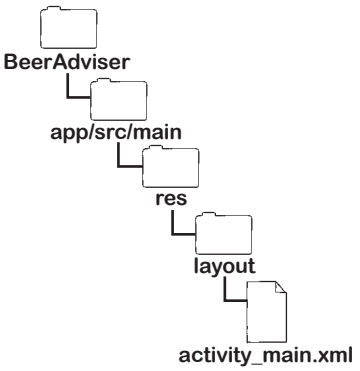
Click on the Code option to open the code editor.

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.

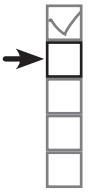



The above code features a linear layout (denoted by the `<LinearLayout>` element) and a text view (denoted by the `<TextView>` element). You'll find out more about these elements later in the chapter, but for now, all you need to know is that the linear layout is used to arrange UI components in a vertical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

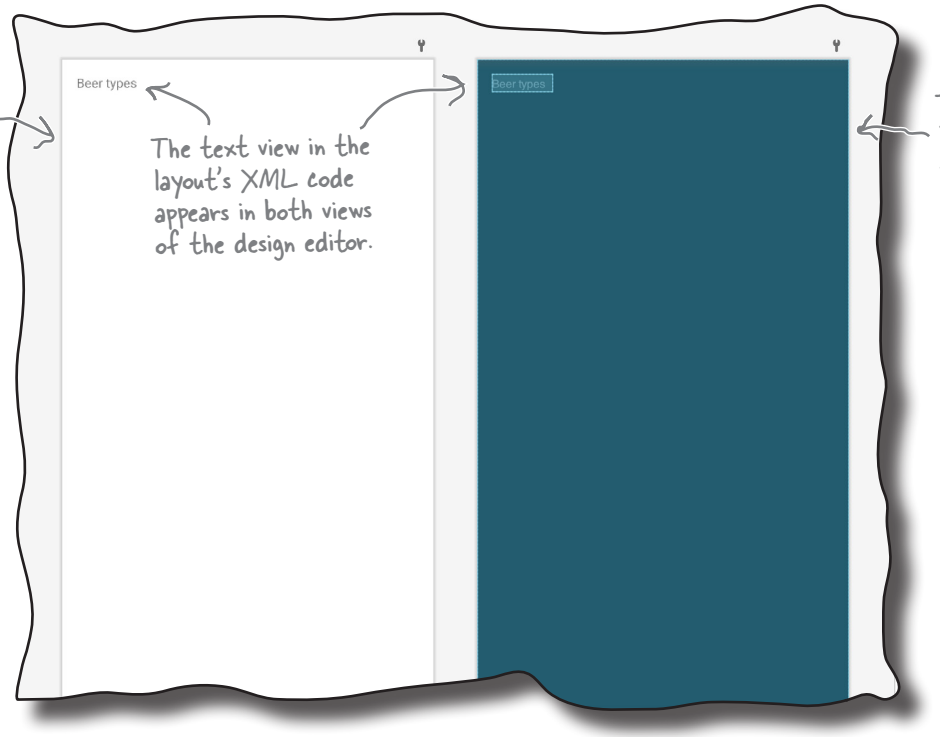
# A closer look at the design editor



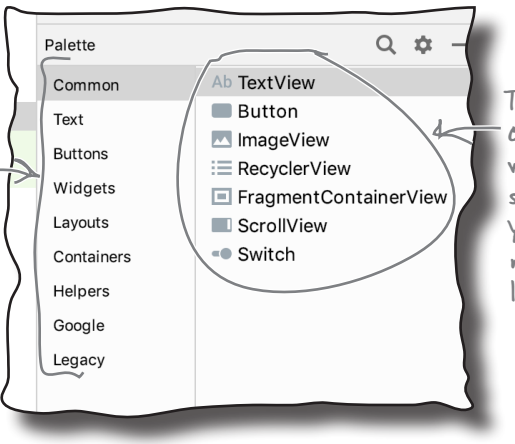
- Create project
- **Update layout**
- Add resources
- Respond to clicks
- Write logic

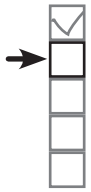
As you learned in Chapter 1, the design editor presents you with a more visual way of editing layout code than editing XML. It features two different views of the layouts design. One shows you how the layout will look on an actual device, and the other shows you a blueprint of its structure:

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.

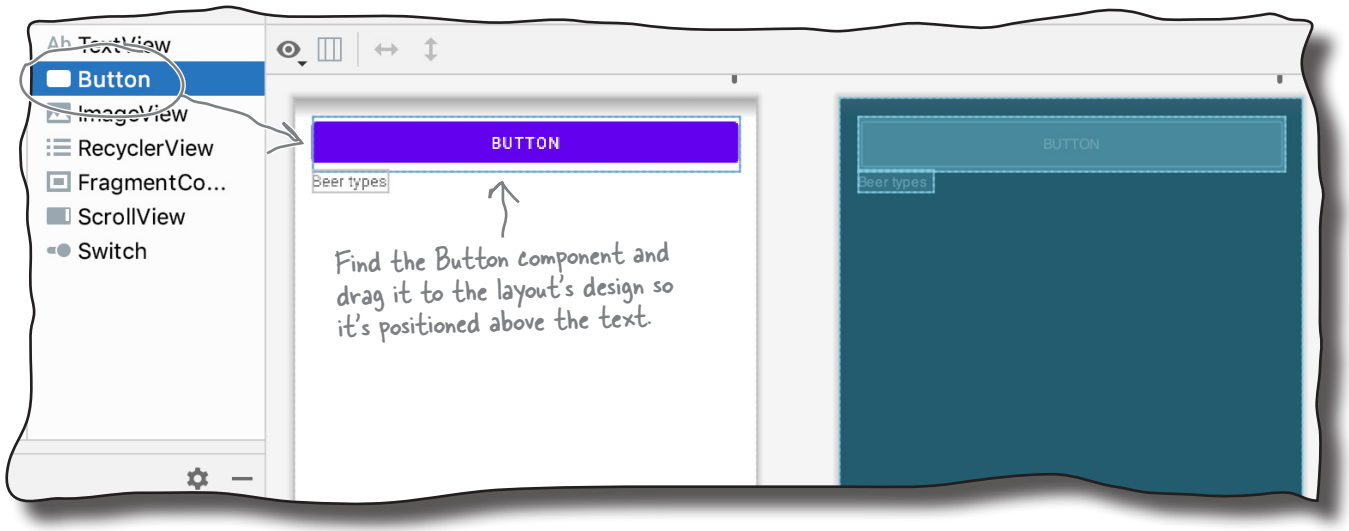




- Create project
- **Update layout**
- Add resources
- Respond to clicks
- Write logic

# 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:



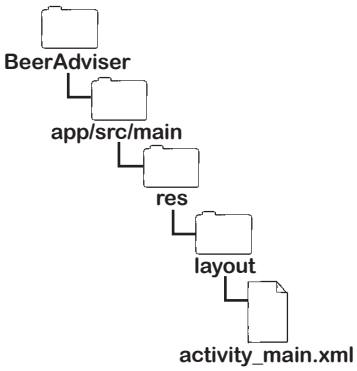

## Changes in the design editor are reflected in the XML

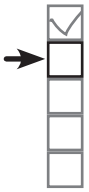
Dragging UI components to the layout like this is a convenient way of updating the layout. If you switch to the code editor, you'll see that adding the button via the design editor has added some more code to the underlying XML:

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

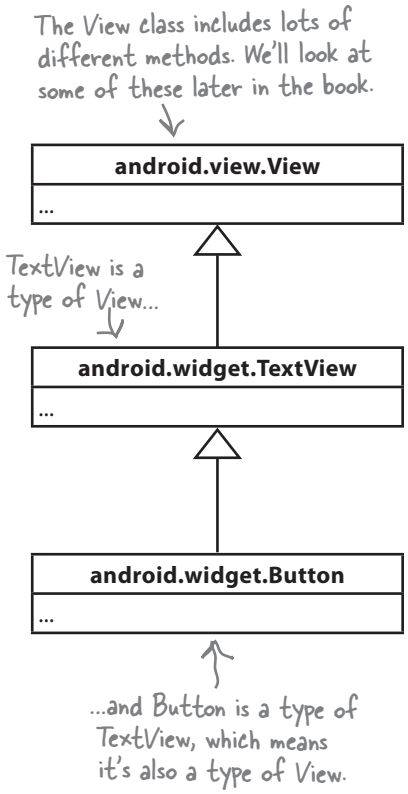
These attributes specify the width and height of the component. `"wrap_content"` means it should be just big enough for the content, and `"match_parent"` means it should be as wide as the layout containing it:

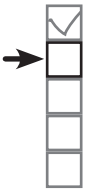
```
android:layout_width="match_parent"
android:layout_height="wrap_content"
```

### 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.

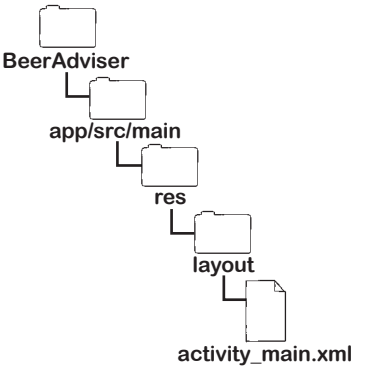
```
    <Button
        android:id="@+id/button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Button" />
```

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

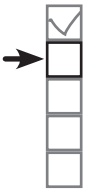
The first element in the layout code is the `<LinearLayout>`. This element tells Android that the different UI components in the layout should be displayed one after another in a single row or column.

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.

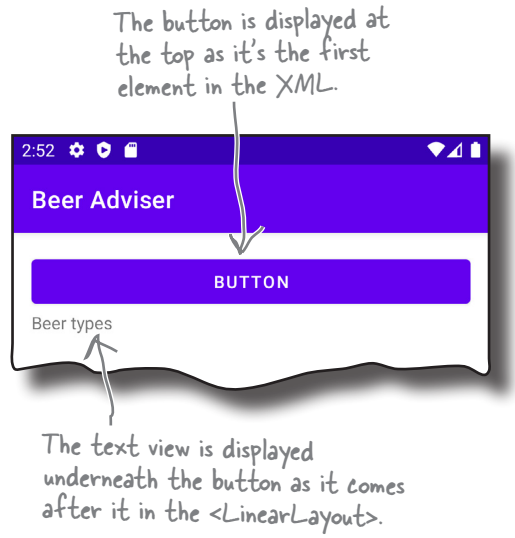
**Using a linear layout means that UI components are displayed in a vertical column or a horizontal row.**

## The <TextView> element

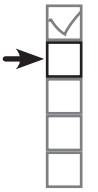
The final element inside the <LinearLayout> is the <TextView>:

```
...
<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.



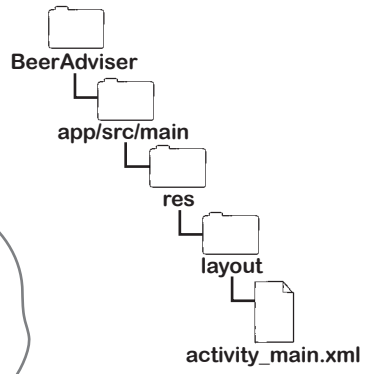
- Create project
- **Update layout**
- Add resources
- Respond to clicks
- Write logic

# 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.

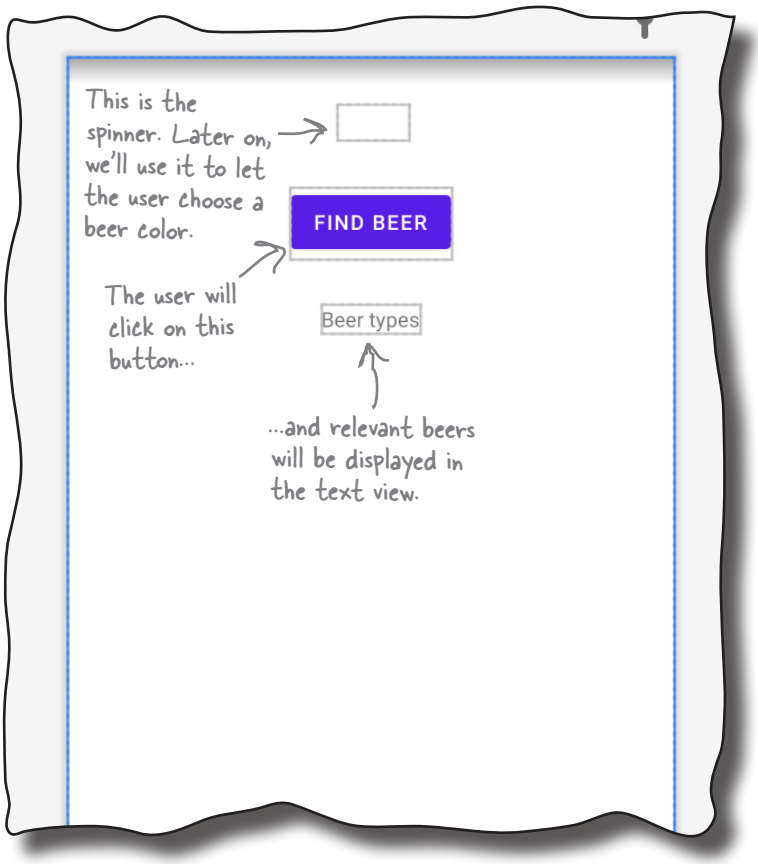
Center the text view and apply a margin.

**Don't forget!**

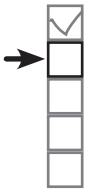
Make sure you update the contents of activity_main.xml with the changes shown here.

# 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.

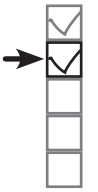


- Create project
- **Update layout**
- Add resources
- Respond to clicks
- Write logic

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.

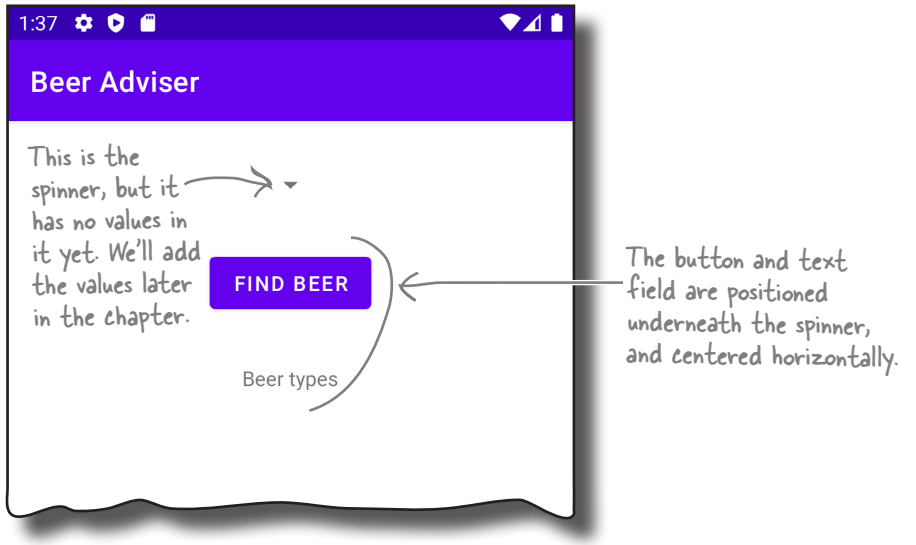


- Create project
- **Update layout**
- Add resources
- Respond to clicks
- Write logic

# 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.

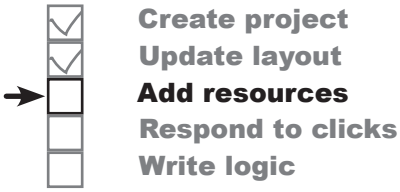
**Q: Why does the button show its text as uppercase?**

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:

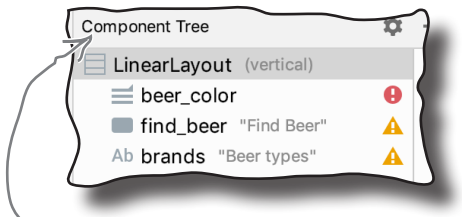
```
android:textAllCaps="false"
```

# There are warnings in the layout...



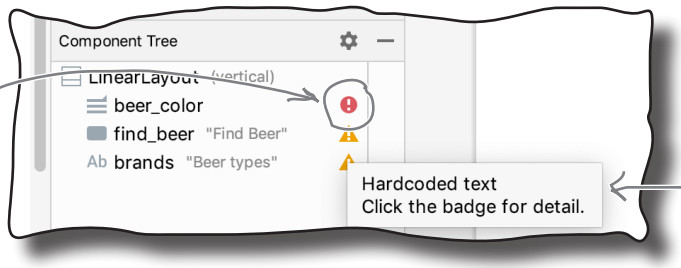

When you develop layouts in Android Studio, the IDE automatically checks your code for errors, and alerts you to potential improvements. An easy way of viewing any warnings or suggestions is to switch to the design editor view of the layout, and check out the component tree panel. This panel is usually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

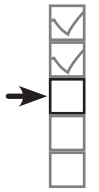
Two of the components in the layout use hardcoded text values.

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.

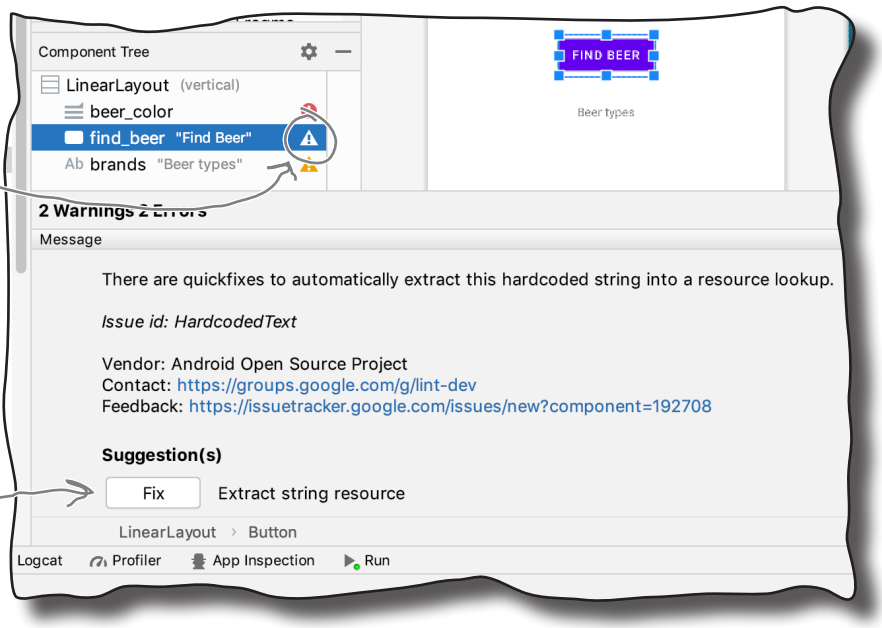
You might need to scroll down to see this button.

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.

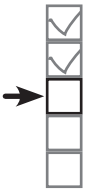
You'll be shown an explanation about why hardcoded text is a problem. Scroll to the end of this explanation, then click on the Fix button:

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

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



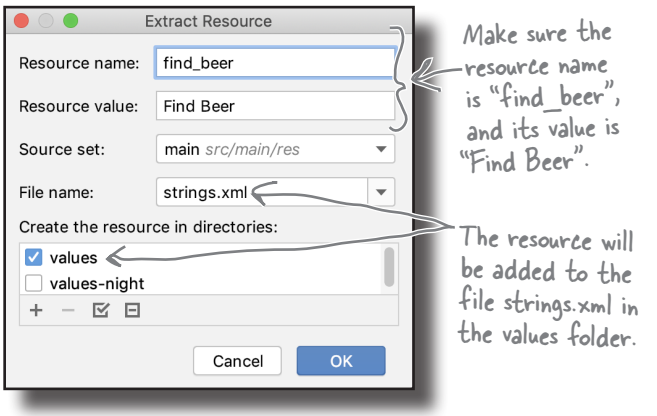
# Extract the String resource



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

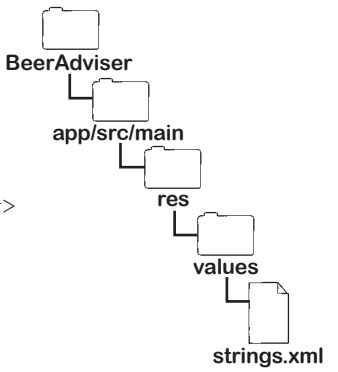
When you click on the Fix button, the Extract Resource window appears. This allows you to specify the name of the `String` resource, its value, and the name of the `String` resource file. Make sure that the resource name is "find_beer", the file name is "strings.xml", the source set is "main", and the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

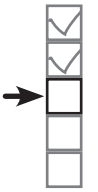
This indicates that this is a String resource.

```
<string name="find_beer">Find Beer</string>
```

Its name is "find_beer".

Its value is Find Beer.

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*.



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

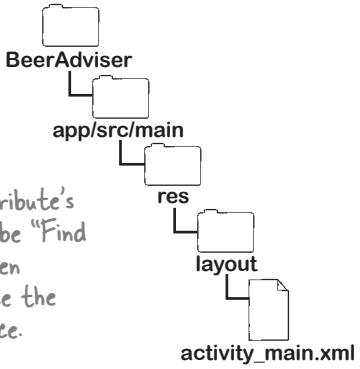
# activity_main.xml uses the String resource

When we told Android Studio to extract the hardcoded text, Android Studio automatically updated the `find_beer` button in *activity_main.xml* so that it uses the extracted `String` resource.

Here's the updated code for the button:

```
<Button
    android:id="@+id/find_beer"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:layout_margin="16dp"
    android:text="@string/find_beer" />
```

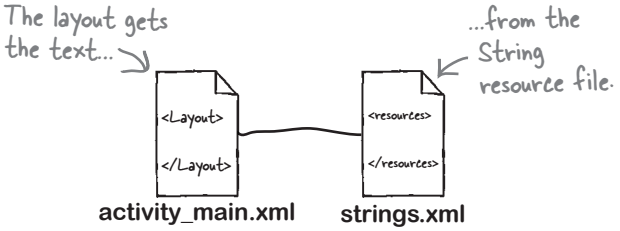
The text attribute's value used to be "Find Beer". It's been changed to use the String resource.




As you can see, the `find_beer` button's `text` attribute has been changed to `"@string/find_beer"`. So what does this mean?

Let's start with the first part, `@string`. This is just a way of telling Android to look up a text value from a `String` resource file. Here, it's the file *strings.xml* that you saw earlier.

The second part, `find_beer`, tells Android to **look up the value of a resource with the name `find_beer`**. So `"@string/find_beer"` is like saying "look up the `String` resource with the name `find_beer`, and use the associated text value."




Display the text...

```
android:text="@string/find_beer" />
```

...for the String resource find_beer.

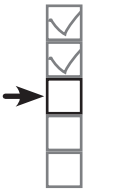
## You also can extract String resources manually

You've now learned what changes Android Studio makes to the code when you ask it to extract hardcoded text into a `String` resource. You can also make these changes yourself by directly updating the code in *strings.xml* and *activity_main.xml*.

Let's see how by changing the hardcoded "Beer types" text that's used by the `brands` component's `text` attribute so that it uses a `String` resource.

In your own projects, it's likely that you'll just use the wizard. We're showing you how to manually edit the XML because we need to make sure that the code you create matches ours, and updating the XML is the best way of doing this.

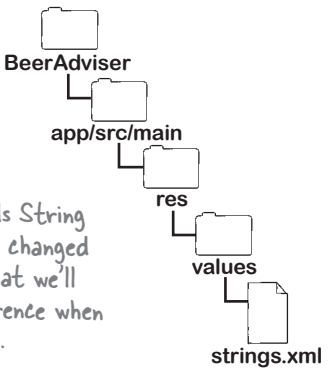
# Add and use a new String resource




We'll start by creating a new `String` resource named `brands`. Open *strings.xml* (located in the *app/src/main/res/values* folder), and add a new line for the `String` resource so that it includes the change shown here (in bold):

```
<resources>
    <string name="app_name">Beer Adviser</string>
    <string name="find_beer">Find Beer</string>
    <string name="brands">No beer selected</string>
</resources>
```

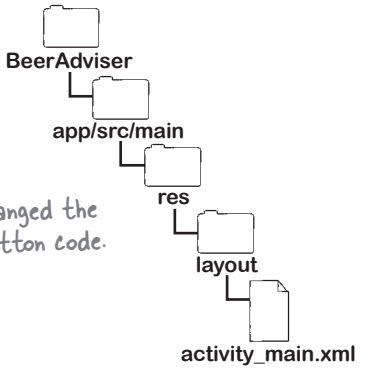
Add the brands String resource. We've changed the value so that we'll notice a difference when we run the app.




Once you've added the `String` resource, open *activity_main.xml* and update the code for the `brands` text view so that it uses the new resource. Update the code so that it includes the change shown here (in bold):

```
...
<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/find_beer"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:layout_margin="16dp"
    android:text="@string/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~~@string/brands" />
...
```

We've not changed the spinner or button code.




Update the text view's text attribute so that it uses the brands String resource instead of the hardcoded text. This will display the value of the resource in the text view.

There's a summary about using `String` resources on the next page. After that, we'll take the app for a test drive.

# String Resources Up Close

*strings.xml* is the default resource file used to hold name/value pairs of `String`s so that they can be referenced throughout the app. It has the following format:

The <string> element identifies the name/value pairs as Strings.

The <resources> element identifies the contents of the file as resources.

```
<resources>
    <string name="app_name">Beer Adviser</string>
    <string name="find_beer">Find Beer!</string>
    <string name="brands">No beer selected</string>
</resources>
```

There are two things that allow Android to recognize *strings.xml* as being a `String` resource file:

- **The file is held in the folder *app/src/main/res/values*.**
  XML files held in this folder contain values that can be used throughout the app, such as `String`s and colors.

- **The file has a <resources> element, which contains one or more <string> elements.**
  The format of the file itself indicates that it's a resource file containing `String`s. The `<resources>` element tells Android that the file contains resources, and the `<string>` element identifies each `String` resource.

This means that you don't need to call the `String` resource file *strings.xml*; if you want, you can call it something else, or split the `String`s into multiple files.

Each name/value pair takes the form:

```
<string name="string_name">string_value</string>
```

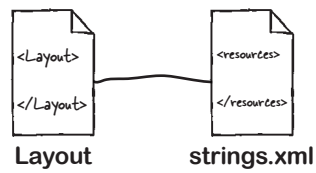
where `string_name` is the identifier of the `String`, and `string_value` is the `String` value itself.

A layout can retrieve the value of the `String` using:

```
"@string/string_name"
```

This is the name of the String whose value we want to return.

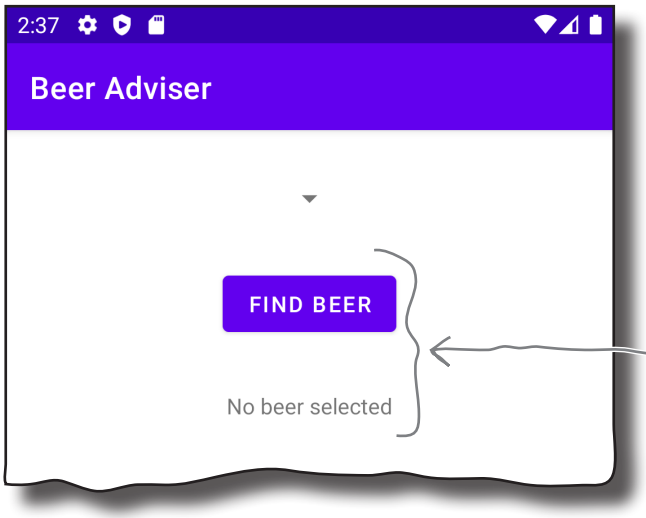
"@string" tells Android to look for a String resource of this name.



# Test Drive

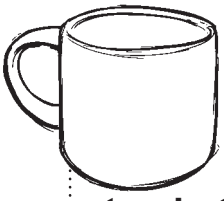
Now that we've updated the layout to use `String` resources instead of hardcoded text values, let's run the app to see how it's looking. Run the app as before by choosing the "Run 'app'" command from the Run menu.

When we run the app, the text that's displayed on the button and in the text view has been updated so that it uses the `String` values we added to *strings.xml*:




The button and text view text attributes now use String resources instead of hardcoded text.

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



## Relax

**When you rerun the app, don't worry if Android Studio asks if you want to terminate the process "app".**

It simply wants to know if it can exit the old version of the app that's currently running so it can run the new one. Just click on the Terminate option, and wait for your app to load.

## there are no Dumb Questions

**Q: Do I have to put my text values in a `String` resource file?**

A: It's not mandatory, but Android gives you warning messages if you hardcode text values. Using a `String` resource file might seem like a lot of effort at first, but it makes things like localization much easier. It's also easier to use `String` resources to start off with, rather than patching them in afterward.

**Q: How does separating out the `String` values help with localization?**

A: Suppose you want the application to be in English by default, but in French if the device language is set to French. Rather than hardcode different languages into the app, you can have one `String` resource file for English text, and another resource file for French text.

**Q: How does the app know which `String` resource file to use?**

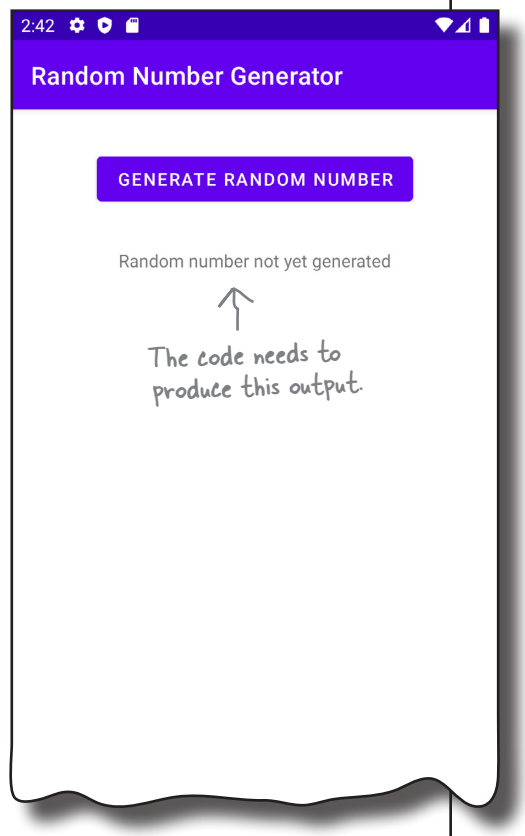
A: Put the default English `String` resource file in the *app/src/main/res/values* folder as normal, and the French one in a new folder named *app/src/main/res/values-fr*. If the device is set to French, it will use the `String` resources in the *app/src/main/res/values-fr* folder. If the device is set to any other language, it will use the `String` resources in *app/src/main/res/values* instead.

# Sharpen your pencil

The code below (a layout file and `String` resource file) is from an app that's used to generate random numbers. Complete the code so it produces the output shown below.

### activity_main.xml

```
<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">

    <.....
        android:id="@+id/generate_number"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="16dp"
        android:.....="@string/button_text" />

    <.....
        android:id="@+id/random_number"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="16dp"
        android:text="....." />
</LinearLayout>
```

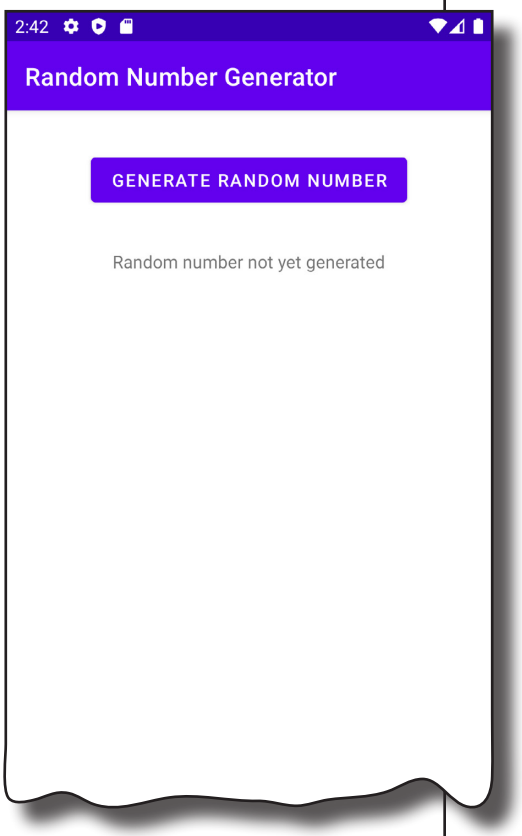



### strings.xml

```
<resources>
    <string name="app_name">Random Number Generator</string>
    <string name=".....">Generate random number</string>
    <string name="no_number">Random number not yet generated</string>
</resources>
```

# Sharpen your pencil Solution

The code below (a layout file and `String` resource file) is from an app that's used to generate random numbers. Complete the code so it produces the output shown below.




**activity_main.xml**

```
<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">

    <Button
        android:id="@+id/generate_number"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="16dp"
        android:text="@string/button_text" />

    <TextView
        android:id="@+id/random_number"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="16dp"
        android:text="@string/no_number" />
</LinearLayout>
```

Display a Button.

Update the text attribute.

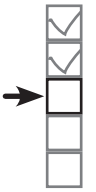
Display a TextView.

Use the no_number String resource.

**strings.xml**

```
<resources>
    <string name="app_name">Random Number Generator</string>
    <string name="button_text">Generate random number</string>
    <string name="no_number">Random number not yet generated</string>
</resources>
```

The button uses a String resource with a name of button_text.



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

# Add values to the spinner

The app's current layout includes a spinner, but when we click on it, it's empty. This is because we haven't yet told the spinner what values it should display. Each time you use a spinner in the layout code, you must specify an associated list of values, or it won't include any, and Android Studio may give you warning messages.

You specify a list of values for a spinner in a similar way to how you specify the text that's displayed on a button or in a text view: by adding a resource to *strings.xml* and referring to the resource in the layout. Instead of specifying a single value in a `String` resource, however, you add multiple `String`s to an *array* resource, and use this array for the spinner's list of values.

**Resources are noncode assets, such as images or Strings, used by the app.**

## Adding an array resource is similar to adding a String

As you already know, you add a `String` resource to *strings.xml* using:

```
<string name="string_name">string_value</string>
```

where `string_name` is the name of the `String`, and `string_value` is the value that's displayed in the app.

To add an array of `String`s to a `String` resource file, you use the following syntax:

```
<string-array name="string_array_name">
    <item>string_value1</item>
    <item>string_value2</item>
    <item>string_value3</item>
    ...
</string-array>
```

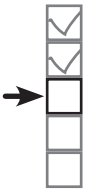
This is the name of the array.

These are the values in the array. You can add as many as you need.

where `string_array_name` is the name of the array, and `string_value1`, `string_value2`, and `string_value3` are the individual `String` values that make up the array.

In this app, we want to add a `String` array resource where each item in the array is a beer color. We'll then attach this array to the spinner so that when the user clicks on the spinner, the beer colors will be displayed.

Let's add the new `String` array.



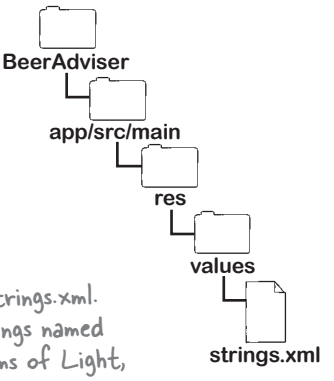
Create project
Update layout
**Add resources**
Respond to clicks
Write logic

# Add the string-array to strings.xml

To add the `String` array, open up *strings.xml*, and add the code below (in bold). This adds a `string-array` resource named `beer_colors` that we'll attach to the spinner:

```
<resources>
    <string name="app_name">Beer Adviser</string>
    <string name="find_beer">Find Beer</string>
    <string name="brands">No beer selected</string>
    <string-array name="beer_colors">
        <item>Light</item>
        <item>Amber</item>
        <item>Brown</item>
        <item>Dark</item>
    </string-array>
</resources>
```

Add this string-array to strings.xml. It defines an array of Strings named beer_colors with array items of Light, Amber, Brown, and Dark.




## Get the spinner to display the array's values

A layout references a `String` array resource using similar syntax to how it would retrieve the value of a `String` resource. Rather than use:

```
"@string/string_name"
```

Use @string to reference a String, and @array to reference an array.

you use the syntax:

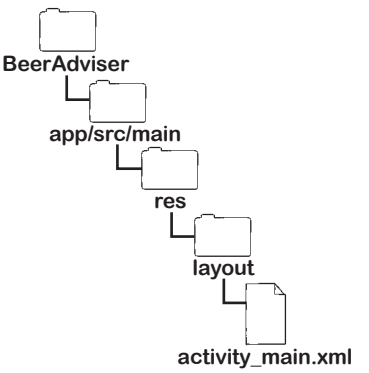
```
"@array/array_name"
```

where `array_name` is the name of the array.

Let's use this in the layout. Go to the layout file *activity_main.xml* and add an `entries` attribute to the spinner like this (in bold):

```
...
<Spinner
    android:id="@+id/color"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:layout_margin="16dp"
    android:entries="@array/beer_colors" />
...
```

This means "the entries for the spinner come from array beer_colors."

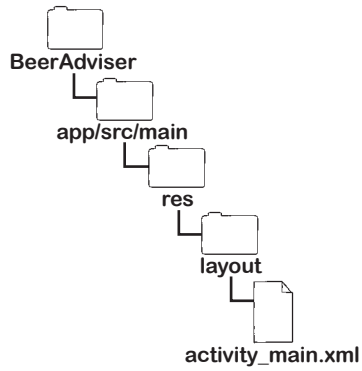



We'll show you the full code for *activity_main.xml* on the next page.

# The full code for activity_main.xml

Below is the entire code for *activity_main.xml*. Make sure the code for this file includes all of the code shown here.

```
<?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"
        android:entries="@array/beer_colors" />

    <Button
        android:id="@+id/find_beer"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="16dp"
        android:text="@string/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="@string/brands" />
</LinearLayout>
```




The beer_color spinner gets its entries from the beer_colors String array.

The find_beer button gets its text from the find_beer String resource.

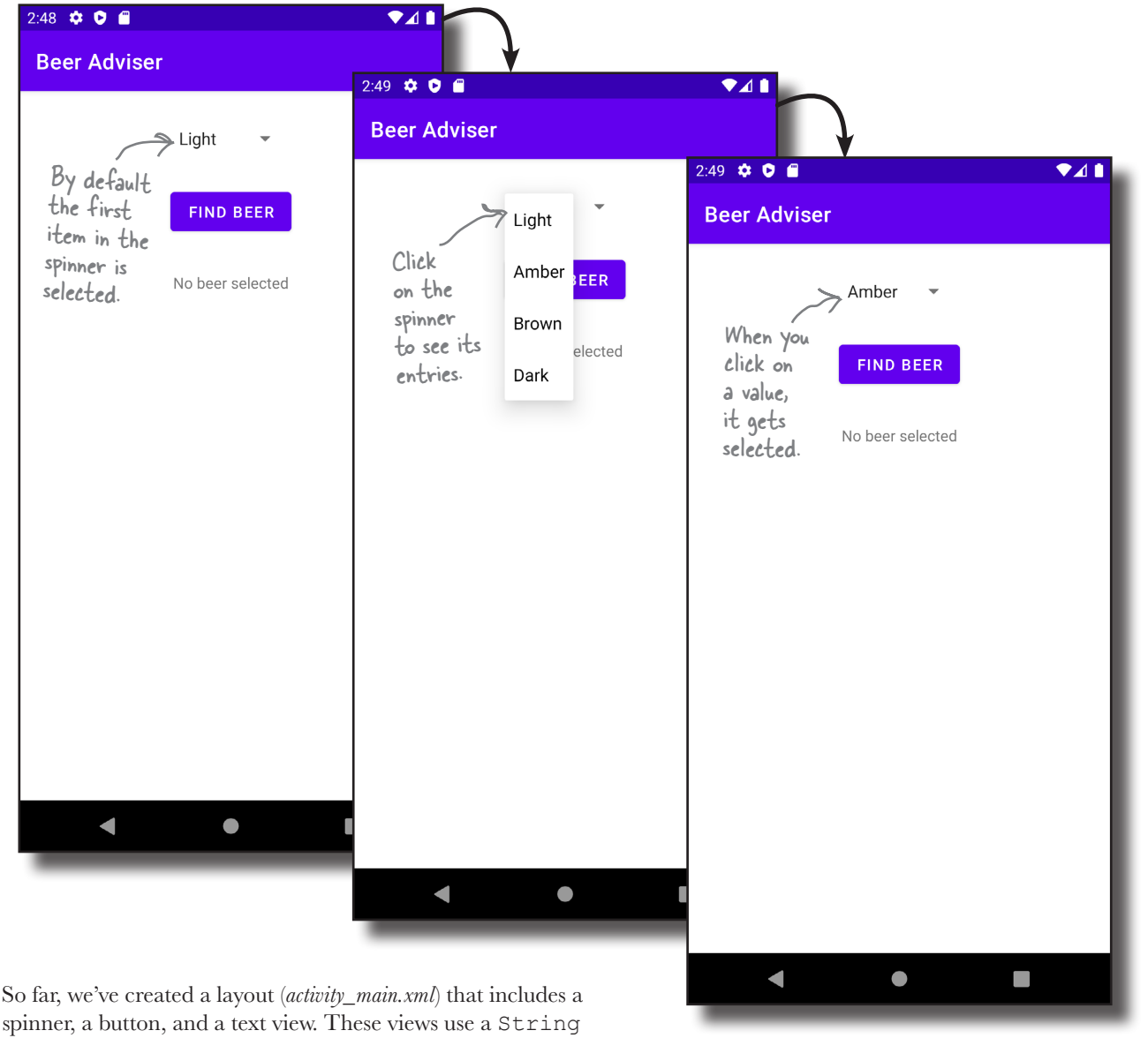
The brands text view gets its text from the brands String resource.

Let's take the app for a test drive.

# Test Drive

Let's see what impact the changes have had on the app. Run the app, and you should get something like this:

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




So far, we've created a layout (*activity_main.xml*) that includes a spinner, a button, and a text view. These views use a `String` resource file (*strings.xml*) for their `String` and array values.

The next thing we need to do is get the app to update the `brands` text view each time the user clicks the button.

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

# We need to make the app interactive

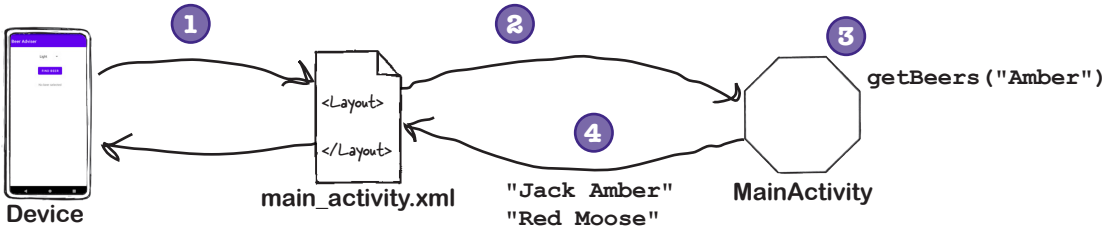
The Beer Adviser app has the correct appearance and includes all the views we need, but it doesn't yet provide any beer recommendations. To make the app interactive, we need to make the app do something when the user clicks the `find_beer` button. We want the app to behave something like this:

1. **The user chooses a beer color from the spinner and clicks the button.**

2. **Code in MainActivity responds to the button being clicked.**

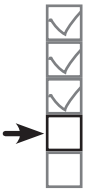
3. **MainActivity passes the selected beer color to a method we'll create named getBeers.**
   The `getBeers()` method finds matching brands for the beer color.

4. **MainActivity updates the brands text view so that it displays the list of recommended beers on the device.**




To make the app respond to the user in this way, we need to update the code in *MainActivity.kt*, as this is the code that's responsible for how the app behaves. Android Studio created this file for us when we created the project, so let's take a look at the current code.

# What the MainActivity code looks like



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

Android Studio created *MainActivity.kt* for us when we created the project. Open this file now (if it's not already open) by going to the *app/src/main/java* folder and double-clicking on it.

Here is the code for *MainActivity.kt* that Android Studio created for us:

```
package com.hfad.beeradviser

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 we specified when we created the project.

The class extends AppCompatActivity.

This is the onCreate() method. It's called when the activity is first created.

setContentView() tells Android which layout the activity uses. In this case, it's activity_main.xml.

The above code is all you need to create a basic activity. As you can see, it's a class that extends `AppCompatActivity`, and overrides its `onCreate()` method.

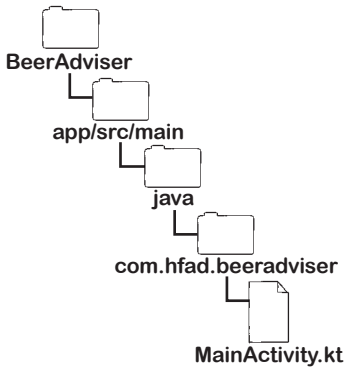
All activities (not just this one) have to extend an activity class such as `AppCompatActivity`. You'll find out more about this in Chapter 5, but for now, all you need to know is that when a class extends `AppCompatActivity`, it transforms your plain old Kotlin class into a full-fledged, card-carrying Android activity.

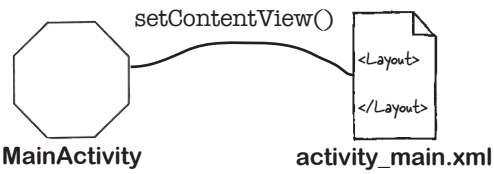
All activities must also implement the `onCreate()` method. This method gets called when the activity object gets created, and it's used to perform basic setup such as what layout the activity is associated with. This is done via a call to `setContentView()`. In the example above, the code:

```
setContentView(R.layout.activity_main)
```

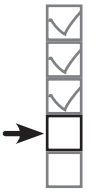
tells Android that this activity uses *activity_main.xml* as its layout.

You now know what the current `MainActivity` code does. So how do we get it to respond when the user clicks the `find_beer` button?




The setContentView() method is the glue that connects an activity to its layout.



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

# A button can listen for on-click events...

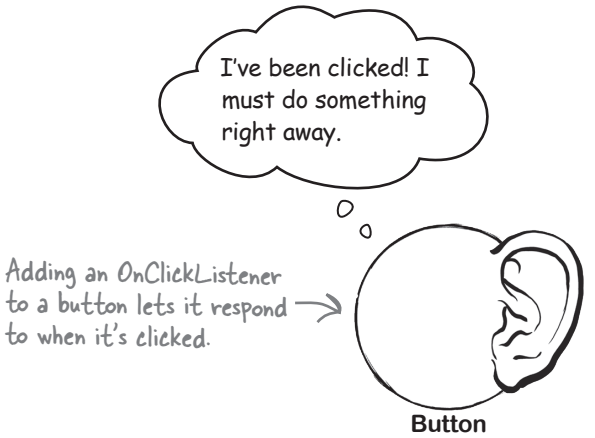
Each time the user does something in your app it's called an **event**. There are many different event types in Androidville, such as clicking a button, swiping the screen, or pressing a hardware key on the device.

In this app, we want to know when the user clicks on the `find_beer` button so that we can get the app to do something in response. We can get the app to listen for the button's **on-click** event so that each time it occurs, we can update the text in the `brands` text view.

## ...using an OnClickListener

You can make the app listen for a button's on-click events by adding an **`OnClickListener`** to the button. Each time the button is clicked, the `OnClickListener` "hears" the click, and responds to it.

**You can make a button respond to on-click events by adding an OnClickListener to the button.**

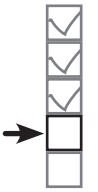



You specify what action the `OnClickListener` should take by passing it a block of code—a lambda—that specifies what should happen when the button is clicked.

If your Kotlin lambda knowledge feels a little rusty, we suggest that you refer to Head First Kotlin.

Here, we want to update the `brands` text view each time the `find_beer` button is clicked. This means that we need to add an `OnClickListener` to the `find_beer` button, and pass it a lambda that tells it how to update the text view.

Let's find out how to do this.



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

# Get a reference to the button...

To add an `OnClickListener` to the `find_beer` button, you first need to get a reference to the button in the activity code. You do this using a method named **`findViewById`**.

The `findViewById` method lets you get a reference to any view in the layout that has an ID. Simply specify the view's type and ID, and the method returns a reference to it.

In this app, we want to get a reference to the button with an ID of "find_beer", so we use the code:

```
val findBeer = findViewById<Button>(R.id.find_beer)
```

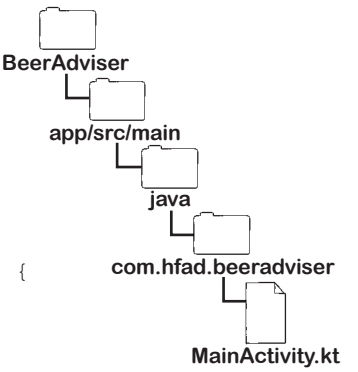
Specify the type here, in this case Button.

We want the button with an ID of find_beer.

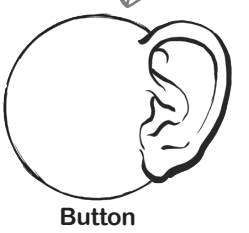
## ...and call its setOnClickListener method

Once you have a reference to the button, you can add an `OnClickListener` to it by calling its `setOnClickListener()` method using code like this:

```
...
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val findBeer = findViewById<Button>(R.id.find_beer)
        findBeer.setOnClickListener {
            //Code that runs when the button is clicked
        }
    }
}
```




This adds an OnClickListener to the find_beer button so that it can respond to on-click events.




Notice that we're adding the `OnClickListener` code to `MainActivity`'s `onCreate()` method. `onCreate()` runs when the activity gets created, so adding the call to `setOnClickListener()` here means that the `find_beer` button will be able to start responding to clicks as soon as possible.

Now that we've added an `OnClickListener` to the button, let's make it do something when the button is clicked.

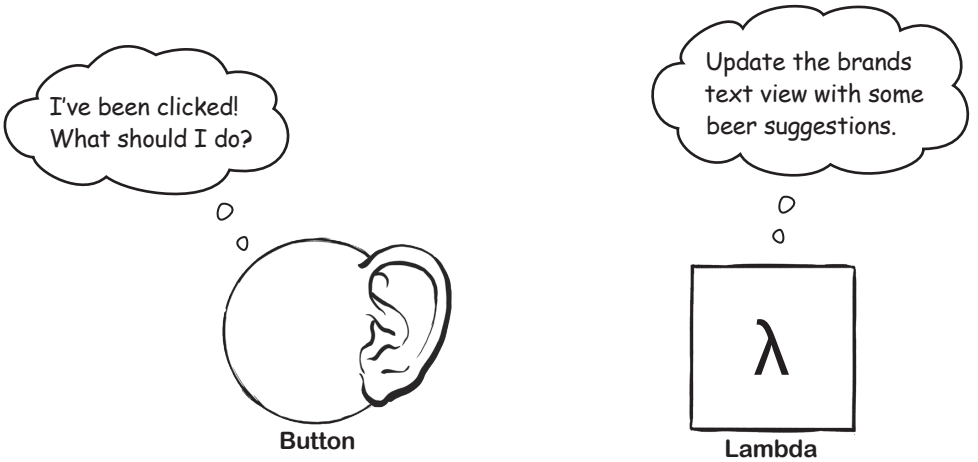
# Pass a lambda to the setOnClickListener method

You make the button do something when it's clicked by passing a lambda to its `setOnClickListener()` method. The lambda specifies what you want to happen each time the button is clicked.

```
findBeer.setOnClickListener {
    //Your lambda code goes here
}
```

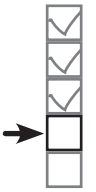
The code you add to the lambda tells the button's OnClickListener what to do when it hears a click.

So if you want the button to update some text or perform some other action, you put the code to do this in the lambda:




We ultimately want the activity to display a list of beer recommendations, but for now, let's make the button update the `brands` text view with the value that's selected in the `beer_color` spinner. This will let us test that the button's `OnClickListener` works before we focus on getting some *real* beer advice.

In order to do this, there are two more things we need to know: how to edit the text in a text view, and how to get the value that's selected in the spinner.




# How to edit a text view's text

As you've already learned, you change the text that's displayed in a text view by updating its `text` attribute in the layout XML. To update the text using activity code, we can therefore get a reference to the text view and update its `text` property.

To edit the text that's displayed in the `brands` text view to say "Gottle of geer", for example, you could use the following code:

```
val brands = findViewById<TextView>(R.id.brands)
brands.text = "Gottle of geer"
```

Find the TextView with an ID of brands.

Set the text to "Gottle of geer".

In the Beer Adviser app, we want to update the text view's `text` property to the value that the user has selected in the `beer_color` spinner. To do this, we need to find out how to get this value.

## How to get the spinner's value

You can get the current value in a spinner using its **`selectedItem`** property. To get the current value from the `beer_color` spinner, for example, you would use the code:

```
val beerColor = findViewById<Spinner>(R.id.beer_color)
```

Find the Spinner with an ID of beer_color.

The `selectedItem` property can hold any type of value, not just `String`s, so you need to convert its value before you can use it.

In the Beer Adviser app, for example, we know that the `beer_color` spinner holds an array of `String` values, so the item the user selects must be a `String`. To use this value, we therefore need to convert it to a `String` using code like this:

```
val color = beerColor.selectedItem.toString()
```

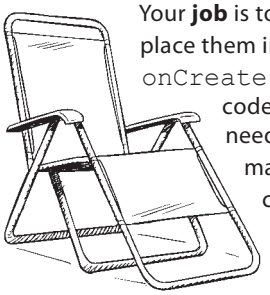
The selectedItem property can hold any type of value, so you need to convert it to the correct type before you can use it. Here, it's holding a String value, so we're converting it to a String.

You could also use it in a `String` template using the following code instead:

```
val color = "${beerColor.selectedItem}"
```

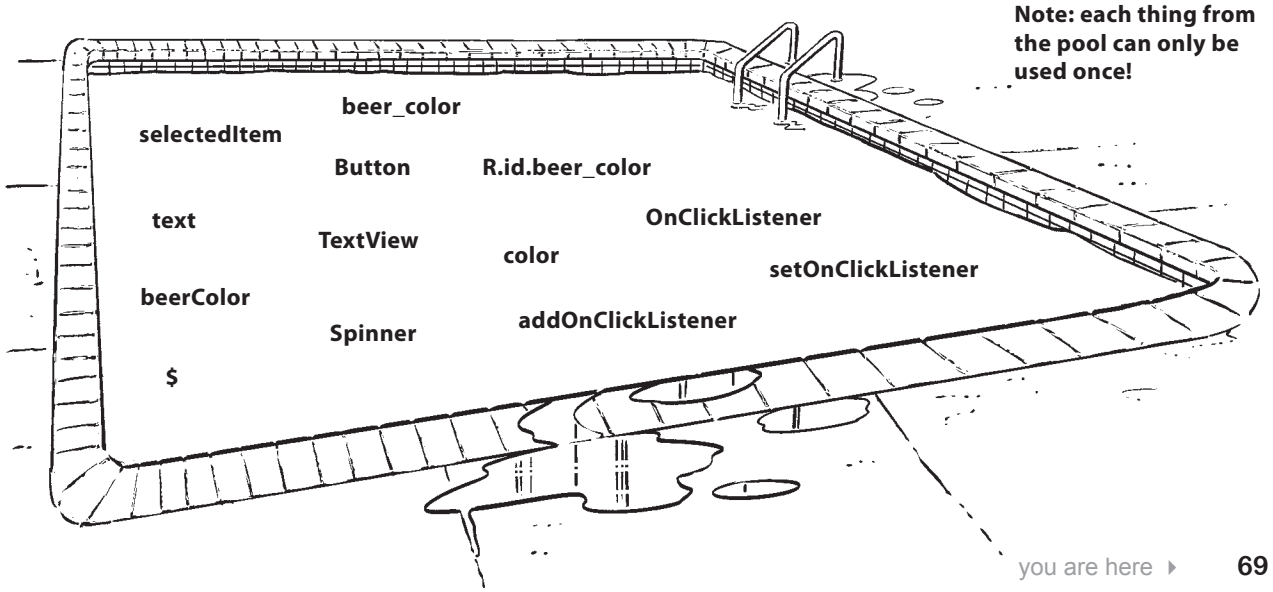
You now know enough to make the `find_beer` button get the `beer_color` spinner's value, and display it in the `brands` text view. Before we show you the full code, see if you can piece it together yourself by having a go at the following exercise.

# Pool Puzzle

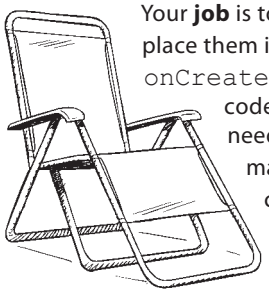


Your **job** is to take code snippets from the pool and place them into the blank lines in `MainActivity`'s `onCreate()` method. You may **not** use the same code snippet more than once, and you won't need to use all the snippets. Your **goal** is to make the `find_beer` button respond to clicks by getting it to update the `brands` text view with the value selected in the `beer_color` spinner.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    val findBeer = findViewById<Button>(R.id.find_beer)
    findBeer.                                    {
        val beerColor = findViewById<Spinner>(                    )
        val color = beerColor.                                    
        val brands = findViewById<                >(R.id.brands)
        brands.          = "Beer color is                         "
    }
}
```



# Pool Puzzle Solution



Your **job** is to take code snippets from the pool and place them into the blank lines in `MainActivity`'s `onCreate()` method. You may **not** use the same code snippet more than once, and you won't need to use all the snippets. Your **goal** is to make the `find_beer` button respond to clicks by getting it to update the `brands` text view with the value selected in the `beer_color` spinner.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    val findBeer = findViewById<Button>(R.id.find_beer)
    findBeer.setOnClickListener {
        val beerColor = findViewById<Spinner>(R.id.beer_color)
        val color = beerColor.selectedItem
        val brands = findViewById<TextView>(R.id.brands)
        brands.text = "Beer color is $color"
    }
}
```

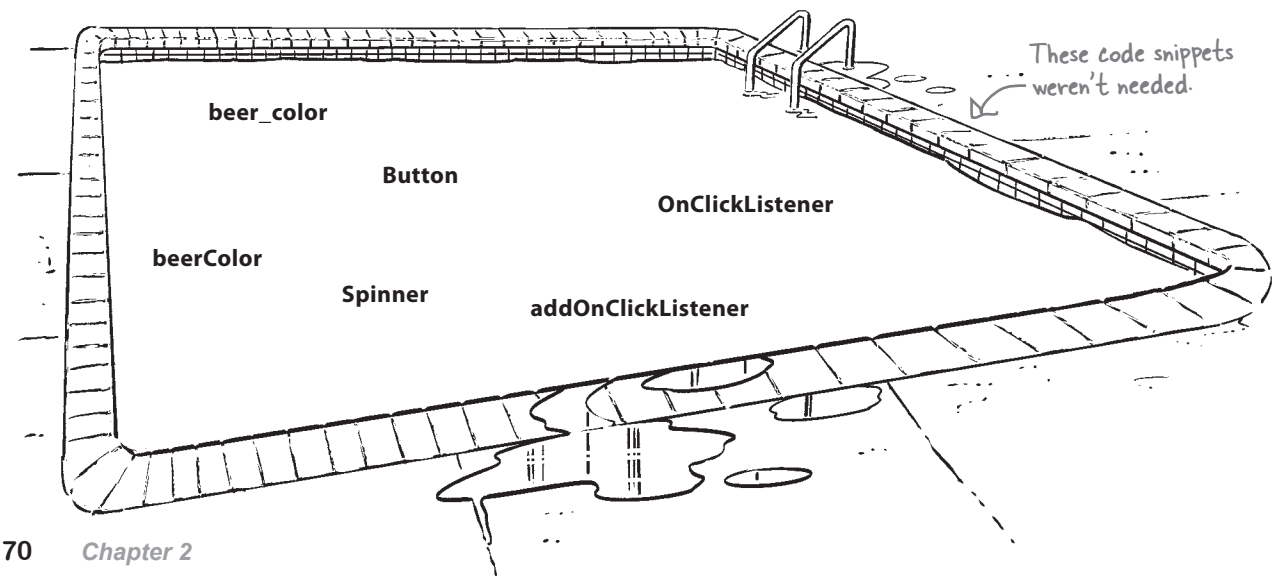
Add an OnClickListener.

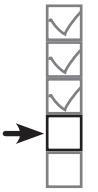
Get a reference to the spinner with an ID of beer_color.

brands is a TextView.

Update the brands text property.

Use color in a String template.



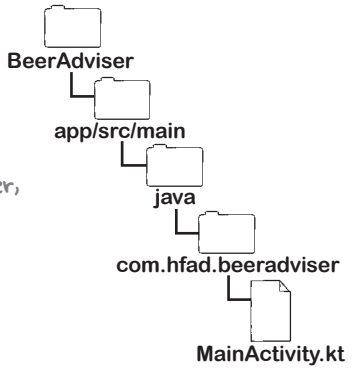


Create project
Update layout
Add resources
**Respond to clicks**
Write logic

# The updated code for MainActivity.kt

We want to update `MainActivity` so that the `find_beer` button responds to clicks. Each time the button is clicked, we want to update the text that's displayed in the `brands` text view so that it includes the beer color the user selects in the spinner.

Below is the updated *MainActivity.kt* code featuring the code you pieced together in the previous exercise. Update *MainActivity.kt* to include these changes (in bold):




```
package com.hfad.beeradviser

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.widget.Button
import android.widget.Spinner
import android.widget.TextView

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val findBeer = findViewById<Button>(R.id.find_beer)
        findBeer.setOnClickListener {
            val beerColor = findViewById<Spinner>(R.id.beer_color)
            val color = beerColor.selectedItem
            val brands = findViewById<TextView>(R.id.brands)
            brands.text = "Beer color is $color"
        }
    }
}
```

Import the Button, Spinner, and TextView classes.

Add an OnClickListener to the button.

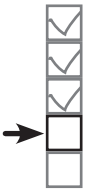
Get the selected item from the spinner.

Update the text view's text property so it includes the value from the spinner.

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

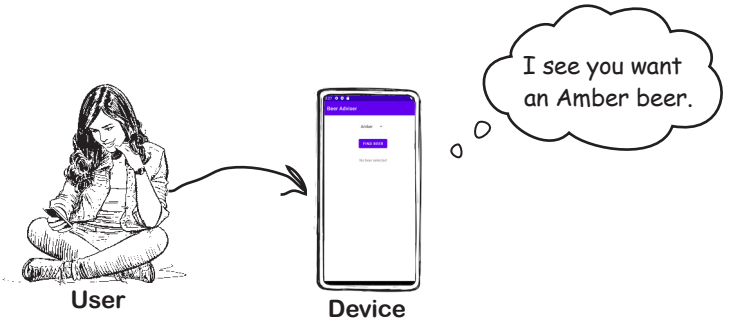
# What happens when you run the code

When we run the app, choose a beer color, and click on the Find Beer button, the following things happen:

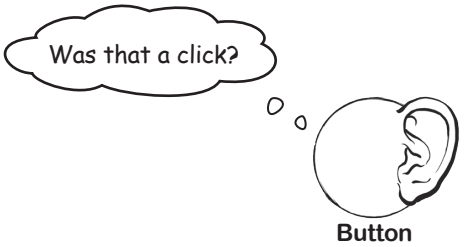


- Create project
- Update layout
- Add resources
- **Respond to clicks**
- Write logic

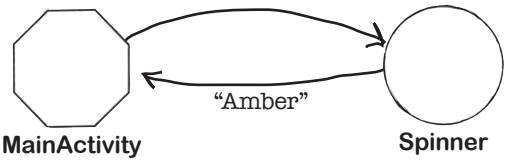
**1** **The user chooses a beer color from the spinner and clicks on the Find Beer button.**



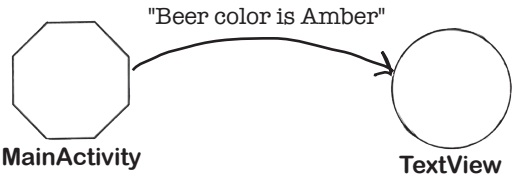

**2** **The button's OnClickListener hears that it's been clicked.**




**3** **The OnClickListener code in MainActivity retrieves the currently selected value of the spinner (in this case Amber).**




**4** **It then updates the text property of the text view to reflect that the user has chosen Amber in the spinner.**

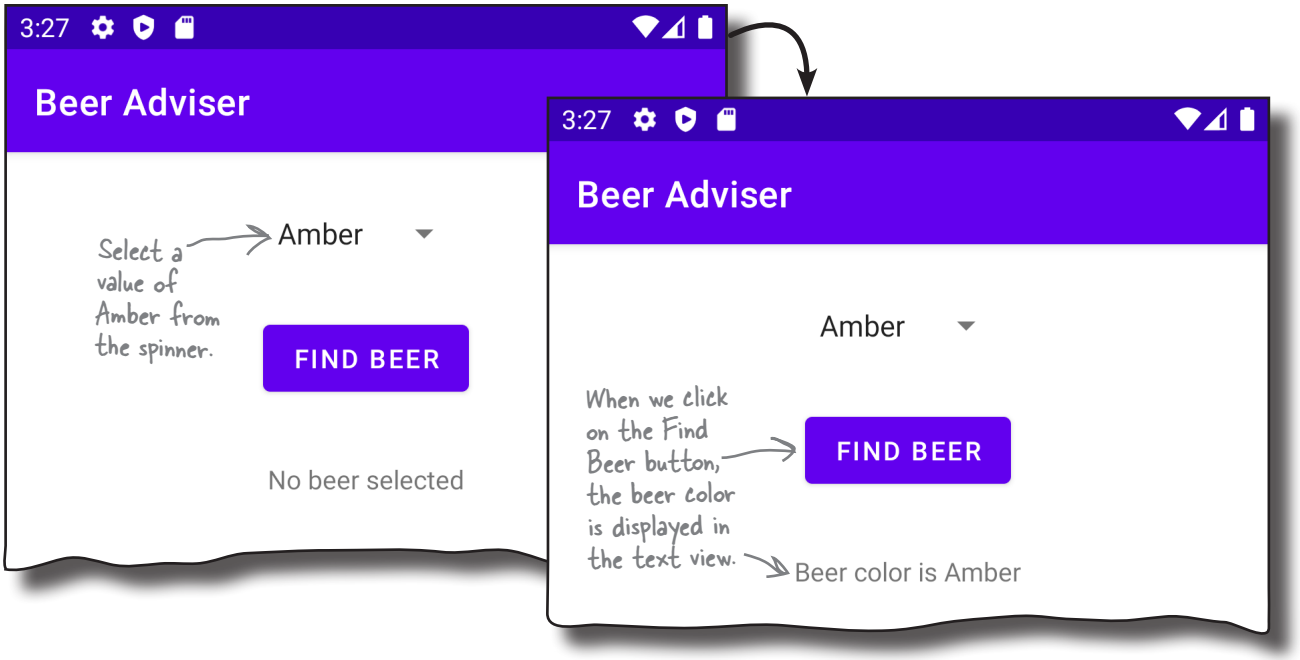


- Create project
- Update layout
- Add resources
- **Respond to clicks**
- Write logic

# Test Drive

Make sure you've updated *MainActivity.kt*, and then run the app.

When we choose a beer color from the spinner and click on the Find Beer button, the value we selected is displayed in the text view.




## there are no Dumb Questions

**Q:** **You said that we get a reference to the `brands` text view by passing the `findViewById` method a value of `R.id.brands`. You also said that we tell `MainActivity` to use *activity_main.xml* for its layout by passing the `setContentView` method a value of `R.layout.activity_main`.**

**`R.id.brands` and `R.layout.activity_main` both refer to something called `R`. So what's `R`?**

**A:** `R` refers to *R.java*. It's a special file that's automatically generated when you build the app. Android uses *R.java* to keep track of any resources used within the app such as layouts, `String` resources, and views.

When you pass the `findViewById` method a value of `R.id.brands`, Android looks up the `brands` ID from `R`, and uses this to return a reference to the `brands` view. Similarly, when you pass the `setContentView` method a value of `R.layout.activity_main`, Android looks up the correct layout from `R`, and assigns it to the activity.

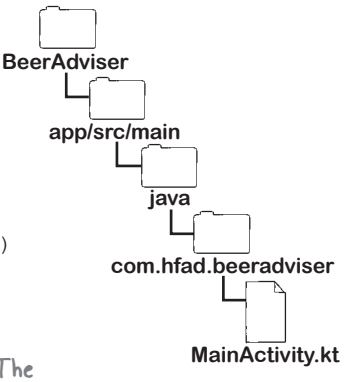
You don't change any of the code in *R.java* yourself: Android Studio automatically generates it for you.

# Add the getBeers() method

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

Now that we know that the `find_beer` button can respond to clicks, let's change its behavior so that each time the user clicks the button, it provides some real beer advice based on the value that's selected in the spinner. We'll do this by adding a new `getBeers()` method to *MainActivity.kt*, which we can then call in the button's `OnClickListener` code.

The `getBeers()` method is pure Kotlin code. It has one `String` parameter for the beer color, and returns a `List<String>` of beer suggestions. Add the `getBeers()` method (shown below in bold) to *MainActivity.kt*:



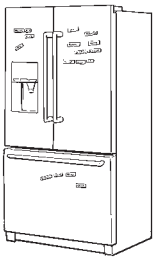

```
...
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val findBeer = findViewById<Button>(R.id.find_beer)
        findBeer.setOnClickListener {
            ...
        }
    }
```

Add the getBeers() method to MainActivity.kt. The method is pure Kotlin; nothing Androidy about it.

```
    fun getBeers(color: String): List<String> {
        return when (color) {
            "Light" -> listOf("Jail Pale Ale", "Lager Lite")
            "Amber" -> listOf("Jack Amber", "Red Moose")
            "Brown" -> listOf("Brown Bear Beer", "Bock Brownie")
            else -> listOf("Gout Stout", "Dark Daniel")
        }
    }
}
```

Next, we need to update the lambda that's passed to the `find_beer` button's `setOnClickListener()` method. We'll make it pass the selected beer color from the spinner to the `getBeers()` method, and update the `brands` text view with its return value.

See if you can piece together the code to do this by having a go at the following exercise.



# Activity Magnets

Somebody used fridge magnets to complete the code for `MainActivity`, but a freak kitchen whirlwind has dislodged some of the magnets. Can you piece the code back together again?

The code needs to call the `getBeers()` method, and display each item it returns in the `brands` text view. Each item should be displayed on a new line.

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val findBeer = findViewById<Button>(R.id.find_beer)
        findBeer.setOnClickListener {
            val beerColor = findViewById<Spinner>(R.id.beer_color)
            val color = beerColor.selectedItem

            val ..... = getBeers( ..... )

            val beers = beerList.reduce { str, item -> str + '\n' + item}
            val brands = findViewById<TextView>(R.id.brands)

            brands.text = .....
        }
    }

    fun getBeers(color: String): List<String> {
        return when (color) {
            "Light" -> listOf("Jail Pale Ale", "Lager Lite")
            "Amber" -> listOf("Jack Amber", "Red Moose")
            "Brown" -> listOf("Brown Bear Beer", "Bock Brownie")
            else -> listOf("Gout Stout", "Dark Daniel")
        }
    }
}
```

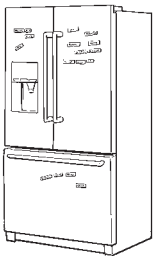
Kotlin's reduce function initializes a String—str—with the first item in beerList. It then loops through the remaining items in beerList, and adds each one to str, separated by a new line.

beers

color

.toString()

beerList



# Activity Magnets Solution

Somebody used fridge magnets to complete the code for `MainActivity`, but a freak kitchen whirlwind has dislodged some of the magnets. Can you piece the code back together again?

The code needs to call the `getBeers()` method, and display each item it returns in the `brands` text view. Each item should be displayed on a new line.

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val findBeer = findViewById<Button>(R.id.find_beer)
        findBeer.setOnClickListener {
            val beerColor = findViewById<Spinner>(R.id.beer_color)
            val color = beerColor.selectedItem

            val beerList = getBeers( color .toString() )

            val beers = beerList.reduce { str, item -> str + '\n' + item}
            val brands = findViewById<TextView>(R.id.brands)

            brands.text = beers
        }
    }
    fun getBeers(color: String): List<String> {
        return when (color) {
            "Light" -> listOf("Jail Pale Ale", "Lager Lite")
            "Amber" -> listOf("Jack Amber", "Red Moose")
            "Brown" -> listOf("Brown Bear Beer", "Bock Brownie")
            else -> listOf("Gout Stout", "Dark Daniel")
        }
    }
}
```

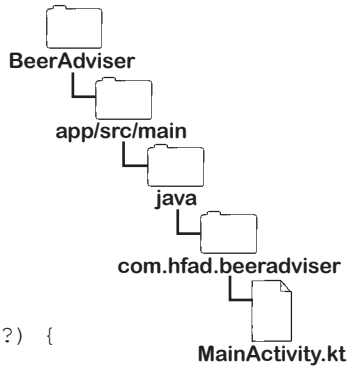
Convert the selected item to a String, and pass it to the getBeers method.

Display the beers String in the brands text view.

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

# The full code for MainActivity.kt

Below is the full code for `MainActivity`. Apply the changes shown in bold to *MainActivity.kt*.




```
package com.hfad.beeradviser

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.widget.Button
import android.widget.Spinner
import android.widget.TextView

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val findBeer = findViewById<Button>(R.id.find_beer)
        findBeer.setOnClickListener {
            val beerColor = findViewById<Spinner>(R.id.beer_color)
            val color = beerColor.selectedItem
            val beerList = getBeers(color.toString())
            val beers = beerList.reduce { str, item -> str + '\n' + item }
            val brands = findViewById<TextView>(R.id.brands)
            brands.text = ~~"Beer color is $color"~~ beers
        }
    }

    fun getBeers(color: String): List<String> {
        return when (color) {
            "Light" -> listOf("Jail Pale Ale", "Lager Lite")
            "Amber" -> listOf("Jack Amber", "Red Moose")
            "Brown" -> listOf("Brown Bear Beer", "Bock Brownie")
            else -> listOf("Gout Stout", "Dark Daniel")
        }
    }
}
```

Use the getBeers method to get a list of beers.

Build a String, displaying each beer on a new line.

Display the beers in the Brands text view.

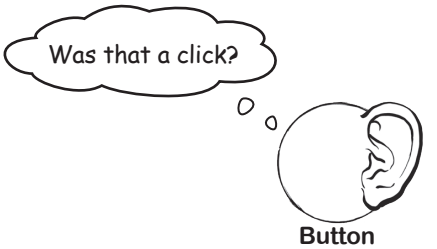
Before we take the code for a final test drive, let's go through what the code does when it runs.

# What happens when you run the code

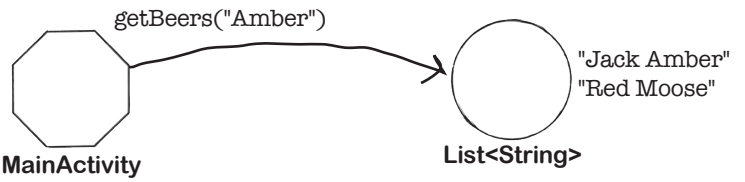
- Create project
- Update layout
- Add resources
- Respond to clicks
- **Write logic**

The following things happen when the app runs:

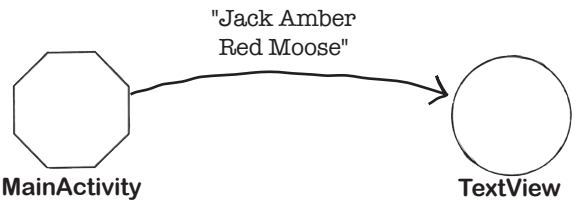
1. **When the user clicks on the Find Beer button, the button's OnClickListener hears the click.**




2. **The OnClickListener code in MainActivity calls the getBeers() method, passing in the beer color selected in the spinner.**
The `getBeers()` method returns a list of beers, which `MainActivity` saves in a separate variable.




3. **MainActivity formats the list of beers and uses it to set the text property in the text view.**



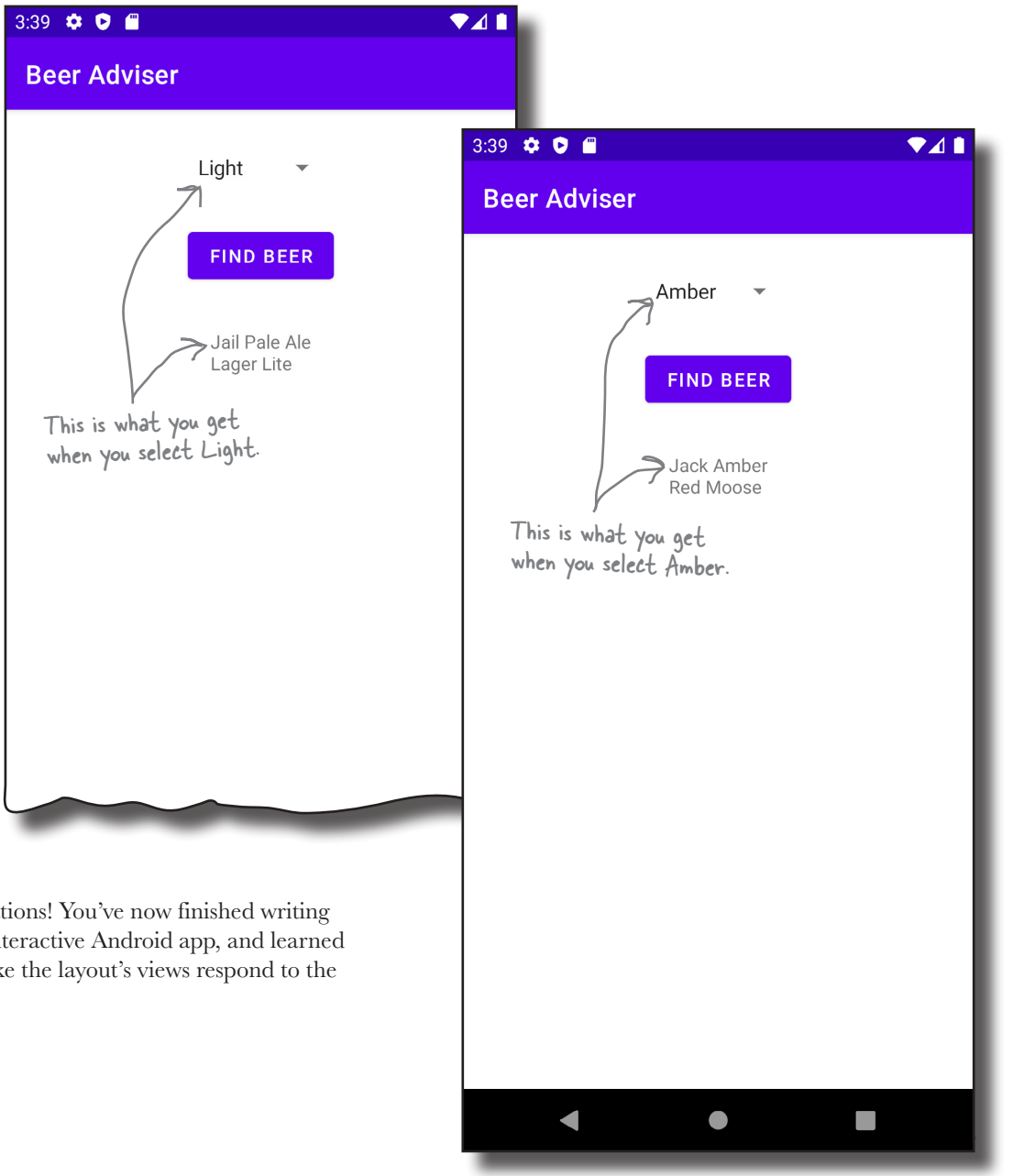

Let's take the app for a test drive.

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

# Test Drive

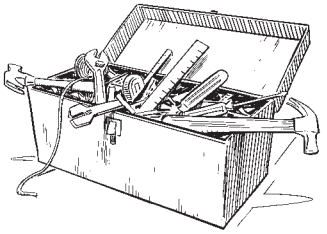
Once you've made the changes to the app, go ahead and run it.

When we try selecting different beer types and clicking on the Find Beer button, the app displays a selection of suitable beers. We get one set of beers when we choose the Light option, and another selection when we choose the Amber option.




Congratulations! You've now finished writing your first interactive Android app, and learned how to make the layout's views respond to the user.

# Your Android Toolbox



**You've got Chapter 2 under your belt and now you've added building interactive Android apps to your toolbox.**

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

## Bullet Points

- The `<Button>` element is used to add a button.
- The `<Spinner>` element is used to add a spinner, which is a drop-down list of values.
- All the UI components you add to a layout are types of view. They inherit from the Android `View` class.
- *strings.xml* is a `String` resource file. It's used to separate out text values from the layouts and activities, and supports localization.
- Add a `String` to *strings.xml* using:

```
<string name="name">
  string
</string>
```

- Reference a `String` in the layout using:

```
"@string/name"
```

- Add an array of `String` values to *strings.xml* using:

```
<string-array name="array_name">
    <item>string1</item>
    ...
</string-array>
```

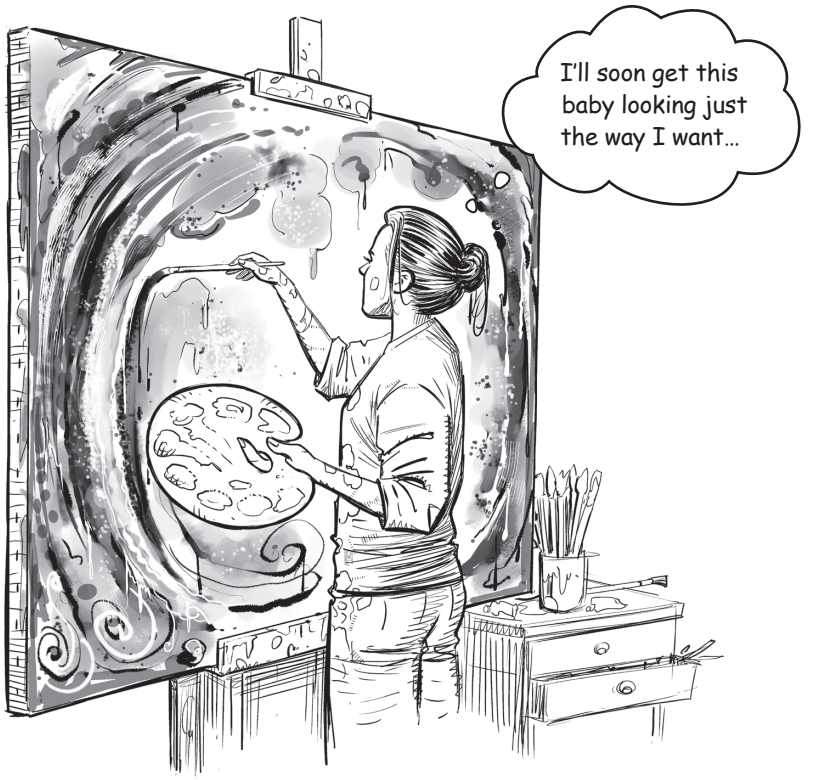
- Reference a `string-array` in the layout using:

```
"@array/array_name"
```

- Use the `findViewById` method to get a reference to a view by its ID.
- Make a button respond to clicks by calling its `setOnClickListener` method.
- The `setOnClickListener` method takes one parameter—a lambda—which describes what should happen when the button is clicked.
- Change the text that's displayed in a view by updating its `text` property.
- Get the selected item in a spinner using its `selectedItem` property.
- The `selectedItem` property can hold any type of value, so you need to convert its value before you can use it.

# 3 layouts

# 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**.

# It all starts with a layout

As you already know, layout files are written in XML, and they let you define what your app looks like.

Each time you write a layout, you need to do three things:

1. **Specify the type of layout.**
You tell Android how you want any views (such as buttons and text views) to be arranged by specifying a type of layout. A linear layout, for example, arranges views in a linear column or row, one after another.

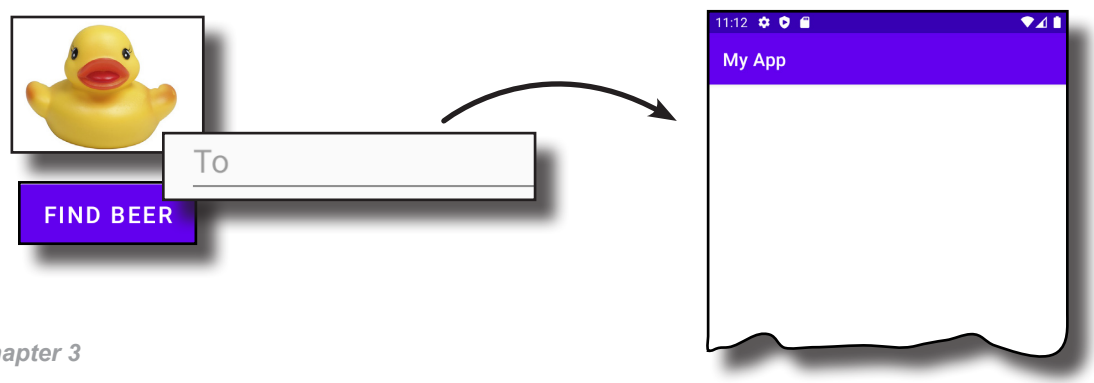
```
<LinearLayout ...>


</LinearLayout>
```

2. **Specify the views.**
Each layout contains one or more views, which your app uses to display information or interact with the user.

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Click me" />
```

3. **Tell an activity to use the layout.**
You tell Android which activity uses the layout you've just defined by adding Kotlin code like this to the activity.

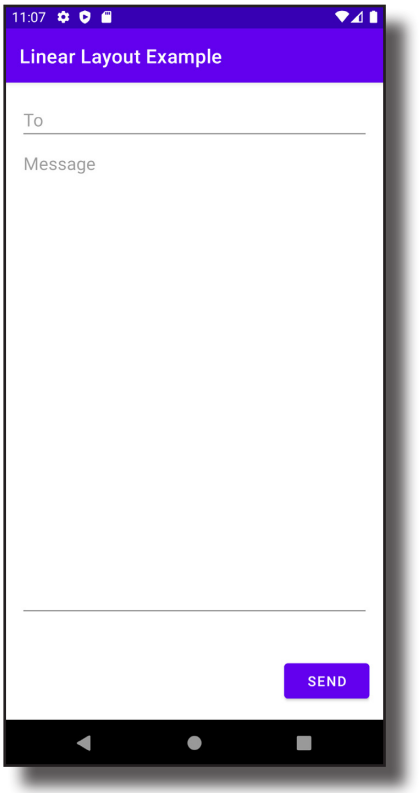
```
setContentView(R.layout.activity_main)
```



# Android has different types of layout

Android has different types of layout, and each one has its own policy about how its views are arranged. A linear layout, for example, will always lay views out in a linear row or column, while a frame layout stacks its views, one on top of another. Which type of layout you use depends on how you want views to be arranged on the device screen.

A linear layout arranges its views in a single column or row.

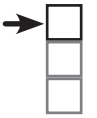


A frame layout stacks its views, one on top of another.

## Use the best layout for your screen design

All the apps you've seen so far have used linear layouts to arrange views in a single column. In this chapter, we're going to take a deeper look at linear layouts, and introduce you to two other types: a frame layout and a scroll view.

Let's start with linear layouts.

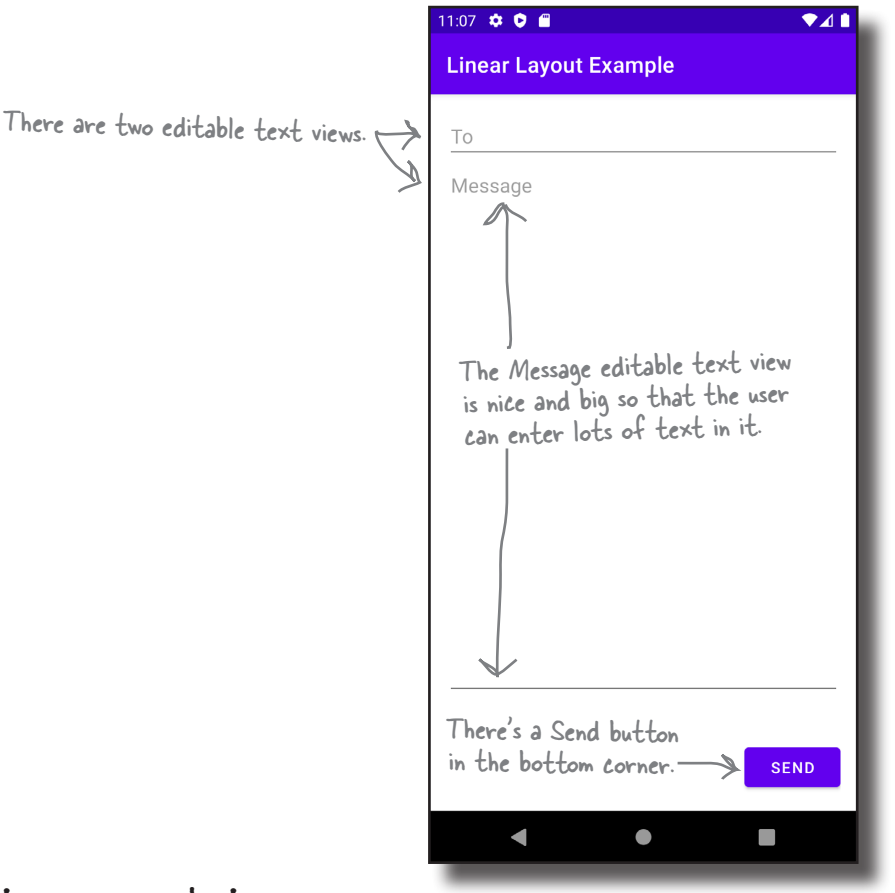


**Linear layout**
Frame layout
Scroll view

# Let's build a linear layout

We're going to use a linear layout to build the layout shown below. A linear layout is a good choice here, because the views are arranged in a single column. As you already know, a linear layout arranges views one after another in a vertical column or a horizontal row.

The layout is composed of two editable text views (text views that allow you to enter text) and a button. Here's what we want the layout to look like:




## Create a new project

We'll use a new Android Studio project for the linear layout app.

Create a new project using the same steps you used in the previous chapters. Choose the Empty Activity option, enter a name of "Linear Layout Example" and a package name of "com.hfad.linearlayoutexample", and accept the default save location. Make sure the language is set to Kotlin and the minimum SDK is API 21 so that it will run on most Android devices.

# How to define a linear layout

As you already know, you define a linear layout using a `<LinearLayout>` element. The code looks like this:

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

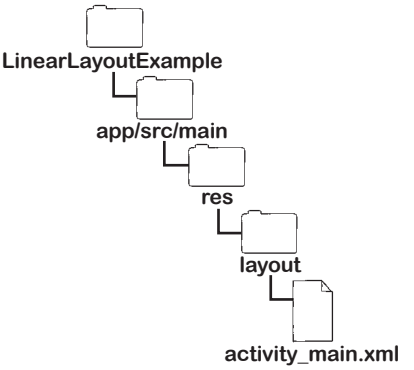
This line specifies the XML format.

You use <LinearLayout> to define a linear layout.

The layout_width and layout_height specify what size you want the layout to be.

The orientation specifies whether you want to display views vertically or horizontally.

There may be other attributes too.



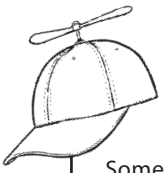

The `<LinearLayout>` element contains various different attributes that it needs to specify the layout's appearance and behavior.

The first is **`xmlns:android`**. This defines a namespace named `android`, and it needs to be set to `"http://schemas.android.com/apk/res/android"` as above. Defining this namespace gives your layout access to the elements and attributes that your layout needs, and you need to define it in every layout file you create.

The next two attributes are **`android:layout_width`** and **`android:layout_height`**, and these specify how wide and high you want the layout to be. These attributes are mandatory for *all* types of layout and view.

You can set `android:layout_width` and `android:layout_height` to `"wrap_content"`, `"match_parent"`, or a specific size such as 8dp—that's 8 density-independent pixels. `"wrap_content"` means that you want the layout to be just big enough to hold all of the views inside it, and `"match_parent"` means that you want the layout to be as big as its parent—in this case, as big as the device screen minus any padding (there's more about padding a few pages ahead). You will usually set the layout width and height to `"match_parent"`.

The next attribute sets the linear layout's orientation. We'll look at options for this attribute next.

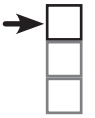


## Serious Coding

Some devices create very sharp images by using very tiny pixels. Other devices are cheaper to produce because they have fewer, larger pixels.

You use **density-independent pixels (dp)** to avoid creating interfaces that are overly small on some devices, and overly large on others. A measurement in density-independent pixels is roughly the same size across all devices.

You set text sizes using **scalable pixels (sp)**. Scalable pixels are like density-independent pixels except they're used for fonts.



**Linear layout**
Frame layout
Scroll view

# Orientation can be vertical or horizontal

You specify the direction in which you wish to arrange views using the **`android:orientation`** attribute.

You arrange views vertically in a single column using:

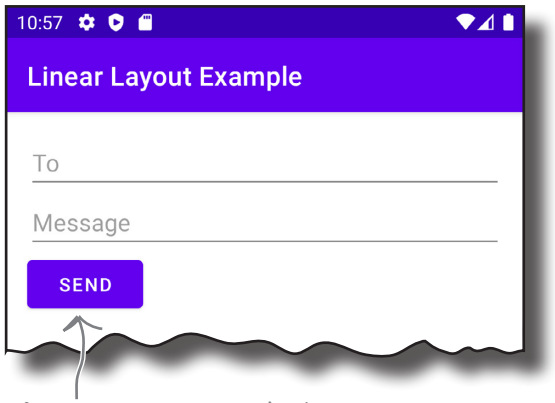
```
android:orientation="vertical"
```

and you arrange views horizontally in a single row with:

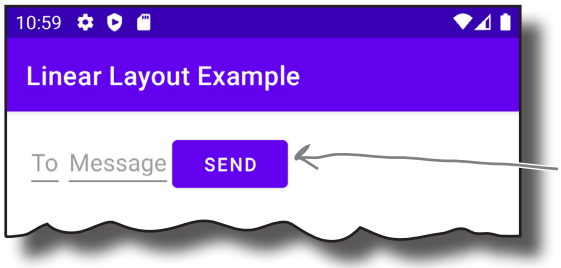
```
android:orientation="horizontal"
```

If the orientation is horizontal, the order in which the views are arranged depends on the device language settings.

If the device language is set to one that is read from left to right, such as English, the views are displayed in a horizontal row from left to right like this:




If the orientation is vertical, views are arranged in a single vertical column.



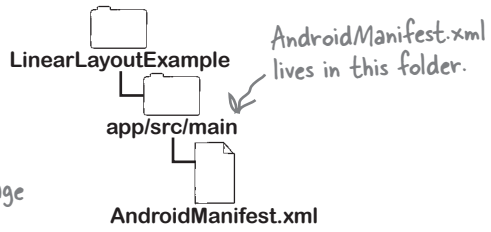

If the orientation is horizontal, views are arranged in a horizontal row. Here, the device language is set to English, so the views are arranged from left to right.

If the language on the device is set to one that's read from right to left instead, like Arabic, you can choose to display the views from right to left, so that the first view appears against the rightmost edge of the layout. You enable this feature by including a property named **`supportsRtl`** in a file named *AndroidManifest.xml* file, and setting it to `"true"`:

supportsRtl means "supports right to left".

```
<manifest ...>
    <application
        ...
        android:supportsRtl="true">
        ...
    </application>
</manifest>
```

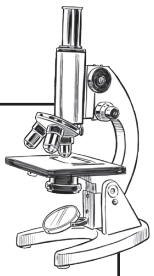
This adds right-to-left language support to your app. Android Studio usually adds this line for you.



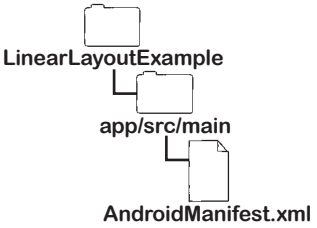

AndroidManifest.xml lives in this folder.

Before we look at what other attributes you can use with linear layouts, let's take a quick detour to find out more about *AndroidManifest.xml*.

# Anatomy of AndroidManifest.xml



Every Android app must include a file called *AndroidManifest.xml*, which lives in the *app/src/main* folder of your project. The *AndroidManifest.xml* file contains essential information about your app that the Android operating system, the Android build tools, and the Google Play Store need to know about. This includes the app's package name, details of any activities, and any permissions the app requires. Android Studio creates the *AndroidManifest.xml* file for you when you create your project.




A typical *AndroidManifest.xml* looks like this:

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.LinearLayoutExample">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

This is the package name that's used by the app.

Android Studio provides your app with default icons.

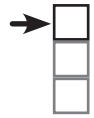
The app supports right-to-left languages.

The theme affects the appearance of your app. You'll find out more about this in Chapter 8.

The app contains a single activity named MainActivity.

MainActivity is the app's main activity, and it starts when the app is launched.

Android Studio usually adds the `supportsRtl` property to *AndroidManifest.xml* automatically, so that your app supports right-to-left languages by default.



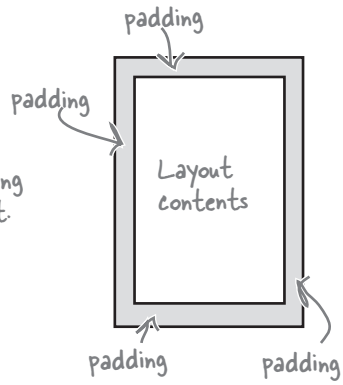
**Linear layout**
Frame layout
Scroll view

# Use padding to add space to the layout's edges

Once you've specified the type of layout you want to use, you can optionally use one or more **padding** attributes to add some extra space between each of the layout's edges and its contents. The following code, for example, uses the `android:padding` attribute to add 16dp to each edge:

```
<LinearLayout ...
    android:padding="16dp" >

</LinearLayout>
```

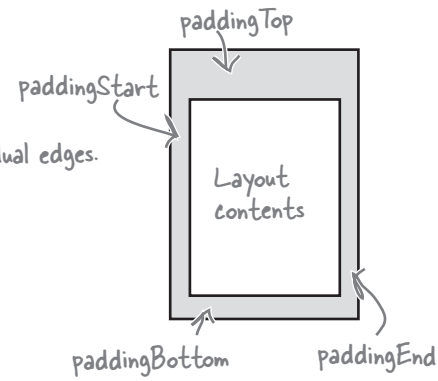
This adds the same padding to all edges of the layout.




If you want to add different amounts of padding to different edges, you can specify the edges individually. The following code, for example, adds padding of 32dp to the top of the layout, and 16dp to the other edges:

```
<LinearLayout ...
    android:paddingBottom="16dp"
    android:paddingStart="16dp"
    android:paddingEnd="16dp"
    android:paddingTop="32dp" >

</LinearLayout>
```

Add padding to the individual edges.

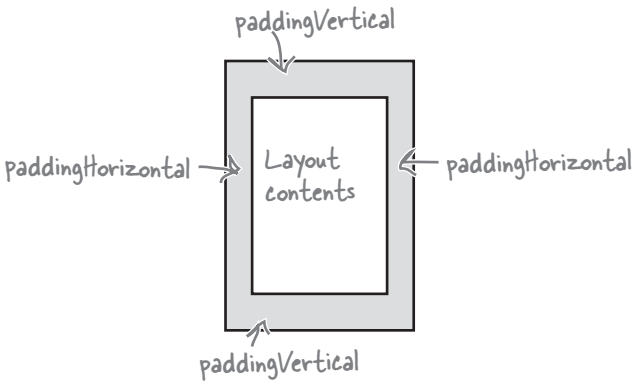



The `android:paddingStart` attribute adds padding to the start edge of the layout. For left-to-right languages (such as English), the start edge is on the left, while if the device language is set to one that's read from right to left—and the app supports right-to-left languages—the start edge is on the right.

The `android:paddingEnd` attribute adds padding to the end edge of the layout. This is on the right for left-to-right languages, and on the left for right-to-left languages (where the app supports this feature).

If you want to apply the same amount of padding to the horizontal or vertical edges, you can also use `android:paddingHorizontal` and `android:paddingVertical`. These attributes add padding to the horizontal and vertical edges of the layout, respectively.

Now that you've learned how to add padding to a linear layout, let's add some to the layout we're building.



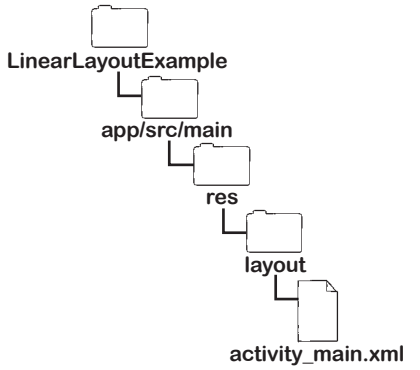
# The layout code so far

In the app, we're going to use a linear layout with a vertical orientation, and add padding of 16dp to each edge so that there's a bit of space between the layout's edges and its contents.

Open *activity_main.xml*, and replace its code so that it matches the code below:

Replace the current layout code so it uses a linear layout.

```
<?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:orientation="vertical"
    android:padding="16dp"
    tools:context=".MainActivity" >

</LinearLayout>
```




We've now defined an empty linear layout; let's go ahead and add some views to it.

## there are no Dumb Questions

**Q: I see the above layout code includes an extra `tools` namespace. What's that for?**

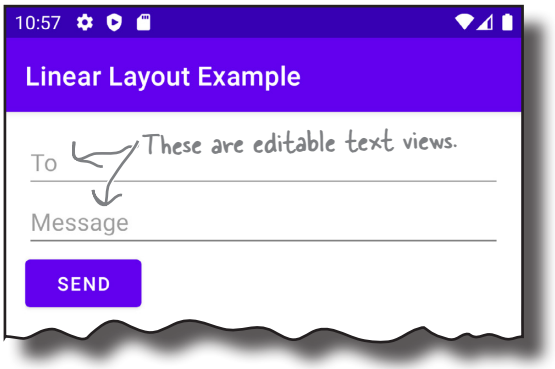
A: The `tools` namespace includes a set of attributes that are used by the tools included in Android Studio. It doesn't change the appearance of your app on the device screen: it's just there to make your coding life a little easier.

**Padding puts extra space between the layout's edges and its contents.**

**You can also use padding with views. This adds extra space between the view's edges and its contents**

# An edit text lets you enter text

The linear layout needs to display a button and two editable text views (which are used to enter text). You already know how to use buttons, so before we update the layout, let's find out how to include an editable text view.




An editable text view is a type of text view that lets you input text. You add one to your layout using an **<EditText>** element, and the code looks like this:

```
<EditText
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="To"
    android:inputType="text" />
```

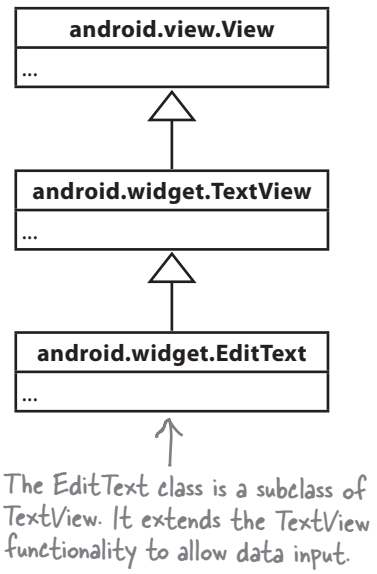
The above code creates an editable text view that's as wide as its parent, and just high enough to hold its contents.

The **android:hint** attribute is used to define hint text. This is displayed when the editable text view is empty, and it gives users a hint as to what sort of text they should enter. In the above example, we've hardcoded the hint text, but in the real world you'd want to include this as a `String` resource instead.

The `android:inputType` attribute specifies what type of data you expect the user to enter so that Android can provide the correct keyboard type. In the above example, we're using:

```
android:inputType="text"
```

which lets user enter a single line of text. Here are some more useful input types you might want to use:




| Value | What it does |
|---|---|
| **text** | Lets the user enter a single line of text. |
| **textMultiLine** | Lets the user enter multiple lines of text. |
| **phone** | Provides a phone number keypad. |
| **textPassword** | Displays a text entry keypad, and your input is concealed. |
| **textCapSentences** | Capitalizes the first word of a sentence. |

You can find out more in the online Android developer documentation at https://developer.android.com/training/keyboard-input/style.

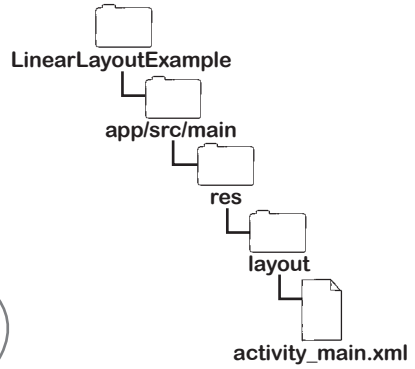
Now that you've learned how to use editable text views, let's add views to the layout code in the app we're building.

# Add views to the layout XML

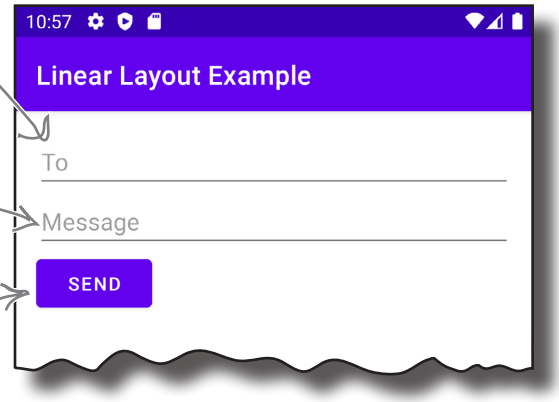
When you define a linear layout, you list views in the layout in the order you want them to display.

In the app we're building, we want to show two editable text views with a button underneath. The layout code looks like this, so update the code for *activity_main.xml* to include these changes (in bold):

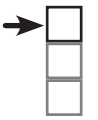
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp"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

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="To"
        android:inputType="text" />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Message"
        android:inputType="textMultiLine" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Send" />

</LinearLayout>
```




In the real world, you'd want to use String resources for the hint and text attributes.



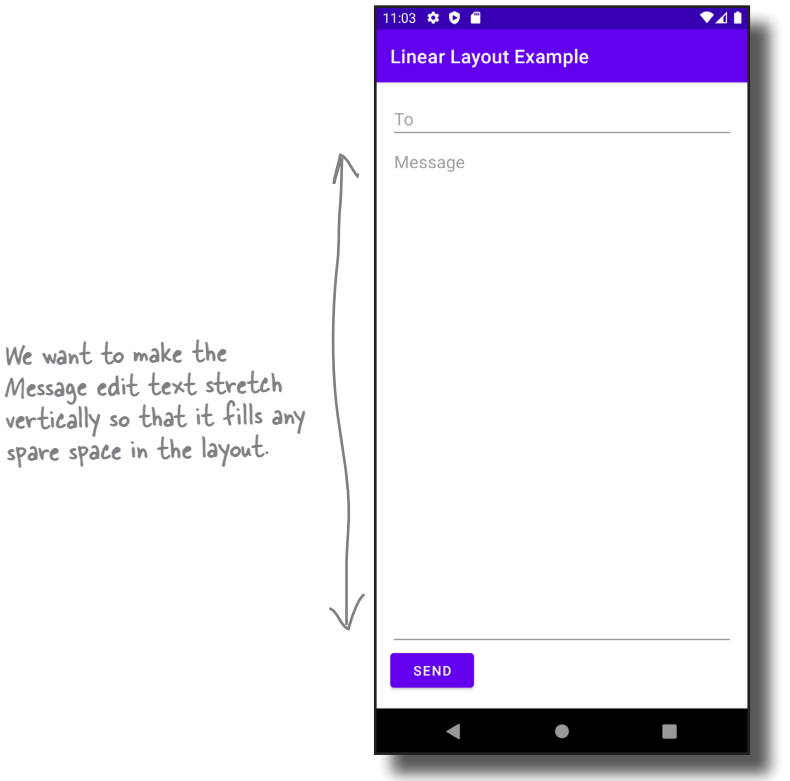

Those are all the views the layout needs. So what's next?



**Linear layout**
Frame layout
Scroll view

# Make a view streeeeetch by adding weight

All of the views in the current layout take up just as much vertical space as they need for their content. But what we *really* want is to make the Message edit text stretch to take up any vertical space in the layout that's not being used by the other views like this:




To do this, we need to allocate some **weight** to the Message area. Allocating weight to a view is a way of telling it to stretch to take up extra space in the layout.

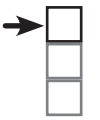
You assign weight to a view using:

```
android:layout_weight="number"
```

where `number` is some number greater than 0.

When you allocate weight to a view, the layout first makes sure that each view has enough space for its content: each button has space for its text, each edit text has space for its hint, and so on. Once it's done that, the layout takes any extra space and divides it proportionally between the views with a weight of 1 or greater.

Let's see how to apply this to the layout we're building.



Linear layout
Frame layout
Scroll view

# How to add weight to one view

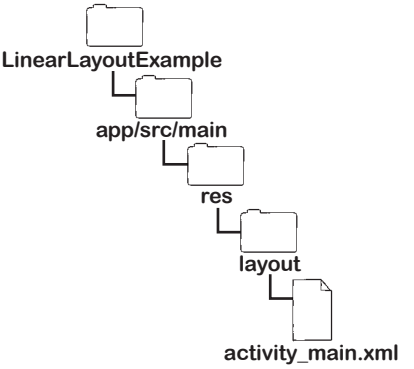
We need the Message edit text to take up any extra space in the layout that's not used by either of the other two views. To do this, we'll set its `android:layout_weight` attribute to `1`. As this is the only view in the layout with a weight value, this will make the text field stretch vertically to fill the remainder of the screen.

Here's the code; update *activity_main.xml* to include the changes (in bold):

```
<LinearLayout ... >
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="To"
        android:inputType="text" />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:hint="Message"
        android:inputType="textMultiLine" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Send" />
</LinearLayout>
```

This edit text and the button have no weight set. They'll take up as much room as their content needs, but no more.

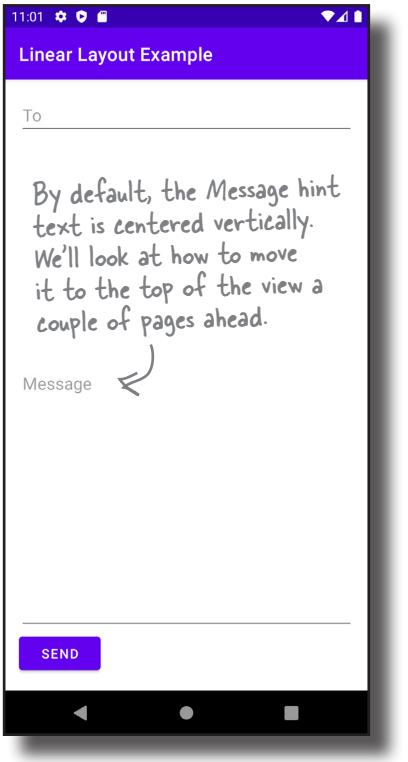
The Message edit text is the only view with any weight. It will expand to fill the space that's not needed by any of the other views.



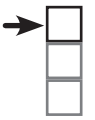

The height of the view will be determined by the linear layout based on the layout_weight. Replacing the layout_height with "0dp" is more efficient than setting it to "wrap_content", as this way Android doesn't have to work out the value of "wrap_content".

Giving the Message edit text a weight of `1` means that it takes up all of the extra space that's not used by the other views in the layout. This is because neither of the other two views has been allocated any weight in the layout XML.

The Message view has a weight of 1. As it's the only view with its weight attribute set, it expands to take up any extra vertical space in the layout.




In this example, we only need to give a weight to a single view. Before we update the layout further, let's see what happens when we need to assign weight to multiple views.



**Linear layout**
Frame layout
Scroll view

# How to add weight to multiple views

When you assign weight to multiple views, the linear layout uses the weight you assign to each view to work out what proportion of the remaining space each view should take up.

As an example, suppose we give the To edit text a weight of 1, and the Message edit text a weight of 2, like this:

```
<LinearLayout ... >
    <EditText
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:hint="To"
        android:inputType="text" />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="2"
        android:hint="Message"
        android:inputType="textMultiLine" />

       <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Send" />
</LinearLayout>
```

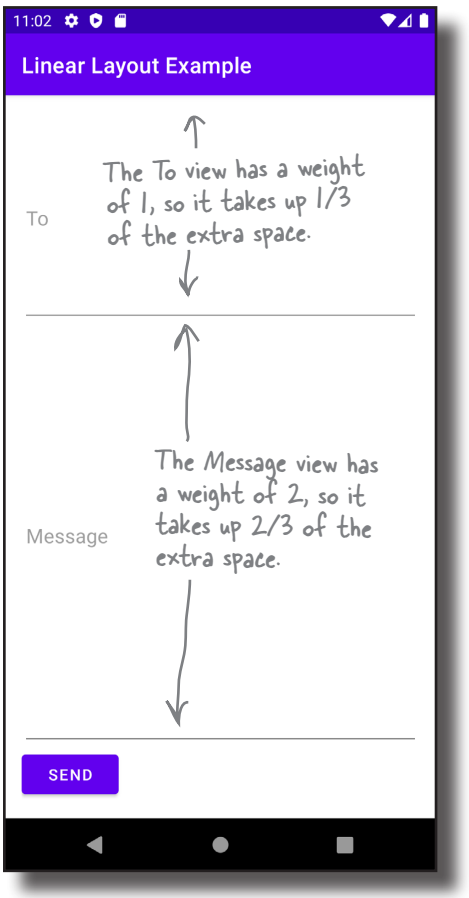
The linear layout spots that the To and Message editable text views have weights, and it uses these to work out how much space each view should take up.

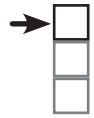
It starts by adding together the `android:layout_weight` attributes for each view. In this example, the To and Message views have weights of 1 and 2 respectively, giving a total of 3.

The proportion of extra space taken up by each view is the view's weight divided by the total weight. The To view has a weight of 1, so this means it will take up 1/3 of the remaining space in the layout. The Message view has a weight of 2, so it will take up 2/3 of the remaining space.

Now that you've learned how to use weight, let's continue to update the layout.

This is just an example; we're not really going to change the layout so it looks like this.





**Linear layout**
Frame layout
Scroll view

# The gravity attribute controls the position of a view's contents

The next thing we'll do is move the hint text that's displayed inside the Message edit text. At the moment, it's centered vertically inside the view. We want to change it so that the text appears at the top of the edit text field, and we can achieve this using the **`android:gravity`** attribute.

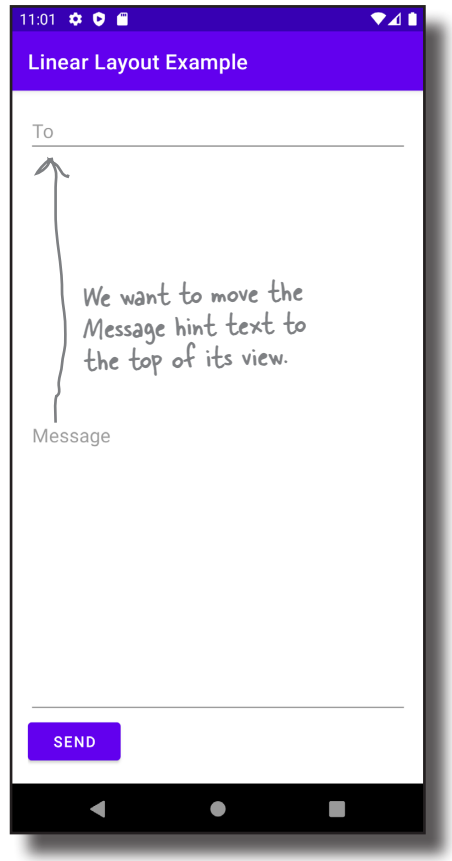
The `android:gravity` attribute lets you specify how you want to position the contents of a view inside the view—for example, how you want to position text inside a text view. If you want a view's contents to appear at the top of the view, as we do here, we can set its `android:gravity` attribute to `"top"` like this:

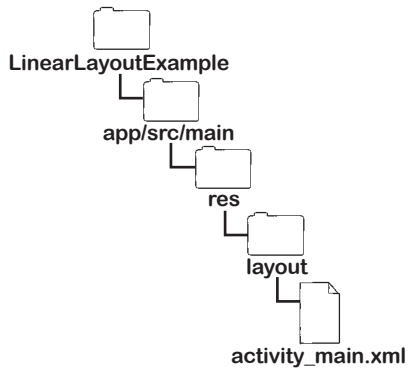
```
android:gravity="top"
```

We'll add an `android:gravity` attribute to the Message edit text so that the hint text moves to the top of the view. Here's the code; update *activity_main.xml* to include the change (in bold):

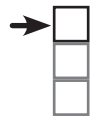
```
<LinearLayout ... >
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="To"
        android:inputType="text" />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:gravity="top"
        android:hint="Message"
        android:inputType="textMultiLine" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Send" />
</LinearLayout>
```

Moves the hint text to the top of the edit text




You'll find a list of other values you can use with the `android:gravity` attribute on the next page.



**Linear layout**
Frame layout
Scroll view

# Values you can use with the android:gravity attribute

Here are some more of the values you can use with the `android:gravity` attribute. Add the attribute to your view, and set its value to one of the values below:

```
android:gravity="value"
```

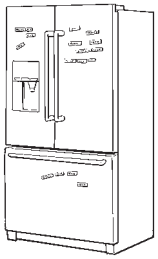
| | |
|---|---|
| **top** | Puts the view's contents at the top of the view. |
| **bottom** | Puts the view's contents at the bottom of the view. |
| **start** | Puts the view's contents at the start of the view. |
| **end** | Puts the view's contents at the end of the view. |
| **center_vertical** | Centers the view's contents vertically. |
| **center_horizontal** | Centers the view's contents horizontally. |
| **center** | Centers the view's contents vertically and horizontally. |
| **fill_vertical** | Makes the view's contents fill the view vertically. |
| **fill_horizontal** | Makes the view's contents fill the view horizontally. |
| **fill** | Makes the view's contents fill the view vertically and horizontally. |

You can also apply multiple gravities to a view by separating each value with a "|". To sink a view's contents to the bottom-end corner, for example, you'd use:

```
android:gravity="bottom|end"
```

Now that you know how to position a view's contents using gravity, have a go at the following exercise on the next page.

android:gravity lets you say where you want the view's contents to appear inside the view.

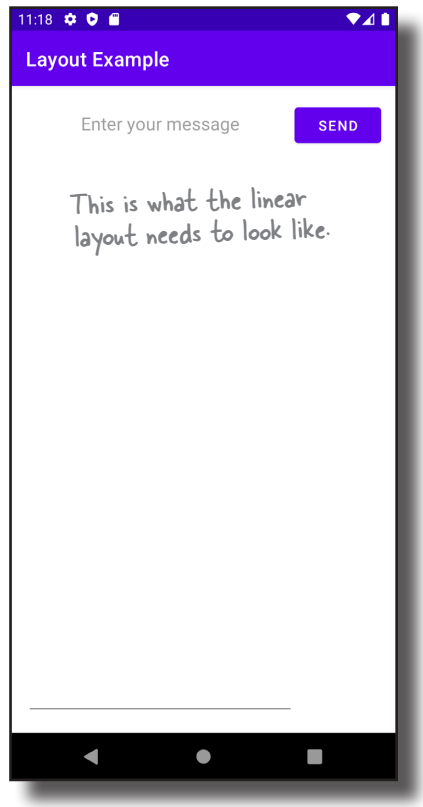


# Layout Magnets

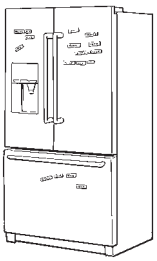
Somebody used fridge magnets to create a linear layout that, when run, produces the output below. Unfortunately, a passing sharknado has dislodged some of the magnets. Can you piece the code back together again?

Hint: You won't need to use all of the magnets.

```
<?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= .....

    android:layout_height= .....

    .....
    android:padding="16dp"
    tools:context=".MainActivity">

    <EditText
        android:layout_width= .....

        android:layout_height= .....

        .....

        .....

        android:hint="Enter your message"
        android:inputType="textMultiLine" />

    <Button
        android:layout_width= .....

        android:layout_height= .....

        android:text="Send" />
</LinearLayout>
```




"vertical"
"horizontal"
"0dp"
"top|start"
"wrap_content"
"match_parent"
android:orientation
android:gravity
android:layout_weight
"top|center_horizontal"
"1"
"wrap_content"
=
"match_parent"
android:weight
=
=
"match_parent"



# Layout Magnets Solution

Somebody used fridge magnets to create a linear layout that, when run, produces the output below. Unfortunately, a passing sharknado has dislodged some of the magnets. Can you piece the code back together again?

```
<?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:orientation = "horizontal"
    android:padding="16dp"
    tools:context=".MainActivity">

    <EditText
        android:layout_width= "0dp"
        android:layout_height= "match_parent"
        android:layout_weight = "1"
        android:gravity = "top|center_horizontal"
        android:hint="Enter your message"
        android:inputType="textMultiLine" />

    <Button
        android:layout_width= "wrap_content"
        android:layout_height= "wrap_content"
        android:text="Send" />
</LinearLayout>
```

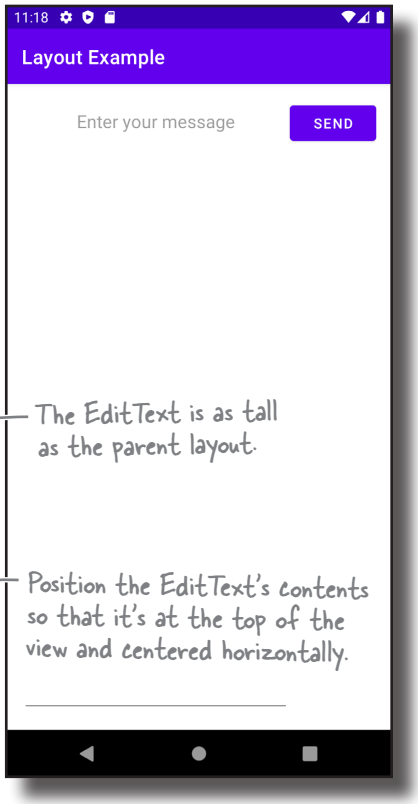
The views are displayed in a horizontal row.

Use any horizontal space that's not required by the Button.

The EditText is as tall as the parent layout.

Position the EditText's contents so that it's at the top of the view and centered horizontally.

Make the Send button just large enough for its contents.




You didn't need to use these magnets.

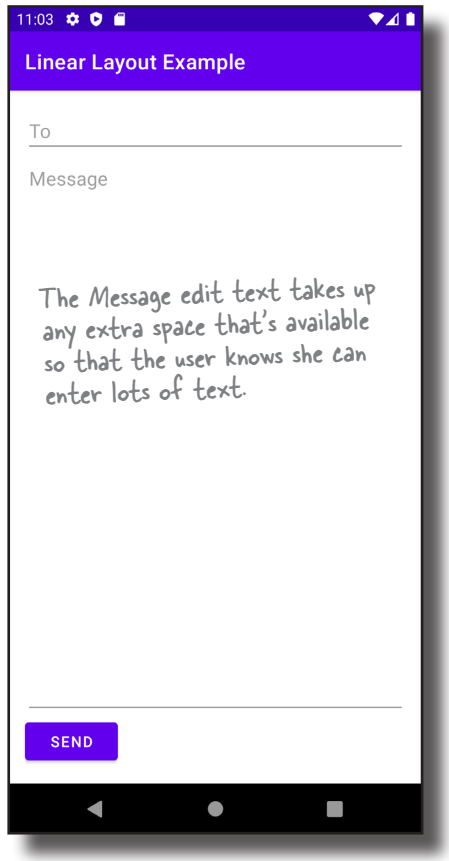
"top|start"

android:weight

"vertical"

# The story so far

So far, we've added three views to the linear layout, and adjusted their position by adding `layout_weight` and `gravity` attributes to the Message edit text. These attributes mean that the edit text uses up any extra space that's not being used by either of the other views, and its hint text is displayed at the top of the view:

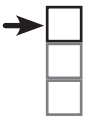



Our linear layout is nearly complete, but there are just two more changes we're going to make.

1. **Move the Send button to the end edge.**
   For left-to-right languages, this will move the button over to the right.

2. **Add more space between the Message edit text and the top of the button.**

Let's see how to do this, starting with moving the button to the end.



**Linear layout**
Frame layout
Scroll view

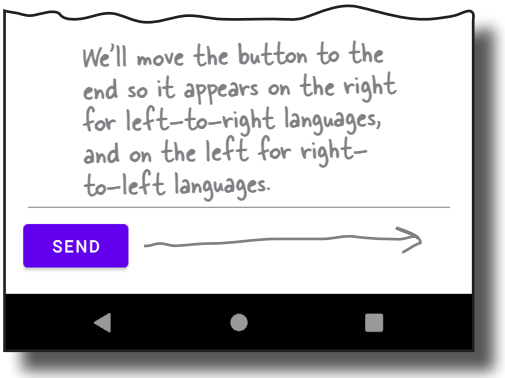
# layout_gravity controls the position of a view within a layout

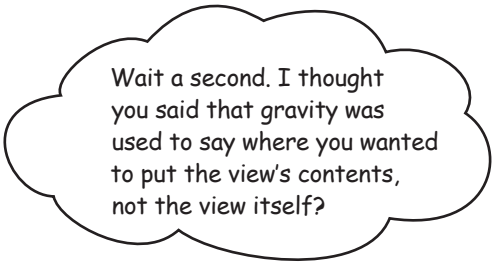
To move the button to the end edge of the layout, we'll add an **`android:layout_gravity`** attribute to the button.

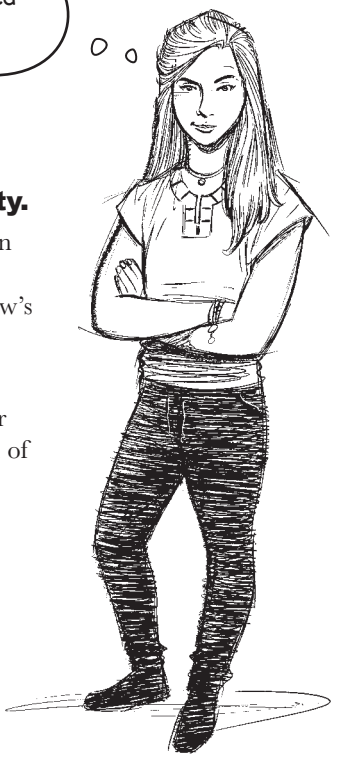
The `android:layout_gravity` attribute lets you specify where you want a view in a linear layout to appear in its enclosing space. You can use it to push a view to the right, for instance, or center the view horizontally.

To move the Send button to the end edge of the layout, we'll set the `android:layout_gravity` attribute to `"end"` using code like this:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="end"
    android:text="Send" />
```



**Linear layouts have two attributes that sound similar to one another, gravity and layout_gravity.**

Previously, we used the `android:gravity` attribute to position the Message hint text inside an edit text. This is because the `android:gravity` attribute lets you say where you want a view's **contents** to appear.

`android:layout_gravity` deals with the **placement of the view itself**, and lets you control where views appear in their available space. In our case, we want the view to move to the end of its available space, so we're using:

```
android:layout_gravity="end"
```

Take a look at the next page, and you'll see a list of some of the other values you can use with the `android:layout_gravity` attribute.

# More values you can use with the android:layout_gravity attribute

Here are some of the values you can use with the `android:layout_gravity` attribute. Add the attribute to your view, and set its value to one of the values below:

```
android:layout_gravity="value"
```

| Value | What it does |
|---|---|
| **top, bottom, start, end** | Puts the view at the top, bottom, start, or end of its available space. |
| **center_vertical, center_horizontal** | Centers the view vertically or horizontally in its available space. |
| **center** | Centers the view vertically and horizontally in its available space. |
| **fill_vertical, fill_horizontal** | Grows the view so that it fills its available space vertically or horizontally. |
| **fill** | Grows the view so that it fills its available space vertically and horizontally. |

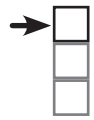
You can assign multiple values to a view's `android:layout_gravity` attribute by separating each value with a "`|`". To move a view to the bottom-end corner of its available space, for example, you'd use the code:

```
android:layout_gravity="bottom|end"
```

Now that you know how change a view's position using the `android:layout_gravity` attribute, let's find out how to add more space between views.

android:layout_gravity lets you say where you want views to appear in their available space.

android:layout_gravity deals with the placement of the view itself, whereas android:gravity controls the view's contents.



**Linear layout**
Frame layout
Scroll view

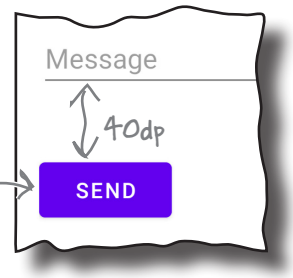
# Use margins to add space between views

When you position views using a linear layout, the layout doesn't leave much space between them. You can increase the amount of space around a view by adding one or more **margins** to it.

Suppose you have two views in a linear layout—an edit text that's positioned above a button. If you wanted to increase the space between the two views, you could add a margin of 40dp to the top of the button using the `android:layout_marginTop` attribute like this:

```
<LinearLayout...>
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Message"
        android:inputType="textMultiLine" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="40dp"
        android:text="Send" />
</LinearLayout>
```

Adding a margin to the top of the button adds extra space between the two views.




Here's a list of the types of margin you can use to give your views extra space. Add the attribute to the view, and set its value to the size of margin you want:

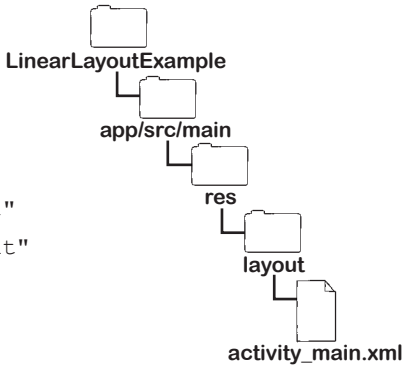
```
android:attribute="8dp"
```

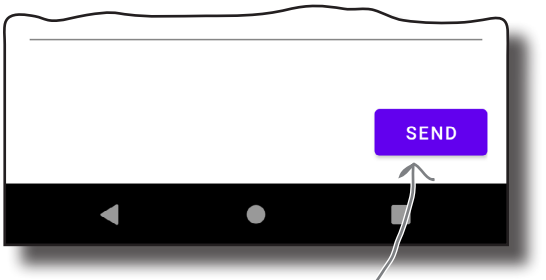
| Attribute | What it does |
|---|---|
| **layout_marginTop** | Adds extra space to the top of the view. |
| **layout_marginBottom** | Adds extra space to the bottom of the view. |
| **layout_marginStart** | Adds extra space to the start of the view. |
| **layout_marginEnd** | Adds extra space to the end of the view. |
| **layout_margin** | Adds equal space to each side of the view. |
| **layout_marginVertical, layout_marginHorizontal** | Adds equal space to the vertical (top and bottom) or horizontal (start and end) edges of the view. |

# The full linear layout code

Now that you know a view's `layout_gravity` and `margin` attributes, let's use them in the linear layout code to reposition the button and add some space to its top edge. Here's the full code for *activity_main.xml*; update the code to include the changes below (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:orientation="vertical"
    android:padding="16dp"
    tools:context=".MainActivity" >

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="To"
        android:inputType="text" />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:gravity="top"
        android:hint="Message"
        android:inputType="textMultiLine" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="end"
        android:layout_marginTop="40dp"
        android:text="Send" />
</LinearLayout>
```




Move the button to the end of its available space, and add a margin of 40dp above it.

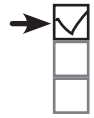
We've now finished writing the linear layout code, so let's take it for a test drive.

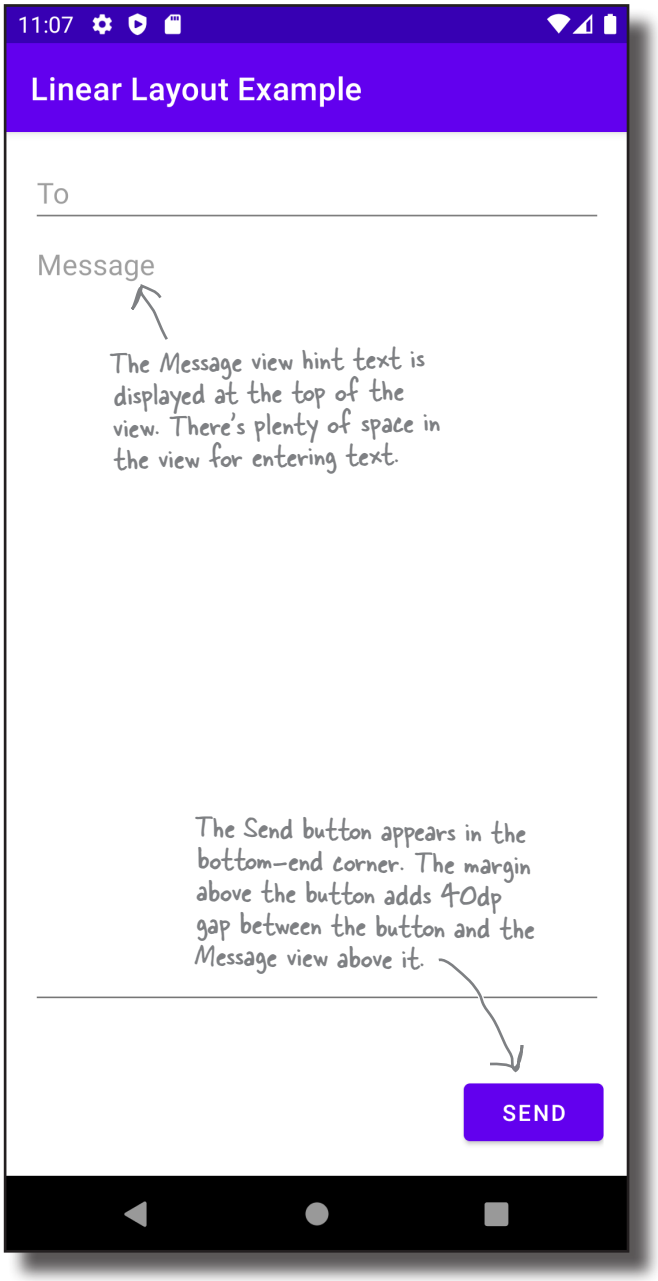
# Test Drive

Once you've made the changes to your app, go ahead and run it.

The app displays a linear layout featuring three views: two editable text views and a button. The views are displayed in a vertical column. The To view appears at the top, the Send button is displayed in the bottom end corner, and the Message view takes up any extra space, allowing for a margin of 40dp at the top edge of the button.

Now that you know how to create a linear layout, and control how its views are displayed, have a go at the exercise on the next page.

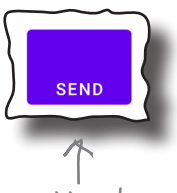


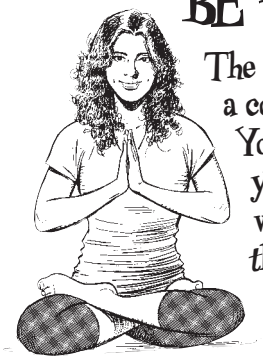




## there are no Dumb Questions

**Q: Could I have used a `paddingTop` attribute with the button instead of `layout_marginTop`? Would that have had the same effect?**

A: You can use padding attributes such as `paddingTop` with views, but it has a different effect. As you've seen, `layout_marginTop` increases the amount of space *above* the view. `paddingTop`, however, increases the height of the view by adding extra space *inside it*:



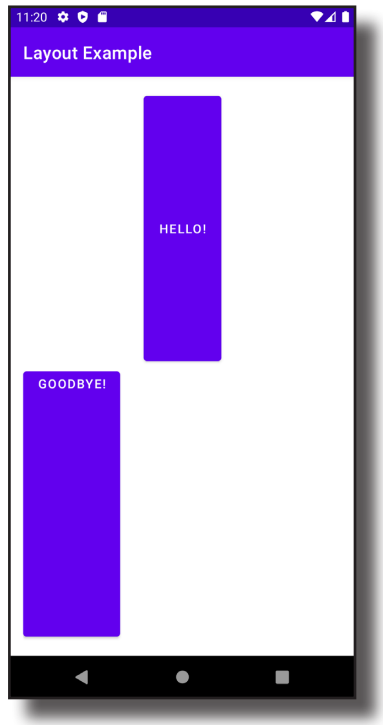


# BE the Layout

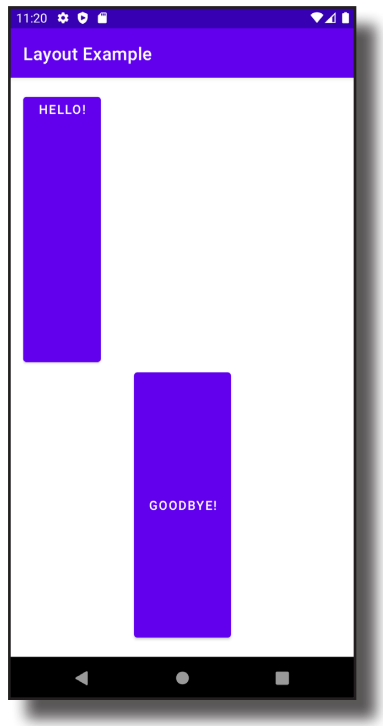
The code below describes a complete linear layout. Your job is to play like you're the layout, and say which screen (A or B) the layout will produce when run.

```
<?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:orientation="vertical"
    android:padding="16dp"
    tools:context=".MainActivity">

    <Button
        android:layout_width="wrap_content"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:layout_gravity="center_horizontal"
        android:text="Hello!" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:gravity="center_horizontal"
        android:text="Goodbye!" />

</LinearLayout>
```

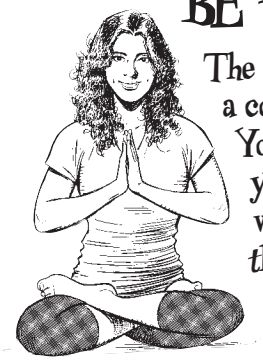
A




B



# BE the Layout Solution

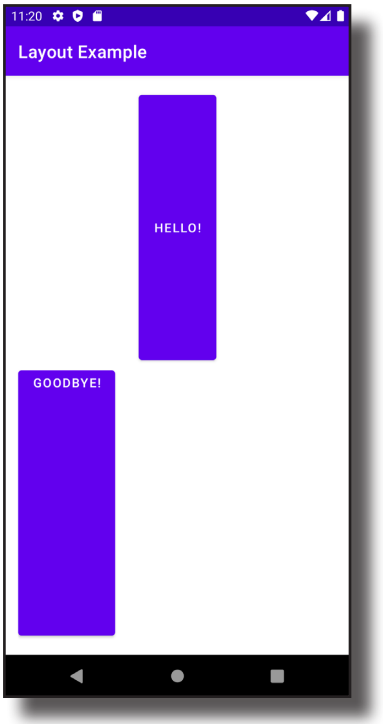


The code below describes a complete linear layout. Your job is to play like you're the layout, and say which screen (A or B) the layout will produce when run.

```
<?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:orientation="vertical"
    android:padding="16dp"
    tools:context=".MainActivity">

    <Button
        android:layout_width="wrap_content"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:layout_gravity="center_horizontal"
        android:text="Hello!" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:gravity="center_horizontal"
        android:text="Goodbye!" />

</LinearLayout>
```

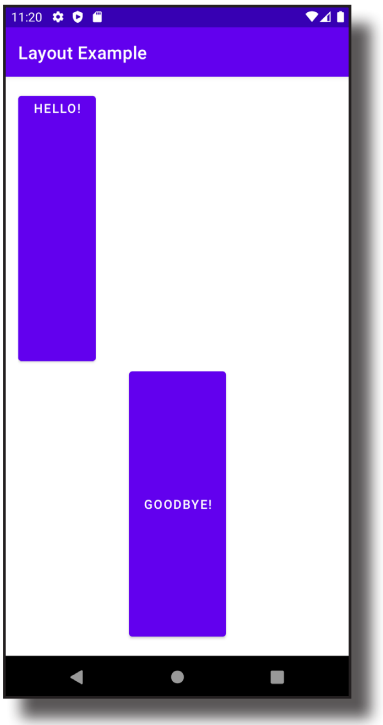
A

The layout code produces screen A. The "Hello!" button is centered horizontally, and the text of the "Goodbye!" button is centered horizontally in the view.




B

You'd get screen B if you switched the gravity and layout_gravity properties.



# Your activity code tells Android which layout it uses

So far in this chapter, you've learned how to use a linear layout and fine-tune the way in which its views are displayed. Before we introduce you to another type of layout, let's go behind the scenes and look at what happens to the layout when the app runs.

As you already know, when Android runs an app, it launches the app's main activity. In the current app, this is an activity named `MainActivity`.

When the activity launches, its `onCreate()` method runs. This method includes the following code that specifies which layout the activity should use:

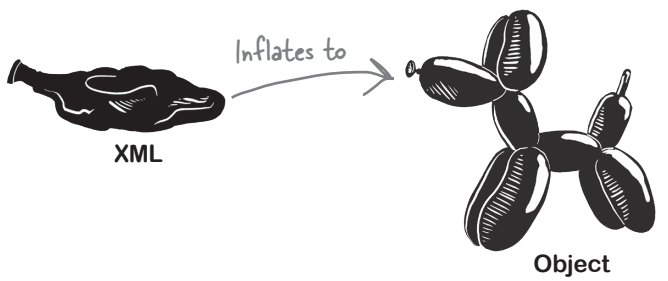
```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
}
```

This tells the activity to use the layout file activity_main.xml.

As you can see, the above code tells Android which layout to use by passing the name of a layout to a method named `setContentView()`. The method then displays the layout on the screen.

## The layout's views are inflated into objects

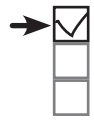
As well as displaying the layout on the device screen, the `setContentView()` method converts the views in the layout's XML into objects. This process is called **layout inflation** because it *inflates* each view into an object:




Layout inflation is important because it's what allows your activity code to manipulate the views in your layout. Behind the scenes, each view is rendered to an object, which you can interact with using activity code.

Let's see how layout inflation works with the linear layout code we've built.

**When you run your app, Android instantiates the layout XML by converting each of the layout's items into an object. This is known as layout inflation.**



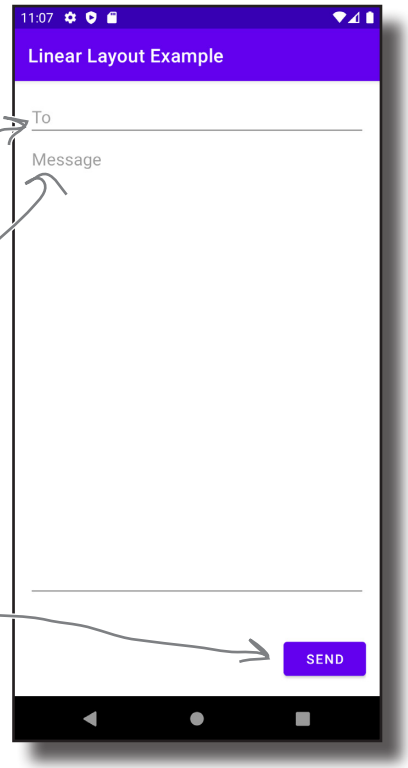
**Linear layout**
Frame layout
Scroll view

# Layout inflation: an example

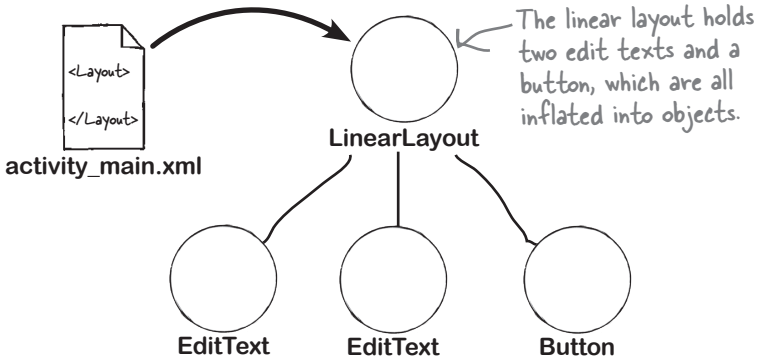
As you already know, the linear layout code displays two editable text views and a button inside a linear layout:

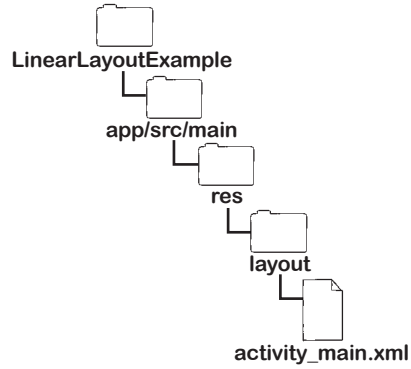
We've left out some of the details, but the code follows this basic structure.

```
<LinearLayout ... >

    <EditText
        ...
        android:hint="To"
        android:inputType="text" />

    <EditText
        ...
        android:hint="Message"
        android:inputType="textMultiLine" />

    <Button
        ...
        android:text="Send" />

</LinearLayout>
```




When the app runs, the views in the layout get inflated into objects. The linear layout is inflated to a `LinearLayout` object, the editable text views are inflated to `EditText` objects, and the button is inflated to a `Button`:




Now that you know how layout inflation works, let's find out how to use a new type of layout: a **frame layout**.

# A frame layout stacks its views

As you already know, a linear layout arranges its views in a single row or column. Each view is allocated its own space on the screen, and they don't overlap one another.

Sometimes, however, you *want* your views to overlap. As an example, suppose you want to display an image with some text overlaid on top of it. You wouldn't be able to achieve this just using a linear layout.

If you want a layout whose views can overlap, a simple option is to use a frame layout. Instead of displaying its views in a single row or column, it stacks them, one on top of another. It's often used to hold just a single view.

We'll examine how frame layouts work by creating a useful app which displays some text on top of a duck image. We'll start by creating a new project.

We'll use a frame layout to stack a text view on top of a duck image.

## Create a new project

Create a new Android Studio project using the same steps you used earlier. Choose the Empty Activity option, enter a name of "Frame Layout Example" and a package name of "com.hfad.framelayoutexample", and accept the default save location. Make sure the language is set to Kotlin, and that the minimum SDK is API 21 so that it will run on most Android devices.

Now that we've created the project, let's define a frame layout.

## How to define a frame layout

You define a frame layout using the `<FrameLayout>` element like this:

You use <FrameLayout> to define a frame layout.

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

</FrameLayout>
```

These are the same attributes we used for the linear layout. They are required for all layouts and views.

Just like a linear layout and any other type of view or view group, the `android:layout_width` and `android:layout_height` attributes specify the layout's width and height, and are mandatory. You can optionally add `padding` attributes too, although we've not done so here.

The above code creates an empty frame layout, so let's go ahead and add a duck image to it.

Linear layout
**Frame layout**
Scroll view

# Add an image to your project

We're going to display an image called *duck.webp* in the frame layout, but first, we need to add the file to the project.

To do this, you need to create a *drawable* resource folder (if Android Studio hasn't already created one for you). This is the default folder for storing image resources in your app. Switch to the Project view of Android Studio's explorer, select the *app/src/main/res* folder, go to the File menu, choose the New… option, then click on the option to create a new Android resource directory. When prompted, choose a resource type of "drawable", name the folder "drawable", and click on OK.

Next, download the file *duck.webp* from *tinyurl.com/hfad3*, then add it to the *app/src/main/res/drawable* folder.

We're going to display *duck.webp* in an image view (a view that displays an image) that we'll add to the frame layout. An image view is defined using the `<ImageView>` element like this:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout ...>
    <ImageView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:scaleType="centerCrop"
        android:src="@drawable/duck"
        android:contentDescription="Duck image" />
</FrameLayout>
```

Add a view that holds an image.

Use the image named "duck" that's in the drawable folder.

Crop the image's edges so it fits in the available space.

Add a content description for accessibility. In a real-world duck app, you'd want to add this text as a String resource.

The `<ImageView>` element includes the `android:layout_width` and `android:layout_height` attributes, which you're already familiar with, along with three new ones.

The `android:src` attribute specifies what image should be displayed in the image view. We've set this to `"@drawable/duck"` so that it uses *duck.webp* in the *drawable* folder.

The `android:contentDescription` attribute provides a text description of the image for accessibility.

Finally, the `android:scaleType` attribute describes how you want to scale the image. We've used `"centerCrop"`, which crops the edges of the image.

That's everything that you need to know in order to display the duck image in a frame layout. Before we find out how to display some text on top of it, let's take a closer look at how you use image—or *drawable*—resources.

## Serious Coding

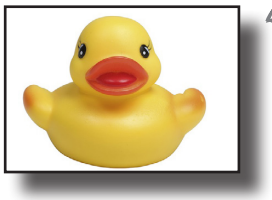
Android's preferred image format is **WebP**, which combines a small file size with a minimum loss of quality.

You can convert images to WebP in Android Studio by right-clicking on the image, and choosing the "Convert to WebP" option.

# Drawable Resources Up Close

As you've just learned, you can add an image to your project by putting it in your app's *app/src/main/res/drawable* folder. Doing this includes the image in your project as a drawable resource, which you can display in your layout using code like this:

```
<ImageView
    android:id="@+id/image"
    android:layout_width="200dp"
    android:layout_height="100dp"
    android:src="@drawable/duck"
    android:contentDescription="Duck image" />
```



Adding duck.webp to the drawable folder includes it in your project as a drawable resource.

If you want, you can use different versions of the image depending on the screen density of the device. This allows you to display higher-resolution images on higher-density screens, and lower-resolution images on lower-density screens.

To do this, you first create different *drawable* folders in *app/src/main/res* for the different screen densities. Each folder has a suffix that relates to a specific screen density:

| | |
|---|---|
| `drawable-ldpi` | Low-density screens, around 120 dpi. |
| `drawable-mdpi` | Medium-density screens, around 160 dpi. |
| `drawable-hdpi` | High-density screens, around 240 dpi. |
| `drawable-xhdpi` | Extra-high-density screens, around 320 dpi. |
| `drawable-xxhdpi` | Extra-extra-high-density screens, around 480 dpi. |
| `drawable-xxxhdpi` | Extra-extra-extra high-density screens, around 640 dpi. |
| `drawable-nodpi` | Images in this folder can be used by screens of all densities, but the image isn't scaled. |

Depending on what version of Android Studio you're running, the IDE may create some of these folders for you automatically.

You then add the different resolution images to each of the *drawable* folders, making sure that each of the image files has the same name. Android decides which version of the image to use at runtime, depending on the screen density of the device it's running on. If the device has an extra-high-density screen, for example, it will use the image located in the *drawable-xhdpi* folder.

If an image is added to just one of the folders, Android will use the same image file for all devices. If you want your app to use the same image regardless of screen density, you'd normally put it in the *drawable* folder.

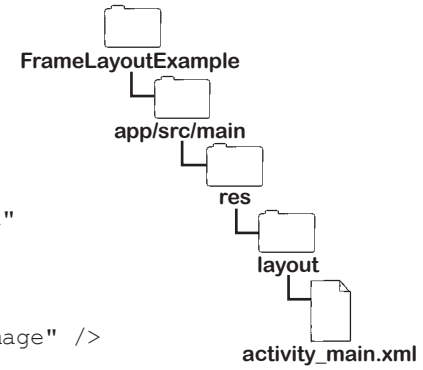
Linear layout
**Frame layout**
Scroll view

# A frame layout stacks views in the order they appear in the layout XML

When you define a frame layout, you list views in the order you want them to be stacked. The first view is displayed first, the second is stacked on top of it, and so on.

Here, we're going to display a text view on top of the image view, so we'll add it below the image view in the XML. Update the *activity_main.xml* code so that it matches the code shown here:

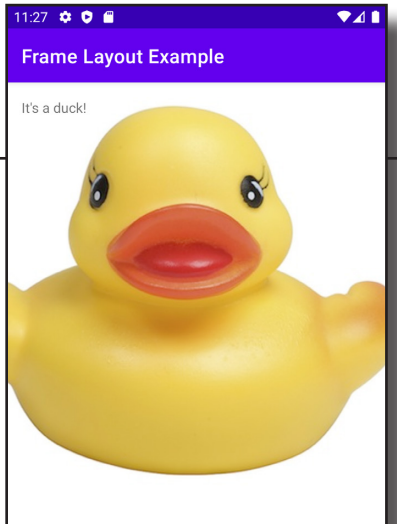
```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <ImageView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:scaleType="centerCrop"
        android:src="@drawable/duck"
        android:contentDescription="Duck image" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        android:text="It's a duck!" />
</FrameLayout>
```




Add this text view.

## Test Drive

When we run the app, an image of a duck appears on the device with the text "It's a duck!" in the top corner.



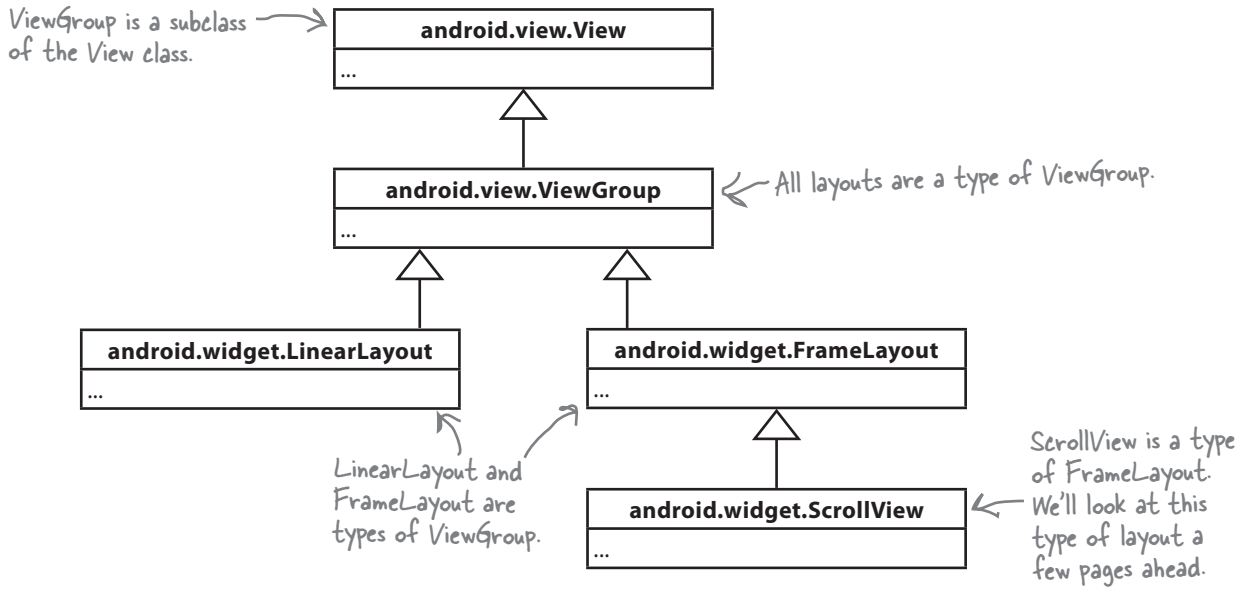

Notice how the "It's a duck!" text has been stacked on top of the image.

- Linear layout
- **Frame layout**
- Scroll view

# All layouts are a type of ViewGroup...

Even though they display their views in different ways, you may have noticed that linear layouts and frame layouts have a lot in common with one another. They can each hold views, for example, and they also have their own policy that determines how views are displayed.

There's a good reason for this commonality. Behind the scenes, all layouts—including linear layouts and frame layouts—are subclasses of the `android.view.ViewGroup` superclass:




## ...and a ViewGroup is a type of View

The `ViewGroup` class is a subclass of `View` that can hold other views. Behind the scenes, every layout is a subclass of `ViewGroup`, which means that **every layout is also a type of `View`**.

This class hierarchy means that all views and layouts share common attributes and behavior. They can all be displayed on the screen, for instance, and you get to say how tall or wide they should be. It's why you need to specify values for each view and layout's `android:layout_height` and `android:layout_width` attributes. These attributes are mandatory for all views, and as a layout is a type of view, they are also mandatory for all layouts.

**Every UI component you add to a layout is a type of View: an object that takes up space on the screen.**

**Every layout is a type of ViewGroup: a type of View that can contain other Views.**

Linear layout
**Frame layout**
Scroll view

### She's right.

As you already know, a layout is a type of view that can hold other views. As every layout is also a type of view, this means that **layouts can hold other layouts**.

Being able to nest layouts inside other layouts is useful because it lets you design more complex user interfaces. You can, for example, add horizontal rows to a vertical linear layout by nesting horizontal linear layouts inside the root linear layout. And if you want to arrange text vertically on top of an image, you can do so by nesting a vertical linear layout inside a frame layout.

Let's take a closer look at how this works using a new type of layout: a scroll view.

## there are no Dumb Questions

**Q: You said earlier that I can display an image in my layout using an image view. What if I want to add an image to a button? Can I do that?**

A: Yes. Buttons can include attributes such as `android:drawableStart` and `android:drawableEnd`, which allow you to position an image at the start or end of the button. The following code, for example, adds a duck image at the start of a button:

```
android:drawableStart="@drawable/duck"
```

Android also includes an image button view: a button with an image but no text. We're not covering image buttons in this book, but if it's something you think you might want to do, you can find out more here:

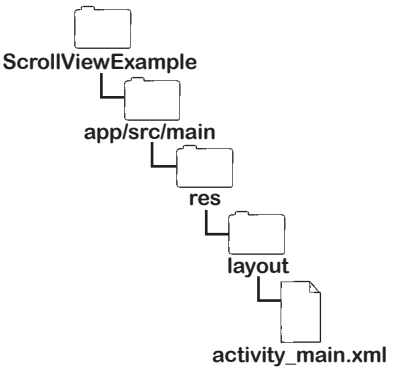
*https://developer.android.com/guide/topics/ui/controls/button*

**Linear layout**
**Frame layout**
**Scroll view**

# A scroll view inserts a vertical scrollbar

A **scroll view** is a type of frame layout that has a vertical scrollbar. It's useful for layouts that are too big for the device they're running on, as it means that you can use it to scroll to any views that don't fit on the screen.

We're going to create an app that uses a scroll view. Create a new project named "Scroll View Example", choosing the Empty Activity option. 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 replace the code in *activity_main.xml* with the code below:




```
<?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:orientation="vertical"
    android:padding="16dp"
    tools:context=".MainActivity" >

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Message"
        android:inputType="textMultiLine" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Send" />
</LinearLayout>
```

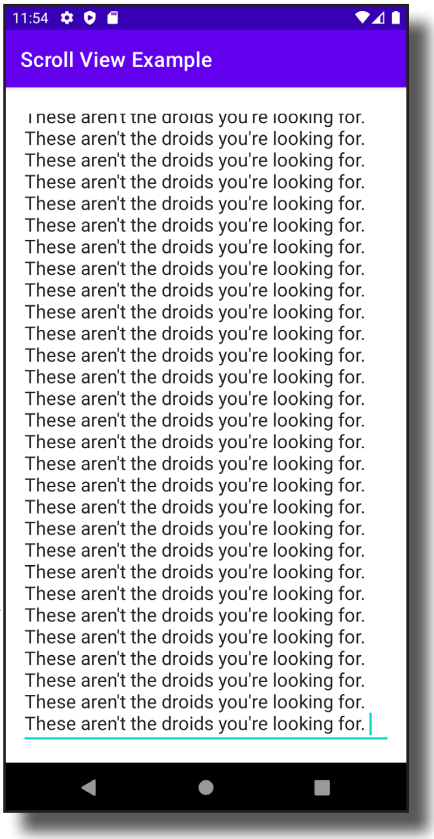
Make sure the activity_main.xml code matches the code shown here.

Set this to "textMultiLine", or you might not be able to enter multiple lines of text in the app.

As you can see, the above layout consists of a simple linear layout that contains a Message edit text and a Send button.

Try running the app, and type lots of text into the edit text. The view expands to fit the content, and it eventually pushes the Send button off the edge of the screen. As there's no scrollbar, you can no longer access the button.

To fix this problem, we'll add a scroll view to the layout. This will give us a scrollbar we can use to access the Send button if it's not visible on the screen.




The edit text includes so much text that it's pushed the button off the edge of the screen.

# How to add a scroll view

Linear layout
Frame layout
**Scroll view**

You add a scroll view to your layout using the **<ScrollView>** element. The <ScrollView> element is used in the same way as <FrameLayout> except that it includes an extra attribute, **fillViewport**, which is used to specify whether the scroll view should fill the device screen.

ScrollView is a subclass of FrameLayout, which means it can do everything a FrameLayout can, and more.

In the code below, we've added a scroll view to the layout code so that it encloses the original linear layout. Update the *activity_main.xml* code to include the changes shown here (in bold):

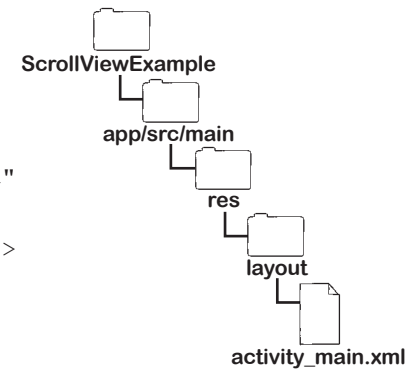
```
<?xml version="1.0" encoding="utf-8"?>
<~~LinearLayout~~ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    ~~android:orientation="vertical"~~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

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:padding="16dp" >

        <EditText
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Message"
            android:inputType="textMultiLine" />

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Send" />
    </LinearLayout>
</ScrollView>
```

Replace the LinearLayout at the root of the layout with a ScrollView.

Delete these lines. The orientation attribute doesn't apply to scroll views, and we don't want to add padding to the scroll view as this will indent the vertical scrollbar.

Set the fillViewport attribute to "true" so that it fills the device screen.

Move the original linear layout so that it's nested inside the scroll view.




Make sure your code still includes this line.

Add a closing tag to the <ScrollView> element.

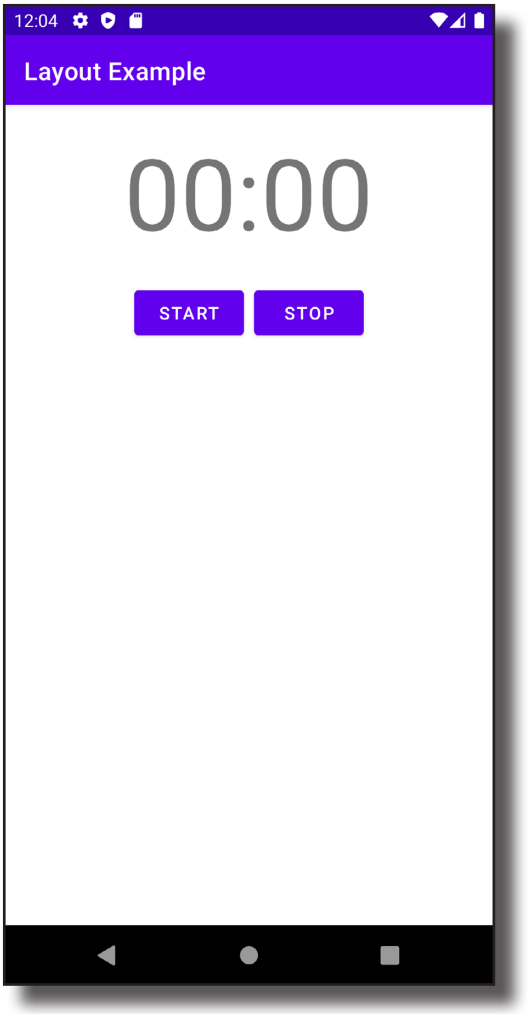
We'll take the code for a test drive after you've had a go at the following exercise.

## Sharpen your pencil

As you've learned, you can nest layouts inside one another to produce more complex layouts. Use this knowledge to say or draw how you might produce the layout shown below.

We're not expecting you to write all the code for the layout, just say how you might construct it.

The layout features a text view with two buttons underneath it.



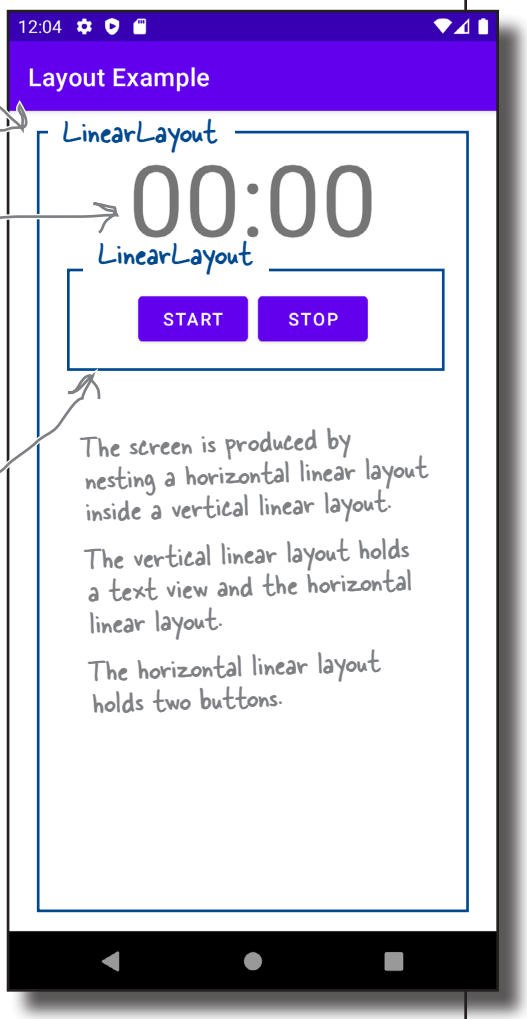
# Sharpen your pencil Solution

As you've learned, you can nest layouts inside one another to produce more complex layouts. Use this knowledge to say or draw how you might produce the layout shown below.

We're not expecting you to write all the code for the layout, just say how you might construct it.

```
<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:orientation="vertical"
    android:padding="16dp"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:textSize="80sp"
        android:text="00:00" />

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:layout_marginTop="20dp"
        android:layout_gravity="center">

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginHorizontal="4dp"
            android:text="Start" />

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginHorizontal="4dp"
            android:text="Stop" />
    </LinearLayout>
</LinearLayout>
```

We've used the textSize attribute to increase the size of the text.

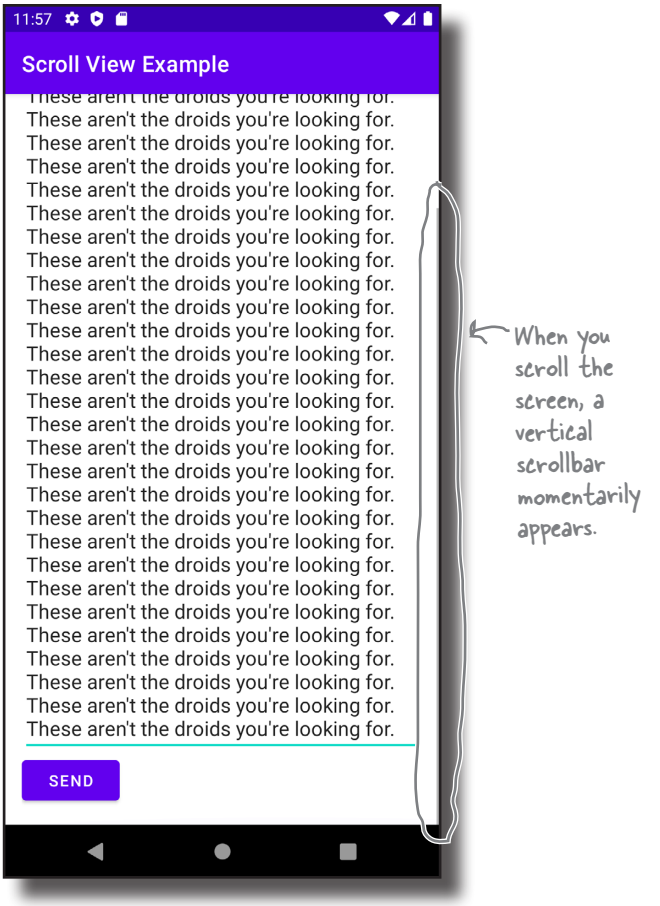


# Test Drive

Once you've added a scroll view to your layout code, go ahead and run the app.

When you enter lots of text into the Message edit text, the Send button gets pushed off the edge of the screen as before. This time, however, we can scroll the device screen to reach the button.

Congratulations! You've now learned how to use different types of layout to control the appearance of your app's UI. You'll build on this knowledge in the next chapter.




All layouts (including linear layouts, frame layouts, and scroll views) can hold other layouts.

You can build complex layouts by nesting one type of layout inside another.

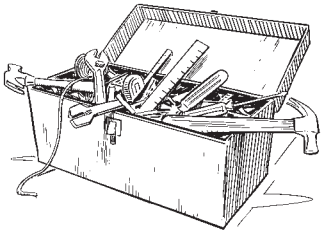
## there are no Dumb Questions

**Q: Does a scroll view just add a vertical scrollbar? What if I want to include a horizontal scrollbar?**

A: Yes, `<ScrollView>` only adds a vertical scrollbar, not a horizontal one.

If you want to add a horizontal scrollbar to your layout, you can do so using the `<HorizontalScrollView>` element instead. A horizontal scroll view works in the same way as a scroll view, except that it adds a horizontal scrollbar instead of a vertical one.

# Your Android Toolbox



**You've got Chapter 3 under your belt and now you've added building layouts to your toolbox.**

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

## Bullet Points

- `<EditText>` defines an editable text view.
- Use `<ImageView>` to display an image.
- A linear layout lists views either horizontally or vertically. You specify the direction using the `android:orientation` attribute.
- In a linear layout, use `android:layout_weight` if you want a view to use up extra space in the layout.
- Use `android:padding` attributes to specify how much padding you want between a view's edges and its contents.
- `android:layout_gravity` lets you say where you want views to appear in their available space.
- `android:gravity` lets you say where you want the contents to appear inside a view.
- Use `android:layout_margin` to add extra space around a view.
- A frame layout stacks views, one on top of another.
- Add scrollbars using a scroll view or a horizontal scroll view. These are both types of frame layout.
- All layouts are subclasses of the `android.view.ViewGroup` class. A view group is a type of view that can contain multiple views.
- Layouts can include other layouts.

# 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**. Let's get designing…

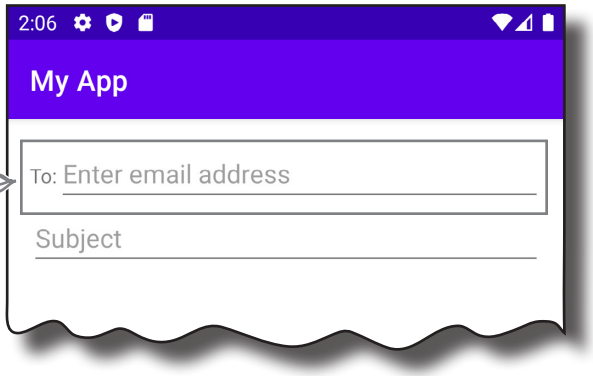
# Nested layouts revisited

In the previous chapter, you learned that you can nest layouts to build more complex screens. The following code, for example, uses nested linear layouts to display a text view and edit text in a horizontal row, with an edit text below them:

```
<?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:orientation="vertical"
    android:padding="16dp"
    tools:context=".MainActivity">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="To:" />

        <EditText
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:hint="Enter email address"
            android:inputType="text" />
    </LinearLayout>

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Subject"
        android:inputType="text" />
</LinearLayout>
```

We're using a linear layout that's oriented vertically.

A nested linear layout arranges a text view and an edit text in a horizontal row.



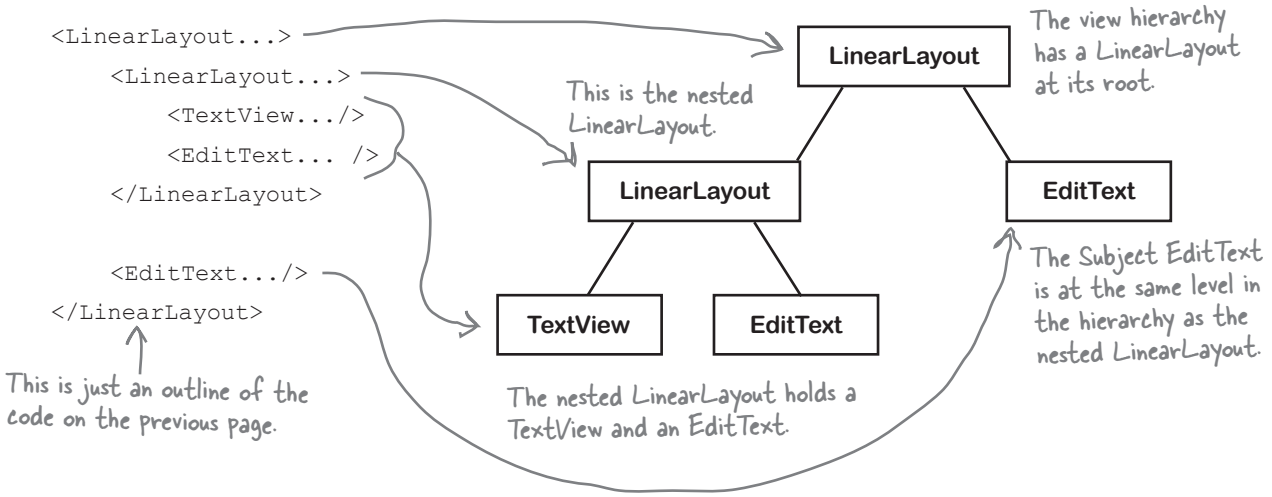

The text view is just wide enough for its contents. The edit text uses layout weight to take up any remaining horizontal space.

The Subject edit text appears below the nested linear layout.

# Nesting layouts comes at a price

The downside to nested layouts is that building complex layouts in this way can be inefficient, make your code harder to read and maintain, and can also slow down your app.

When Android displays a layout on the device screen, it first checks the structure of the layout file, and uses this to build a hierarchy of views. For the nested layout shown on the previous page, for example, it builds a hierarchy of views that contains two linear layouts, two edit texts, and a text view:




Android uses the view hierarchy to help it figure out where each view should be placed on the device screen. Each view needs to be measured, laid out, and drawn on the screen, and Android needs to make sure each view has enough space for its contents, and that any weights are taken into account.

If the layout contains nested layouts, the view hierarchy is more complex, and Android may need to make multiple passes in order to figure out how views need to be arranged. If the layouts are deeply nested, this can lead to bottlenecks in your code, and can leave you with a mass of code that's difficult to read and maintain.

If you have a more complex UI like this, an alternative to using nested layouts is to use a **constraint layout**.

**Each view in the layout needs to be initialized, measured, laid out, and drawn. In deeply nested layouts, this can slow down your app.**

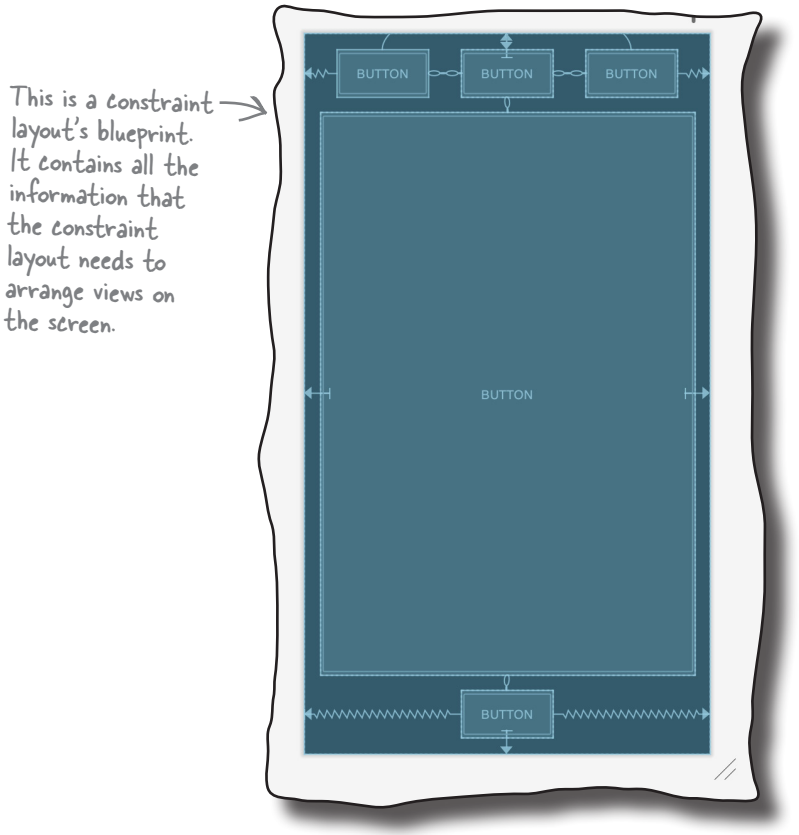
# Introducing the constraint layout

A constraint layout is more complex than a linear or frame layout, but it's way more flexible. It's also much more efficient for complex UIs as it gives you a flatter view hierarchy, which means that Android has less processing to do at runtime.

**Use a constraint layout to build flexible UIs without nesting layouts.**

## You design constraint layouts VISUALLY

Another advantage of using constraint layouts is that they're specifically designed to work with Android Studio's design editor. Unlike linear and frame layouts where you usually hack directly in XML, you build constraint layouts *visually*. You drag and drop views into the design editor's blueprint, and give it instructions for how each view should be displayed:

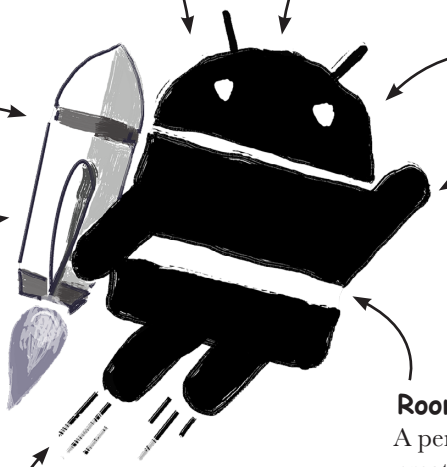



Unlike linear and frame layouts, constraint layouts are part of a suite of libraries known as **Android Jetpack**. You may have already heard something about Jetpack, but what is it?

# Constraint layouts are part of Android Jetpack

Android Jetpack is a collection of libraries that help you follow best practice, reduce boilerplate code, and make your coding life easier. It includes constraint layouts, navigation, the Room persistence library (which helps you build databases) and lots, lots more.

Here are some of our favorite Jetpack components; you'll find out how to use them in later chapters:



**Fragment**
A fragment is like a kind of subactivity that's used to control part of the screen.

**View Model**
This lets you move a screen's business logic into a separate class.

**Navigation**
This includes tools to navigate from screen to screen, and safely pass arguments between them.

**Constraint layout**
Use this to build complex, flexible layouts without the overheads of nesting.

**LiveData**
This lets you build apps that respond to data changes as soon as they happen.

**Recycler view**
An efficient list you can use to display data, and navigate between screens.

**Data binding**
This lets you build responsive layouts that can access Kotlin methods and properties.

**Room**
A persistence library that lets you create and interact with databases.

**Compose**
A toolkit for building native Android apps without using layout files.

Another great thing about Jetpack is that it lets you write code that works consistently across new and old versions of Android. This is great news for your users, as it means that you can include exciting new Android features that will work on older devices.

An example of this is `AppCompatActivity`, which you've already been using to write activity code. We didn't mention it earlier, but `AppCompatActivity` is part of Android Jetpack. It adds new features to activities across new and old versions of Android without you having to worry about backward compatibility.

Yes! You've already been using part of Android Jetpack without knowing it. You'll learn more about using Jetpack through the rest of this book.

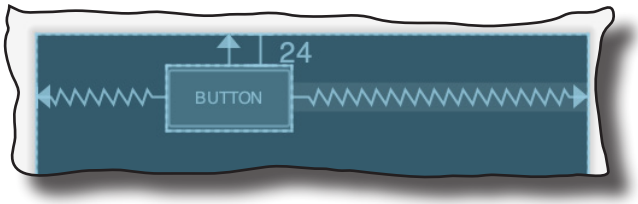
In this chapter, you're going to learn how to use constraint layouts. Let's run through what we're going to do.

# Here's what we're going to do

We're going to break learning about constraint layouts into two main parts:

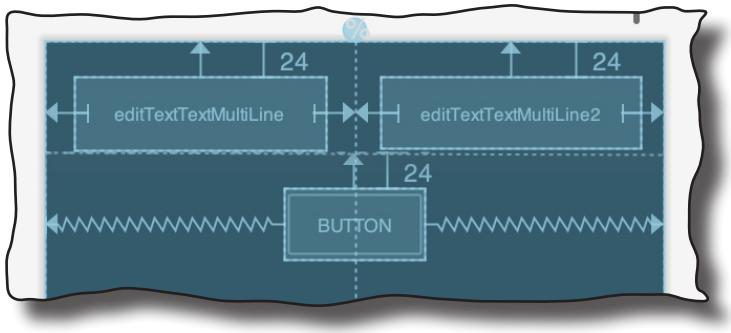
1. **How to position and size a single view.**
   You'll learn how to use constraints and bias to control how and where a single view is shown in its layout.

These are constraints, and you'll learn how to use them in the first step.




2. **How to position and size multiple views.**
   You'll then apply your knowledge to multiple views, and learn more advanced techniques using guidelines, barriers, chains, and flows.

You'll find out how to build more complex constraint layouts like this in the second step.




## Create a new project

We're going to use a new project for the app we're going to build, so create one now using the same steps you used in the previous chapters. Choose the Empty Activity option, enter a name of "My Constraint Layout" and a package name of "com.hfad.myconstraintlayout", and accept the default save location. Make sure the language is set to Kotlin and the minimum SDK is API 21 so it will run on most Android devices.

Now that we've created the project, let's make sure that it has been set up to use constraint layouts.

# Use Gradle to include Jetpack libraries

To make sure that all the Jetpack libraries—including constraint layouts—work across all versions of Android, they're not included in the main Android SDK. Instead, you have to add any libraries you need using **Gradle**. This is a build tool that's used to compile code, configure apps, and fetch any extra libraries that your project requires.

Every time you create a new project, Android Studio creates two Gradle files named ***build.gradle***.

The first version of *build.gradle* lives in the *project* folder, and specifies the basic settings of your app, such as what version of the Gradle plug-in to use.

The second version of *build.gradle* lives in the project's *app* folder. It's where the majority of the app's properties are set, such as the API level.

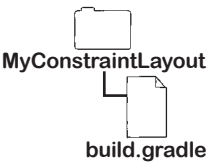
**Behind the scenes, every Android Studio project uses Gradle as its build tool.**

## The project build.gradle needs a Google repository line

Every project needs to know where to find any extra Jetpack libraries it needs, and this is done by adding a reference to the Google repository in the **project**'s *build.gradle* file. Android Studio usually does this for you, but you can make sure it's there by opening the file *MyConstraintLayout/build.gradle*, and looking for the following line (in bold) in the `repositories` section under `allprojects`:

```
allprojects {
    repositories {
        google()
        ...
    }
}
```

All Jetpack libraries come from this repository. Android Studio should add this line for you.

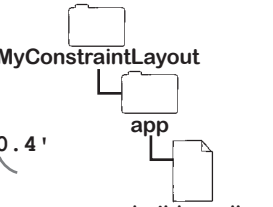



## The app build.gradle includes the constraint layout's library

To use constraint layouts, a reference to its library needs to be included in the **app**'s *build.gradle* file. Android Studio should have already added this for you, but you can double-check by opening the file *MyConstraintLayout/app/build.gradle*, and looking for the following line (in bold) in the `dependencies` section:

```
dependencies {
    implementation 'androidx.constraintlayout:constraintlayout:2.0.4'
    ...
}
```

We're using this version of the constraint layout library.

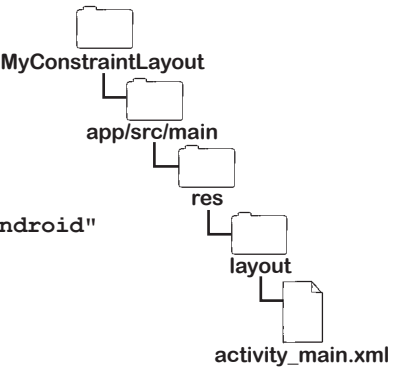



If the file doesn't include this line, add it now, and click on the Sync Now option that appears in the code editor. This syncs any changes you've made with the rest of your project, and adds the library.

# Let's add a constraint layout to activity_main.xml

Now that your project is all set up to use constraint layouts, let's start using one.

You add a constraint layout to a layout file using an **<androidx.constraintlayout.widget.ConstraintLayout>** element. We're going to use one in the layout file *activity_main.xml*, so open this file in the *app/src/main/res/layout* folder, and make sure its code looks like this:




```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

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

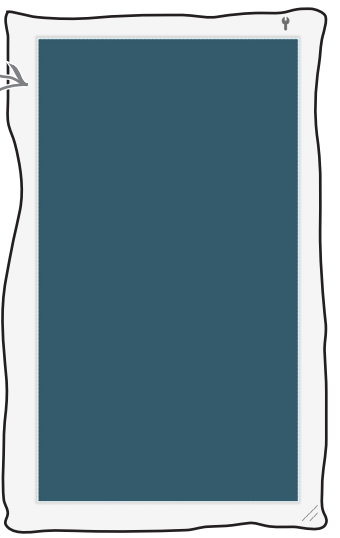
This is how you define a constraint layout.

If Android Studio added any extra views to the layout, delete them.

## Show the layout in the blueprint

We're going to add views to the layout using the design editor's blueprint. Switch to the design editor by clicking on the Design option, click on the Select Design Surface button in the editor's toolbar, and select the Blueprint option. This shows you a blueprint of the layout like this:

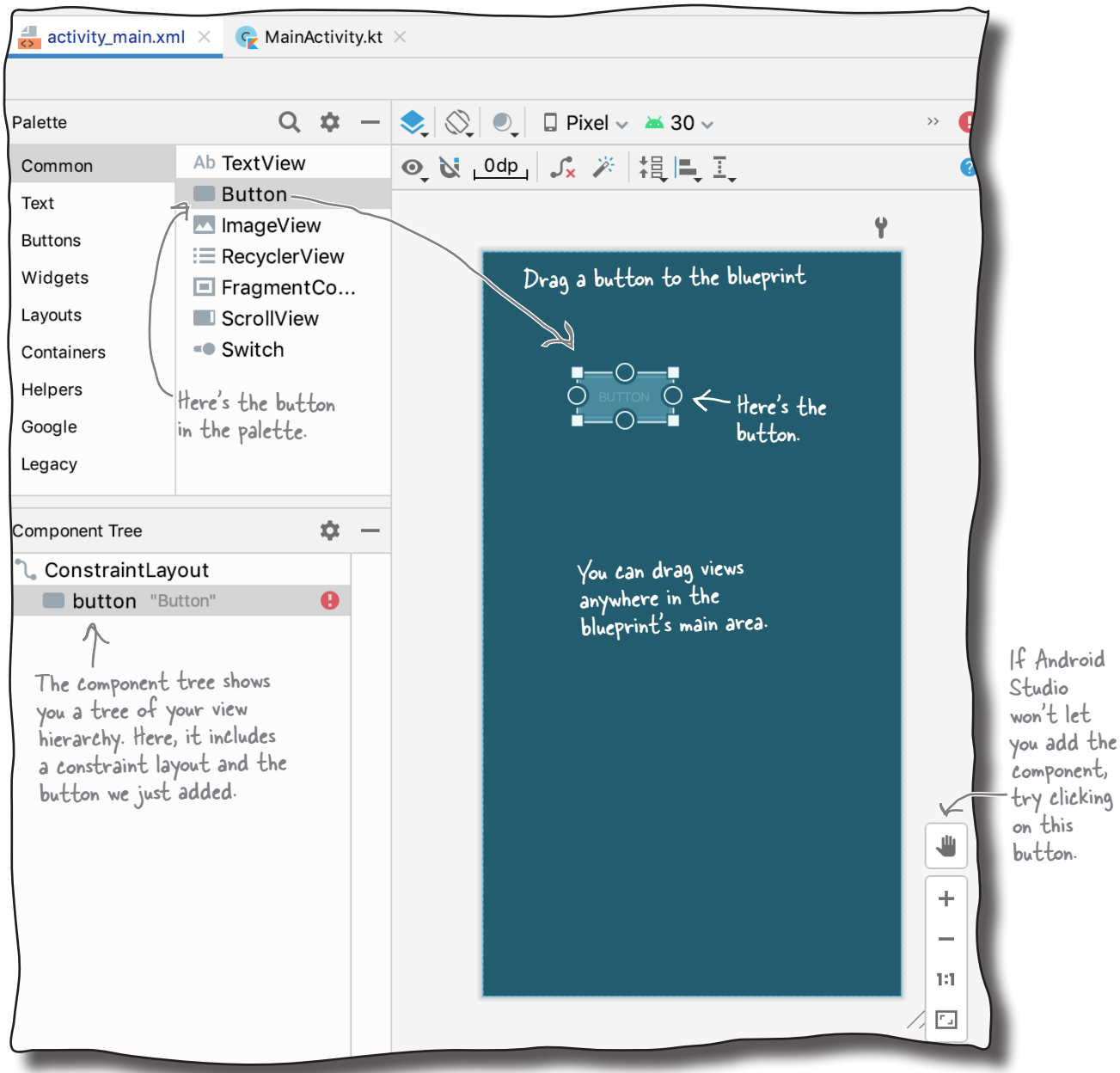
This is the Select Design Surface button. Use it to see the Blueprint view of your layout, the design view, or both.

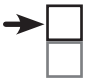


This is the layout's blueprint. It looks empty because we haven't added any views to the constraint layout yet.

# Add a button to the blueprint

We're going to add a button to the layout. To do this, go to the design editor's palette, find the Button component (it's usually in the Common section), and drag it to the blueprint. You can place the button anywhere in the blueprint, just so long as it appears in its main area like this:





**Single views**
Multiple views

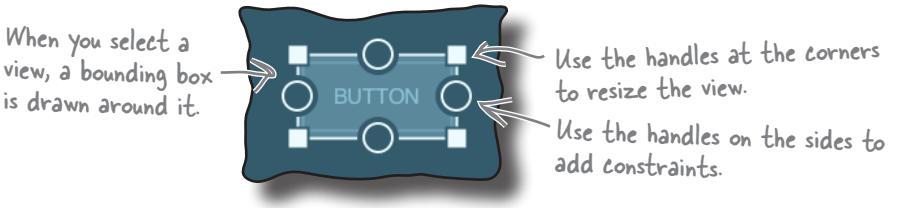
# Position views using constraints

With a constraint layout, you don't specify where views should be positioned by dropping them on the blueprint in a particular place. Instead, you specify placement by defining **constraints**. A **constraint** is a connection or attachment that tells the layout where the view should be positioned. You can use a constraint to attach a view to the start edge of the layout, for example, or underneath another view.

## We'll add a horizontal constraint to the button

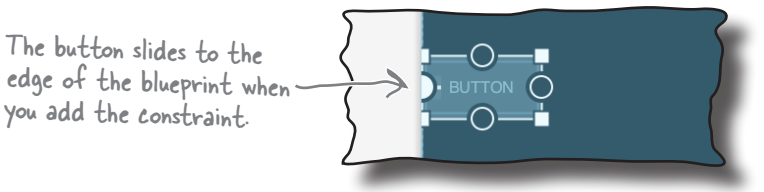
To see how this works, let's add a constraint to attach the button to the left edge of the layout.

First, make sure the button's selected by clicking it. When you select a view, a bounding box is drawn around it, and handles are added to its corners and sides. The handles in the corners let you resize the view, and the handles on the sides let you add constraints:




To add a constraint, you click on one of the view's constraint handles and drag it to whatever you want to attach it to. In this case, we're going to attach the left edge of the button to the left edge of the layout, so click on the left constraint handle and drag it to the left edge of the blueprint:

This adds the constraint, and pulls the button over to the left:

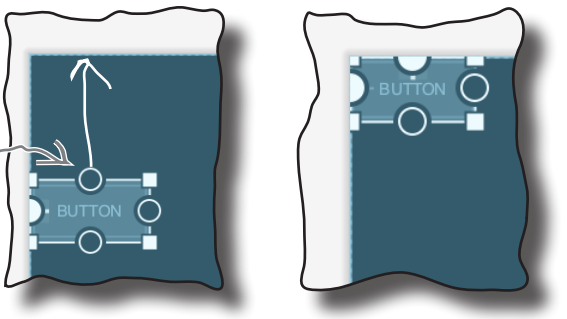



That's how you add a horizontal constraint. Let's see what happens when you add a vertical one.

# Add a vertical constraint too

We'll use a vertical constraint to attach the button to the top of the layout. To do this, click on the button's top constraint handle, and drag it to the top of the blueprint. This adds the vertical constraint, which pulls the button up.

Click on the handle on the button's top edge, and drag it to the top of the blueprint. The button slides to the top of the blueprint.



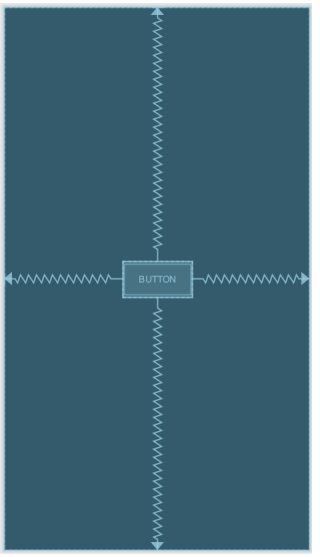
## Exercise

You've seen what happens when you add constraints to the button's left and top edges. What do you think happens when you add opposing constraints to the button's right and bottom edges? Try doing this now and draw your answers below.

# Exercise Solution

You've seen what happens when you add constraints to the button's left and top edges. What do you think happens when you add opposing constraints to the button's right and bottom edges as well? Try doing this now, and draw your answers below.

When you add a constraint that connects the button and blueprint's right edges, the left and right constraints pull the button in opposite directions. The button is centered horizontally.

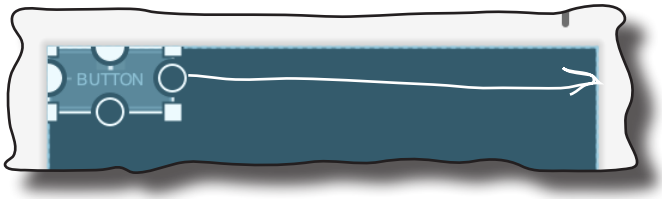



When you add a constraint that connects the button and blueprint's bottom edges, the top and bottom constraints pull the button so that it's centered in the middle of the layout.

# Use opposing constraints to center views

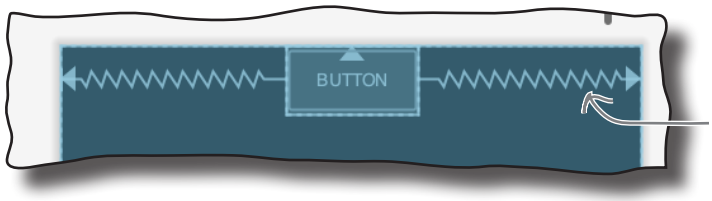
As you've learned, you can use constraints to attach a view to the edge of the blueprint. Each constraint works like a spring that pulls the view to the blueprint's edge.

If you want to position views in the center of the blueprint, you can do so by adding constraints to opposite sides of the view. To center a button horizontally, for example, you add one constraint that pulls the view to the left, and another that pulls it to the right like this:



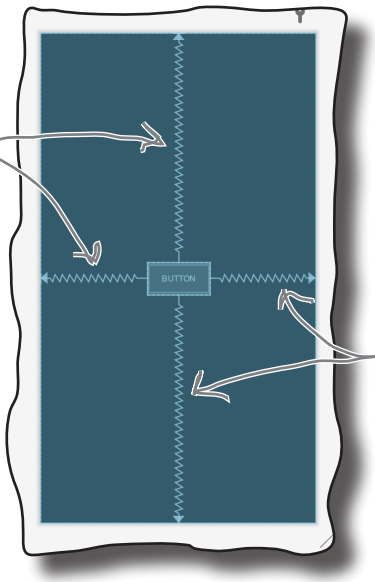

To add a constraint that pulls the button to the right, click on the constraint handle on the button's right edge, and drag it to the right edge of the blueprint.

The two constraints pull the button in opposite directions, which centers it horizontally like so:




Adding the opposing constraint pulls the button to the center.

You can also center a view vertically by adding constraints to its top and bottom edges. And if you wanted to center it horizontally and vertically, you'd add constraints to all four edges like this:




These constraints pull the button to the top and left edges of the blueprint.

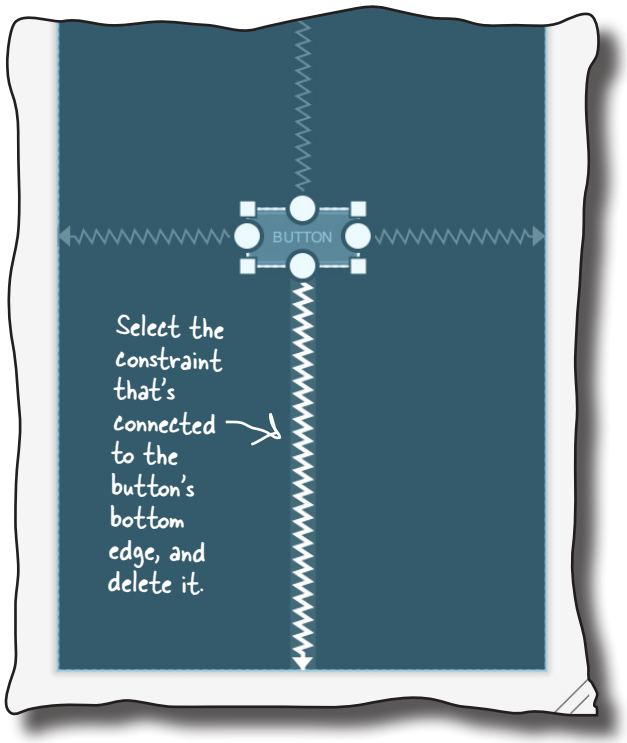
If you can't see all the constraints, click on the View Options button, and select the Select All Constraints option.

These constraints pull the button to the bottom and right edges of the blueprint.

**Single views**
Multiple views

# You can delete constraints you no longer need

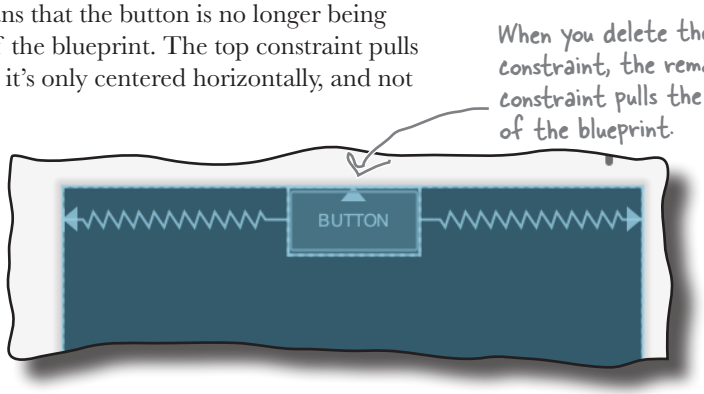
You can remove any constraints you no longer need by selecting them in the blueprint, and deleting them. If you have a button that's centered in the middle of the blueprint, for example, you delete the constraint that's attached to its bottom edge:



**Make sure each view has *at least* one vertical and horizontal constraint.**

*If the view has no vertical constraints, it will be displayed at the top of the layout when you run the app. If the view has no horizontal ones, it will be displayed next to its start edge.*

Deleting this constraint means that the button is no longer being pulled toward the bottom of the blueprint. The top constraint pulls the button to the top so that it's only centered horizontally, and not vertically:



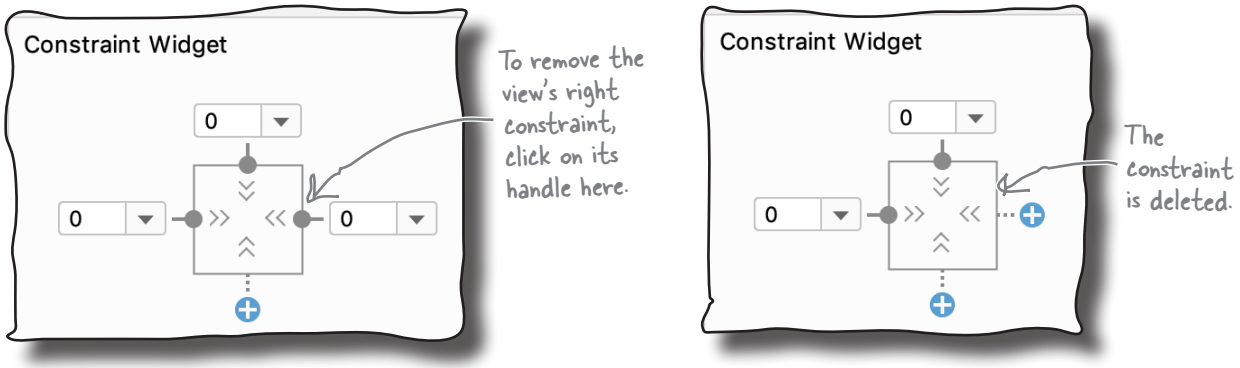

Another way of removing constraints you no longer need is to use the constraint widget tool. Let's see how this works.

# Remove constraints with the constraint widget

The constraint widget is displayed in the Attributes panel at the side of the design editor. It appears when you select a view, and displays a diagram featuring the view's constraints, and the size of any margins.

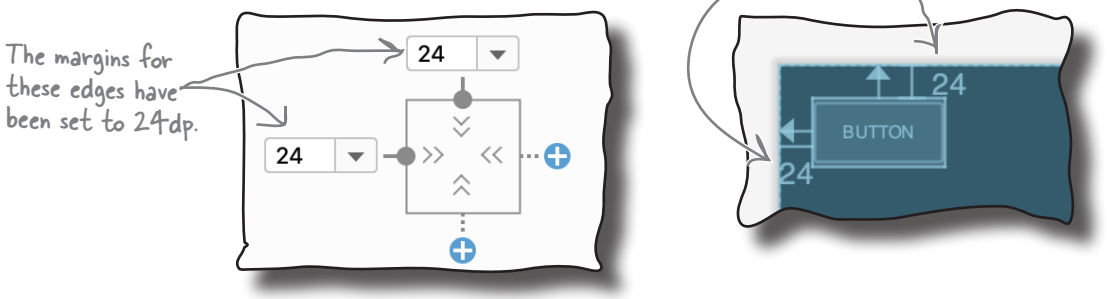
You'll find out more about the Attributes panel a few pages ahead.

To delete a constraint in the constraint widget, select the view in the blueprint you want to remove the constraint from, then click on the constraint's handle in the constraint widget. The constraint is removed, and the view is repositioned in the blueprint.

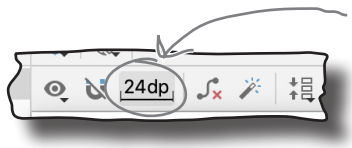



## You can use it to add margins too

You may have noticed that each of the constraints in the constraint widget has a number next to it. This is used to set the margin size for that edge of the view so that there's space between the view and the layout's edge. To change the size of a view's left and top margins to 24dp, for example, you'd update their values in the diagram to 24:




You can set a default size for any new margins using the Default Margins button in the design editor's toolbar. Setting this to 24dp, for example, means that any new constraints that get added will automatically include a margin of 24dp.




This the Default Margins option. Here, we've set a default margin of 24dp.

**Single views**
**Multiple views**

# Changes to the blueprint appear in the XML

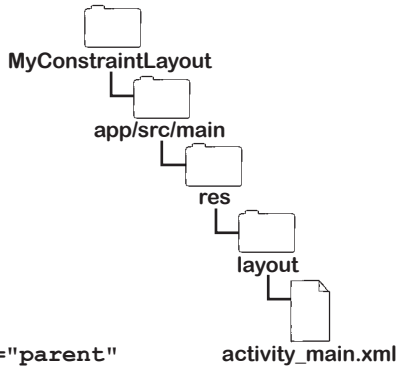
When you add views to a blueprint, and specify constraints and margins, they get added to the layout's underlying XML. To see this, switch to the layout's code view. Your code should look something like this (but don't worry if it's slightly different):

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    ...>
    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="24dp"
        android:layout_marginTop="24dp"
        android:text="Button"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Here's the button we added.

These are all attributes you've seen before.

These lines only apply to constraint layouts.




As you can see, the XML now includes a button. Does its code look familiar to you? If so, nice catch—it includes attributes that you learned about in Chapter 3.

The button's width, height, and margins are specified in exactly the same way as before, and if you want to, you can change their values in the XML instead of using the design editor.

The only unfamiliar code is the two lines that specify the view's constraints on its start and top edges:

```
app:layout_constraintStart_toStartOf="parent"
app:layout_constraintTop_toTopOf="parent" />
```

These lines describe the constraints on the button's left (or start) and top edges.

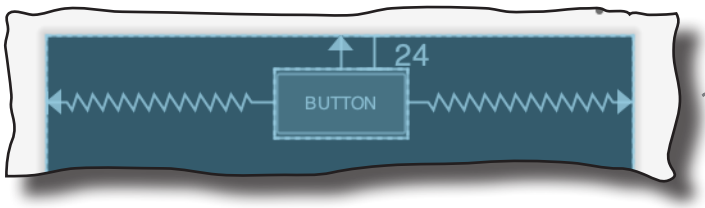
Similar code is generated if you add constraints to the button's remaining edges.

Now that you've had a glimpse of what constraint layout XML looks like switch back to the design editor, and we'll look at some more techniques you can use to position views.

# Views can have bias

As you learned earlier, you can add constraints to opposite sides of a view. This centers the view by default, but you can also control its position relative to each side by changing its **bias**. This tells Android what the proportionate length of each constraint should be on either side of the view.

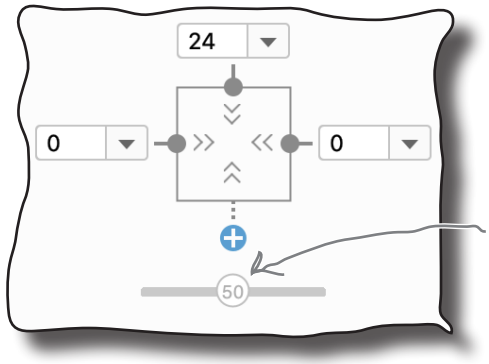
To see this in action, let's change the button's horizontal bias so that it's positioned off-center. First, make sure that the button includes constraints on its left and right sides like this:




Make sure the button includes left and right constraints so that it's centered horizontally.

Then select the button so the constraint widget is displayed.

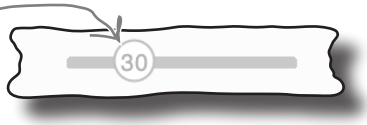
Underneath the widget's diagram of the view, you should see a slider with a number in it. This is a percentage of the view's horizontal bias.



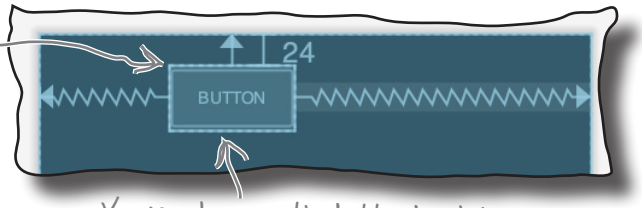

This slider is for the view's horizontal bias. It currently displays 50 as the view is positioned halfway between its horizontal constraints.

To change the bias, simply move the slider. If, say, you move the slider to the left so that the number changes to 30, it moves the button in the blueprint to the left as well:

Moving the slider...




...moves the button.




You can also move the button by clicking and dragging it, but that technique is less accurate.

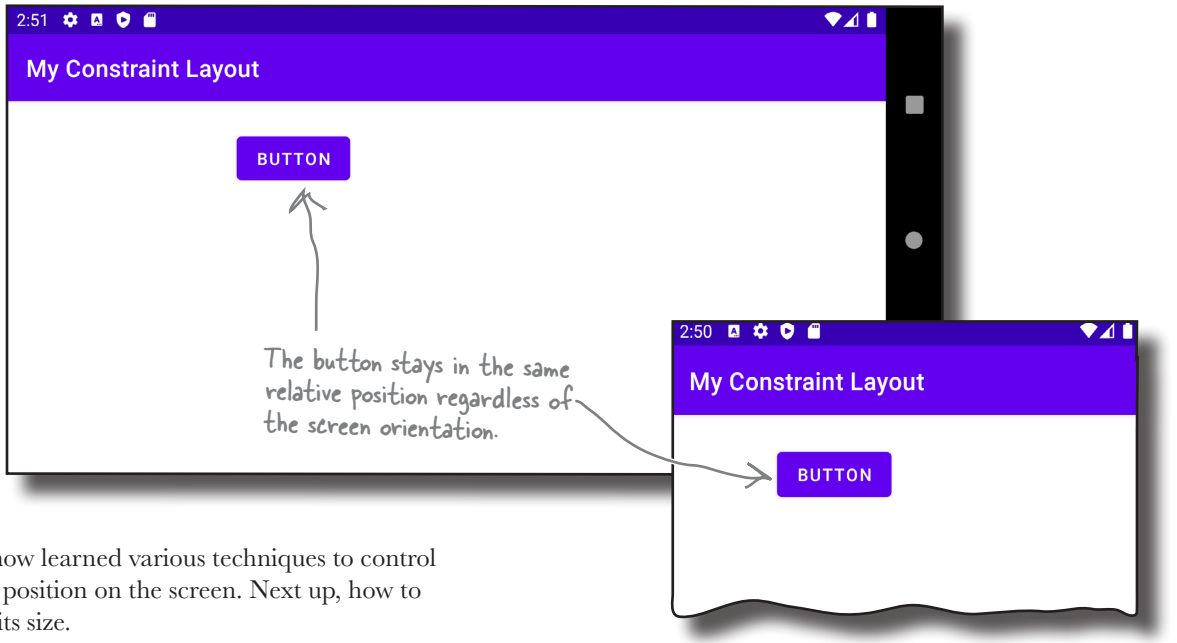
The view maintains this relative position irrespective of screen size and orientation. Let's try this out by taking the app for a test drive.

# Test Drive

When we run the app, a button appears off-center toward the top of the screen. It maintains the same relative position when we rotate the device.




If your screen doesn't rotate when you turn the device, make sure you have the auto-rotate setting switched on. You can find it under Settings → Display.




You've now learned various techniques to control a view's position on the screen. Next up, how to change its size.

## there are no Dumb Questions

**Q: You said that all the Jetpack components (including constraint layouts) have separate libraries that aren't part of the main Android release. Why is this?**

A: It's so you can use new versions and features of these components in older versions of Android.

**Q: When you added the constraint layout library to the app's *build.gradle* file, you said that you were using a particular version. Can I use a different one?**

A: All of the features and code we've used in this chapter work with that version of the constraints layout library. Libraries are always getting fresh updates, but bear in mind that using a different version might break some of the code.

**Q: How will I know if there's a new version?**

A: The *build.gradle* files usually spot when a new version is available, and let you know.

**Q: Is there a way of making the design editor add constraints for me automatically?**

A: There's an "Enable Autoconnection to Parent" button in the design editor toolbar, which you can use to automatically add constraints between each new view and the edge of the blueprint.

There's also an Infer Constraints button, which guesses what constraints are needed based on the position of views in the layout.

We've had mixed results with these options, but feel free to try them out.

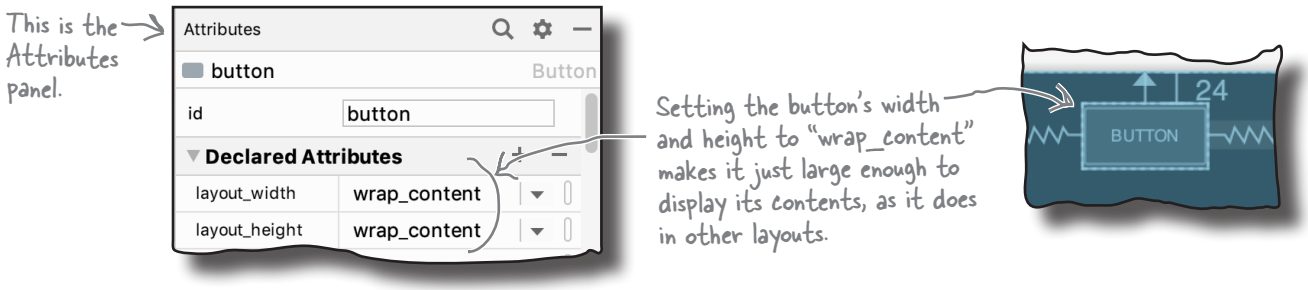
# You can change a view's size

As you might expect, you can change a view's size in a constraint layout by updating its `layout_width` and `layout_height` attributes. You can do this in the layout's XML, or in the design editor's Attributes panel.

The Attributes panel is displayed to the side of the blueprint. When you select a view, it shows you all the attributes that have already been declared (such as `layout_width` and `layout_height`), and lets you set ones that haven't.

## Make the view just big enough

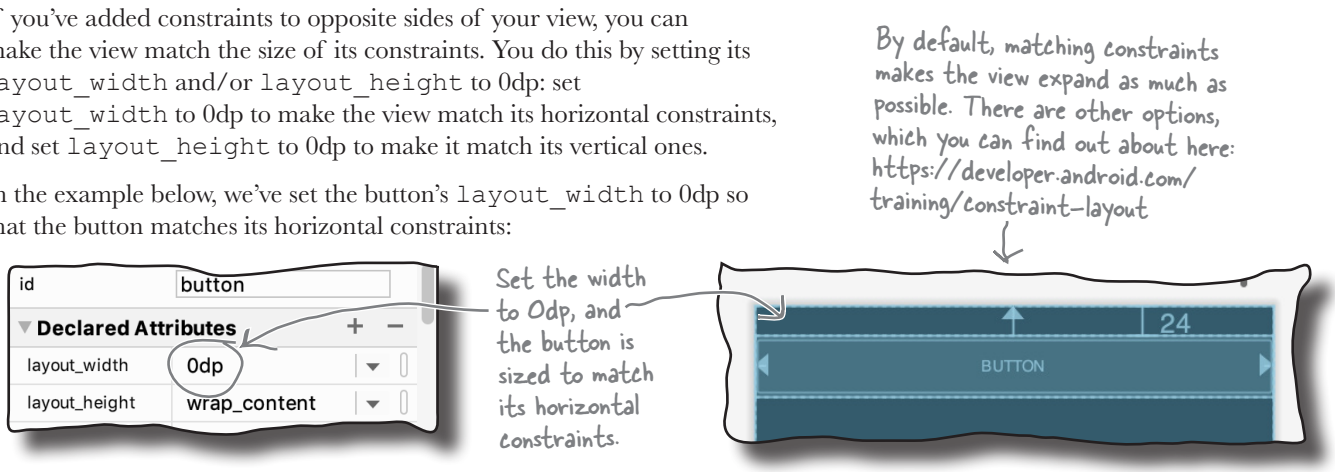
Just like with linear and frame layouts, you make a view just large enough to display its contents by setting its `layout_width` and `layout_height` properties to `wrap_content`. If the view is a button, for example, it makes the button just large enough to hold its text:




## Match the view's constraints

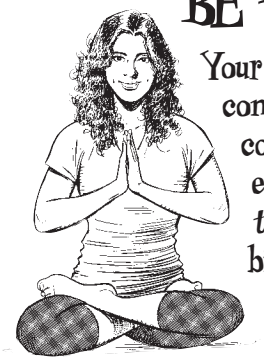
If you've added constraints to opposite sides of your view, you can make the view match the size of its constraints. You do this by setting its `layout_width` and/or `layout_height` to 0dp: set `layout_width` to 0dp to make the view match its horizontal constraints, and set `layout_height` to 0dp to make it match its vertical ones.

In the example below, we've set the button's `layout_width` to 0dp so that the button matches its horizontal constraints:




Now that you've seen how to resize a view, try experimenting with the different techniques, then have a go at the exercise on the next page.

# BE the Constraint

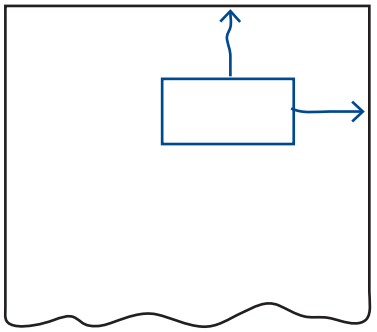


Your job is to play like you're the constraint layout and draw the constraints that are needed to produce each layout. You also need to specify the layout_width, layout_height, and bias (when needed) for each view. We've completed the first one for you.

You need to add the views and constraints to each blueprint.

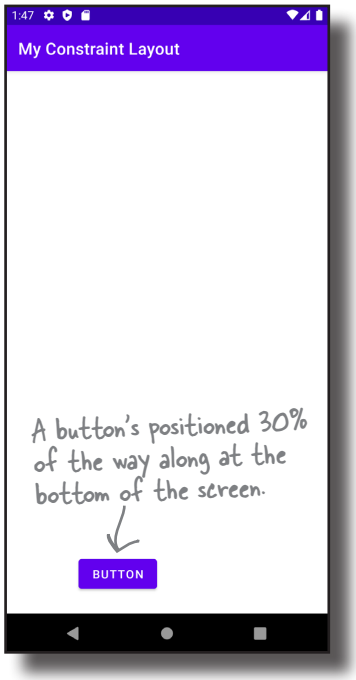
**A**

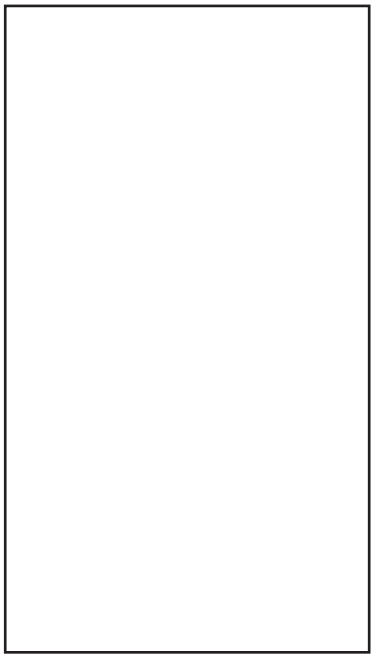
This is how we want the screen to look.



layout_width: wrap_content
layout_height: wrap_content

**B**

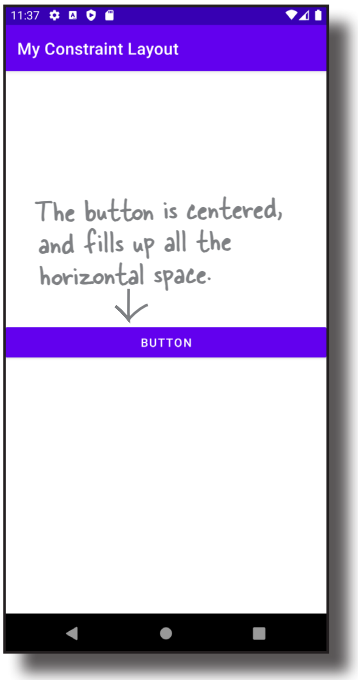


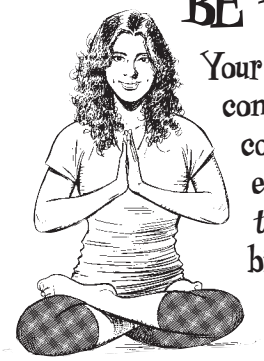
The button fills the available space.

C

D

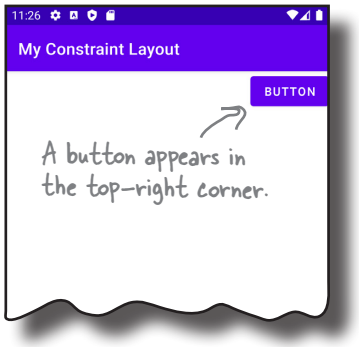


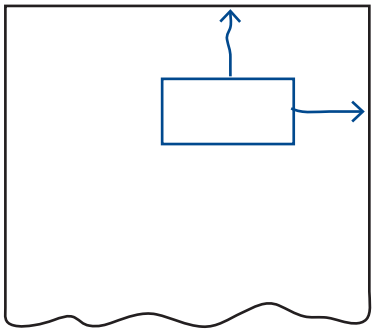
# BE the Constraint Solution



Your job is to play like you're the constraint layout and draw the constraints that are needed to produce each layout. You also need to specify the layout_width, layout_height, and bias (when needed) for each view. We've completed the first one for you.

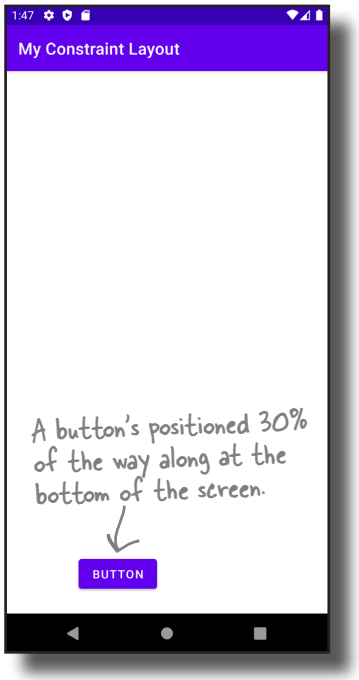
**A**

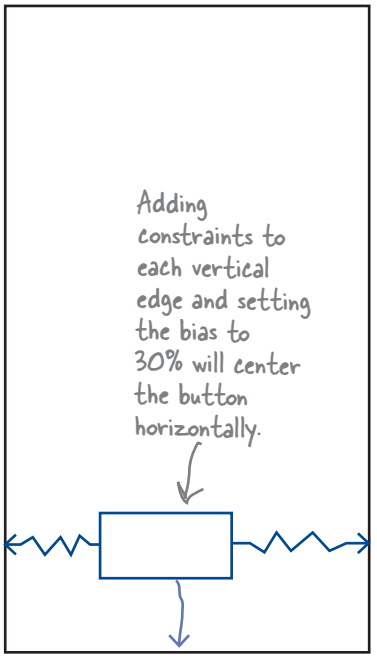


layout_width: wrap_content
layout_height: wrap_content

**B**

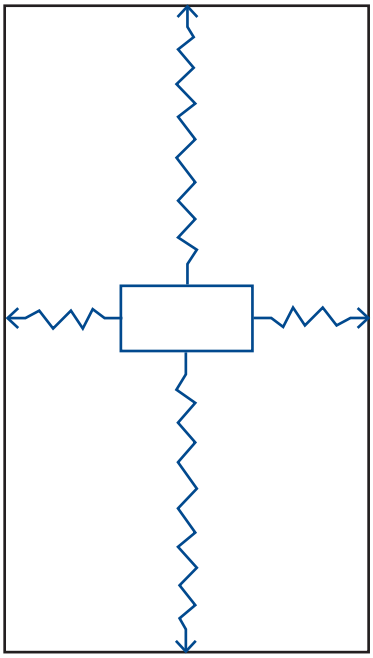



layout_width: wrap_content
layout_height: wrap_content
bias: 30%

The button fills the available space.

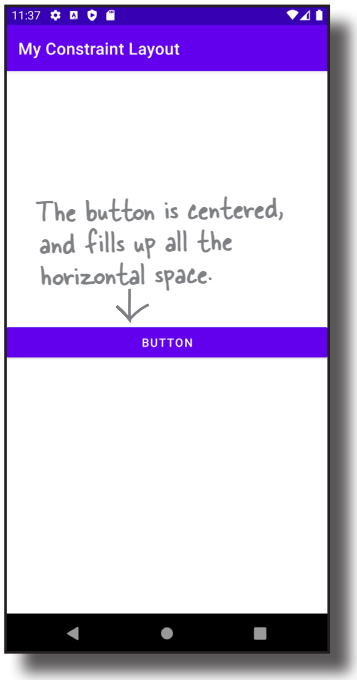
C

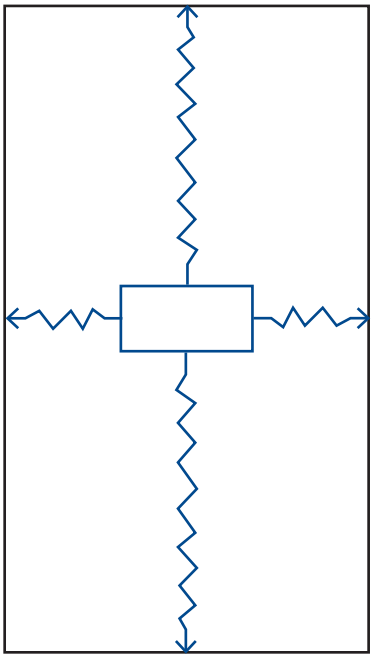


The button needs to stretch in all directions, so it requires constraints on all edges, and its width and height need to be set to 0dp.

layout_width: 0dp
layout_height: 0dp

D



layout_width: 0dp
layout_height: wrap_content

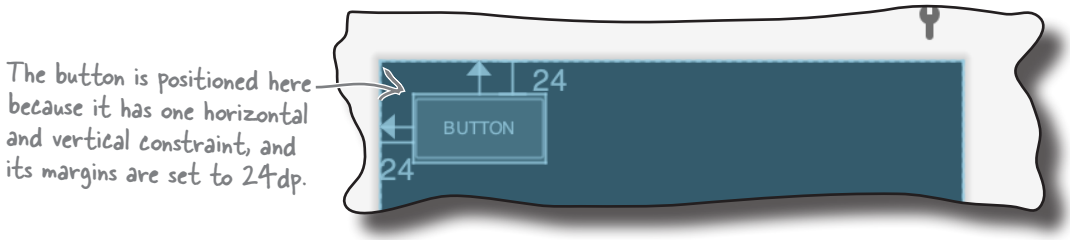
The button needs to stretch horizontally and be centered vertically. It therefore needs constraints on all edges as for C, but its width is set to 0dp, and its height is set to wrap_content.

# Most layouts need multiple views

So far you've seen how to position and size a single view in a constraint layout. Most of the time, however, your layout will need to contain multiple views that are laid out relative to one another.

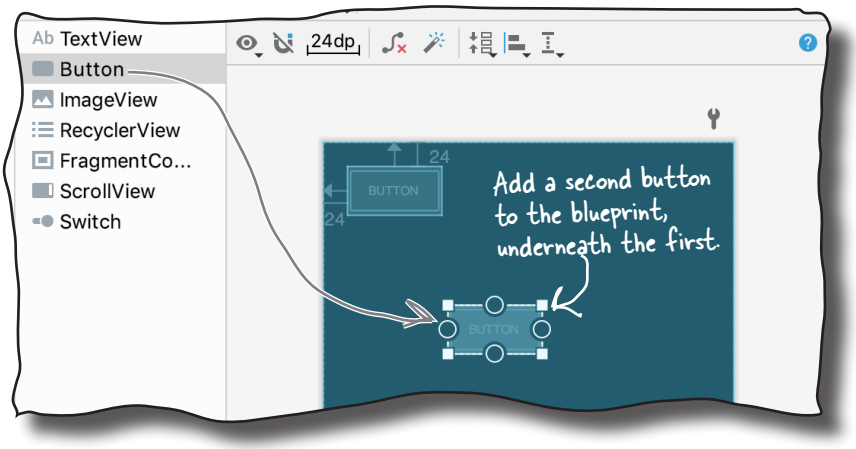
To see how this works, first make sure that your constraint layout includes a single button with two constraints: one that connects its top edge to the top of the blueprint, and another that connects its left edge to the blueprint's left side. Its `layout_width` and `layout_height` properties should be set to `wrap_content`, and the margins for these edges should be set to 24dp.

After you've made these changes, the button should be positioned in the blueprint's top-left corner like this:




## Add a second button to the blueprint

Next, add a second button to the blueprint by dragging one from the palette, and placing it somewhere underneath the first button like this:

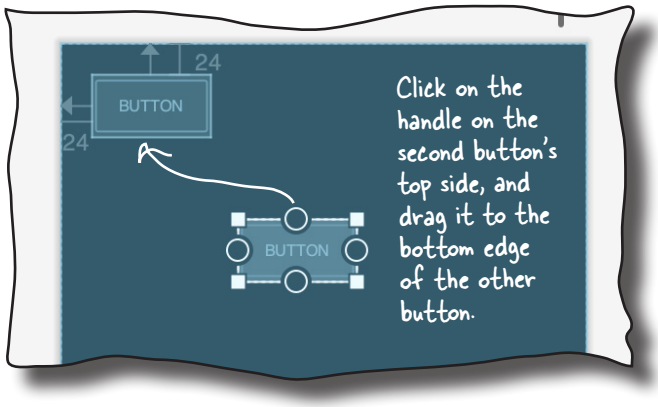



The blueprint now includes two buttons. Let's find out how to position them relative to one another.

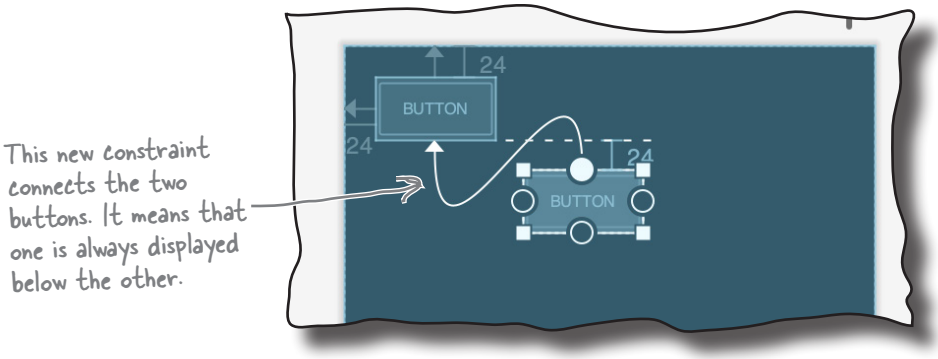
# You can connect views to other views

As you already know, constraints let you attach a view to the edge of its blueprint. You can also use constraints to connect two views together, and this is used to specify how they should be displayed relative to one another.

To see how this works, select the second button in the blueprint, then draw a constraint that goes from the second button's top edge to the first button's bottom edge like this:




When the constraint is added, it pulls the button up so that it's connected to the first, and displayed underneath it:



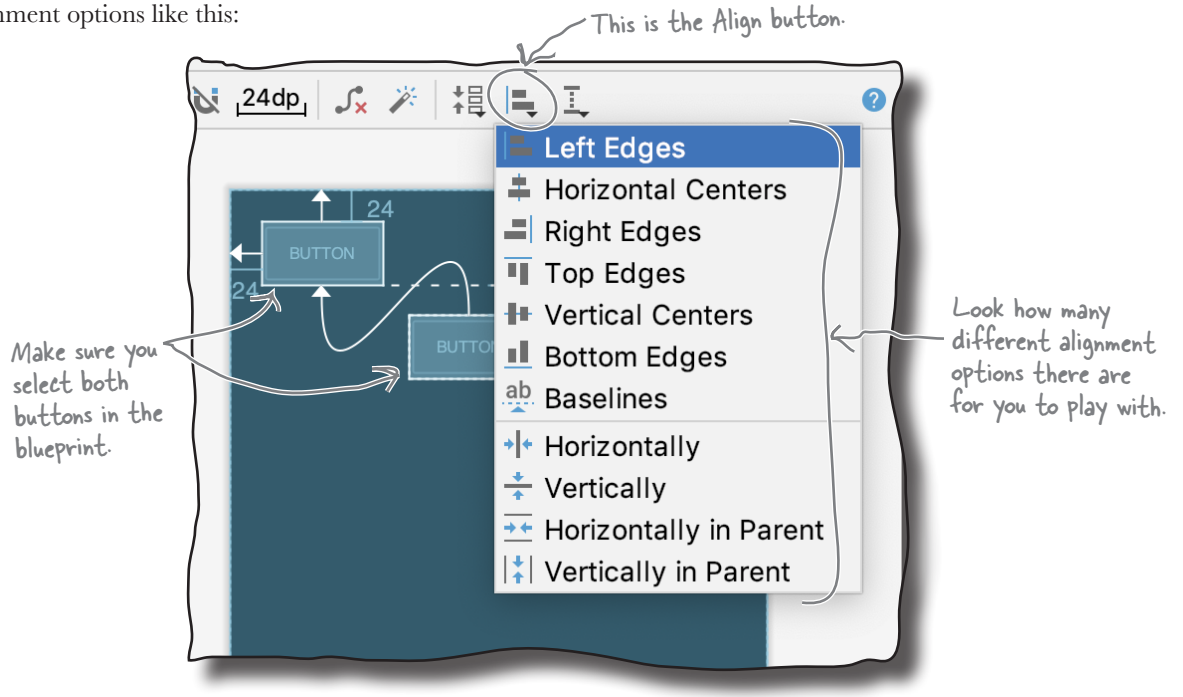

The constraint means that the second button will always be positioned under the first, irrespective of the first button's position on the device screen.

Once you've positioned two views in this way, the next thing you might want to do is make sure that they're aligned. Let's find out how to do this.

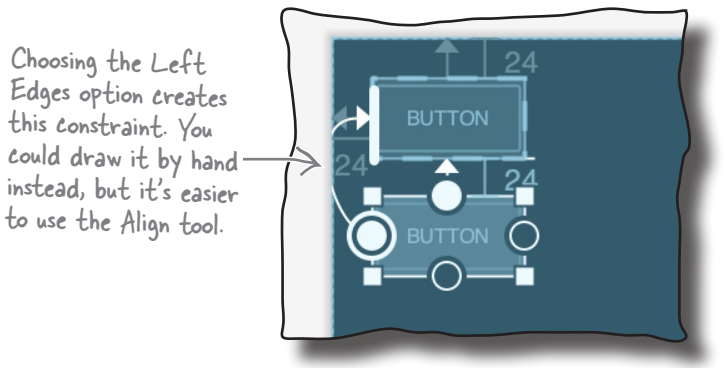
# You can align views too

The simplest way of aligning two views is to use the Align button in the design editor's toolbar.

To see how this works, let's left-align the two buttons in the blueprint so that their left edges line up. First, select both buttons by holding down the Shift key as you click on each one. Then click on the Align button to open up a set of alignment options like this:



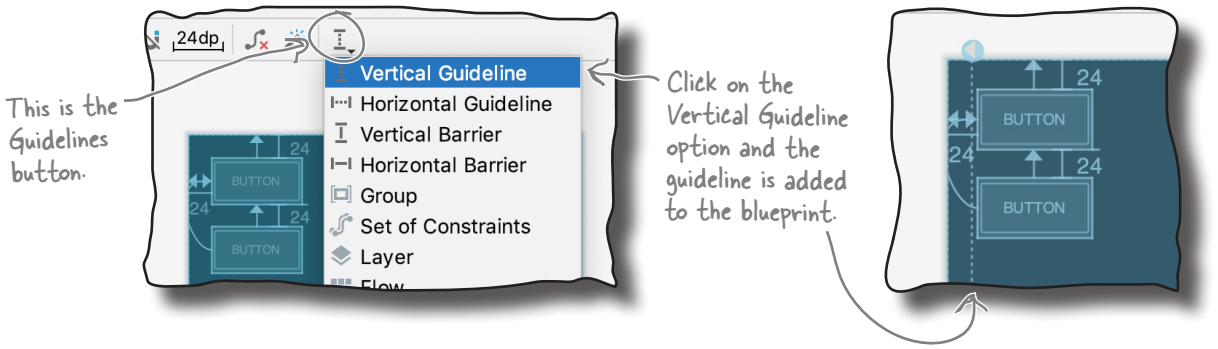

Click on the Left Edges option to left-align the two buttons. This adds a constraint to the blueprint that connects their left edges together like this:



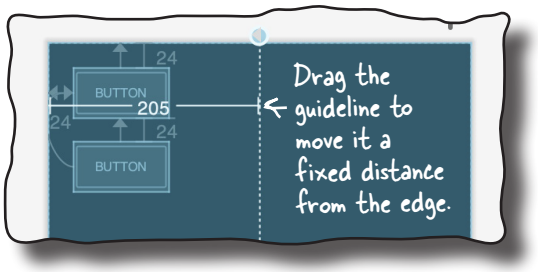
# Align views using guidelines

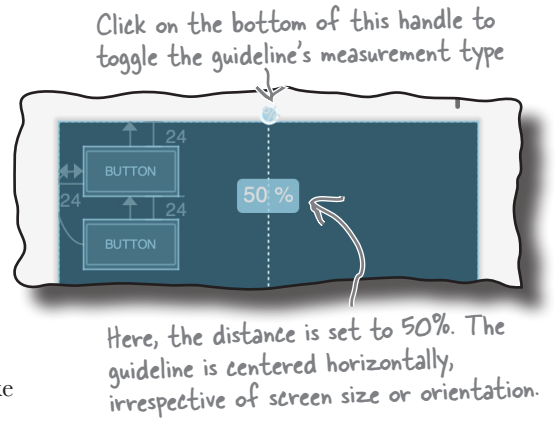
Another technique you can use to align views is a **guideline**. This is a fixed line you add to the blueprint that you can use to constrain views. It's only visible in the design editor, so users don't see it when they run the app.

Let's explore how guidelines work by adding one to the blueprint. Click on the Guidelines button in the design editor's toolbar, and choose the option to add a vertical guideline. This places a vertical guideline in the blueprint:

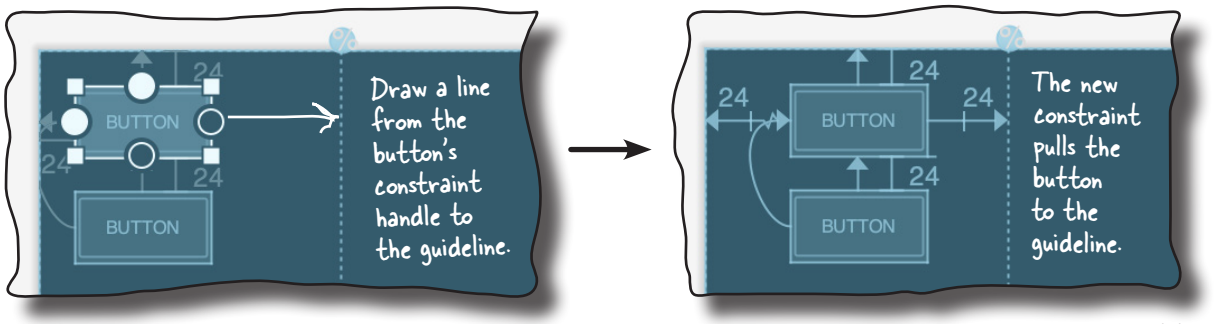



Once the guideline has been added, you can move it elsewhere by dragging it. You can set it to be either a fixed distance from the blueprint's edge, or a fixed percentage:



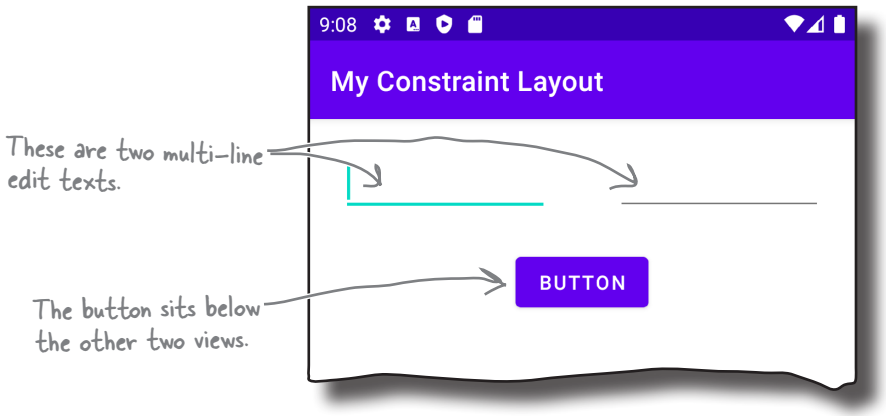

You can then use constraints to attach views to the guideline like this:



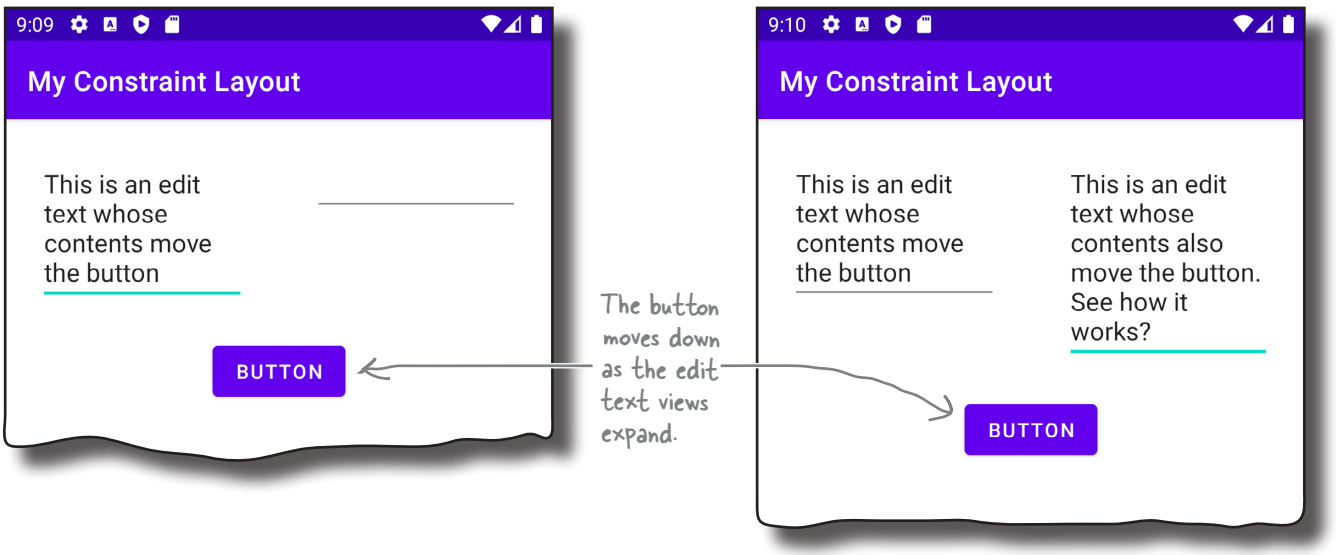
# Guidelines have a fixed position

Guidelines are either positioned a fixed distance from the blueprint's edge, or a fixed percentage between the two. They stay in that position when the app runs, so they're a useful way of aligning views.

In some situations, you need something that's a little more flexible. As an example, suppose you have a layout that includes two multi-line edit texts, side by side, with a button underneath like this:




The edit texts expand vertically as the user enters text. You want the button to move as the views change size so that it's always positioned beneath them like so:

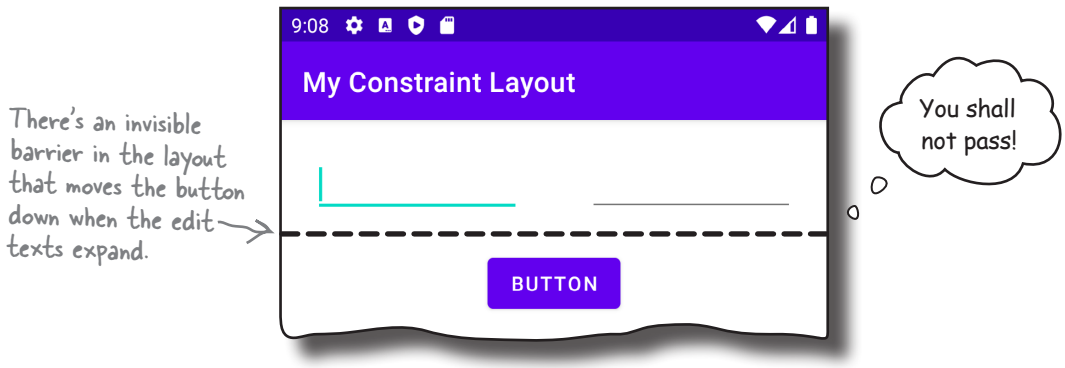



So how can you build this sort of layout?

# Create a movable barrier

To create layouts like this, you can use a **barrier**. This is like a guideline, except that it doesn't have a fixed position. Instead, it forms a barrier against views, and moves when they change size. This repositions any views that are constrained to the barrier.

In the example on the previous page, the two edit texts are placed above a horizontal barrier, and the button is constrained beneath it. As the edit texts expand, the barrier moves down and repositions the button:




## Let's build a layout that uses a barrier

To see how barriers work, let's create this example.

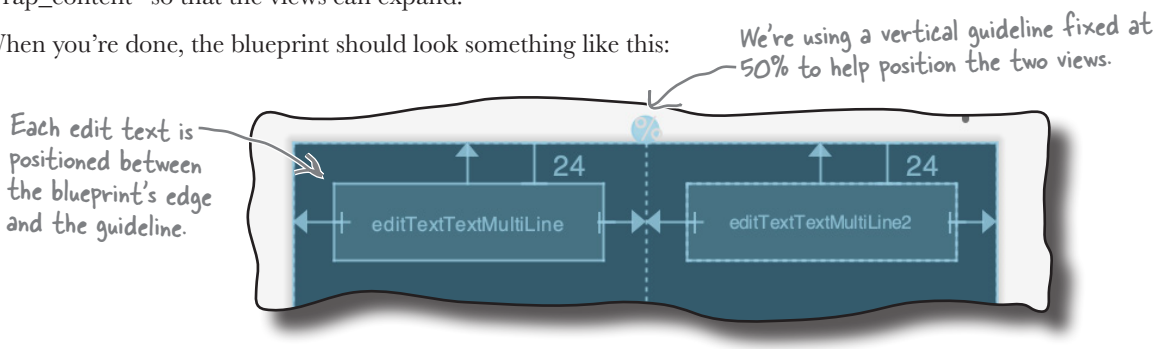
First, delete any views, and make sure the blueprint includes a vertical guideline positioned at 50%. Then drag two multi-line edit texts from the palette, and position them on either side of the guideline.

You can usually find these in the "Text" part of the palette, listed as "Multiline Text."

Next, add vertical constraints to constrain each view to the top of the blueprint, and horizontal constraints to position each one between the blueprint's edge and the guideline.

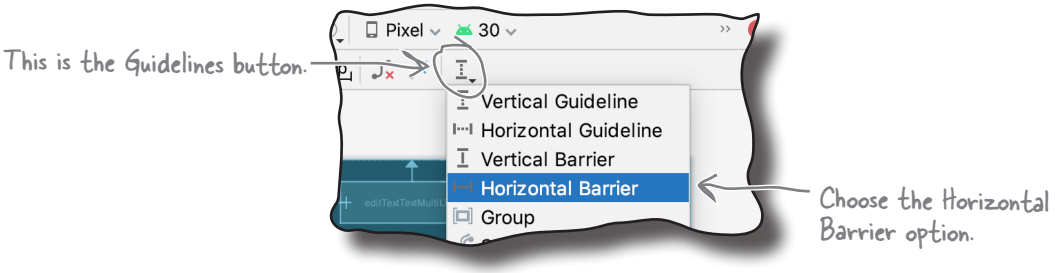
Finally, change the `layout_width` of each edit text to 0dp so that it matches its horizontal constraints, and set their `layout_heights` to "wrap_content" so that the views can expand.

When you're done, the blueprint should look something like this:



# Add a horizontal barrier

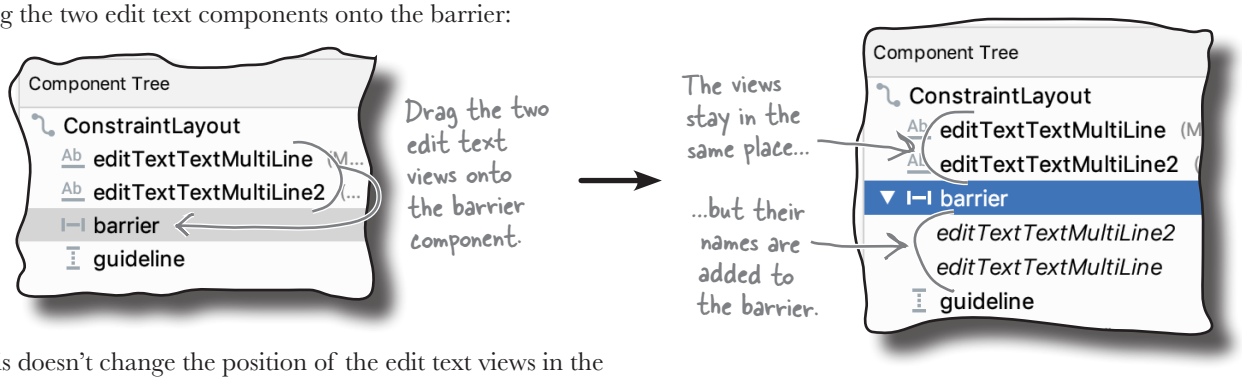
We need to add a horizontal barrier to the blueprint. To do this, click on the Guidelines button in the design editor toolbar, and choose the option to add a horizontal barrier:




This creates the horizontal barrier.

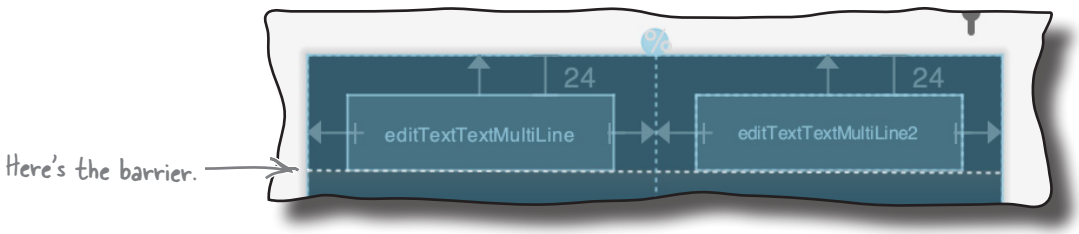
## Place the barrier beneath the views

We want the barrier to move down as the two edit text views expand. To do this, go to the layout's component tree panel, and drag the two edit text components onto the barrier:




This doesn't change the position of the edit text views in the blueprint: it simply tells the barrier that it needs to move with these views.

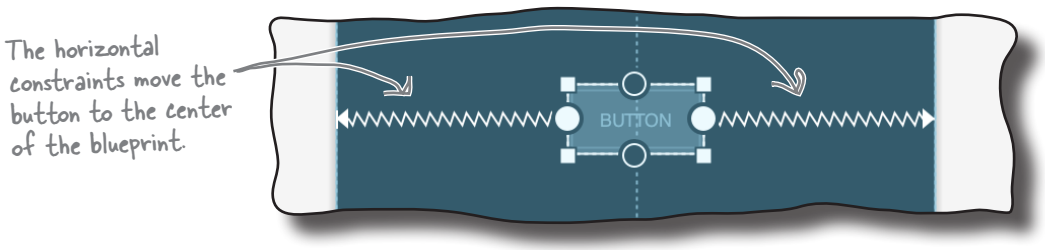
Next, we need to position the barrier so that it's at the bottom of the two views. Select the barrier in the component tree, and use the Attributes panel to change its `barrierDirection` attribute to "bottom." This positions the barrier below the two edit texts so that the blueprint looks like this:



# Constrain a button under the barrier

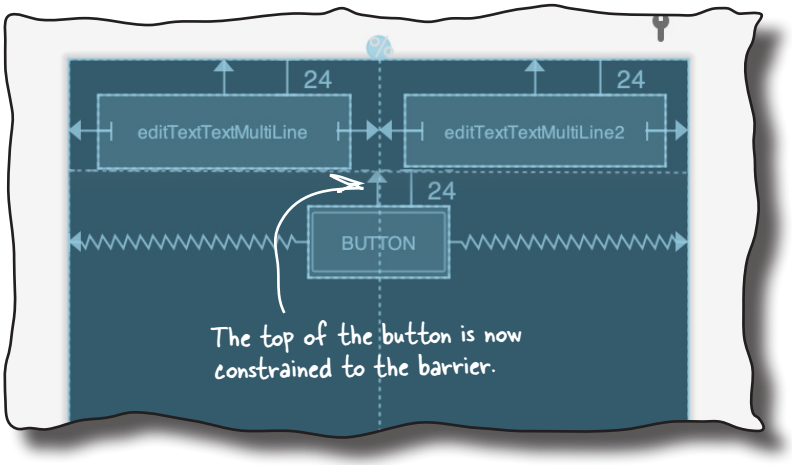
Now that the layout's barrier is in place, let's add the button, and constrain it to the barrier so that it moves down as the edit text views expand.

First drag a button from the palette to the blueprint, and place it somewhere underneath the barrier. Then center it horizontally by adding two horizontal constraints that connect the button's sides to the edges of the blueprint like this:




Next, you need to attach the top of the button to the barrier. You can try doing this by drawing the constraint directly in the blueprint. If, like us, you find this a bit too fiddly, select the button, search for its `layout_constraintTop_toBottomOf` attribute in the Attributes panel, and change its value to the barrier's ID (in our case, this is `@id/barrier`).

After you make this change, the blueprint should look something like this:



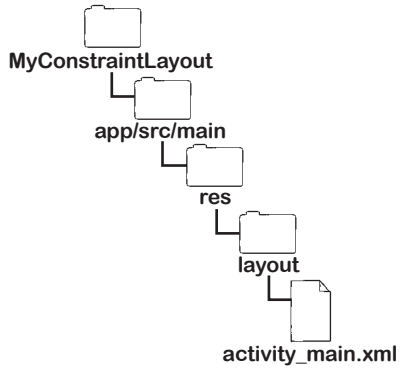

As barriers can be quite tricky to work with at first, we'll show you our complete XML over the next couple pages, and then take the app for a test drive.

**Single views**
**Multiple views**

# The full code for activity_main.xml

Here's our complete code for *activity_main.xml*; if you want your blueprint to look like ours, replace the contents of this file so that it matches the code shown here:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <EditText
        android:id="@+id/editTextTextMultiLine"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="24dp"
        android:layout_marginTop="24dp"
        android:layout_marginEnd="24dp"
        android:ems="10"
        android:gravity="start|top"
        android:inputType="textMultiLine"
        app:layout_constraintEnd_toStartOf="@+id/guideline"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <EditText
        android:id="@+id/editTextTextMultiLine2"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="24dp"
        android:layout_marginTop="24dp"
        android:layout_marginEnd="24dp"
        android:ems="10"
        android:gravity="start|top"
        android:inputType="textMultiLine"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="@+id/guideline"
        app:layout_constraintTop_toTopOf="parent" />
```




These are the two multi-line edit text views. They are constrained to the top of the layout, its edge, and the guideline.

The code continues on the next page.

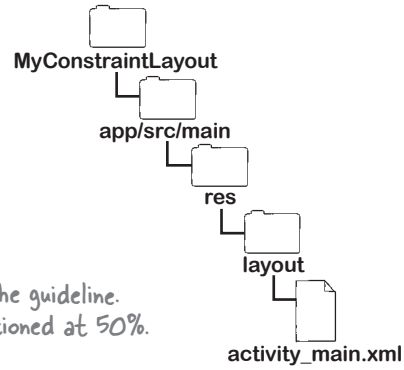
# activity_main.xml continued

```
<androidx.constraintlayout.widget.Barrier
    android:id="@+id/barrier"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:barrierDirection="bottom"
    app:constraint_referenced_ids="editTextTextMultiLine2,editTextTextMultiLine"
    tools:layout_editor_absoluteY="731dp" />

<androidx.constraintlayout.widget.Guideline
    android:id="@+id/guideline"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    app:layout_constraintGuide_percent="0.5" />

<Button
    android:id="@+id/button3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="24dp"
    android:text="Button"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/barrier" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

The barrier moves with the two edit texts, and is attached to their bottom edge.

This is the guideline. It's positioned at 50%.

The button is constrained to each side of the layout, and the barrier.

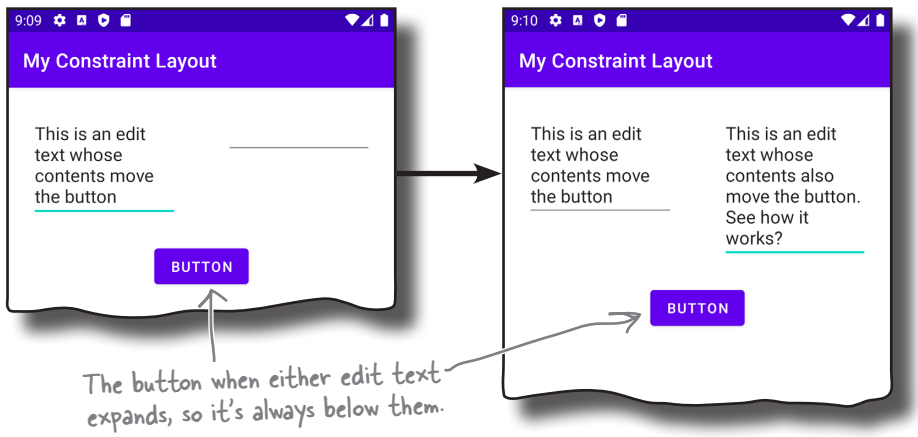


## Test Drive

When we run the app, a button is displayed beneath two edit text views. The button moves down when we type into each edit text, and the views expand.

As you can see, adding a barrier is a bit more complicated than drawing constraints and aligning views, but we think it's worth the extra effort.

So what's next?




The button when either edit text expands, so it's always below them.

**Single views**
**Multiple views**

# Use a chain to control a linear group of views

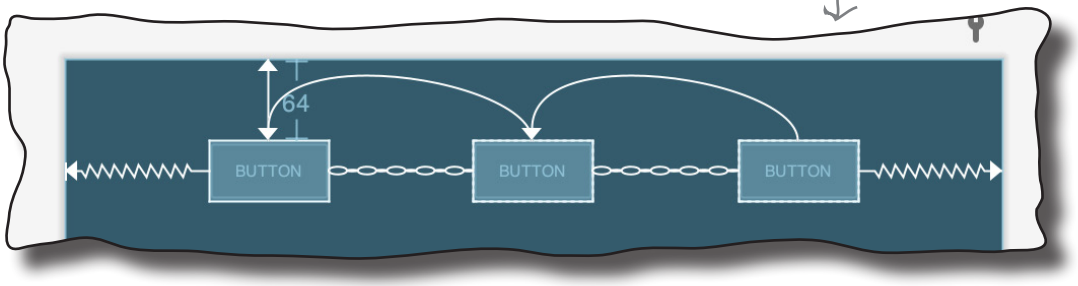
You've now learned how to connect and align views, and use guidelines and constraints. But what if you want to create a row or column of views, and evenly space them out?

In this sort of situation, you can use a **chain**. This is a linear group of views that are linked together with bidirectional constraints. The chain controls each view's position, so you can use it to evenly space the views out, or pack them in the center of the blueprint.

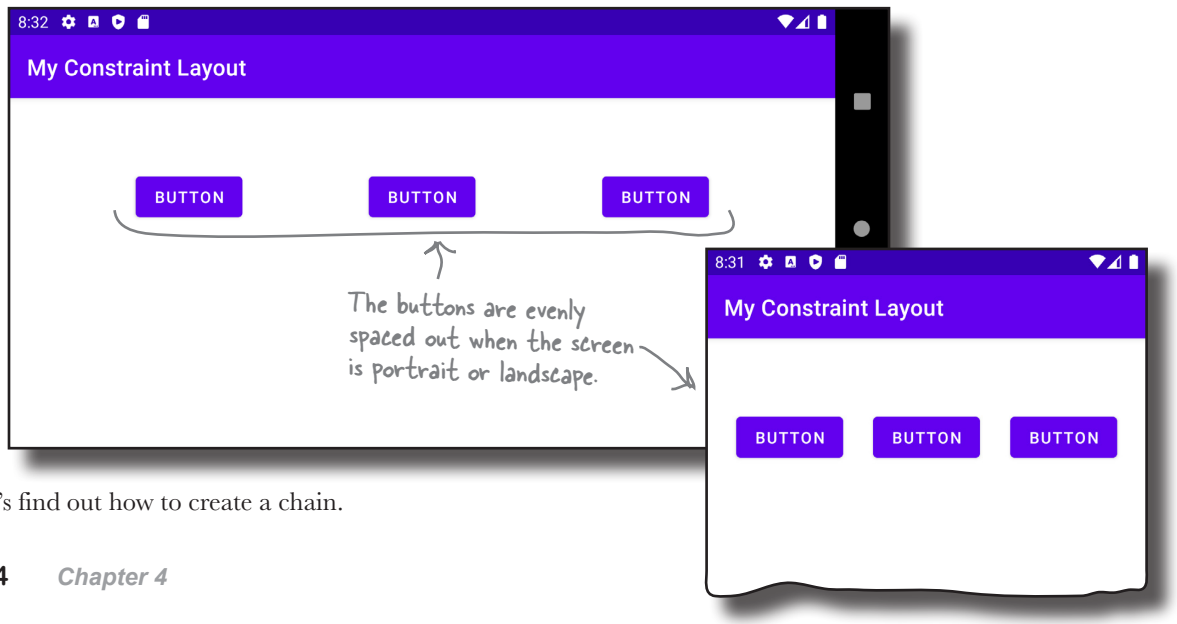
## We're going to create a horizontal chain

To see how this works, we're going to create a chain that controls the position of three buttons. The buttons will be lined up in a horizontal row, and evenly spaced between each side of the blueprint like this:

This shows a horizontal chain of three buttons. Their top edges are aligned, and they're evenly spaced out between the blueprint's edges.




When the app runs, the buttons will maintain their relative positions, regardless of screen size or orientation:



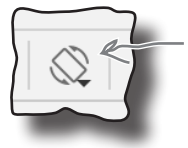

Let's find out how to create a chain.

# The chain will use three buttons

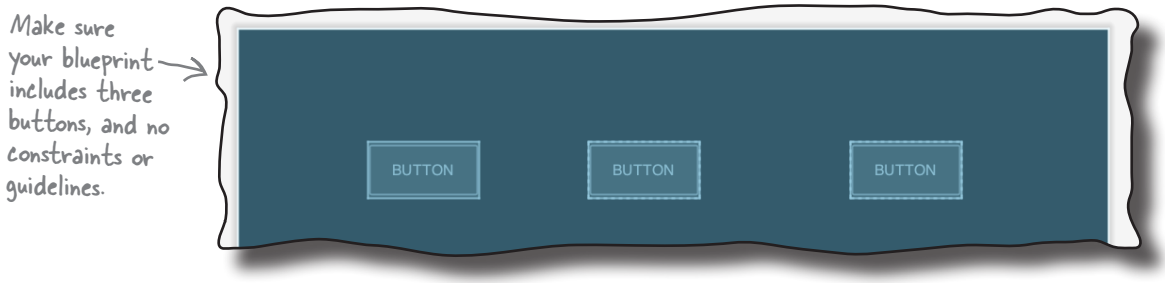
Before we create the chain, first remove all of the constraints that have been added to the blueprint so far. The quickest way of doing this is with the Clear All Constraints button in the design editor's toolbar, so click on this button now.

You also need to get rid of any guidelines, barriers, and edit text views. Do this by selecting each one, and deleting it.

Then add two more buttons to the blueprint so there are three in total, and use the "Orientation for Preview" button in the design editor's toolbar to change the blueprint's orientation to landscape. This will make it easier to see the chain.

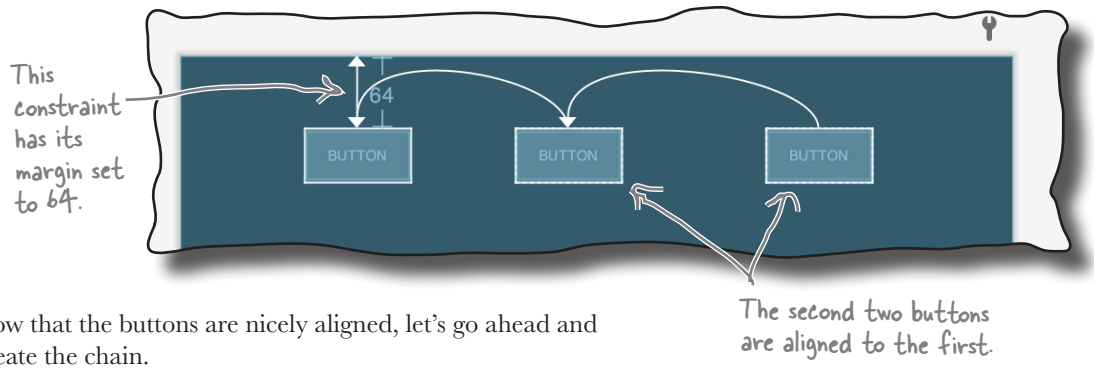



When you've added the buttons, the blueprint should look something like this:




## Align the views we're going to chain

Chains work best when views are aligned. First, add a constraint that connects the first button to the top of the blueprint, and set its margin to 64. Then select all three buttons, and use the Align button in the design editor's toolbar to align their top edges. The blueprint should look like this.

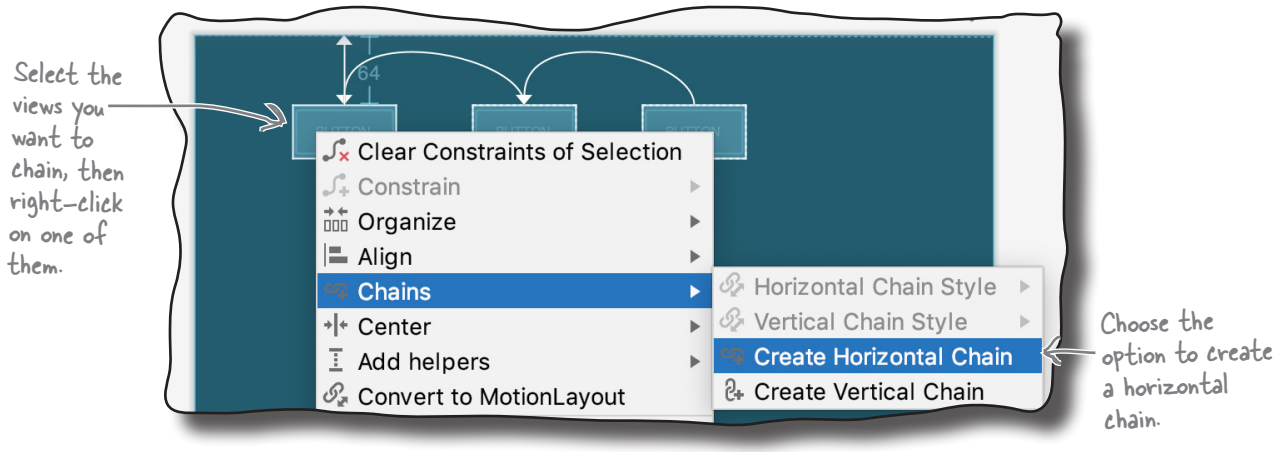



Now that the buttons are nicely aligned, let's go ahead and create the chain.

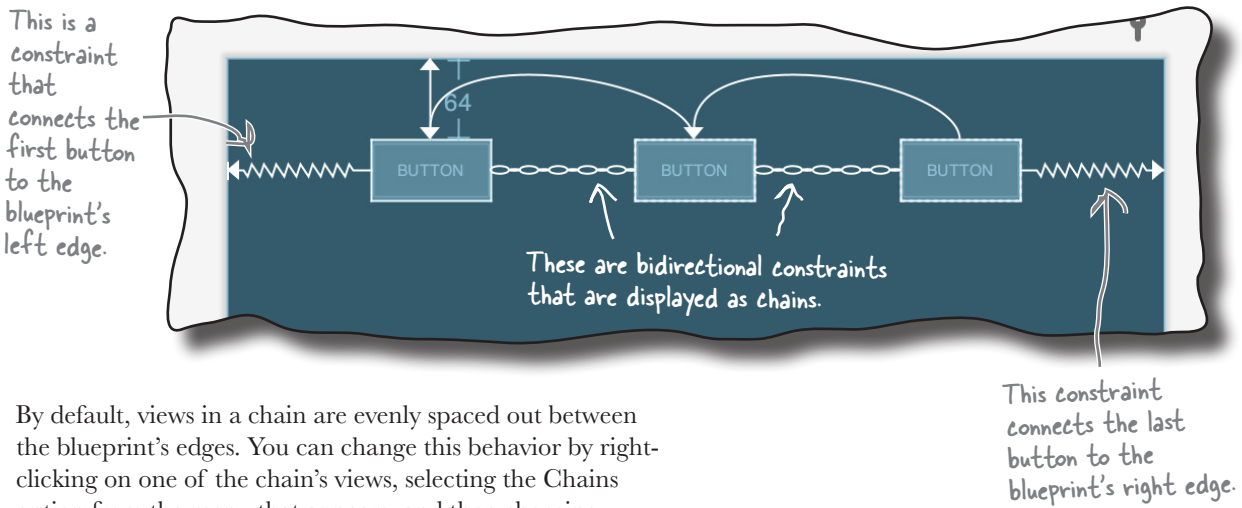
**Single views**
**Multiple views**

# Create the horizontal chain

To create the chain, select all three buttons, then right-click on one of them. From the menu that appears, choose the Chains option, followed by Create Horizontal Chain.




When the horizontal chain is created, it joins the buttons together, and fastens the first and last view to the blueprint's vertical edges. The chain should look something like this:




By default, views in a chain are evenly spaced out between the blueprint's edges. You can change this behavior by right-clicking on one of the chain's views, selecting the Chains option from the menu that appears, and then choosing Horizontal Chain Style.

Possible chain style options include spread, spread inside, and packed. See if you can work out what these options do by having a go at the following exercise.

## Exercise

Right-click on one of the buttons in the horizontal chain, select the Chains option from the menu that appears, and then choose Horizontal Chain Style. What happens to the views when you choose each option?

Draw the blueprint for each option below, and describe how it arranges the views.

**Spread:**

**Spread inside:**

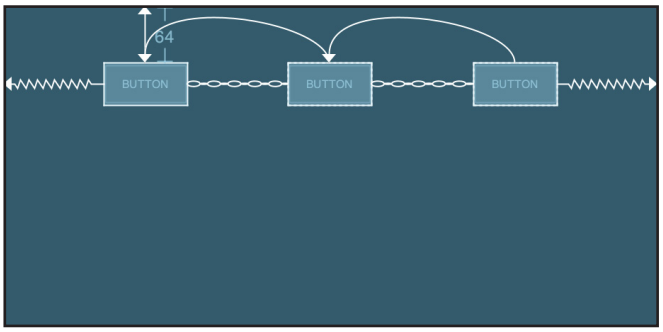
**Packed:**

# Exercise Solution

Right-click on one of the buttons in the horizontal chain, select the Chains option from the menu that appears, and then choose Horizontal Chain Style. What happens to the views when you choose each option?

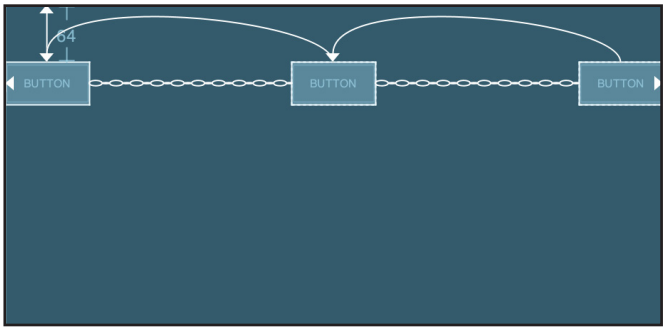
Draw the blueprint for each option below, and describe how it arranges the views.

**Spread:**



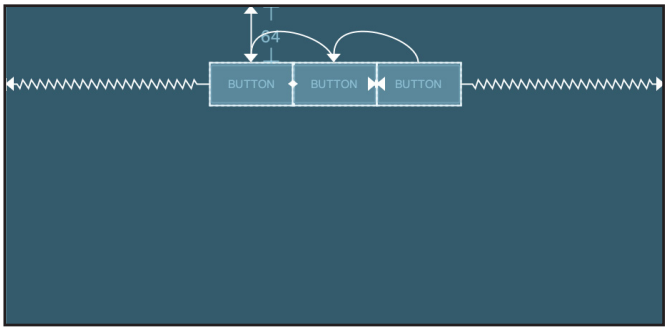

The spread option arranges views so that they are evenly spaced out between the blueprint's edges.

**Spread inside:**




The spread inside option moves the left– and rightmost views to the blueprint's edges, then evenly spaces out the remaining views.

**Packed:**



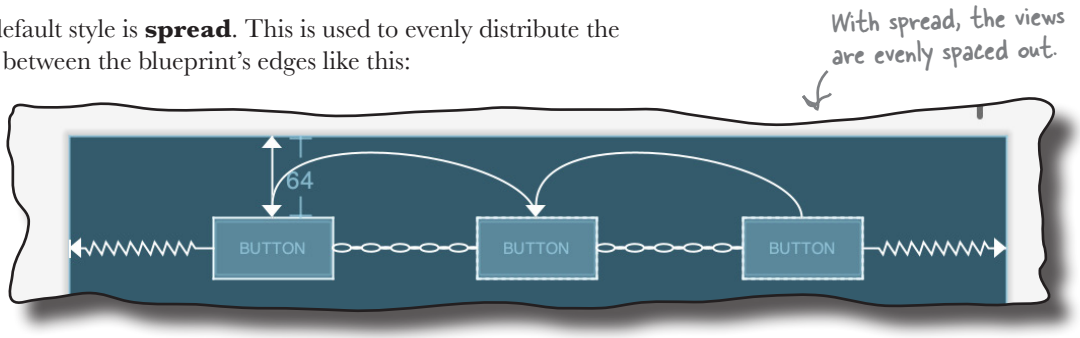

The packed option packs the views together, and centers the whole group.

# There are different styles of chain

As you have discovered, you can choose different chain styles to change how a chain arranges its views.

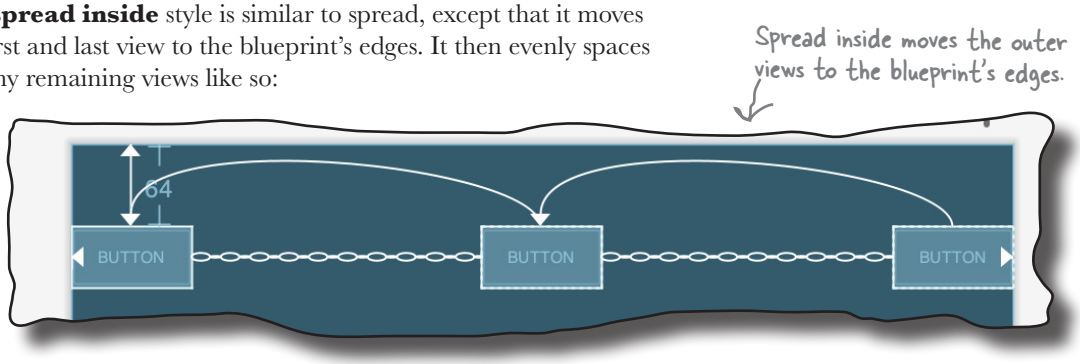
## Spread spaces out views between the blueprint's edges

The default style is **spread**. This is used to evenly distribute the views between the blueprint's edges like this:



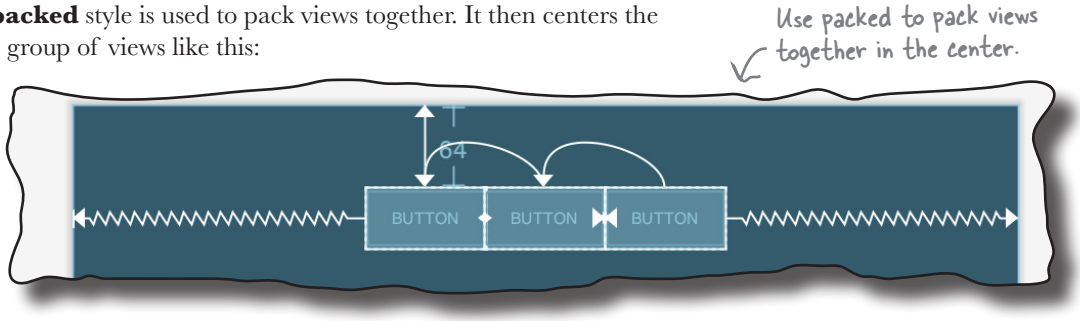

## Spread inside moves the first and last view to the edges

The **spread inside** style is similar to spread, except that it moves the first and last view to the blueprint's edges. It then evenly spaces out any remaining views like so:




## Packed moves the views together

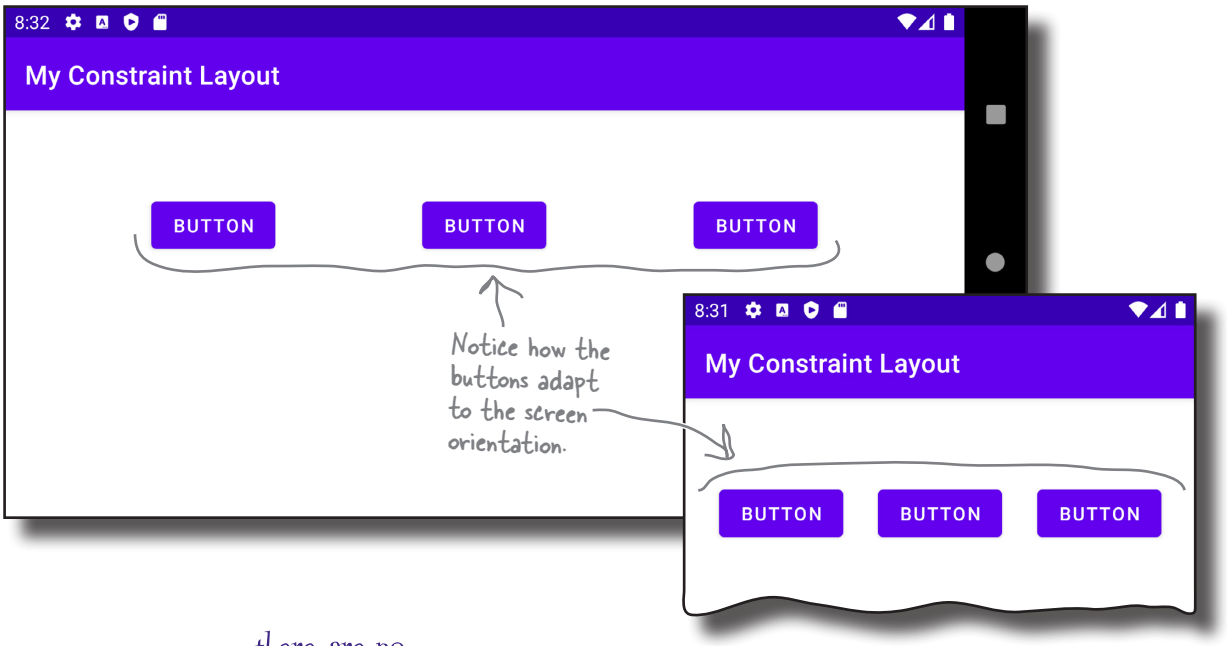
The **packed** style is used to pack views together. It then centers the entire group of views like this:




Now that you've seen what these options do, let's take the app for a test drive.

# Test Drive

When we use the spread style of chain and run the app, the buttons are evenly spread in the device screen. This is irrespective of screen orientation.




## there are no Dumb Questions

**Q: Can I make views a fixed size?**

A: Yes. It's not generally recommended because it means they can't grow or shrink to fit the size of their contents, or the size of the screen.

If you definitely do need a fixed size view, you can either update its attributes, or use its corner handles to resize it in the blueprint.

**Q: Can I attach a chain to a guideline?**

A: Yes, you can.

**Q: What happens if I make a view invisible?**

A: It depends how you do it.

If you set a view's `visibility` attribute to `invisible`, the view won't appear on the screen, and there'll be an empty space in its place.

If you set it to `gone`, the view takes up no space on the screen, and things like chains will act as though it's not there.

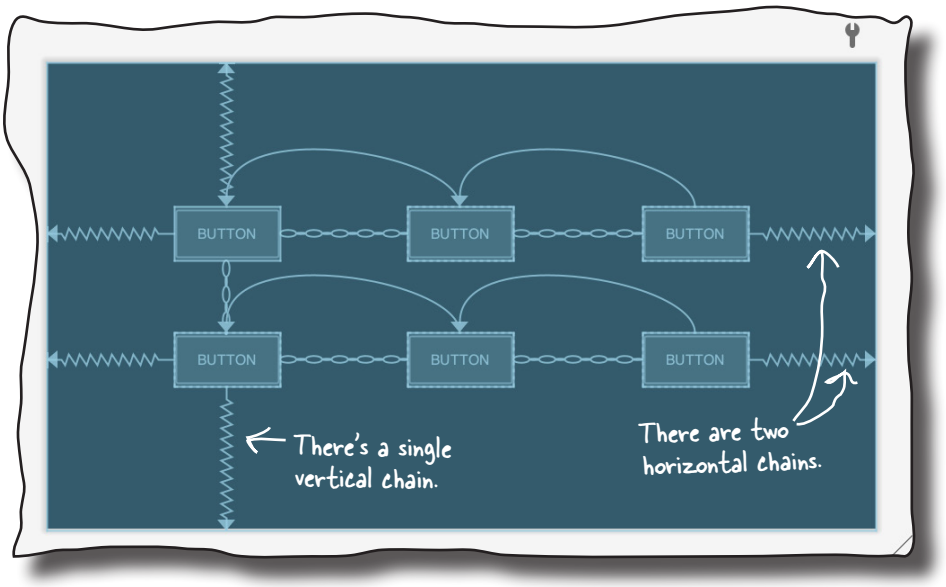
**Always test layouts on a variety of device sizes and orientations to make sure they look and behave how you want.**

So we've created a horizontal chain, and there are options to create vertical chains too. That's got me thinking... Is it possible for a view to belong to a horizontal chain **and** a vertical one? If so, could that be used to create a grid of views?

## She's right.

Constraint layouts can include both horizontal and vertical chains, and a single view can belong to both types. You can use this to arrange views in a grid.

The blueprint below, for example, shows six buttons arranged in a grid. Each row is a horizontal chain, and the leftmost buttons form a vertical chain:



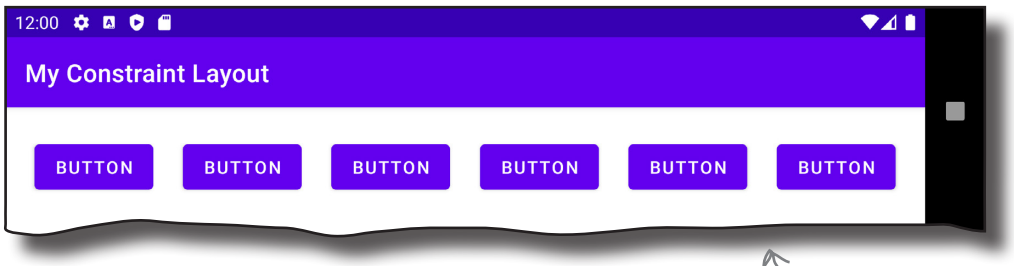

Another way in which you can create grids is to use a **flow**. Let's find out what this is, and how to use it.

Single views
**Multiple views**

# A flow is like a multi-line chain

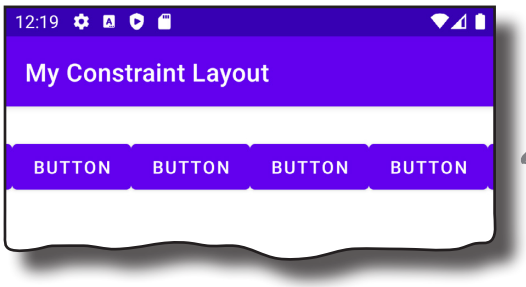
A flow is like a chain that can span multiple rows. It's invaluable when, say, you want to display lots of views in a row, but they might not fit on the screen for some screen sizes or orientations.

As an example, suppose you have a chain that displays six buttons in a horizontal row. When the orientation is landscape, they are displayed like this:



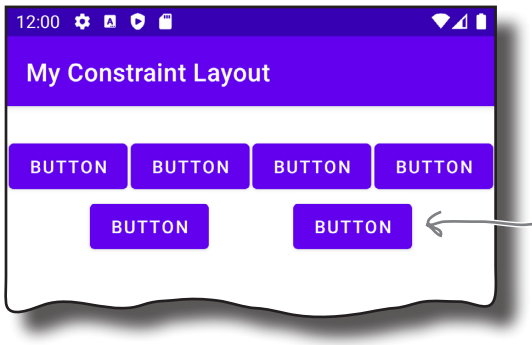

The buttons fit on the screen when the orientation is landscape.

But when the orientation is changed to portrait, there's not enough room to display all of the views:




Here, there's not enough space to display all of the buttons, so they've fallen off the edge of the screen

If you replace the chain with a flow, any views that can't fit on the first row will flow onto a second row like this:

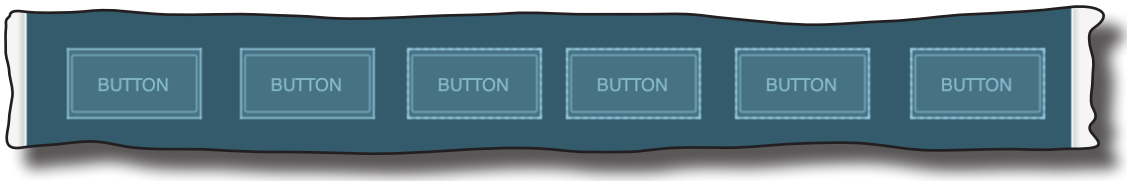



Using a flow means that any views that don't fit on the first row will form a second row.

Let's see how flows work by building the above layout.

# How to add a flow

First, remove any constraints by clicking on the Clear All Constraints button in the design editor's toolbar. Then add extra buttons to the blueprint so there are six in total like this:

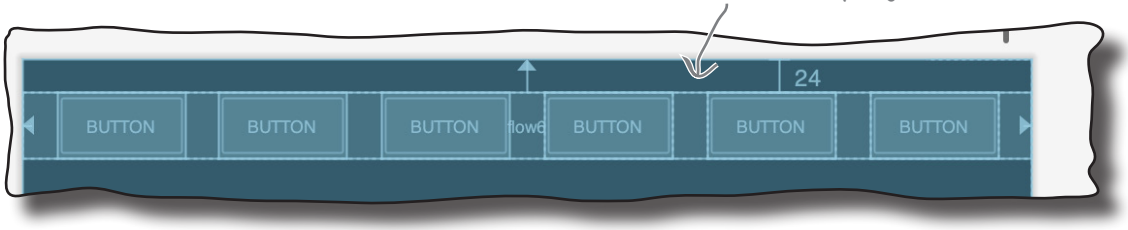



Next, select all of the buttons, click on the Guidelines button in the design editor's toolbar, and choose the Flow option. This adds the flow component.

We now need to tweak the flow component's settings to make it behave the way we want. To do this, select the flow in the component tree, then use the blueprint or constraint widget to add constraints to connect its sides and top to the edges of the blueprint. Change its `layout_width` attribute to "0dp" so that it matches its constraints. Finally, search for its `flow_wrapMode` attribute in the Attributes panel, and set this to "chain".

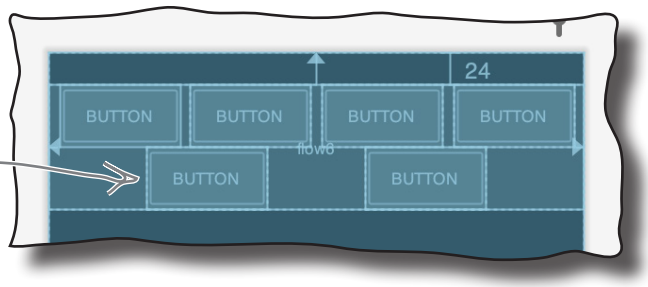
When you've made all of these changes, the blueprint should look something like this when the orientation is landscape:

All of the buttons have been added to a flow. The flow is constrained to the blueprint's sides and top edge.




If you change the orientation to portrait, the blueprint should look like this instead:

Any buttons that don't fit on the first row flow onto the second.




Once you've created a flow, you can tweak the way in which it displays its views. Let's see how.

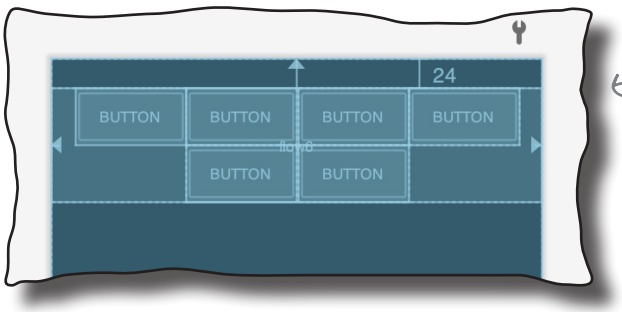
# You can control the flow's appearance

The main way in which you can alter the flow's appearance is with its **`flow_wrapMode`** attribute.

## Use "chain" to create a multi-line chain

If you set the `flow_wrapMode` attribute to `chain`, the flow behaves like a flexible chain that lets its views flow onto extra rows.

With this option, you can make further changes to the flow's appearance by changing the value of its `flow_horizontalStyle` attribute. Possible options for this attribute are spread, spread inside, and packed. These have exactly the same effect as when you used them with chains. The packed option, for example, packs the views together like this:

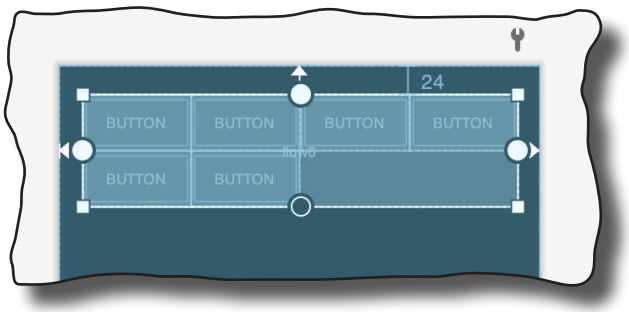



This is what you get if you set flow_wrapMode to chain, and use a packed horizontal style.

## Use "aligned" to line up the views

If you set the `flow_wrapMode` attribute to `aligned`, the views flow onto extra rows, and they are lined up like this:

Notice how the views are aligned. You can use this option to display views in a grid.

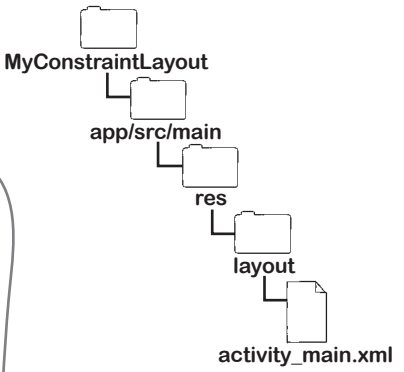



You can also set the `flow_wrapMode` attribute to `none` or leave it unset. This makes the flow behave like a normal chain so that its views don't flow onto a second row.

# The full code for activity_main.xml

Flows can sometimes be a little tricky to get right, so here's our full code for *activity_main.xml*; if you want to, replace the code for this file so it matches the code shown here:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/button3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button" />

    <Button
        android:id="@+id/button4"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button" />

    <Button
        android:id="@+id/button5"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button" />

    <Button
        android:id="@+id/button6"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button" />
```




These are four of the buttons we added.

The code continues on the next page.

# activity_main.xml continued

**Single views**
**Multiple views**

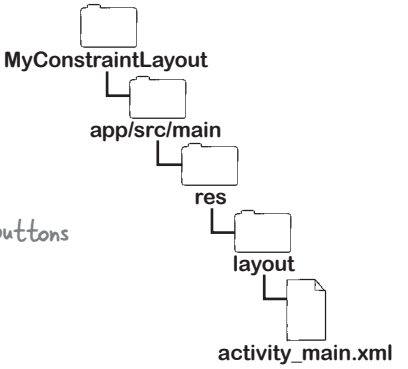
```
    <Button
        android:id="@+id/button7"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button" />

    <Button
        android:id="@+id/button8"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button" />

    <androidx.constraintlayout.helper.widget.Flow
        android:id="@+id/flow"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginTop="24dp"
        app:constraint_referenced_ids=
            "button3,button4,button5,button6,button7,button8"
        app:flow_horizontalStyle="spread"
        app:flow_wrapMode="chain"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Yet more buttons

Here's the flow.

The flow includes these views.

These options determine how the views are arranged.

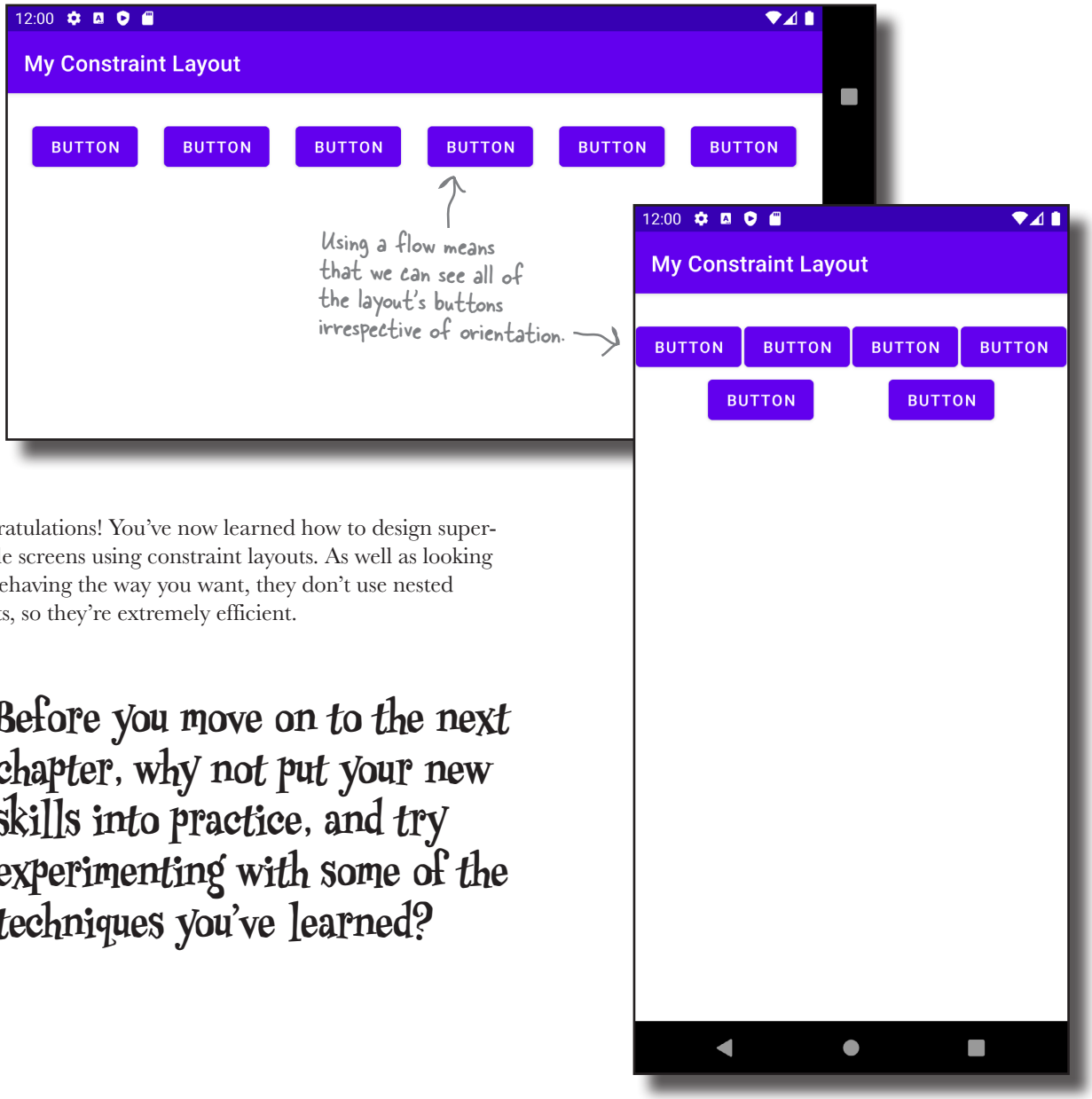



Let's take the app for a test drive.

# Test Drive

When we use the chain style of flow and run the app, the buttons are evenly spread in the device screen when the orientation is landscape.

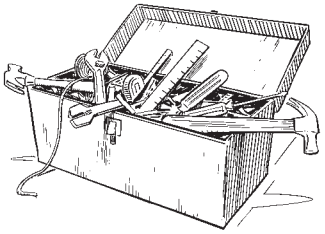
When we change the orientation to portrait, any buttons that don't fit onto the first row flow onto the second.




Congratulations! You've now learned how to design super-flexible screens using constraint layouts. As well as looking and behaving the way you want, they don't use nested layouts, so they're extremely efficient.

## Before you move on to the next chapter, why not put your new skills into practice, and try experimenting with some of the techniques you've learned?

# Your Android Toolbox



**You've got Chapter 4 under your belt and now you've added constraint layouts to your toolbox.**

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

## Bullet Points

- Constraint layouts are designed to work with Android Studio's design editor.
- Position views by adding constraints. Each view needs at least one horizontal and one vertical constraint.
- Center views by adding constraints to opposite sides of the view. Change the view's bias to update its position between the constraints.
- You can change a view's size to match its constraints if the view has constraints on opposing sides.
- You can align views to one another, or use a guideline or barrier.
- A guideline is positioned a fixed distance or percentage from the blueprint's edge.
- A barrier moves as its views change size.
- Use a chain to control a linear group of views. Chains can be horizontal or vertical.
- Horizontal and vertical chains can work together to display views in a grid.
- The spread chain style spreads its views out evenly.
- The spread inside chain style moves its outer views to the edges, and the inner views are spread out evenly.
- The packed chain style packs views together in the center.
- A flow is like a multi-line chain.

# 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. Read on to find out more…

# How do activities really work?

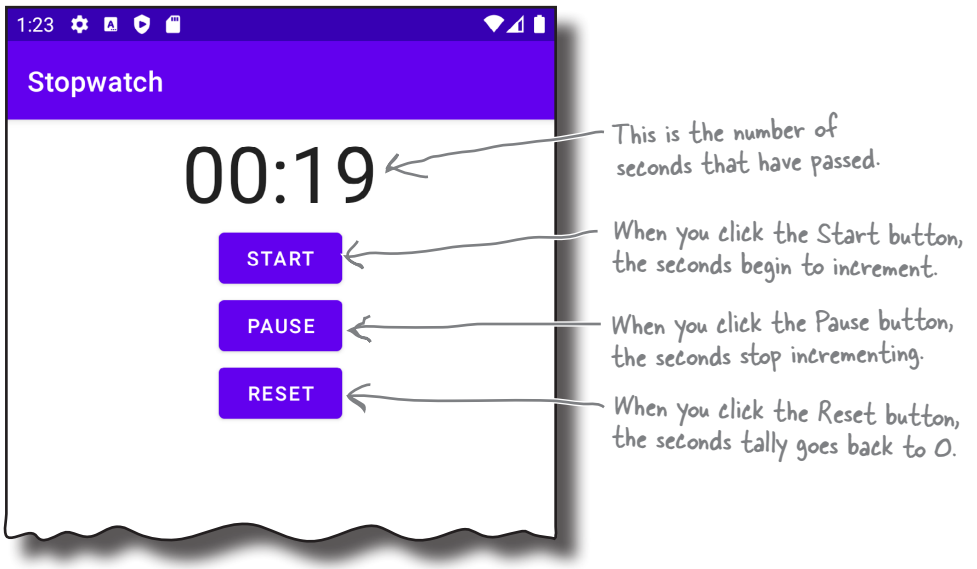
So far you've seen how to create an interactive app by specifying how the app should behave in its activity code. But there are still lots of things you don't yet know about how activities function.

How long does an activity live for? What happens when your activity disappears from the screen? Is it still running? Is it still in memory? And what happens if your app gets interrupted by an incoming phone call?

You want to be able to control the behavior of your activities in a *whole range of different circumstances*, so we're going to take a closer look at how activities work under the hood. We'll look at some common ways in which your apps can break, and how you can fix them using the **activity lifecycle methods**. We'll explore these functions using a Stopwatch app as an example.

## Introducing the Stopwatch app

The Stopwatch app has four views: a view that displays the elapsed time, a Start button that starts the stopwatch, a Pause button that pauses it, and a Reset button that sets the timer value back to 0. Here's what the app looks like:




We'll build the app over the next few pages, and we'll start by creating a new project.

# Create a new project

Create a new Android Studio project using the same steps you used in previous chapters. Choose the Empty Activity option, enter a name of "Stopwatch" and a package name of "com.hfad.stopwatch", and accept the default save location. Make sure the language is set to Kotlin and the minimum SDK is API 21 so it will run on most Android devices.

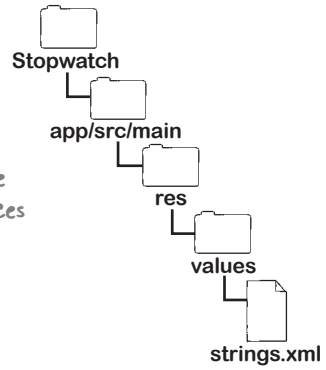
## Add String resources for the text labels

The Stopwatch layout features three buttons labeled Start, Pause, and Reset. We'll add these labels to *strings.xml* as `String` resources so the app can look up their values at runtime.

To add the `String` resources, open the file *strings.xml* in the *app/src/main/res/values* folder, and update it so it includes the changes below:

```
<resources>
    <string name="app_name">Stopwatch</string>
    <string name="start">Start</string>
    <string name="pause">Pause</string>
    <string name="reset">Reset</string>
</resources>
```

We'll use these String resources in the layout.




Now that we've added the `String` resources, let's build the layout.

## The layout includes a Chronometer view

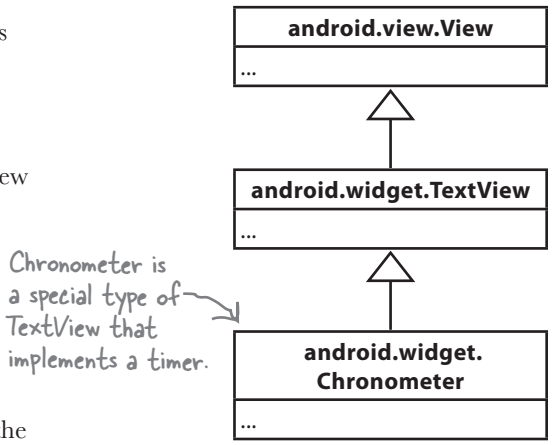
The layout is comprised of three buttons, and a view that displays the number of seconds on the stopwatch. To display the number of seconds, we'll use a **chronometer**.

A chronometer is a type of text view that acts as a simple timer. It has built-in methods you can use to start and stop it, and it displays the elapsed time.

You add a chronometer to your layout using the `<Chronometer>` element. The following code, for example, adds a `Chronometer` view with an ID of "stopwatch" that is just large enough for its contents:

```
<Chronometer
    android:id="@+id/stopwatch"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

Chronometer is a special type of TextView that implements a timer.



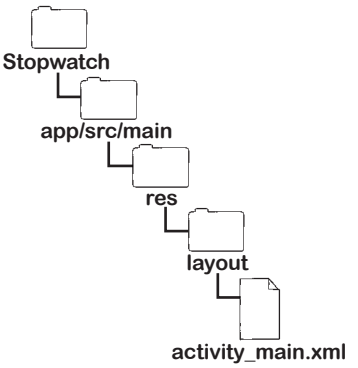

You'll learn more about how the chronometer works when we write the activity code. First, let's look at the full code for the layout.

# The full code for activity_main.xml

The full code for *activity_main.xml* features a `Chronometer` view and three buttons. Update your code in *activity_main.xml* so that it matches the code below:

```
<?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:orientation="vertical"
    android:gravity="center_horizontal"
    tools:context=".MainActivity">

    <Chronometer
        android:id="@+id/stopwatch"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="56sp" />

    <Button
        android:id="@+id/start_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/start" />

    <Button
        android:id="@+id/pause_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/pause" />

    <Button
        android:id="@+id/reset_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/reset" />
</LinearLayout>
```

This centers the contents of the layout horizontally.




This uses the textSize attribute to make the chronometer text nice and big.

This code adds the Start, Pause, and Reset buttons.

**Don't forget!**

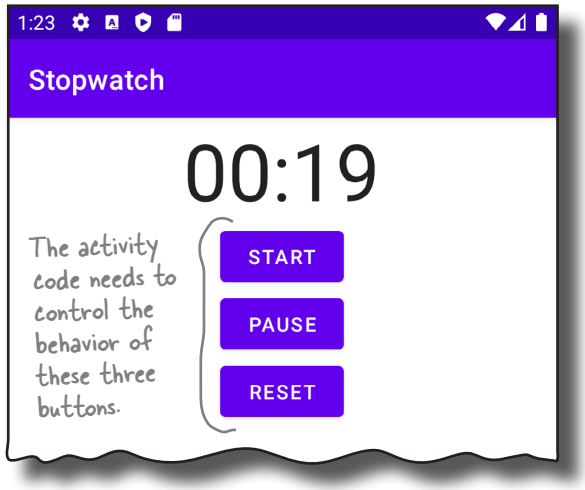
Make sure you update the layout and strings.xml code (on the previous page) before continuing.

The layout is now complete. Next, let's move on to the activity.

# The activity code controls the stopwatch

The activity code needs to specify how the Start, Pause, and Reset buttons control the stopwatch. The Start button needs to start it running (if it's not running already), the Pause button needs to pause it, and the Reset button needs to set it back to 0.

The buttons will control the stopwatch by accessing the `Chronometer` view's built-in properties and methods. Let's take a look at these.




## The Chronometer's key properties and methods

As we said earlier, the `Chronometer` view is a type of `TextView`. It inherits all of `TextView`'s properties and methods, and defines some new ones too.

Here are all the `Chronometer` properties and methods we'll be using in the `MainActivity` code:

### The base property

The `Chronometer base` property is used to set the start time: the time from which the chronometer will start counting.

You set the start time to the current time by setting the `base` property to `SystemClock.elapsedRealtime()` using code like this:

```
stopwatch.base = SystemClock.elapsedRealtime()
```

This sets the time that's displayed to 0.

`SystemClock.elapsedRealtime()` returns the number of milliseconds since the device was booted. Setting the `base` property to this value means that the time that's displayed is set to 0.

### The start() method

The `start()` method makes the chronometer start counting from the base time:

```
stopwatch.start()
```

This starts the chronometer.

### The stop() method

This pauses the chronometer so it stops counting:

```
stopwatch.stop()
```

This pauses the chronometer.

Now that you know about the `Chronometer` view's properties and methods we'll be using, let's look at the activity code.

### Serious Coding

You can find out more about using the Chronometer view in the online documentation here:

*https://developer.android.com/reference/android/widget/Chronometer*

# The full code for MainActivity.kt

The code for `MainActivity` requires an `OnClickListener` for each button. The one for the Start button needs to start the stopwatch if it's not already running, the one for the Pause button needs to pause it, and the Reset button's `OnClickListener` needs to set the display back to 0.

We're using three properties to help us with this: `stopwatch`, `running`, and `offset`.

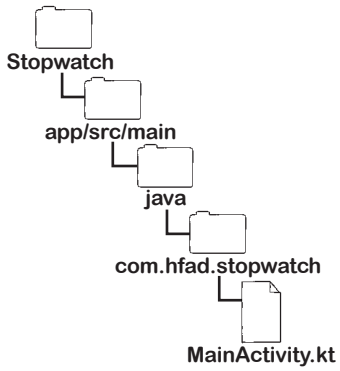
The `stopwatch` property holds a reference to the `Chronometer` view.

The `running` property records whether the stopwatch is running. We'll set it to `true` when the Start button's clicked, and `false` when the user clicks the Pause button.

The `offset` property is used to display the correct time on the stopwatch if the stopwatch has been paused and restarted. Without this, the stopwatch will display the incorrect time.

We've also included two methods (`setBaseTime` and `saveOffset`) to help make the code more readable.

Here's the full code for *MainActivity.kt*; update your code to include all of the changes:




```
package com.hfad.stopwatch

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.os.SystemClock
import android.widget.Button
import android.widget.Chronometer

class MainActivity : AppCompatActivity() {

    lateinit var stopwatch: Chronometer  //The stopwatch
    var running = false  //Is the stopwatch running?
    var offset: Long = 0  //The base offset for the stopwatch

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

        //Get a reference to the stopwatch
        stopwatch = findViewById<Chronometer>(R.id.stopwatch)
```

We're using these libraries in the code, so you need to import them.

We're using three property values to help control the stopwatch.

We can only instantiate the stopwatch property after the call to setContentView. Before this point, the Chronometer view doesn't exist.

The code continues on the next page.

# MainActivity.kt continued

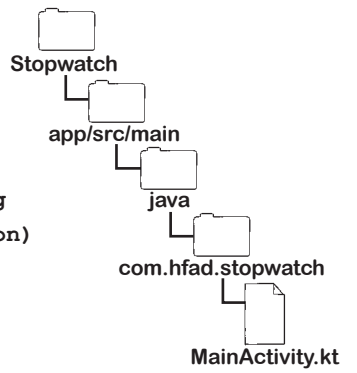
```
        //The start button starts the stopwatch if it's not running
        val startButton = findViewById<Button>(R.id.start_button)
        startButton.setOnClickListener {
            if (!running) {
                setBaseTime()
                stopwatch.start()
                running = true
            }
        }

        //The pause button pauses the stopwatch if it's running
        val pauseButton = findViewById<Button>(R.id.pause_button)
        pauseButton.setOnClickListener {
            if (running) {
                saveOffset()
                stopwatch.stop()
                running = false
            }
        }

        //The reset button sets the offset and stopwatch to 0
        val resetButton = findViewById<Button>(R.id.reset_button)
        resetButton.setOnClickListener {
            offset = 0
            setBaseTime()
        }
    }

    //Update the stopwatch.base time, allowing for any offset
    fun setBaseTime() {
        stopwatch.base = SystemClock.elapsedRealtime() - offset
    }

    //Record the offset
    fun saveOffset() {
        offset = SystemClock.elapsedRealtime() - stopwatch.base
    }
}
```

Make sure the stopwatch starts from the correct time, then start it.

Save the time on the stopwatch, then stop it running.

Set the stopwatch time back to 0.

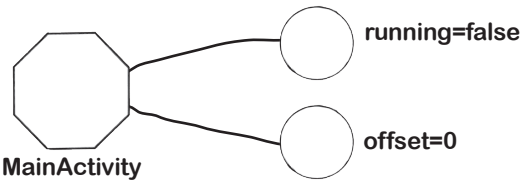
setBaseTime() and saveOffset() are convenience methods to make the code a bit more readable.



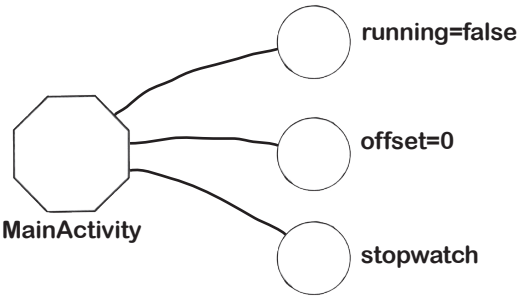

Let's run through how we expect the code to work.

# What happens when you run the app

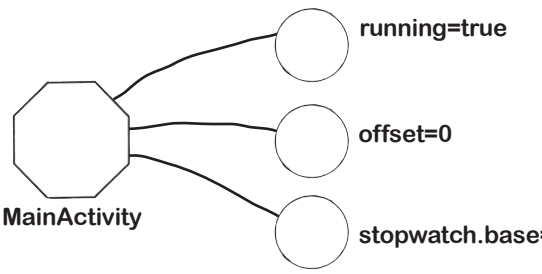
1 **The user runs the app, and MainActivity starts.**
The `running` and `offset` properties are initialized. `running` is set to `false`, and `offset` is set to 0.




2 **MainActivity's onCreate method gets called.**
The *activity_main.xml* layout is linked to the activity, and the `stopwatch` property is given a reference to the `Chronometer` view.



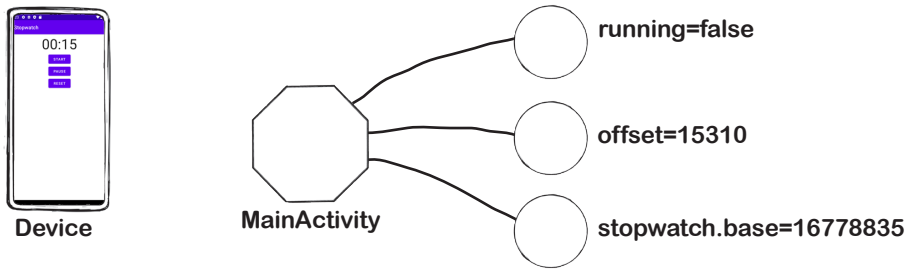

3 **The user clicks the Start button.**
The `stopwatch.base` property is set to `SystemClock.elapsedRealtime()`, its `start()` method is called, and `running` is set to `true`. The stopwatch starts running.



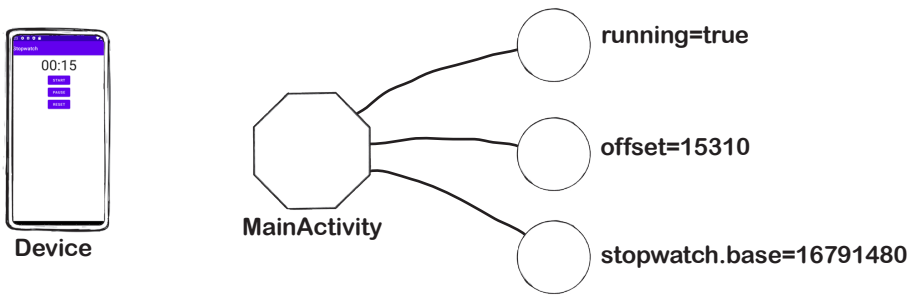

The stopwatch uses the number of milliseconds since the device was booted as its base time.

# The story continues

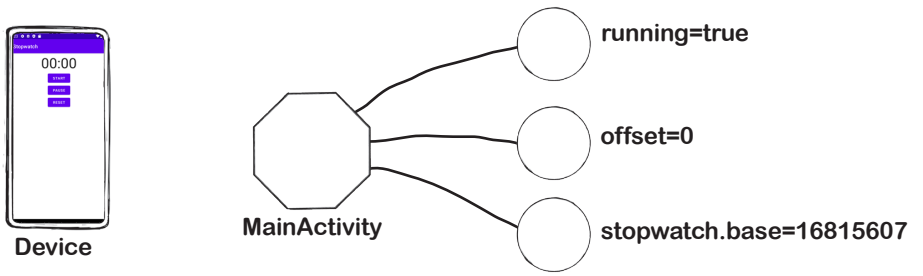
4 **The user clicks the Pause button.**
`MainActivity` updates the `offset` property and calls the stopwatch `stop()` method and `running` is set to `false`. The stopwatch pauses.




5 **The user clicks the Start button again.**
`MainActivity` uses the `offset` value to adjust the `stopwatch.base` property, calls its `start()` method, and sets `running` to `true`. The stopwatch starts running again.



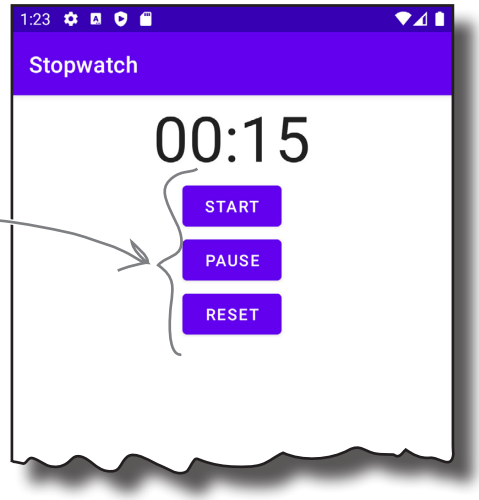

6 **The user clicks the Reset button.**
The `offset` property is set to `0`, and the `stopwatch.base` property is reset to `SystemClock.elapsedRealtime()`.




Let's take the app for a test drive.

# Test Drive

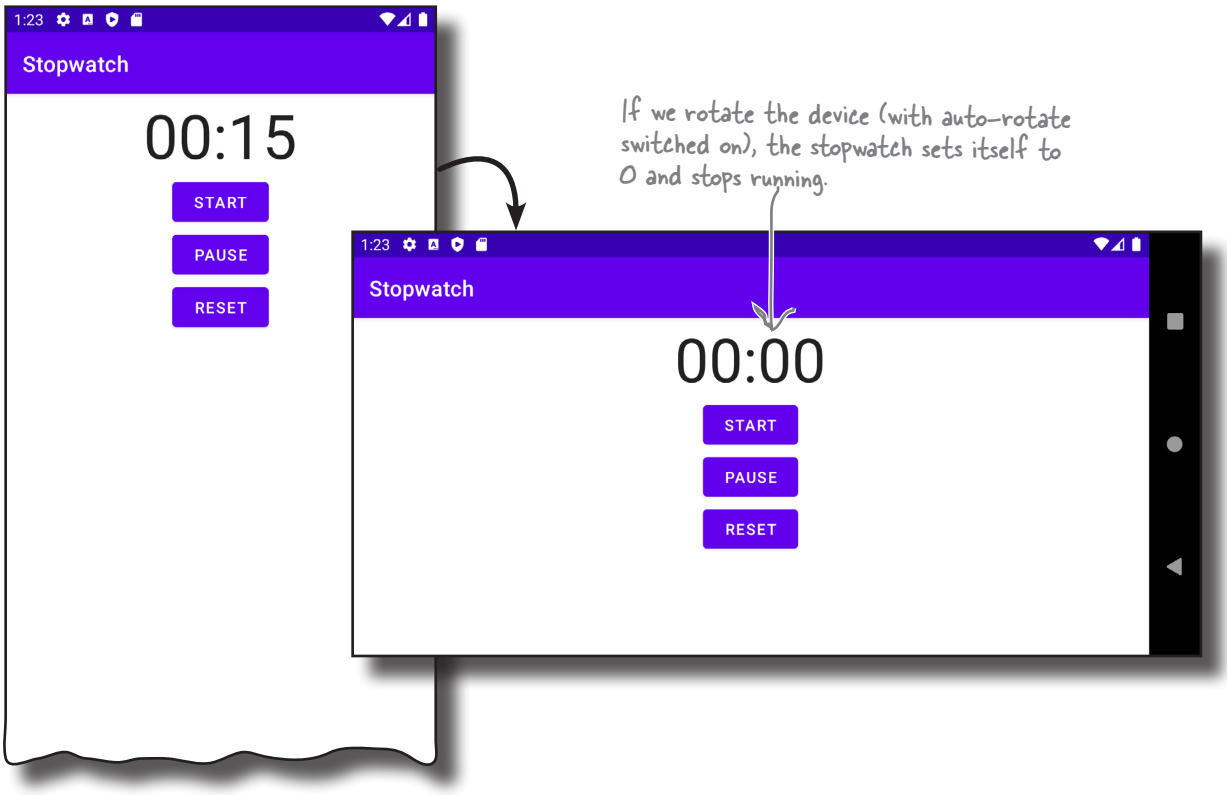
When we run the app, it seems to work as we expect it to. We can start, pause, and reset the stopwatch without any problems.




These buttons work as you'd expect. The Start button starts the stopwatch, the Pause button pauses it, and the Reset button sets the stopwatch back to 0.

## But there's just one problem...

When we turn the device so the screen rotates, the app goes wrong. The stopwatch sets itself back to 0 and stops running.

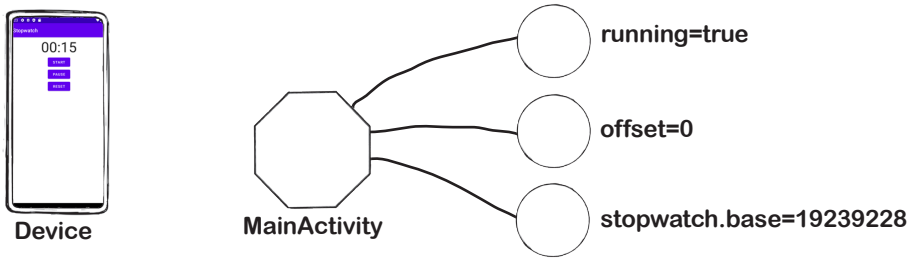



So why does the app break when we rotate the screen? Let's go behind the scenes to find out more.

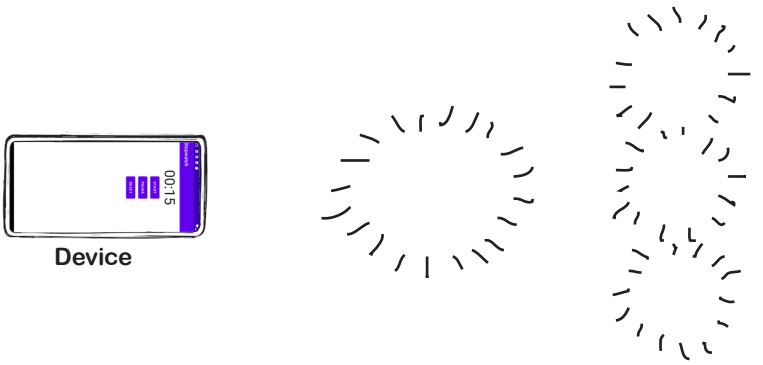
# What happens when the app runs

The following things happen when the code runs:

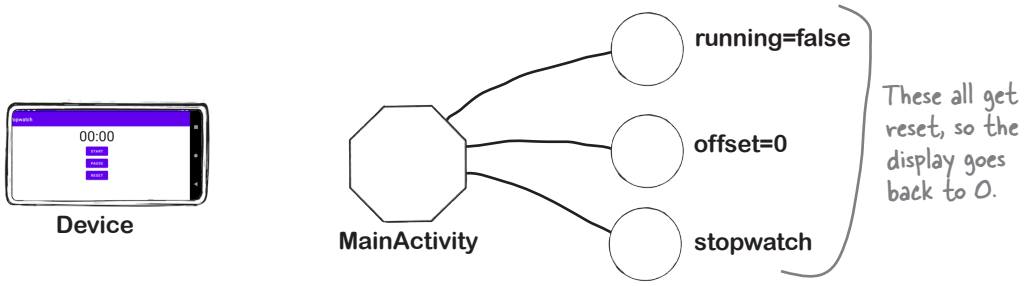
1. **Click on the Start button to start the stopwatch.**
The stopwatch starts, and the `running` property is set to `true`.




2. **Rotate the device screen.**
Android sees that the screen orientation has changed, and it **destroys `MainActivity`**. Its property values are lost.



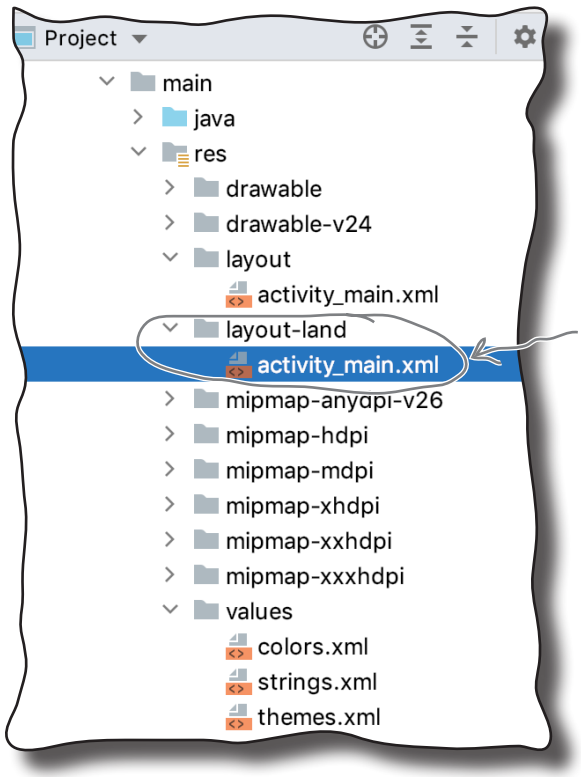

3. **MainActivity restarts.**
The activity is recreated, all its properties are reinitialized, and `onCreate()` runs again. The stopwatch doesn't restart as `onCreate()` doesn't tell it to.



# Rotating the screen changes the device configuration

When Android runs your app and starts an activity, it takes into account the **device configuration**. By this we mean the configuration of the physical device (such as the screen size, screen orientation, and whether there's a keyboard attached) and also configuration options specified by the user (such as the locale).

Android needs to know what the device configuration is when it starts an activity because the configuration can impact what resources are needed for the application. You may need to use a different layout if the device screen is in landscape mode rather than portrait, for instance, and a different set of `String` values if the locale is France.




Android apps can have multiple versions of the same resource file. If you want to use a different version of activity_main.xml when the device is in landscape mode, for example, you put the file in the layout-land resources folder.

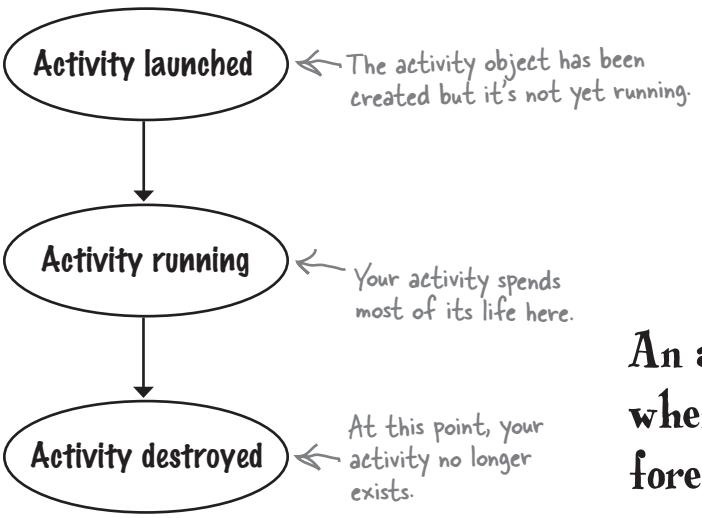
When the device configuration changes, anything that displays a user interface needs to be updated to match the new configuration. If you rotate the screen, for example, Android spots the orientation has changed, and classes this as a change to the device configuration. It destroys the current activity, and then recreates it so that resources appropriate to the new configuration get picked up.

The device configuration includes options specified by the user (such as the locale), and options relating to the physical device (such as the orientation and screen size). A change to any of these options results in the activity being destroyed and then recreated.

# An activity's states

When Android creates and destroys an activity, the activity moves from being launched to running to being destroyed.

The main state of an activity is when it's ***running*** or ***active***. An activity is running when it's in the foreground of the screen, it has the focus, and the user can interact with it. The activity spends most of its life in this state. An activity starts running after it has been launched, and at the end of its life, the activity is ***destroyed***.




When an activity moves from being launched to being destroyed, it triggers key activity lifecycle methods: the **`onCreate()`** and **`onDestroy()`** methods. These are lifecycle methods that the activity inherits, and that you can override.

The `onCreate()` method gets called immediately after your activity is launched. This method is where you do all your normal activity setup such as calling `setContentView()`. You should always override this method. If you *don't* override it, you won't be able to tell Android what layout your activity should use.

The `onDestroy()` method is the final call you get before the activity is destroyed. There are a number of situations in which an activity can get destroyed—for example, if it's been told to finish, if the activity is being recreated due to a change in device configuration, or if Android has decided to destroy the activity in order to save space.

We'll take a closer look at how these methods fit into the activity states on the next page.

An activity is running when it's in the foreground of the screen.

onCreate() gets called when the activity is first created, and it's where you do your normal activity setup.

onDestroy() gets called just before your activity gets destroyed.

# The activity lifecycle: from create to destroy

Here's an overview of the activity lifecycle from birth to death. As you'll see later in the chapter, we've left out some of the details because at this point we're just focusing on the `onCreate()` and `onDestroy()` methods.

1. **The activity gets launched.**
   The activity object is created and its constructor is run.

2. **The onCreate() method runs immediately after the activity is launched.**
   The `onCreate()` method is where any initialization code should go, as this method always gets called after the activity has launched but before it starts running.

3. **An activity is running when it's visible in the foreground and the user can interact with it.**
   This is where an activity spends most of its life.

4. **The onDestroy() method runs immediately before the activity is destroyed.**
   The `onDestroy()` method enables you to perform any final cleanup such as freeing up resources.

5. **After the onDestroy() method has run, the activity is destroyed.**
   The activity ceases to exist.

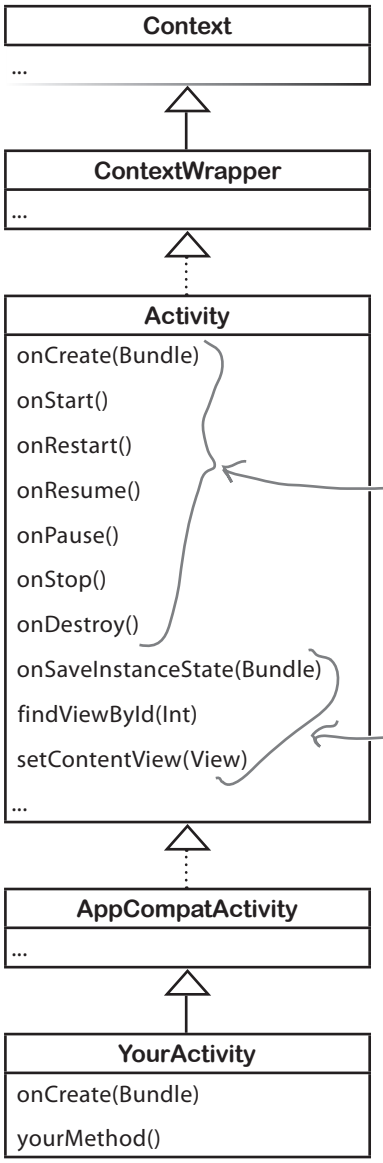
If your device is extremely low on memory, onDestroy() might not get called before the activity is destroyed.

The `onCreate()` and `onDestroy()` methods are two of the activity lifecycle methods. So where do these methods come from?

# Your activity inherits the lifecycle methods

The activity lifecycle methods are defined in the `android.app.Activity` class. Each activity you create is a subtype of this class, so it inherits these methods.

Here's a diagram showing the main classes in your activity's class hierarchy:




**Context abstract class**
(android.content.Context)
This gives you access to global information about the application environment, such as application resources and operations.

**ContextWrapper class**
(android.content.ContextWrapper)
This is a proxy implementation for the `Context`.

**Activity class**
(android.app.Activity)
The `Activity` class implements default versions of the lifecycle methods. It also defines methods such as `findViewById()` and `setContentView()`.

These are the activity lifecycle methods. You'll learn more about them through the rest of the chapter.

These aren't lifecycle methods, but they're still very useful. You've already used most of them in earlier chapters.

**AppCompatActivity class**
(androidx.appcompat.app.AppCompatActivity)
This is an activity that's part of Android Jetpack. It lets you use new features in older versions of Android.

**YourActivity class**
(com.hfad.foo)
Most of the behavior of your activity is handled by superclass methods it inherits. All you do is override the methods you need.

Now that you know more about how your activity gets its lifecycle methods, let's see how you deal with device configuration changes.

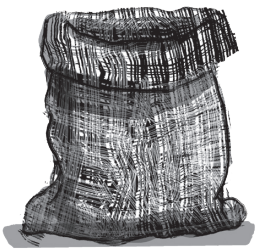
# Save the current state in a Bundle

As you saw earlier, the Stopwatch app went wrong when we rotated the device. `MainActivity` was destroyed and recreated, so all the property values were lost, and the `Chronometer` view was reset.

To fix this bug, we'll save the state of the `Chronometer` and property values before the activity gets destroyed, and restore them when the activity gets recreated. We'll save the values in a type of object called a **`Bundle`**.

A `Bundle` is a type of object that's used to hold key/value pairs. Before the activity is destroyed, Android lets you put key/value pairs into a `Bundle`. This `Bundle` is then picked up by the new instance of the activity when it's recreated. Using a `Bundle` therefore gives you a way of reinstating an activity's state when you rotate the device screen.

Before we show you how the `Bundle` is passed between activity instances, let's look at how you add and retrieve values.




A Bundle is used to save name/value pairs. Bundles are useful for restoring the state of any property values when your activity is destroyed and recreated.

## Add values using put methods

You add key/value pairs to a `Bundle` using `put` methods. The `putLong` method puts a `Long` value into a `Bundle`, `putBoolean()` adds a `Boolean`, and so on. The following code, for example, puts a key of "answer" and an `Int` value of 42 into a `Bundle` named `bundle`:

```
bundle.putInt("answer", 42)
```

You can save multiple key/value pairs to a `Bundle`. Each key must be of type `String?`, and the value must match the method call. You can't, say, put an `Int` value into a `Bundle` using the `putString()` method without upsetting the compiler.

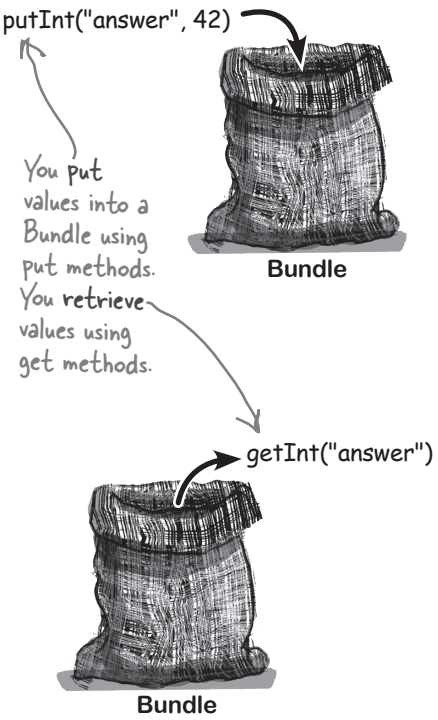
## Retrieve values using get methods

Once you've put a key/value pair into a `Bundle`, you can retrieve the value using the `Bundle`'s `get` methods. The `getLong()` method lets you get a `Long` value, for example, and you use the `getBoolean()` method to retrieve a `Boolean`.

Each `get` method takes one parameter: the name of the key whose value you want to get. The following code, for example, gets the value that's saved against a key of "answer" in a `Bundle` named `bundle`:

```
val answerOfLife = bundle.getInt("answer")
```

Now that you know how to add and retrieve values using a `Bundle`, let's find out how to use one to save and restore an activity's state.



# Save the state using onSaveInstanceState()

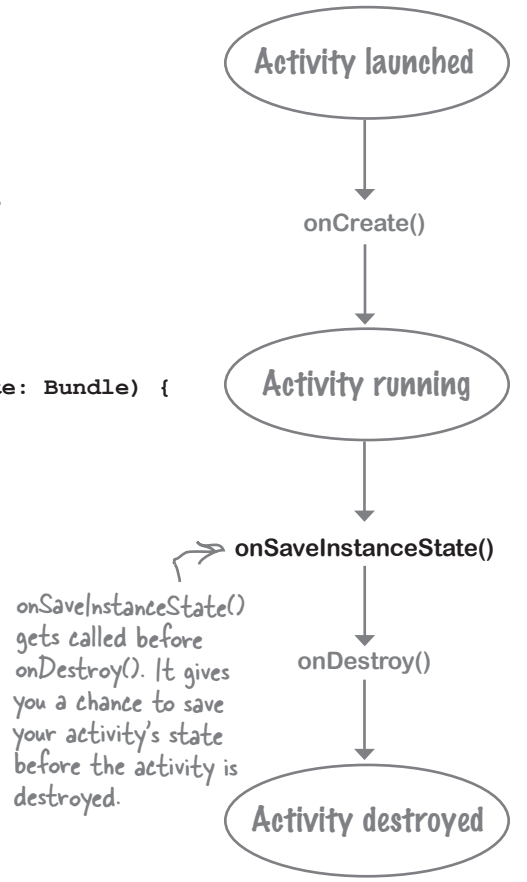
Bundles are passed from one instance of an activity to another using two methods: **onSaveInstanceState()** and **onCreate()**. onSaveInstanceState() lets you add values to a Bundle before the activity is destroyed, and onCreate() picks the Bundle up again when the activity has been recreated.

Each activity inherits the onSaveInstanceState() method from its Activity superclass. It gets called just before the activity is destroyed, and takes one parameter: a Bundle.

You override the onSaveInstanceState() method in your activity using code like this:

```
override fun onSaveInstanceState(savedInstanceState: Bundle) {
    savedInstanceState.putInt("answer", 42)
    super.onSaveInstanceState(savedInstanceState)
}
```

The above example puts a key of "answer" and an Int value of 42 into the onSaveInstanceState() method's Bundle (named savedInstanceState). It then calls the superclass version of onSaveInstanceState(), which saves the state of the view hierarchy.




## Restore the state using onCreate()

As you already know, the onCreate() method gets called when the activity is recreated, and it takes one parameter, a Bundle?. If the activity's being created from scratch, the Bundle? will be null, but if the activity's being recreated, it's the same Bundle object that was used by onSaveInstanceState().

You use the onCreate() method to get values from the Bundle using code like this:

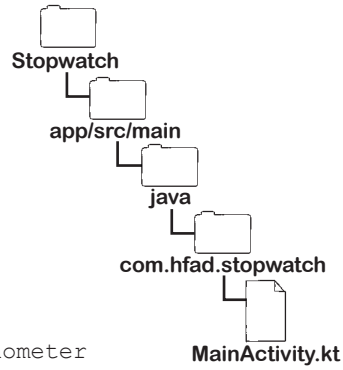
```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    if (savedInstanceState != null) {
        var answer = savedInstanceState.getInt("answer")
    }
    ...
}
```

We only want to reinstate the value of answer if the activity's been destroyed and recreated, so the Bundle passed to onCreate() is not null.

So what code changes do we need to make to the Stopwatch app?

# The updated MainActivity.kt code

We've updated the `MainActivity` code so that if the user rotates the device, its state gets saved in a `Bundle` using the `onSaveInstanceState()` method, and restored via `onCreate()`. Update your *MainActivity.kt* code to include these changes (below in bold):




```
package com.hfad.stopwatch

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.os.SystemClock
import android.widget.Button
import android.widget.Chronometer

class MainActivity : AppCompatActivity() {

    lateinit var stopwatch: Chronometer  //The chronometer
    var running = false  //Is the chronometer running?
    var offset: Long = 0  //The base offset for the chronometer

    //Add key Strings for use with the Bundle
    val OFFSET_KEY = "offset"
    val RUNNING_KEY = "running"
    val BASE_KEY = "base"

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

        //Get a reference to the stopwatch
        stopwatch = findViewById<Chronometer>(R.id.stopwatch)

        //Restore the previous state
        if (savedInstanceState != null) {
            offset = savedInstanceState.getLong(OFFSET_KEY)
            running = savedInstanceState.getBoolean(RUNNING_KEY)
            if (running) {
                stopwatch.base = savedInstanceState.getLong(BASE_KEY)
                stopwatch.start()
            } else setBaseTime()
        }
```

We're going to use these three constants as the names for any values we add to the Bundle.

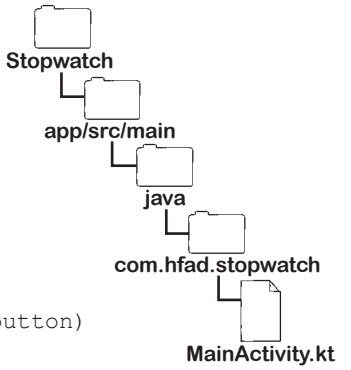
Restore the state of offset and running.

Set the time on the stopwatch, and start it if it should be running.

The code continues on the next page.

# MainActivity.kt continued

```
        //The start button starts the stopwatch if it's not running
        val startButton = findViewById<Button>(R.id.start_button)
        startButton.setOnClickListener {
            if (!running) {
                setBaseTime()
                stopwatch.start()
                running = true
            }
        }

        //The pause button pauses the stopwatch
        val pauseButton = findViewById<Button>(R.id.pause_button)
        pauseButton.setOnClickListener {
            if (running) {
                saveOffset()
                stopwatch.stop()
                running = false
            }
        }

        //The reset button sets the offset and stopwatch to 0
        val resetButton = findViewById<Button>(R.id.reset_button)
        resetButton.setOnClickListener {
            offset = 0
            setBaseTime()
        }
    }

    override fun onSaveInstanceState(savedInstanceState: Bundle) {
        savedInstanceState.putLong(OFFSET_KEY, offset)
        savedInstanceState.putBoolean(RUNNING_KEY, running)
        savedInstanceState.putLong(BASE_KEY, stopwatch.base)
        super.onSaveInstanceState(savedInstanceState)
    }

    ...
}
```



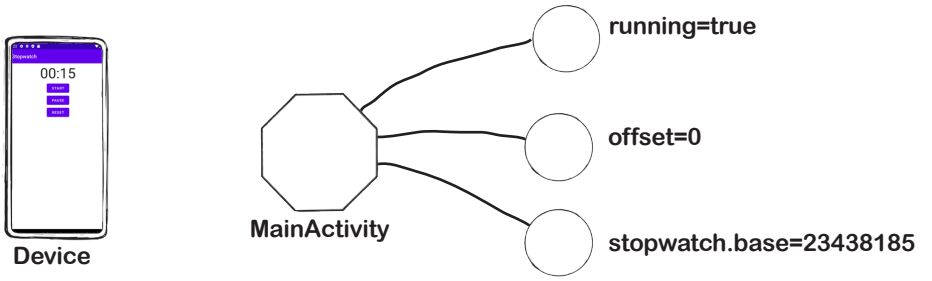

Use the onSaveInstanceState method to save the state of the offset, running, and stopwatch.base properties.

We've left out the code for the saveOffset() and setBaseTime() methods as you don't need to update them.

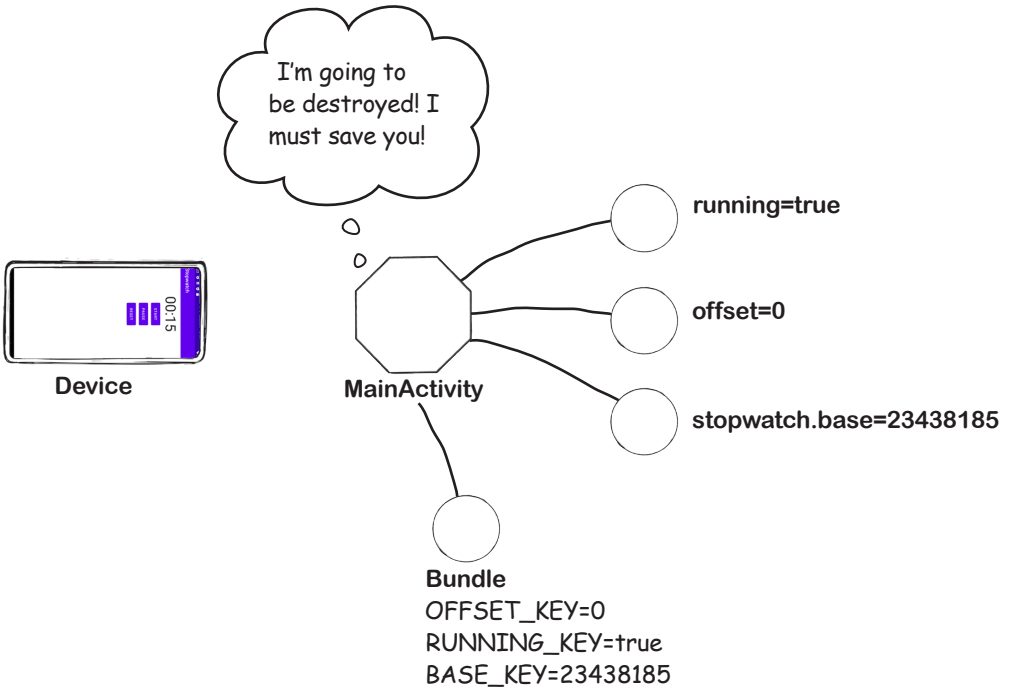
Let's run through what the code does, and take the app for a test drive.

# What happens when you run the app

1 **The user starts the app, and clicks on the Start button.**
The stopwatch starts and the `running` property is set to `true`.

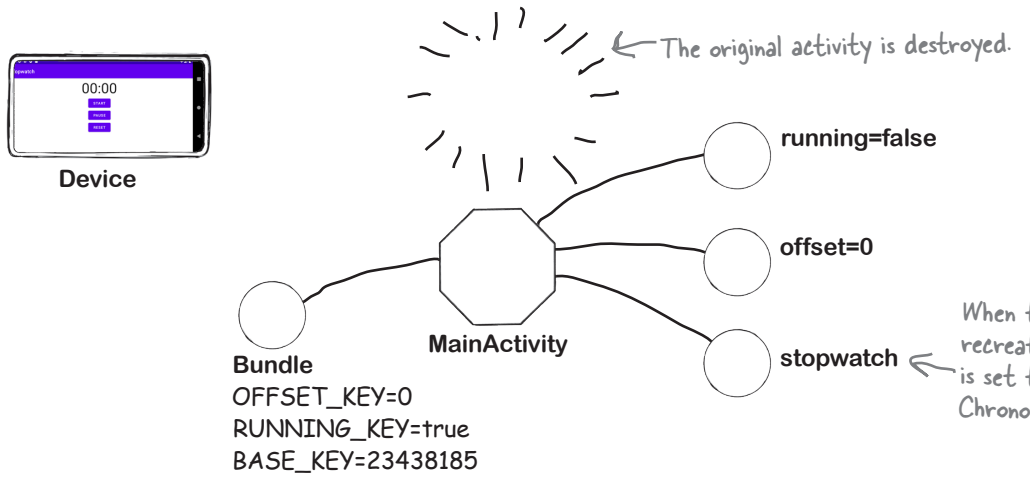



2 **The user rotates the device screen.**
Android views this as a configuration change, and gets ready to destroy the activity. Before the activity is destroyed, `onSaveInstanceState()` gets called. The `onSaveInstanceState()` method saves the `offset`, `running` and `stopwatch.base` values to a `Bundle`.

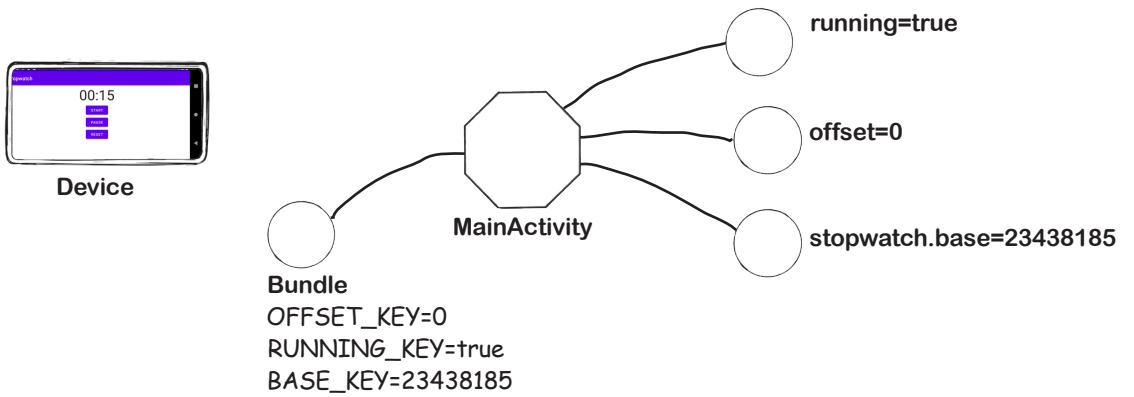


# The story continues

3 **Android destroys the activity, and then recreates it.**
The activity's property values get reset.




4 **The onCreate() method gets called.**
It uses the values saved in the `Bundle` to set the property values.



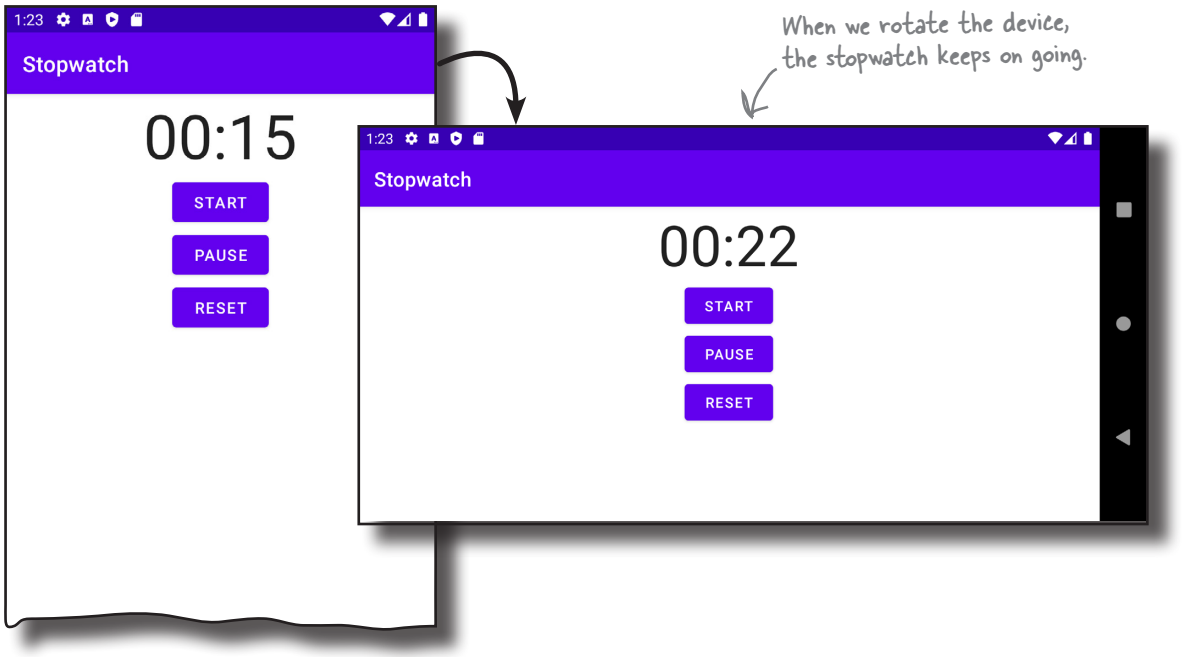

5 **The stopwatch starts running again and displays the correct time.**

Let's take the app for a test drive.

# Test Drive

Make the changes to your activity code, then run the app.

When we click on the Start button, the timer starts, and it continues when we rotate the device screen.




## there are no Dumb Questions

**Q: Remind me, why does Android recreate the activity just because I rotate the screen?**

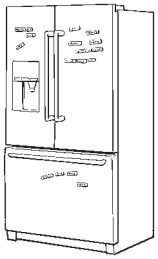
A: The `onCreate()` method is normally used to set up the screen. If your code in `onCreate()` depends upon the screen configuration (for example, if you have different layouts for landscape and portrait), then you would want `onCreate()` to be called every time the configuration changes. Also, if the user changes their locale, you might want to recreate the UI in the local language.

**Q: Why doesn't Android store every instance variable automatically? Why do I have to write all that code myself?**

A: It's because you might not want every instance variable to be stored. For example, you might have a variable that stores the current screen width. You would want that variable to be recalculated the next time `onCreate()` is called.

**Q: You mentioned `AppCompatActivity` is part of Android Jetpack. Is it in a separate library? Do I need to update the *build.gradle* files?**

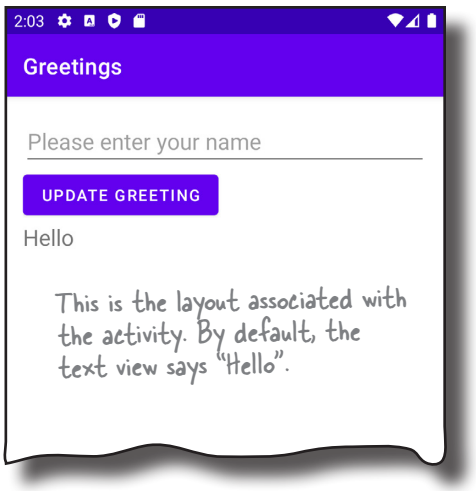
A: `AppCompatActivity` is part of the `AppCompat` library. We've not told you to update the *build.gradle* files because Android Studio should have already included this library for you. If you want to double-check, look for an `AppCompat` dependency in the app *build.gradle* file.



# Activity Magnets

The activity below is for a new Greetings app. You enter your name in an edit text (with the ID `name`), and when you click on a button (with the ID `button`), the text in the text view (with the ID `hello`) is updated to include your name.

When you run the app and rotate the screen, the text in the `hello` text view gets lost. Use the magnets below to make sure the text gets reinstated in this situation.



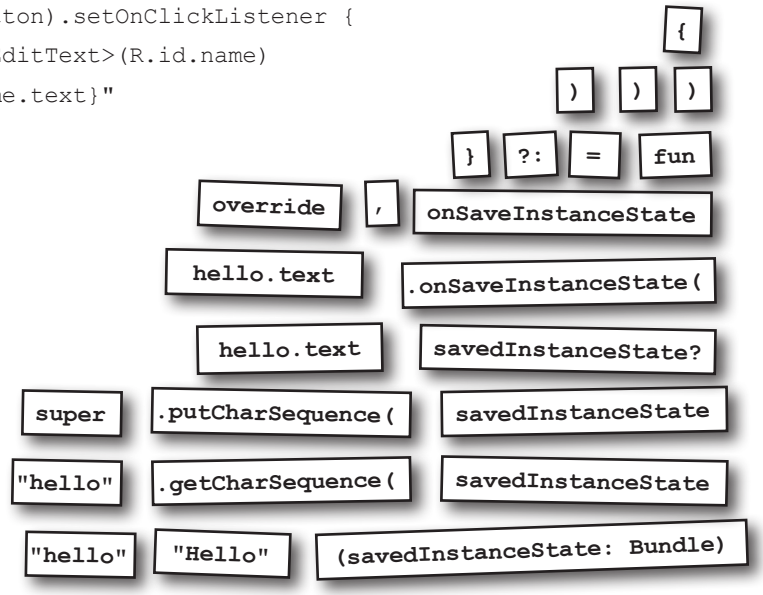

```
...
class MainActivity : AppCompatActivity() {

    lateinit var hello: TextView

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

        hello = findViewById<TextView>(R.id.hello)
```

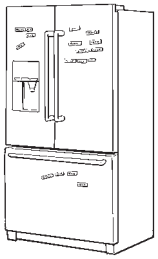
Some code needs to go here.

```
        findViewById<Button>(R.id.button).setOnClickListener {
            val name = findViewById<EditText>(R.id.name)
            hello.text = "Hello ${name.text}"
        }
    }
```

Some more code needs to go here.

```
}
```

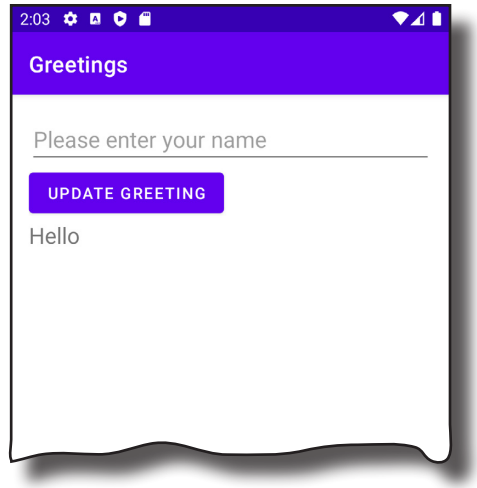




# Activity Magnets Solution

The activity below is for a new Greetings app. You enter your name in an edit text (with the ID `name`), and when you click on a button (with the ID `button`), the text in the text view (with the ID `hello`) is updated to include your name.

When you run the app and rotate the screen, the text in the `hello` text view gets lost. Use the magnets below to make sure the text gets reinstated in this situation.




```
...
class MainActivity : AppCompatActivity() {

    lateinit var hello: TextView

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

        hello = findViewById<TextView>(R.id.hello)
```

Reinstate the hello text if savedInstanceState is not null.

```
hello.text = savedInstanceState?.getCharSequence("hello") ?: "Hello"
```

```
        findViewById<Button>(R.id.button).setOnClickListener {
            val name = findViewById<EditText>(R.id.name)
            hello.text = "Hello ${name.text}"
        }
    }
```

Override the onSaveInstanceState method.

```
override fun onSaveInstanceState(savedInstanceState: Bundle) {
    savedInstanceState.putCharSequence("hello", hello.text)
    super.onSaveInstanceState(savedInstanceState)
}
```

Call the method in the superclass.

Put the value of hello.text in the savedInstanceState Bundle.

```
}
```

# There's more to an activity's life than create and destroy

So far we've looked at the create and destroy parts of the activity lifecycle (and a little bit in between), and you've seen how to deal with configuration changes such as screen orientation. But there are other events in an activity's life that you might want to deal with to get the app to behave the way you want.

As an example, suppose the stopwatch is running and you get a phone call. Even though the stopwatch isn't visible, it will continue running. But what if you want the stopwatch to pause while it's hidden, and resume once the app is visible again?

Even if you don't really want your stopwatch to behave like this, just play along with us. It's a great excuse to look at more lifecycle methods.

## Start, stop, and restart

Fortunately, it's easy to handle actions that relate to an activity's visibility if you use the right lifecycle methods. In addition to the `onCreate()` and `onDestroy()` methods, which deal with the overall lifecycle of the activity, there are other lifecycle methods that deal with an activity's visibility.

Specifically, there are three key lifecycle methods that can run when an activity becomes visible or invisible to the user: `onStart()`, `onStop()`, and `onRestart()`. Just as with `onCreate()` and `onDestroy()`, your activity inherits them from the Android `Activity` class.

Here's an overview of these methods:

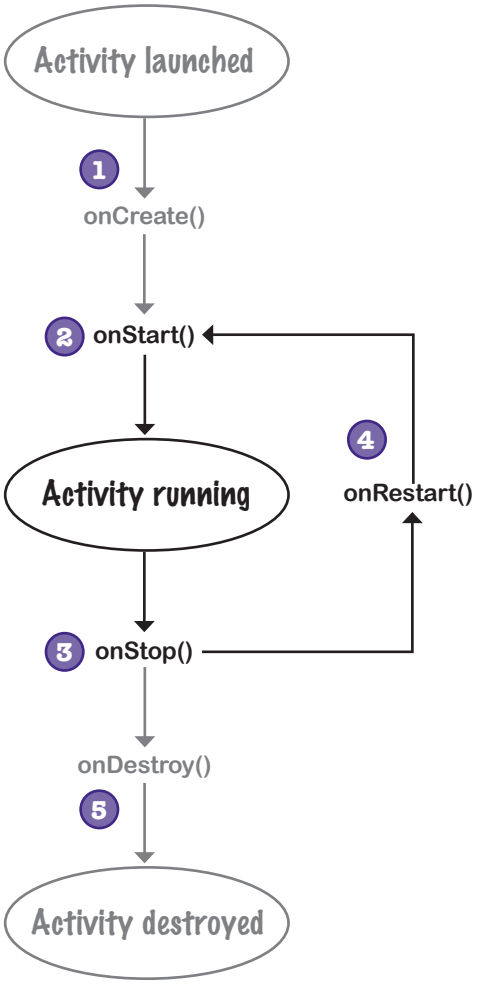
**An activity's state is stopped if it's completely hidden by another activity and isn't visible to the user. The activity still exists in the background and maintains all state information.**

| Method | When it's called |
|---|---|
| **onStart()** | When your activity becomes visible. |
| **onStop()** | When the activity has stopped being visible. This might be because it's completely hidden by another activity that's appeared on top of it, or because the activity is about to be destroyed. |
| **onRestart()** | After the activity has been made invisible, before it gets made visible again. |

We'll take a closer look at how these fit in with the `onCreate()` and `onDestroy()` methods on the next page.

# The visible lifecycle

Let's build on the lifecycle diagram you saw earlier in the chapter, this time including the `onStart()`, `onStop()`, and `onRestart()` methods (the new bits are in black):




1. **The activity gets launched, and the onCreate() method runs.**
   Any activity initialization code in the `onCreate()` method runs. At this point, the activity isn't yet visible, as no call to `onStart()` has been made.

2. **The onStart() method runs. It gets called when the activity is about to become visible.**
   After the `onStart()` method has run, the user can see the activity on the screen.

3. **The onStop() method runs when the activity stops being visible to the user.**
   After the `onStop()` method has run, the activity is no longer visible.

4. **If the activity becomes visible to the user again, the onRestart() method gets called followed by onStart().**
   The activity may go through this cycle many times if the activity repeatedly loses visibility and then becomes visible again.

5. **Finally, the activity is destroyed.**
   The `onStop()` method will get called before `onDestroy()`.

# We need to implement two more lifecycle methods

There are two things we need to do to update the Stopwatch app. First, we need to implement the activity's `onStop()` method so that the stopwatch pauses when the app isn't visible. Once we've done that, we need to implement `onRestart()` so that the stopwatch starts again when the app is visible.

Let's start with the `onStop()` method.

## Implement onStop() to pause the stopwatch

You override the `onStop()` method by adding the following method to your activity:

```
override fun onStop() {
    super.onStop()
    //Code to run when the activity stops
}
```

The line:

```
super.onStop()
```

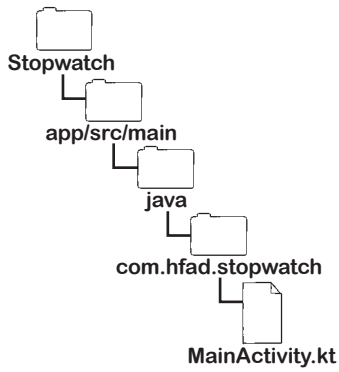
calls the `onStop()` method in the `Activity` superclass. You need to add this line of code whenever you override the `onStop()` method to make sure the activity gets to perform any other actions in the superclass `onStop()` method. If you bypass this step, Android will generate an exception. This applies to all of the lifecycle methods; if you override any of the `Activity` lifecycle methods in your activity, you must call the superclass method or the compiler will get upset.

We need to get the stopwatch to stop if it's running when the `onStop()` method is called. To do this, we need to call the `saveOffset()` method—which makes sure we can later restart the stopwatch from the same displayed time—and then stop the stopwatch. Here's the code:

```
override fun onStop() {
    super.onStop()
    if (running) {
        saveOffset()
        stopwatch.stop()
    }
}
```

We'll add this code to MainActivity.kt a couple pages ahead.

So now the stopwatch stops when the activity is no longer visible. The next thing we need to do is get the stopwatch to start again when the activity becomes visible.




**When you override any activity lifecycle method in your activity, you need to call the Activity superclass method or the compiler will complain.**

# Restart the stopwatch when the app becomes visible

When the activity becomes visible again, two key lifecycle methods get called: `onStart()` and `onRestart()`.

`OnStart()` is called when the activity is about to become visible. This can be when the activity is first created, or when it becomes visible again after losing its visibility.

You implement the `onStart()` method using code like this:

```
override fun onStart() {
    super.onStart()
    //Code to run when the activity starts
}
```

As with the `onStop()` method, the line:

```
super.onStart()
```

calls the `onStart()` method in the `Activity` superclass.

The `onRestart()` method is called before `onStart()`, but only when the activity becomes visible again after previously losing visibility. It's *not* called when the activity first becomes visible.

You implement the `onRestart()` method using code like this:

```
override fun onRestart() {
    super.onRestart()
    //Code to run when the activity restarts
}
```

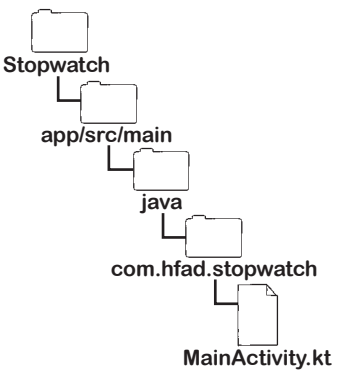
In the Stopwatch app, we want the stopwatch to start running again when the activity regains visibility, so we'll implement the `onRestart()` method. The code looks like this:

```
override fun onRestart() {
    super.onRestart()
    if (running) {
        setBaseTime()
        stopwatch.start()
        offset = 0
    }
}
```

We'll add this method to MainActivity.kt on the next page.

**onStart() gets called whenever the activity becomes visible.**

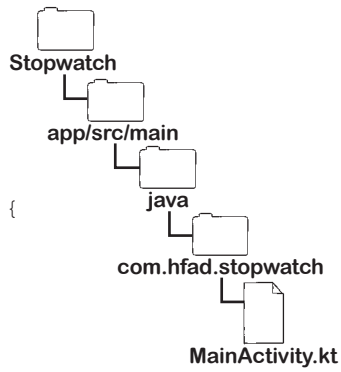
**onRestart() only gets called when the activity becomes visible again after losing visibility.**



# The updated MainActivity.kt code

Here is the code for *MainActivity.kt*, including the `onStop()` and `onRestart()` methods; update the code so it includes the changes below (in bold):

```
...
class MainActivity : AppCompatActivity() {

    lateinit var stopwatch: Chronometer  //The chronometer
    var running = false  //Is the chronometer running?
    var offset: Long = 0  //The base offset for the chronometer

    //Add key Strings for use with the Bundle
    val OFFSET_KEY = "offset"
    val RUNNING_KEY = "running"
    val BASE_KEY = "base"

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
    }

    override fun onStop() {
        super.onStop()
        if (running) {
            saveOffset()
            stopwatch.stop()
        }
    }

    override fun onRestart() {
        super.onRestart()
        if (running) {
            setBaseTime()
            stopwatch.start()
            offset = 0
        }
    }

    ...
}
```



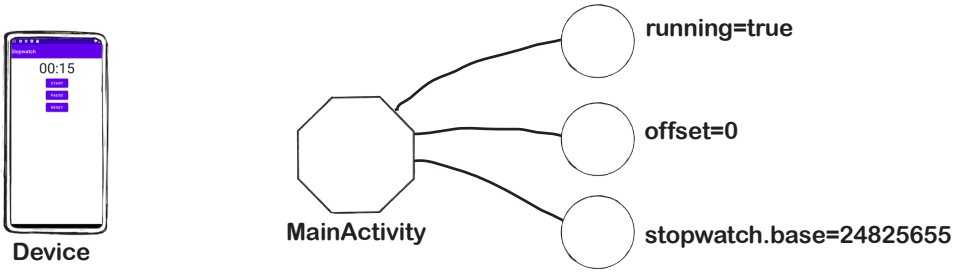

You don't need to change the onCreate() method so we've left out the code.

Make sure you add the onStop() and onRestart() methods to your code.

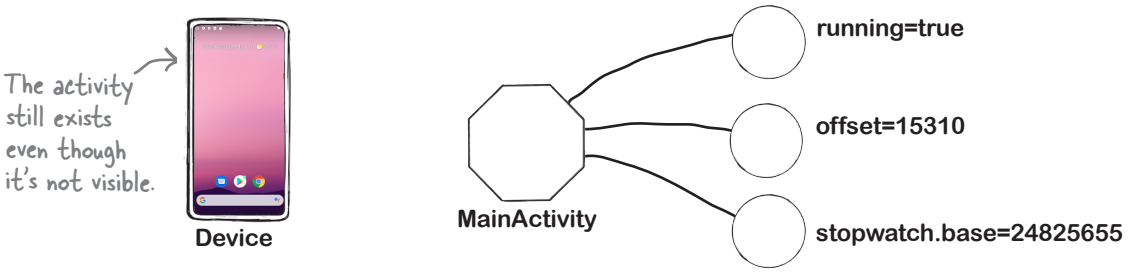
We've left out the code for the saveOffset(), setBaseTime(), and onSaveInstanceState() methods as you don't need to update them.

# What happens when you run the app

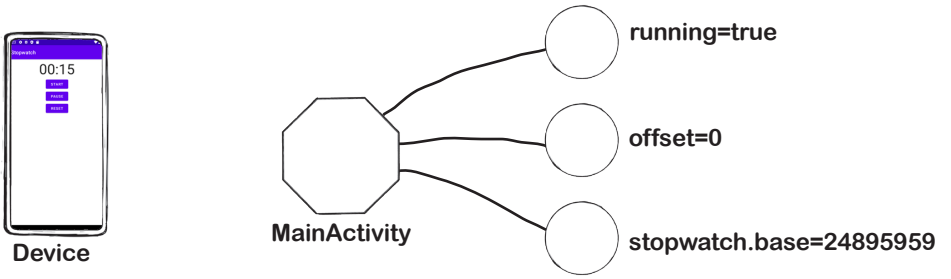
1. **The user starts the app, and clicks the Start button to set the stopwatch going.**
The stopwatch starts, and the `running` property is set to `true`.




2. **The user navigates to the device's home screen so the Stopwatch app is no longer visible.**
The `onStop()` method gets called, the `offset` property is updated, and the stopwatch is stopped.

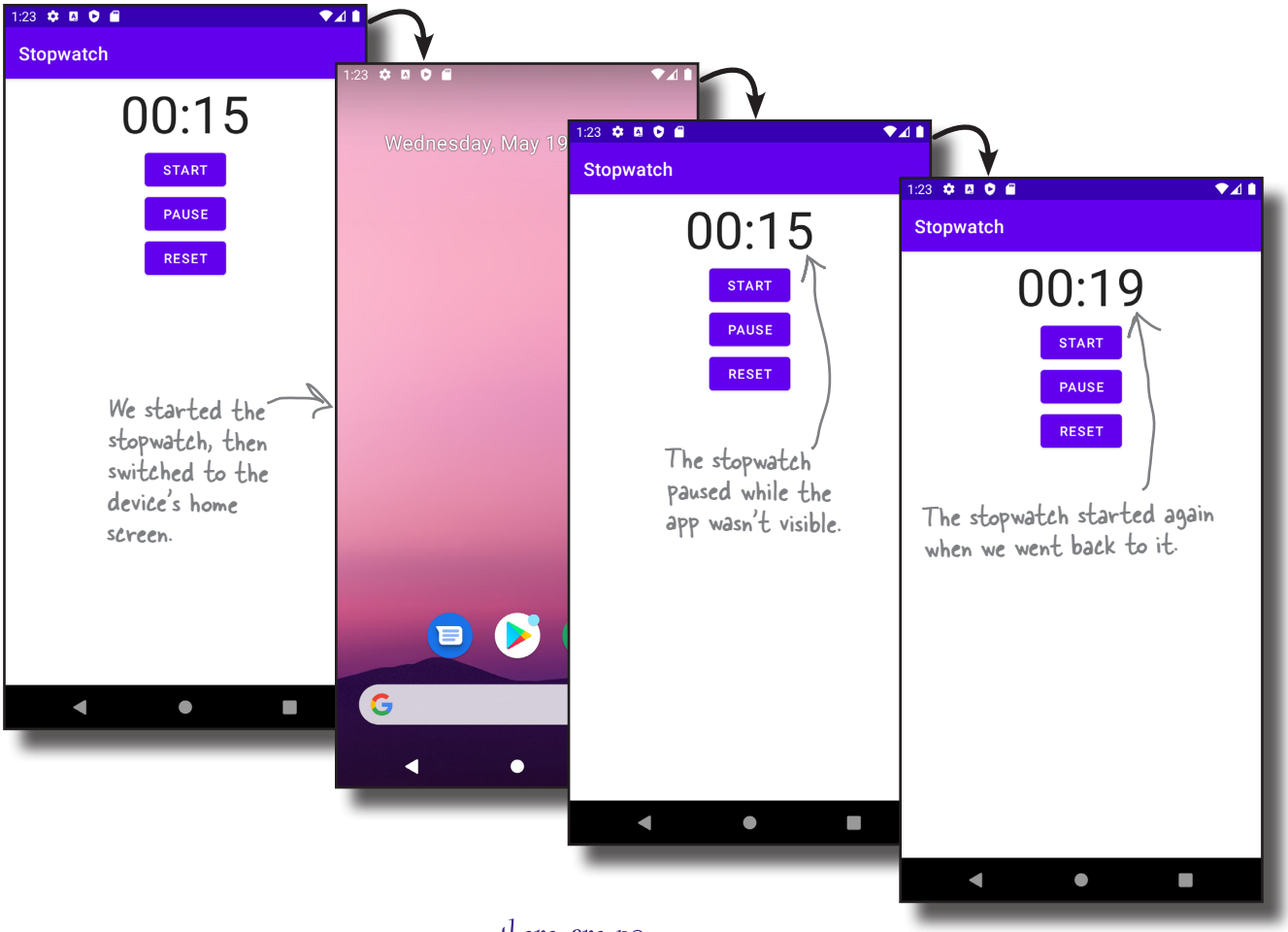



3. **The user navigates back to the Stopwatch app.**
The `onRestart()` method gets called, the `stopwatch.base` property is updated, the stopwatch starts, and the `offset` property is set to `0`. The stopwatch starts from the correct time.



# Test Drive

When we run the app and click on the Start button the timer starts. It stops when the app is no longer visible, and restarts when the app becomes visible again.




## there are no Dumb Questions

**Q:** **Could we have used the `onStart()` method instead of `onRestart()` to set the stopwatch running again?**

**A:** Yes. When an activity becomes visible after having previously been invisible, both `onRestart()` and `onStart()` get called, so we could have used either method.

**Q:** **So why did we use `onRestart()` and not `onStart()`?**

**A:** Unlike `onRestart()`, `onStart()` also gets called when the activity first becomes visible after it's been created. This means that it's slightly more efficient to use `onRestart()` than it is to use `onStart()`.

# Exercise

The activity code on the right changes the value of a property named `x` at various stages of the activity lifecycle. See if you can determine which lifecycle methods get called and the final value of `x` after each of the actions below.

1. The user opens the app and it starts running.

2. The user opens the app and it starts running. The user then rotates the device.

3. The user opens the app and it starts running. The user navigates to the Android Home screen, then returns to the app.

```
package com.hfad.lifecycleexercise

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

class MainActivity : AppCompatActivity() {

    var x = 0

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        x = savedInstanceState?.getInt("x") ?: 2
    }

    override fun onRestart() {
        super.onRestart()
        x -= 7
    }

    override fun onStart() {
        super.onStart()
        x += 6
    }

    override fun onStop() {
        super.onStop()
        x *= 2
    }

    override fun onSaveInstanceState(savedInstanceState: Bundle) {
        savedInstanceState.putInt("x", x + 1)
        super.onSaveInstanceState(savedInstanceState)
    }
}
```

# Exercise Solution

The activity code on the right changes the value of a property named `x` at various stages of the activity lifecycle. See if you can determine which lifecycle methods get called and the final value of `x` after each of the actions below.

1. The user opens the app and it starts running.

   onCreate(): x = 2

   onStart():. x = 8

   The final value of x is 8.

2. The user opens the app and it starts running. The user then rotates the device.

   onCreate(): x = 2

   onStart(): x = 8

   onStop(): x = 16

   onSaveInstanceState(): x = 16

   onCreate(): x = 17

   onStart(): x = 23

   The final value of x is 23.

   While onSaveInstanceState() saves a value of 17 to the Bundle, x isn't set until the onCreate() method runs.

   Here, onRestart() doesn't get called because the app doesn't lose visibility.

3. The user opens the app and it starts running. The user navigates to the Android Home screen, then returns to the app.

   onCreate(): x = 2

   onStart(): x = 8

   onStop(): x = 16

   onRestart(): x = 9

   onStart(): x = 15

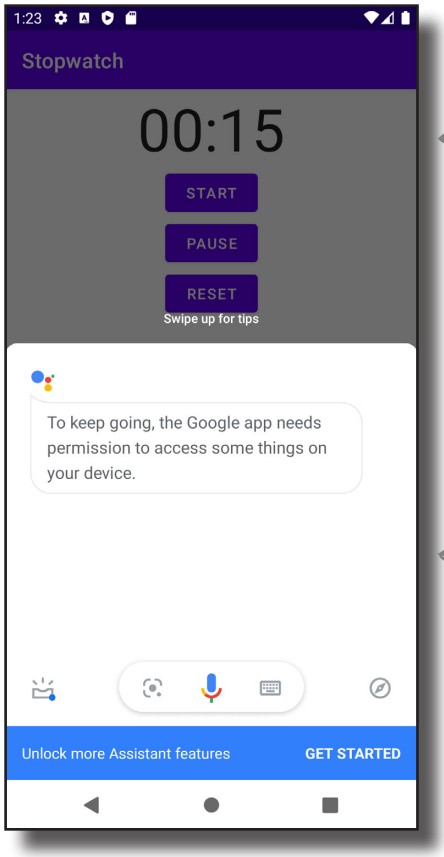
   The final value of x is 15.

```
package com.hfad.lifecycleexercise

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

class MainActivity : AppCompatActivity() {

    var x = 0

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        x = savedInstanceState?.getInt("x") ?: 2
    }

    override fun onRestart() {
        super.onRestart()
        x -= 7
    }

    override fun onStart() {
        super.onStart()
        x += 6
    }

    override fun onStop() {
        super.onStop()
        x *= 2
    }

    override fun onSaveInstanceState(savedInstanceState: Bundle) {
        savedInstanceState.putInt("x", x + 1)
        super.onSaveInstanceState(savedInstanceState)
    }
}
```

# What if an activity is only partially visible?

So far you've seen what happens when an activity gets created and destroyed, and you've also seen what happens when an activity becomes visible, and when it becomes invisible. But there's one more situation you need to consider: **when an activity is visible but doesn't have the focus**.

When an activity is visible but doesn't have the focus, the activity is **paused**. This can happen if another activity appears on top of your activity that isn't full-size or that's transparent. The activity on top has the focus, but the one underneath is still visible and is therefore paused.




The stopwatch activity is still visible, but it's partially obscured and no longer has the focus. When this happens, it pauses.

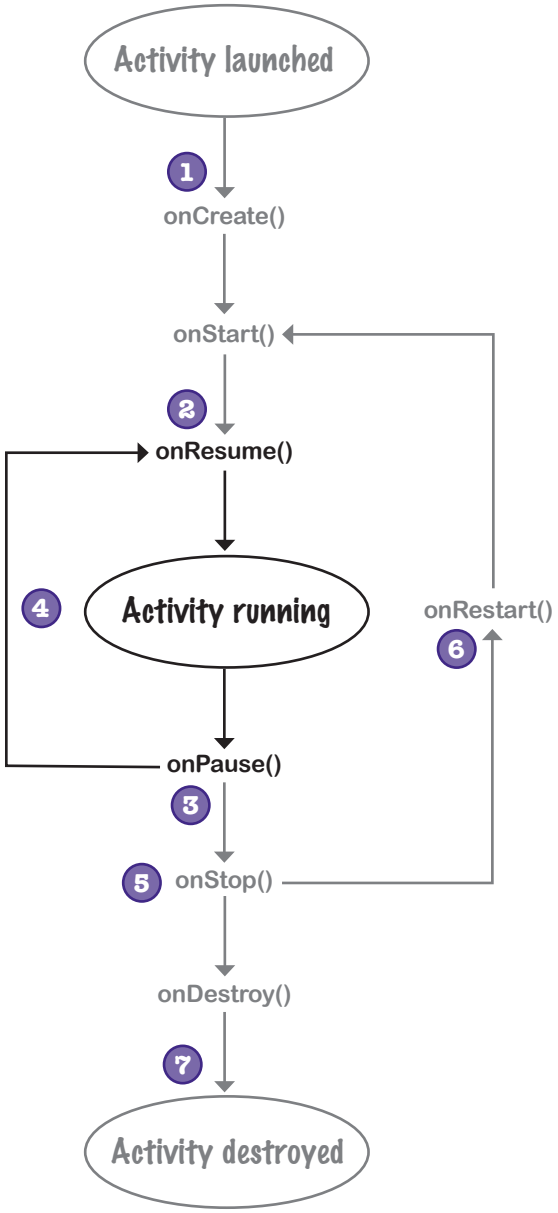
This is an activity from the Google Assistant app that's appeared on top of the stopwatch.

**An activity has a state of paused if it's lost the focus but is still visible to the user. The activity is still alive and maintains all its state information.**

There are two lifecycle methods that get involved when the activity is paused and when it becomes active again: **`onPause()`** and **`onResume()`**. `onPause()` gets called when your activity is visible but another activity has the focus. `onResume()` is called immediately before your activity is about to start interacting with the user. Let's look at how these methods fit in with the rest of the activity lifecycle.

# The foreground lifecycle

Let's build on the lifecycle diagram you saw earlier in the chapter, this time including the `onResume()` and `onPause()` methods (the new bits are in black):




1. **The activity gets launched, and the onCreate() and onStart() methods run.**
   At this point, the activity is visible, but it doesn't have the focus.

2. **The onResume() method runs. It gets called when the activity is about to move into the foreground.**
   After the `onResume()` method has run, the activity has the focus and the user can interact with it.

3. **The onPause() method runs when the activity stops being in the foreground.**
   After the `onPause()` method has run, the activity is still visible but doesn't have the focus.

4. **If the activity moves into the foreground again, the onResume() method gets called.**
   The activity may go through this cycle many times if the activity repeatedly loses and then regains the focus.

5. **If the activity stops being visible to the user, the onStop() method gets called.**
   After the `onStop()` method has run, the activity is no longer visible.

6. **If the activity becomes visible to the user again, the onRestart() method gets called, followed by onStart() and onResume().**
   The activity may go through this cycle many times.

7. **Finally, the activity is destroyed.**
   As the activity moves from running to destroyed, the `onPause()` and `onStop()` methods get called before the activity is destroyed.

# Pause the stopwatch if the activity's paused

Let's get back to the Stopwatch app.

So far we've made the stopwatch pause if the Stopwatch app isn't visible, and made it restart when the app becomes visible again. We did this by overriding the `onStop()` and `onRestart()` methods.

Let's get the app to have the same behavior if the app is only partially visible. We'll get the stopwatch to pause if the activity is paused, and start again when the activity is resumed. So which lifecycle methods do we need to implement?

The easy answer is that we need to use the `onPause()` and `onResume()` methods, but we can take this one step further. We'll use these methods to *replace* the calls to `onStop()` and `onRestart()` that we've already implemented.

If you look again at the lifecycle diagram, calls are made to `onPause()` and `onResume()`—in addition to `onStop()`, `onStart()`, and `onRestart()`—whenever an activity is stopped and restarted. Because we want the app to have the same behavior irrespective of whether the activity has been paused or stopped, we can use the `onPause()` and `onResume()` methods for both situations.

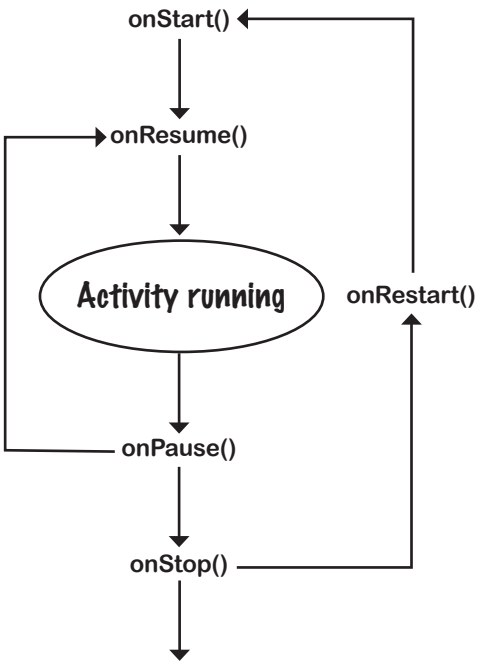
You implement the `onPause()` and `onResume()` methods using code like this:

```
override fun onPause() {
    super.onPause()
    //Code to run when the activity pauses
}

override fun onResume() {
    super.onResume()
    //Code to run when the activity resumes
}
```

As before, the calls to `super.onPause()` and `super.onResume()` are mandatory. They call the `onPause()` and `onResume()` methods in the `Activity` superclass, and without these method calls, the code won't compile.

We want to replace the `onStop()` and `onRestart()` methods in *MainActivity.kt* with implementations of `onPause()` and `onResume()`. See if you can figure out the required code changes by having a go at the following exercise.

The onResume() method is called when the activity is restarted or resumed. As we want the app to do the same thing irrespective of whether it's started or resumed, we can implement the onResume() method instead of onRestart().




The onPause() method is called when the activity is paused or stopped. This means we can implement the onPause() method instead of onStop().

## Sharpen your pencil

We want the stopwatch to pause each time `MainActivity` loses the focus, and start when it regains it. Update the *MainActivity.kt* code below to include the required code changes.

We're not showing you the entire MainActivity.kt code here, just the bits you might want to change.

```
...
class MainActivity : AppCompatActivity() {

    lateinit var stopwatch: Chronometer  //The chronometer
    var running = false  //Is the chronometer running?
    var offset: Long = 0  //The base offset for the chronometer
    ...

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
    }

    override fun onStop() {
        super.onStop()
        if (running) {
            saveOffset()
            stopwatch.stop()
        }
    }

    override fun onRestart() {
        super.onRestart()
        if (running) {
            setBaseTime()
            stopwatch.start()
            offset = 0
        }
    }

    ...
}
```

## Sharpen your pencil Solution

We want the stopwatch to pause each time `MainActivity` loses the focus, and start when it regains it. Update the *MainActivity.kt* code below to include the required code changes.

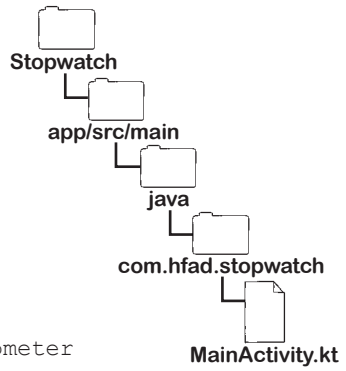
```
...
class MainActivity : AppCompatActivity() {

    lateinit var stopwatch: Chronometer  //The chronometer
    var running = false  //Is the chronometer running?
    var offset: Long = 0  //The base offset for the chronometer
    ...

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
    }

    override fun onStop() onPause() {
        super.onStop()
        super.onPause()
        if (running) {
            saveOffset()
            stopwatch.stop()
        }
    }

    override fun onRestart() onResume() {
        super.onRestart()
        super.onResume()
        if (running) {
            setBaseTime()
            stopwatch.start()
            offset = 0
        }
    }

    ...
}
```

We need to replace onStop() with onPause(), and call up to the superclass version of the onPause() method.

Likewise, we need to replace onRestart() with onResume().

# The complete MainActivity.kt code

Here's the full *MainActivity.kt* code for the finished app; update your code to include the changes (in bold):



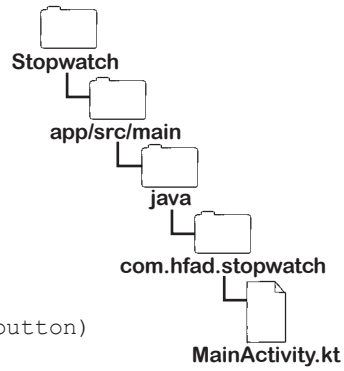

```
package com.hfad.stopwatch

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.os.SystemClock
import android.widget.Button
import android.widget.Chronometer

class MainActivity : AppCompatActivity() {

    lateinit var stopwatch: Chronometer  //The chronometer
    var running = false  //Is the chronometer running?
    var offset: Long = 0  //The base offset for the chronometer

    //Add key Strings for use with the Bundle
    val OFFSET_KEY = "offset"
    val RUNNING_KEY = "running"
    val BASE_KEY = "base"

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

        //Get a reference to the stopwatch
        stopwatch = findViewById<Chronometer>(R.id.stopwatch)

        //Restore the previous state
        if (savedInstanceState != null) {
            offset = savedInstanceState.getLong(OFFSET_KEY)
            running = savedInstanceState.getBoolean(RUNNING_KEY)
            if (running) {
                stopwatch.base = savedInstanceState.getLong(BASE_KEY)
                stopwatch.start()
            } else setBaseTime()
        }
```

You don't need to change any of the code on this page.

The code continues on the next page.

# MainActivity.kt continued

```
        //The start button starts the stopwatch if it's not running
        val startButton = findViewById<Button>(R.id.start_button)
        startButton.setOnClickListener {
            if (!running) {
                setBaseTime()
                stopwatch.start()
                running = true
            }
        }

        //The pause button pauses the stopwatch
        val pauseButton = findViewById<Button>(R.id.pause_button)
        pauseButton.setOnClickListener {
            if (running) {
                saveOffset()
                stopwatch.stop()
                running = false
            }
        }

        //The reset button sets the offset and stopwatch to 0
        val resetButton = findViewById<Button>(R.id.reset_button)
        resetButton.setOnClickListener {
            offset = 0
            setBaseTime()
        }
    }

    override fun ~~onStop()~~ onPause() {
        ~~super.onStop()~~
        super.onPause()
        if (running) {
            saveOffset()
            stopwatch.stop()
        }
    }
```



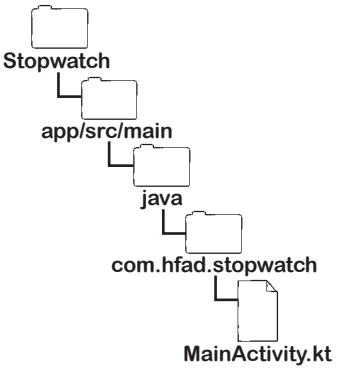

Replace onStop() with onPause().

The code continues on the next page.

# MainActivity.kt continued

```
override fun ~~onRestart()~~ onResume() {
    ~~super.onRestart()~~
    super.onResume()
    if (running) {
        setBaseTime()
        stopwatch.start()
        offset = 0
    }
}
```

Replace onRestart with onResume().

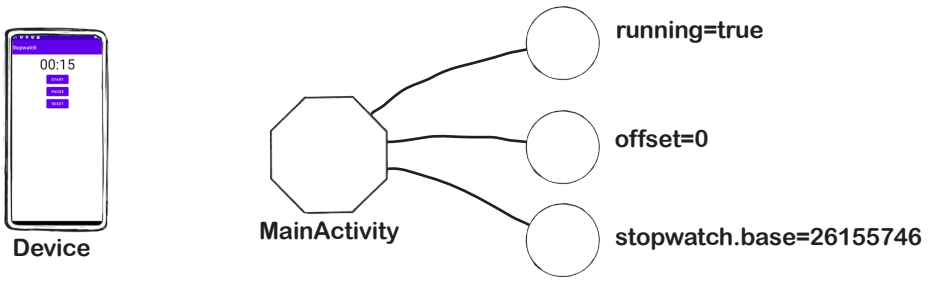



```
    override fun onSaveInstanceState(savedInstanceState: Bundle) {
        savedInstanceState.putLong(OFFSET_KEY, offset)
        savedInstanceState.putBoolean(RUNNING_KEY, running)
        savedInstanceState.putLong(BASE_KEY, stopwatch.base)
        super.onSaveInstanceState(savedInstanceState)
    }

    //Update the stopwatch.base time, allowing for any offset
    fun setBaseTime() {
        stopwatch.base = SystemClock.elapsedRealtime() - offset
    }

    //Record the offset
    fun saveOffset() {
        offset = SystemClock.elapsedRealtime() - stopwatch.base
    }
}
```

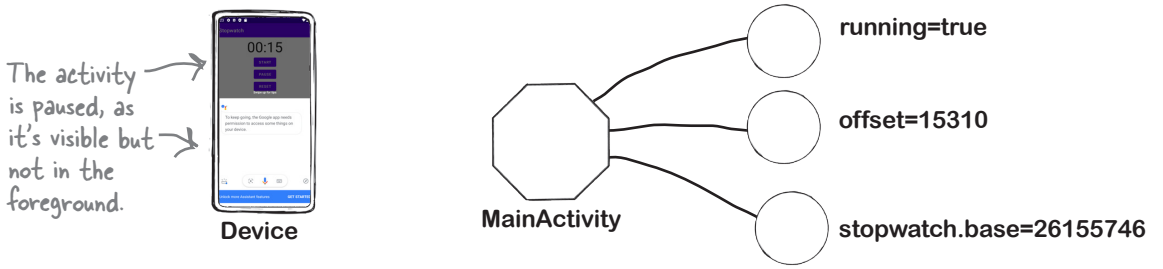
Before we take the app for a test drive, let's go through what happens when the code runs.

# What happens when you run the app

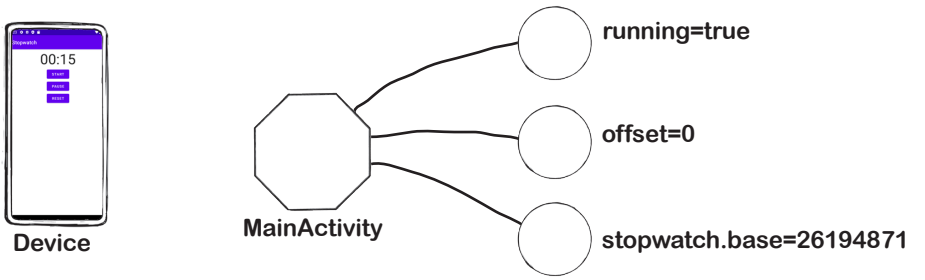
1. **The user starts the app, and clicks the Start button to set the stopwatch going.**
The stopwatch starts and the `running` property is set to `true`.




2. **Another activity appears in the foreground, leaving MainActivity partially visible.**
The `onPause()` method gets called, the `offset` property is updated, and the stopwatch is stopped.

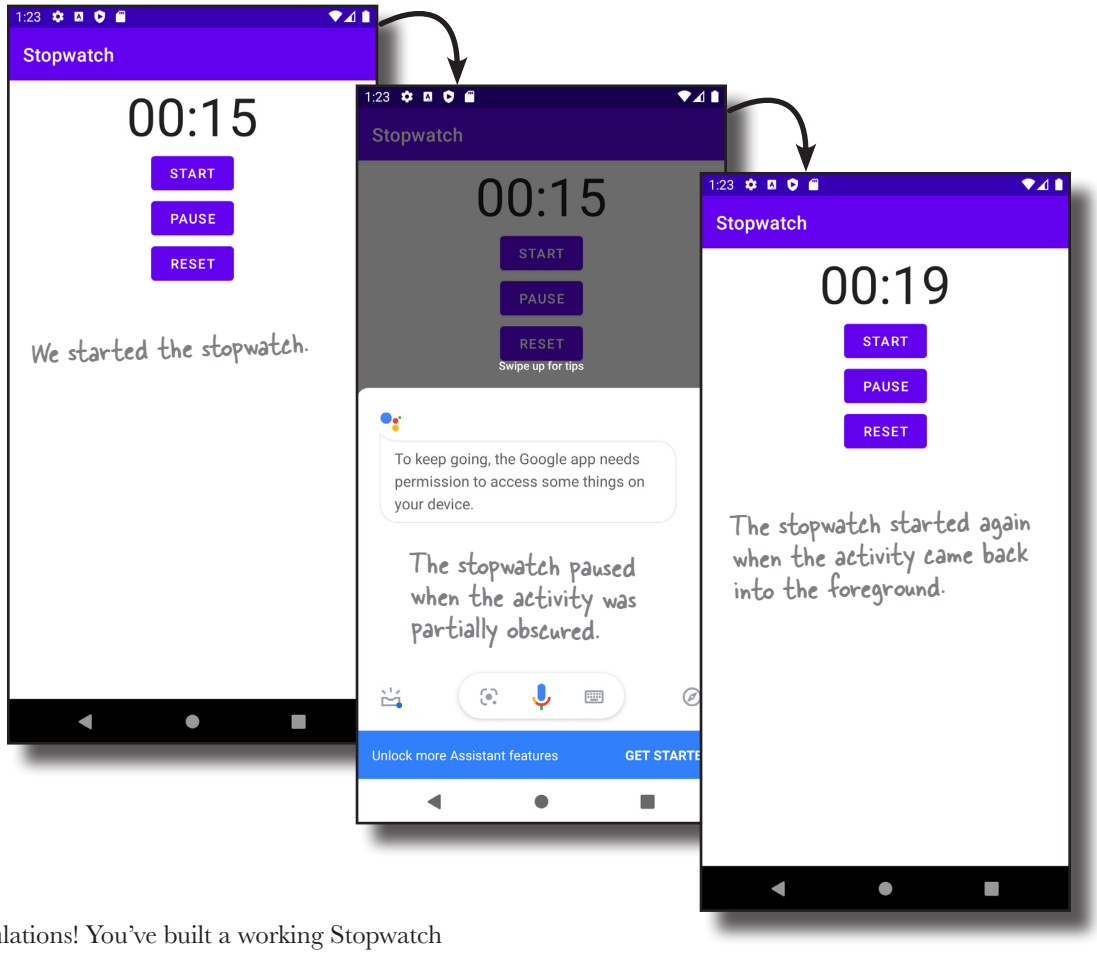



3. **MainActivity returns to the foreground.**
The `onResume()` method gets called, the `stopwatch.base` property is updated, the stopwatch starts, and the `offset` property is set to `0`. The stopwatch starts from the correct time.



# Test Drive

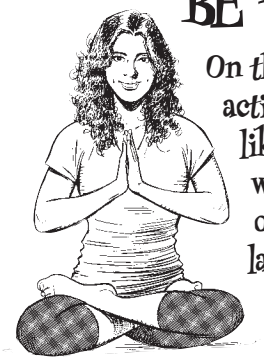
When we run the app and click on the Start button, the timer starts; it pauses when the app is partially obscured by another activity; and it starts again when the app is back in the foreground.




Congratulations! You've built a working Stopwatch app, and learned how to use Android's activity lifecycle methods to take control of your app's behavior.

# Your handy guide to the activity lifecycle methods

| Method | When it's called | Next method |
|---|---|---|
| **onCreate()** | When the activity is first created. Use it for normal setup, such as creating views. It also gives you a `Bundle` that contains the previously saved state of the activity. | `onStart()` |
| **onRestart()** | When your activity has been stopped but just before it gets started again. | `onStart()` |
| **onStart()** | When your activity is becoming visible. It's followed by `onResume()` if the activity comes into the foreground, or `onStop()` if the activity is made invisible. | `onResume()` or `onStop()` |
| **onResume()** | When your activity is in the foreground. | `onPause()` |
| **onPause()** | When your activity is no longer in the foreground because another activity is resuming. The next activity isn't resumed until this method finishes, so any code in this method needs to be quick. It's followed by `onResume()` if the activity returns to the foreground, or `onStop()` if it becomes invisible. | `onResume()` or `onStop()` |
| **onStop()** | When the activity is no longer visible. This can be because another activity is covering it, or because this activity is being destroyed. It's followed by `onRestart()` if the activity becomes visible again, or `onDestroy()` if the activity is being destroyed. | `onRestart()` or `onDestroy()` |
| **onDestroy()** | When your activity is about to be destroyed or because the activity is finishing. | None |



# BE the Activity

On the right, you'll see some activity code. Your job is to play like you're the activity and say which code will run in each of the situations below. We've labeled the code we want you to consider. We've done the first one to start you off.

**User starts the activity and starts using it.**

Code segments A, G, D. The activity is created, then made visible, then receives the focus.

**User starts the activity, starts using it, then switches to another app.**

This one's tough.

**User starts the activity, starts using it, rotates the device, switches to another app, then goes back to the activity.**

```
...
class MainActivity : AppCompatActivity() {

    override fun onCreate(
            savedInstanceState: Bundle?) {
        A //Run code A
        ...
    }

    override fun onPause() {
        B //Run code B
        ...
    }

    override fun onRestart() {
        C //Run code C
        ...
    }

    override fun onResume() {
        D //Run code D
        ...
    }

    override fun onStop() {
        E //Run code E
        ...
    }

    fun onRecreate() {
        F //Run code F
        ...
    }

    override fun onStart() {
        G //Run code G
        ...
    }

    override fun onDestroy() {
        H //Run code H
        ...
    }
}
```

# BE the Activity Solution

On the right, you'll see some activity code. Your job is to play like you're the activity and say which code will run in each of the situations below. We've labeled the code we want you to consider. We've done the first one to start you off.

**User starts the activity and starts using it.**

Code segments A, G, D. The activity is created, then made visible, then receives the focus.

**User starts the activity, starts using it, then switches to another app.**

Code segments A, G, D, B, E. The activity is created, then made visible, then receives the focus. When the user switches to another app, it loses the focus and is no longer visible to the user.

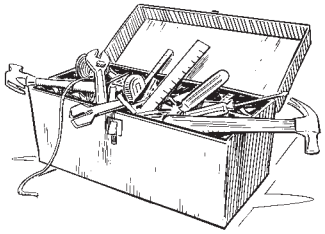
**User starts the activity, starts using it, rotates the device, switches to another app, then goes back to the activity.**

Code segments A, G, D, B, E, H, A, G, D, B, E, C, G, D. First, the activity is created, made visible, and receives the focus. When the device is rotated, the activity loses the focus, stops being visible, and is destroyed. It's then created again, made visible, and receives the focus. When the user switches to another app and back again, the activity loses the focus, loses visibility, becomes visible again, and regains the focus.

```
...
class MainActivity : AppCompatActivity() {

    override fun onCreate(
            savedInstanceState: Bundle?) {
        A //Run code A
        ...
    }

    override fun onPause() {
        B //Run code B
        ...
    }

    override fun onRestart() {
        C //Run code C
        ...
    }

    override fun onResume() {
        D //Run code D
        ...
    }

    override fun onStop() {
        E //Run code E
        ...
    }

    fun onRecreate() {
        F //Run code F
        ...
    }

    override fun onStart() {
        G //Run code G
        ...
    }

    override fun onDestroy() {
        H //Run code H
        ...
    }
}
```

There's no lifecycle method called onRecreate().

# Your Android Toolbox

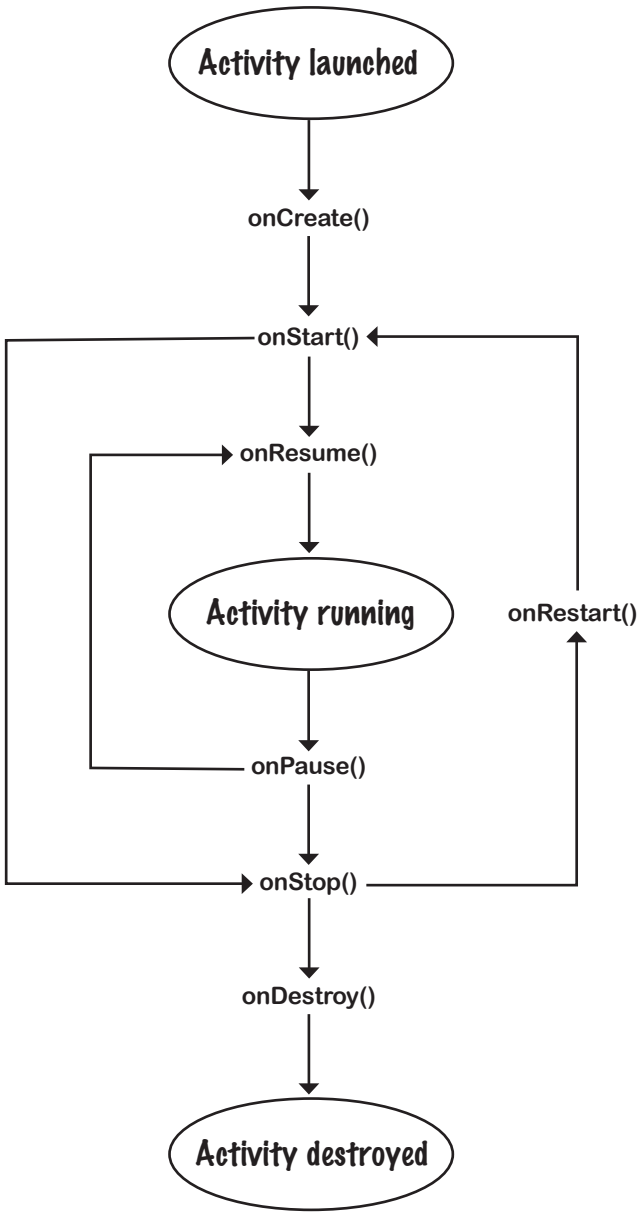


**You've got Chapter 5 under your belt and now you've added the activity lifecycle to your toolbox.**

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

## Bullet Points

- The `Chronometer` view implements a simple timer. `Chronometer` is a subclass of `TextView`.
- A device configuration change results in the activity being destroyed and recreated.
- Your activity inherits the lifecycle methods from the `android.app.Activity` class. If you override any of these methods, you need to call up to the method in the superclass.
- `onSaveInstanceState(Bundle)` enables your activity to save its state before the activity gets destroyed. You can use the `Bundle` to restore state in `onCreate()`.
- You add values to a `Bundle` in `onSaveInstanceState()` using `bundle.put...("name", value)`.
- You retrieve values from the bundle in `onCreate()` using `bundle.get...("name")`.
- `onCreate()` and `onDestroy()` deal with the birth and death of the activity.
- `onRestart()`, `onStart()`, and `onStop()` deal with the visibility of the activity.
- `onResume()` and `onPause()` get called when the activity gains and loses the focus.



# 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.

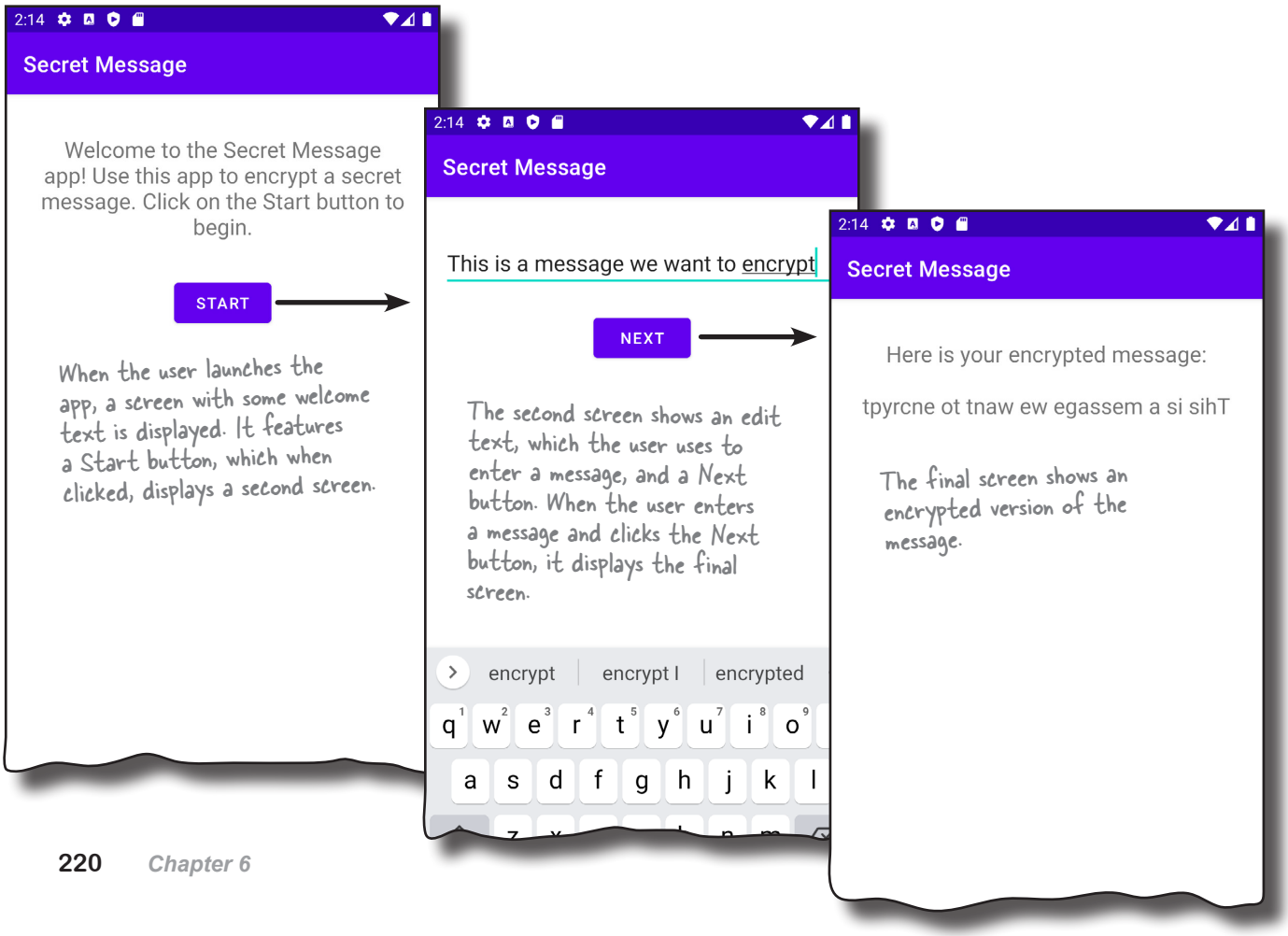
# Most apps need more than one screen

All of the apps you've built so far have one thing in common: they each have a single screen. Each app has a single activity with a corresponding layout, which defines the app's appearance, and how the user can interact with it.

Most apps, however, contain *more* than one screen. An email app, for example, might have one screen that lets you compose an email, and another that displays a list of emails you've received. A calendar app might display a list of events on one screen, and the details of one specific event on another.

We're going to show you how to build multi-screen apps by creating a Secret Message app. The app will comprise of a welcome screen, a second screen which allows the user to enter a message, and a third which displays an encrypted version of the message.

Here's what the app will look like:



# Each screen is a fragment

The Secret Message app has three distinct screens, and we're going to build each screen as a separate **fragment**. A fragment is like a kind of subactivity that's displayed inside an activity's layout. It has Kotlin code that controls its behavior, and an associated layout that defines its appearance.

Here are the three fragments that the app will use:

**A fragment has Kotlin code that controls its behavior, and a layout that specifies its appearance.**

## WelcomeFragment

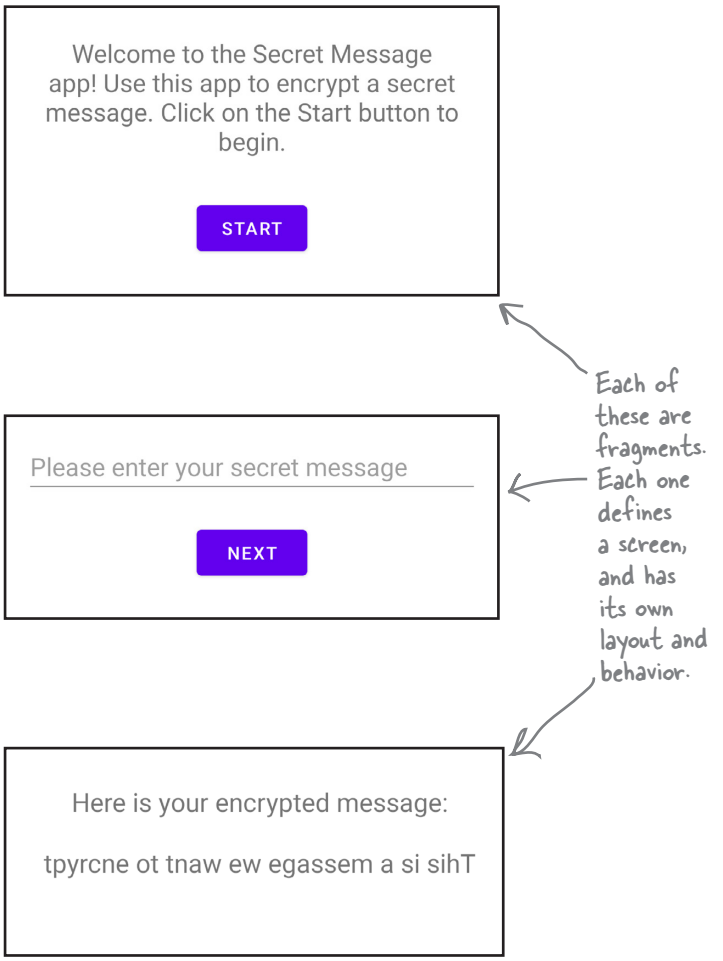
This is the app's main screen. It needs to display some introductory text, and a button. The button will be used to navigate to the next screen—`MessageFragment`.

## MessageFragment

This screen will let the user enter a message into an edit text. When the user clicks on its button, the app will navigate to `EncryptFragment`.

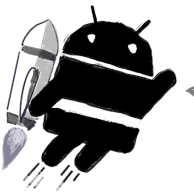
## EncryptFragment

This is the final screen. It encrypts the user's message, and displays the result.




As you can see, the user needs to be able to navigate between all three fragments. So how will this work?

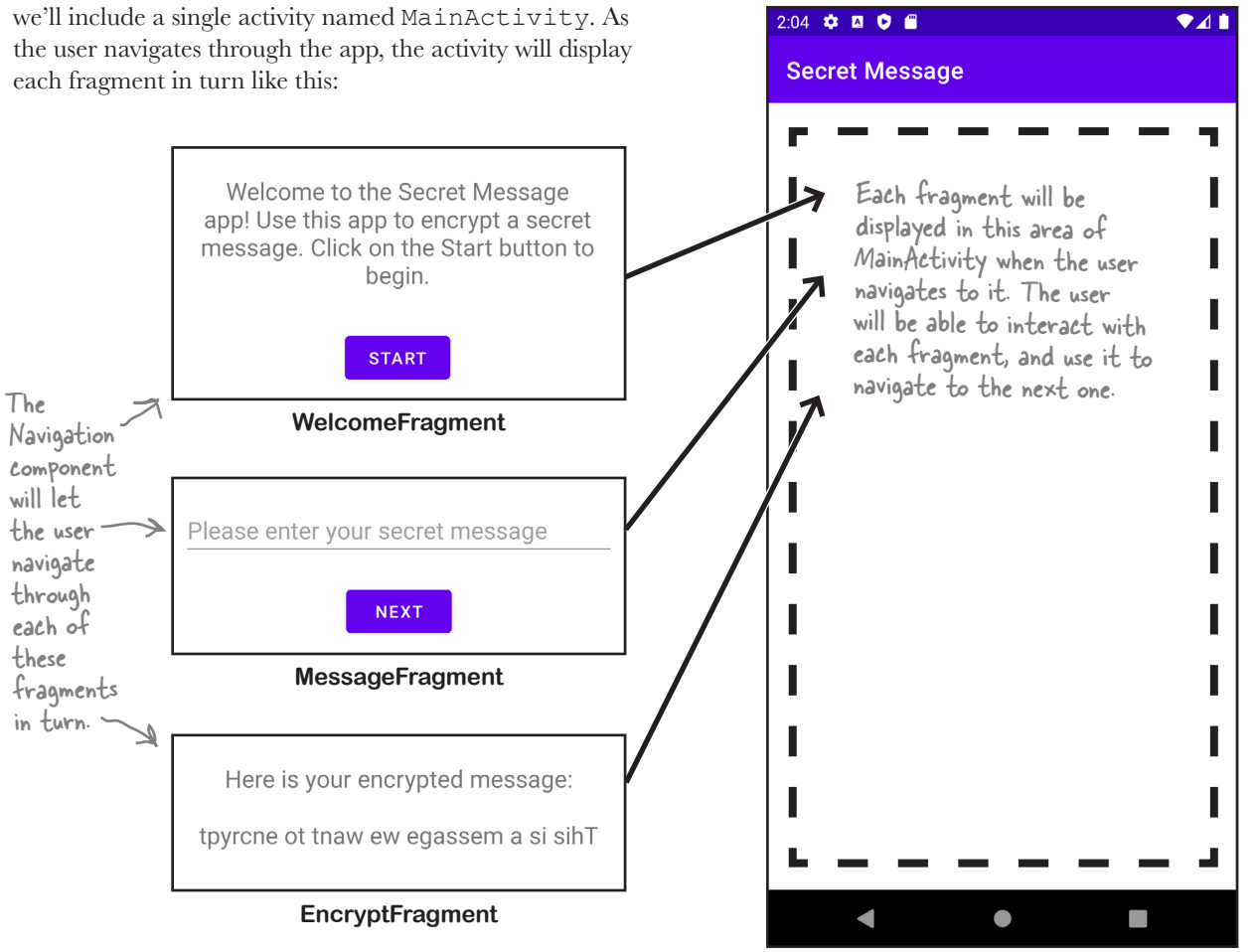
# Navigate between screens using the Navigation component



The Navigation component is part of Android Jetpack.

The best way of navigating between fragments is to use Android's **Navigation component**. The Navigation component is part of Android Jetpack, and it helps you implement navigation in a standard way.

To use the Navigation component in the Secret Message app, we'll include a single activity named `MainActivity`. As the user navigates through the app, the activity will display each fragment in turn like this:




You'll find out more about fragments and the Navigation component as we build the Secret Message app. Let's go through the steps we'll take to create it.

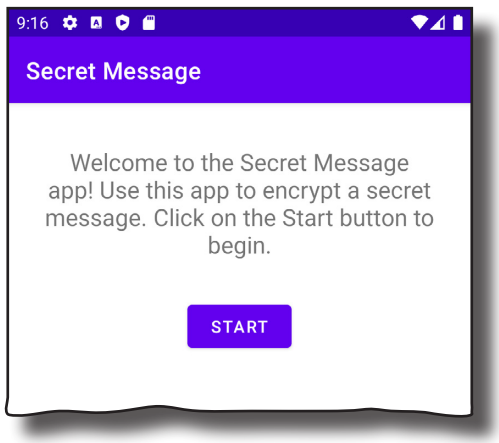
# Here's what we're going to do

We're going to start building the Secret Message app in this chapter, and complete it in the next.

Here's what we'll do in this chapter:

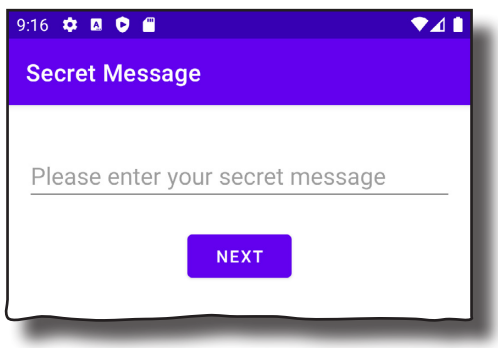
1. **Create and display WelcomeFragment.**
In this step, we'll create `WelcomeFragment`, which will be the app's first screen. We'll display this fragment in `MainActivity`'s layout so the user sees it when the app launches.

In this step, we'll display WelcomeFragment in MainActivity's layout.




2. **Navigate to MessageFragment.**
We'll create a second fragment named `MessageFragment`, and navigate to it when the user clicks on a button in `WelcomeFragment`'s layout. We'll use Android's Navigation component to implement navigation.

When the user clicks on WelcomeFragment's Start button, MainActivity will display MessageFragment.




Let's start by creating a new project for the app.

**WelcomeFragment**
MessageFragment

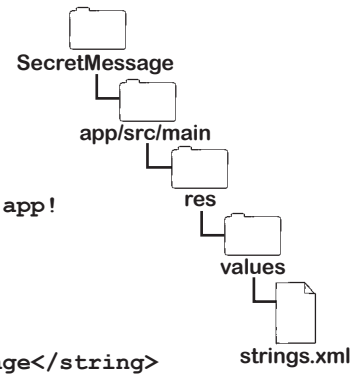
# Create a new project

We're going to use a new project for the Secret Message app, so create one now using the same steps you used in the previous chapters. Choose the Empty Activity option, enter a name of "Secret Message" and a package name of "com.hfad.secretmessage", and accept the default save location. Make sure the language is set to Kotlin and the minimum SDK is API 21 so it will run on most Android devices.

## Add some String resources

Before we create any fragments, we're going to add some `String` resources to the project. We'll use these to display text in the fragments' layouts, such as the labels for buttons, and the welcome message on the first screen.

To add the `Strings`, open the file *strings.xml* in the *SecretMessage/app/src/main/res/values* folder, and then add the following resources (in bold) to the file:




```
<resources>
    <string name="app_name">Secret Message</string>
    <string name="welcome_text">Welcome to the Secret Message app!
        Use this app to encrypt a secret message.
        Click on the Start button to begin.</string>
    <string name="start">Start</string>
    <string name="message_hint">Please enter your secret message</string>
    <string name="next">Next</string>
    <string name="encrypt_text">Here is your encrypted message:</string>
</resources>
```

Make sure you add all of these.

## there are no Dumb Questions

**Q: Why are we using fragments? Can't you create a multi-screen app using multiple activities instead?**

A: Yes, it's possible to create multi-screen apps using multiple activities, and up until a few years ago, this is how you'd typically develop apps of this kind.

Since releasing Android's Navigation component, the Android team recommends using fragments instead. The Navigation component is primarily designed to work with fragments, so this is now the standard way of implementing navigation.

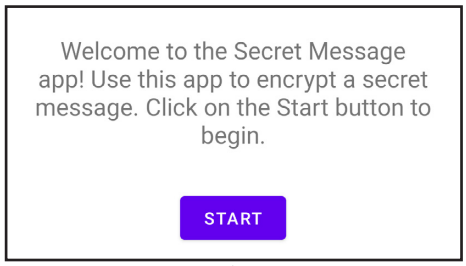
**WelcomeFragment**
**MessageFragment**

# Add WelcomeFragment to the project

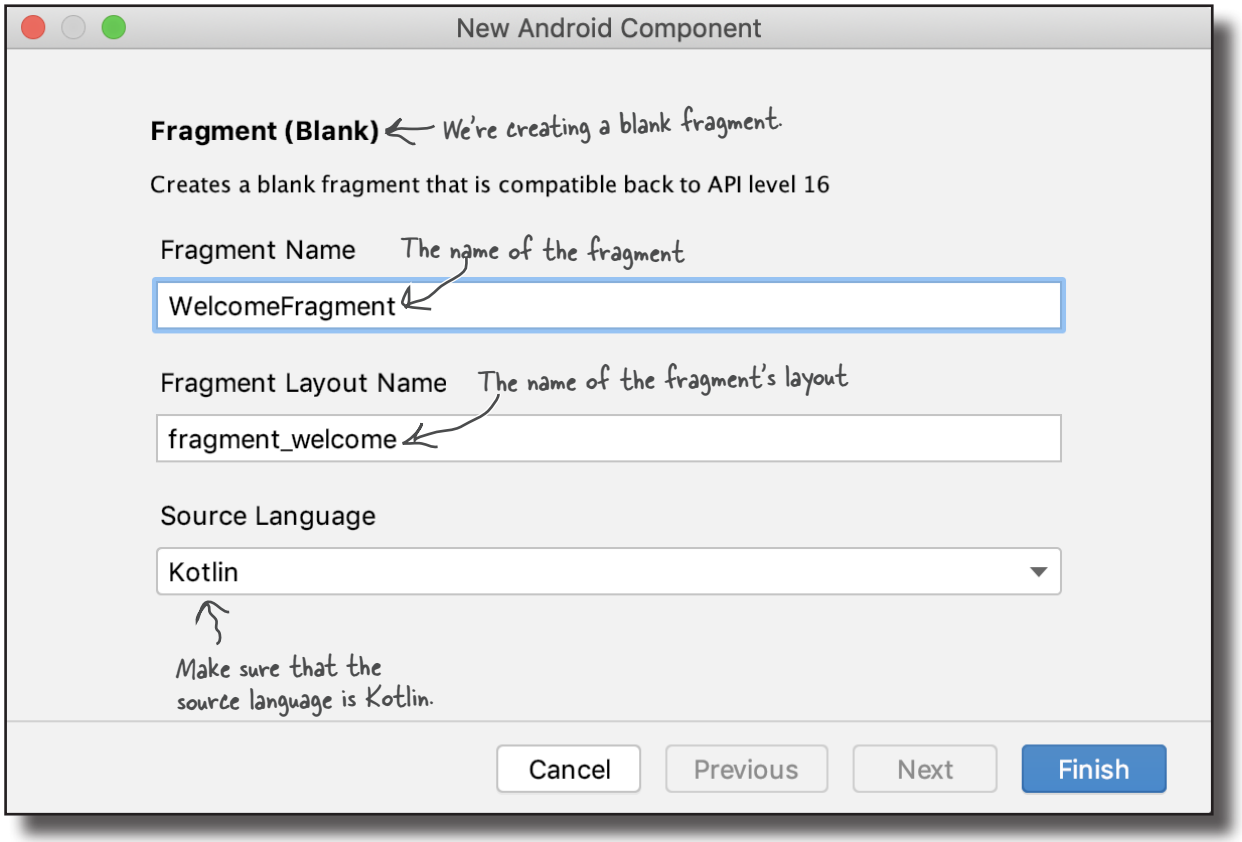
We're going to add a fragment named `WelcomeFragment` to the project. `WelcomeFragment` will be the first screen that the user sees when she opens the app, and we'll use it to display some introductory text about the app, and a button.

To add the fragment, highlight the *com.hfad.secretmessage* package in the *app/src/main/java* folder in the project explorer, go to the File menu, and choose New→Fragment→Fragment (Blank).

You will be asked how you want to configure the new fragment. Name the fragment "WelcomeFragment", and give its layout a name of "fragment_welcome". Then make sure the language is set to Kotlin, and click on the Finish button.




This is what WelcomeFragment will look like.



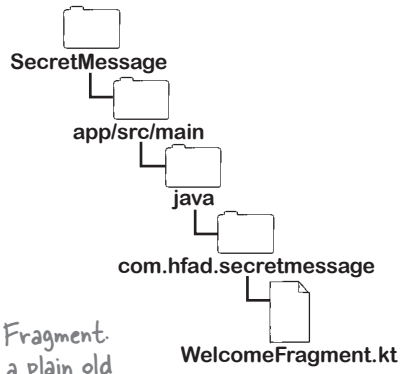

When you click on the Finish button, Android Studio creates the new fragment and adds it to the project.

**WelcomeFragment**
MessageFragment

# What fragment code looks like

When you create a new fragment, Android Studio adds two files to your project: a Kotlin file that controls the fragment's behavior, and a layout file that describes the fragment's appearance.

We'll look at the Kotlin code first. Go to the *com.hfad.secretmessage* package in the *app/src/main/java* folder and open the file *WelcomeFragment.kt*. Then **replace the code** that Android Studio generated with the code below:




```
package com.hfad.secretmessage

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup

class WelcomeFragment : Fragment() {

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.fragment_welcome, container, false)
    }
}
```

WelcomeFragment extends Fragment. This is what turns it from a plain old class into a card-carrying fragment.

The class overrides onCreateView(). This method is called when the fragment needs to be displayed.

This tells Android that the fragment uses fragment_welcome.xml as its layout.

## Fragment code looks similar to activity code

The above code defines a basic fragment. As you can see, the code for a fragment looks very similar to activity code. Instead of extending `AppCompatActivity`, however, it extends **`Fragment`**.

The `androidx.fragment.app.Fragment` class is part of Android Jetpack, and it's used to define a basic fragment. It includes the latest fragment features, while being backward compatible with older versions of Android.

The fragment overrides the `onCreateView()` method, which gets called as soon as Android needs the fragment's layout. Nearly every fragment overrides this method, so let's look at it in more detail.

**Don't forget!**

Make sure you update the contents of WelcomeFragment.kt with the code shown here.

**WelcomeFragment**
MessageFragment

# The fragment's onCreateView() method

The `onCreateView()` method gets called when Android needs access to the fragment's layout. Overriding this method is optional, but as you need to implement it each time you define a fragment with a layout, you'll override it for nearly every fragment you create.

The method takes three parameters:

The onCreateView() method gets called when Android needs the fragment's UI.

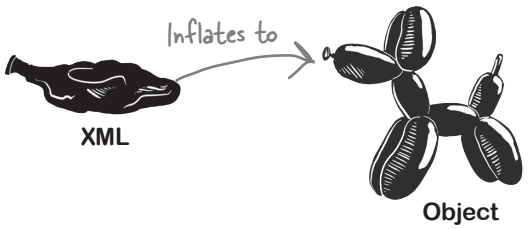
```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                          savedInstanceState: Bundle?): View? {
}
```

The first parameter is a `LayoutInflater` that you use to inflate the fragment's layout. As you learned in Chapter 3, inflating a layout turns its XML views into objects.

The second parameter is a `ViewGroup?`. This is the `ViewGroup` in the activity's layout that displays the fragment.

You'll find out more about this a few pages ahead.

The final parameter is a `Bundle?`. This is used if you've previously saved the fragment's state, and want to reinstate it. It works in a similar way to the `Bundle?` parameter that's passed to an activity's `onCreate()` method.




## Inflate the fragment's layout, and return it

The `onCreateView()` method returns a `View?`, which is an inflated version of the fragment's layout.

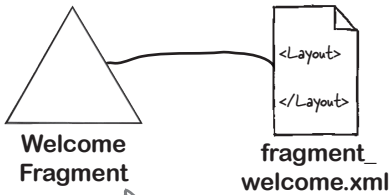
You inflate the layout using the `LayoutInflater`'s `inflate()` method like this:

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                          savedInstanceState: Bundle?): View? {
    return inflater.inflate(R.layout.fragment_welcome, container, false)
}
```

The above code is the fragment equivalent of calling an activity's `setContentView()` method, as it's used to inflate the fragment's layout into a hierarchy of `View` objects. The above code, for example, inflates `WelcomeFragment`'s layout *fragment_welcome.xml*.

Once the fragment's layout has been inflated, the `View` hierarchy is inserted into the activity's layout, and displayed.

Now that you've seen `WelcomeFragment`'s Kotlin code, let's take a look at its layout.



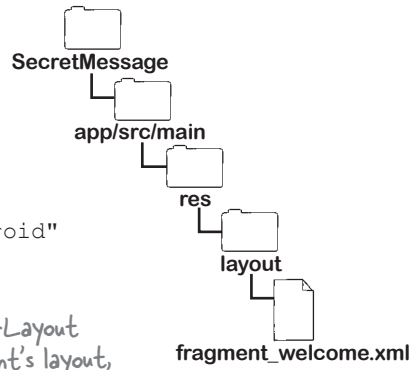

WelcomeFragment uses fragment_welcome.xml for its layout.

# Fragment layout code looks like activity layout code

As we said earlier, fragments use layout files to describe their appearance. There's no difference between activity layout and fragment layout code, so **you can use any of the views and view groups in your fragment layout code that you're already familiar with**.

We're going to replace the default layout code Android Studio has generated for us with a linear layout containing a text view that displays a brief description of the app, and a button that we'll use to navigate to a different fragment later in the chapter.

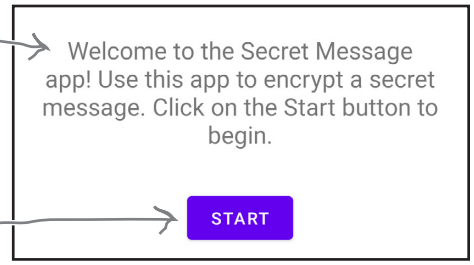
Open the file *fragment_welcome.xml* in the *app/src/main/res/layout* folder, and replace its contents with the code below:




```
<?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:orientation="vertical"
    android:gravity="center_horizontal"
    tools:context=".WelcomeFragment">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:layout_marginTop="20dp"
        android:textSize="20sp"
        android:text="@string/welcome_text" />

    <Button
        android:id="@+id/start"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="32dp"
        android:text="@string/start" />
</LinearLayout>
```

We're using a LinearLayout for WelcomeFragment's layout, but we could have used any of the other layout types you've learned about instead.

This code displays some introductory text, held in the welcome_text String resource.




This code defines the Start button. We'll use this later to navigate to the next fragment.

That's all the code we need for `WelcomeFragment` (and its layout) for now, so let's find out how to display it in the app.

# You display a fragment in a FragmentContainerView

To display a fragment, you need to add it to an activity's layout. In this app, for example, we're going to display `WelcomeFragment` by adding it to `MainActivity`'s layout file *activity_main.xml*.

You add a fragment to a layout using a **`FragmentContainerView`**. This is a type of `FrameLayout` that's used to display fragments, and you add it to a layout file using code like this:

```
<androidx.fragment.app.FragmentContainerView
    android:id="@+id/fragment_container_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:name="com.hfad.secretmessage.WelcomeFragment" />
```

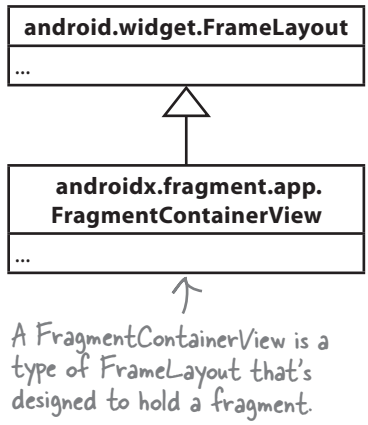
This adds the FragmentContainerView.

The FragmentContainerView needs to have an ID.

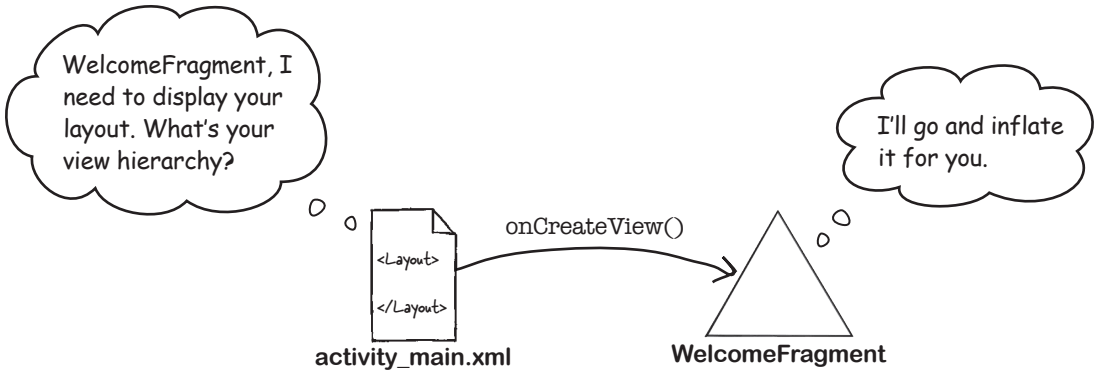
This is the full name of the fragment including its package.

You specify which fragment you want to display by setting the `FragmentContainerView`'s `android:name` attribute to the fully qualified fragment name, including its package. In the Secret Message app, we want to display the fragment named `WelcomeFragment` that's in the *com.hfad.secretmessage* package, so we're setting the `android:name` attribute using:

```
android:name="com.hfad.secretmessage.WelcomeFragment"
```




When Android creates the activity's layout, it populates the `FragmentContainerView` with the `View` object returned by the fragment's `onCreateView()` method. This `View` is the fragment's user interface, so you can think of the `FragmentContainerView` as a placeholder for where the fragment's layout should be inserted:



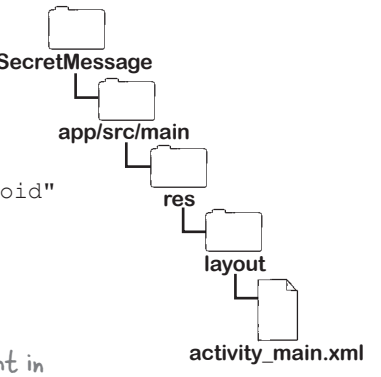

Now that you know how to add a fragment to a layout, let's add `WelcomeFragment` to `MainActivity`'s layout.

**WelcomeFragment**
MessageFragment

# Update the activity_main.xml code

We want `MainActivity` to display `WelcomeFragment`, which means we need to add a `FragmentContainerView` to its layout.

Here's the full code for *activity_main.xml*: replace the code so that it includes the changes shown here:



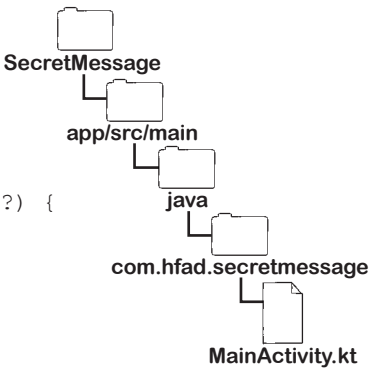

The layout only needs to include a Fragment Container View, so we're adding it as the layout's root.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/fragment_container_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:name="com.hfad.secretmessage.WelcomeFragment"
    tools:context=".MainActivity" />
```

This displays WelcomeFragment in MainActivity's layout.

## The full code for MainActivity.kt

We don't need to add any extra Kotlin code to `MainActivity` to make it display a fragment because the layout's `FragmentContainerView` handles everything. You just need to make sure that the code in *MainActivity.kt* looks like this:



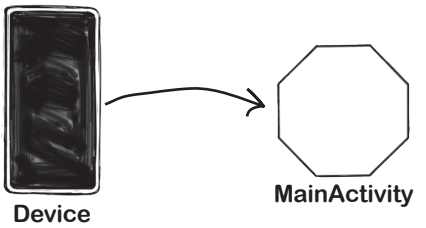

```
package com.hfad.secretmessage

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)
    }
}
```

Let's see what happens when the app runs.

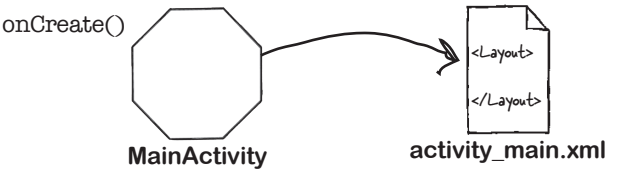
# What the code does

The following things happen when the app runs:

1. **When the app is launched, MainActivity gets created.**

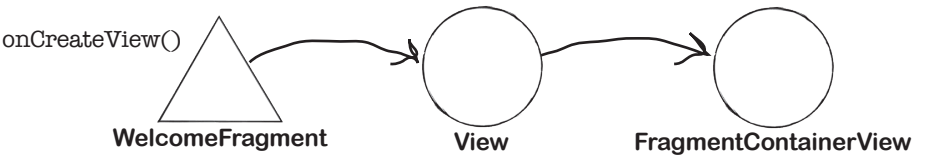



2. **MainActivity's onCreate() method runs.**
The `onCreate()` method specifies that *activity_main.xml* should be used for `MainActivity`'s layout.




3. **activity_main.xml includes a FragmentContainerView.**
Its `android:name` attribute specifies that it needs to display `WelcomeFragment`.

4. **WelcomeFragment's onCreateView() method is called, which inflates its layout.**
`WelcomeFragment`'s inflated view hierarchy is added to the `FragmentContainerView` in `MainActivity`'s layout.



**WelcomeFragment**
MessageFragment

# The story continues

5 **Finally, MainActivity is displayed on the device.**
As the `FragmentContainerView` includes `WelcomeFragment`, this fragment is displayed on the screen.

Device

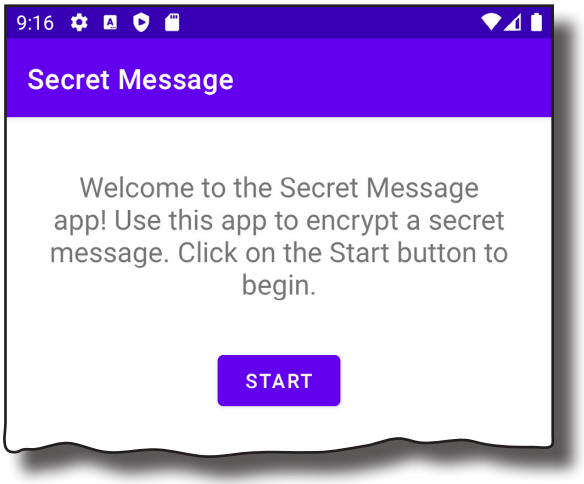
WelcomeFragment is displayed in MainActivity. You'll see it more clearly in the test drive below.

Now that you've seen what happens when the code runs, let's take the app for a test drive.

## Test Drive

When we run the Secret Message app, `MainActivity` is launched. The `FragmentContainerView` in `MainActivity`'s layout includes `WelcomeFragment`, so the fragment's layout is displayed on the device.

WelcomeFragment is displayed on the screen.

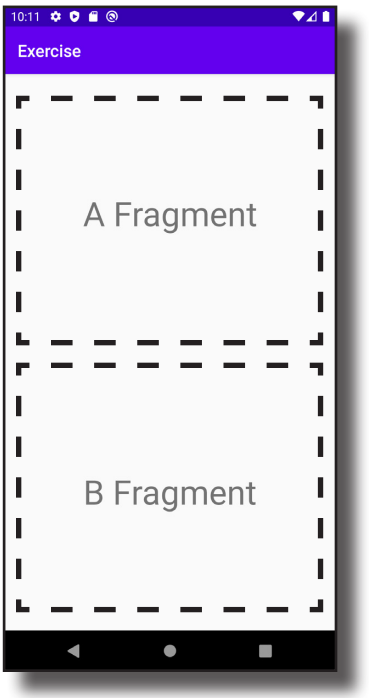



You've now learned how to create and display a fragment. Before we build a second fragment and learn how to navigate to it, have a go at the following exercise.

# Pool Puzzle

Your **job** is to take code snippets from the pool and place them into the blank lines in the linear layout. You may **not** use the same code snippet more than once, and you won't need to use all the snippets. Your **goal** is to make the linear layout display two fragments, `AFragment` and `BFragment`, so that it displays the screen shown on the right. Hint: both fragments are in a package named *com.hfad.exercise*.

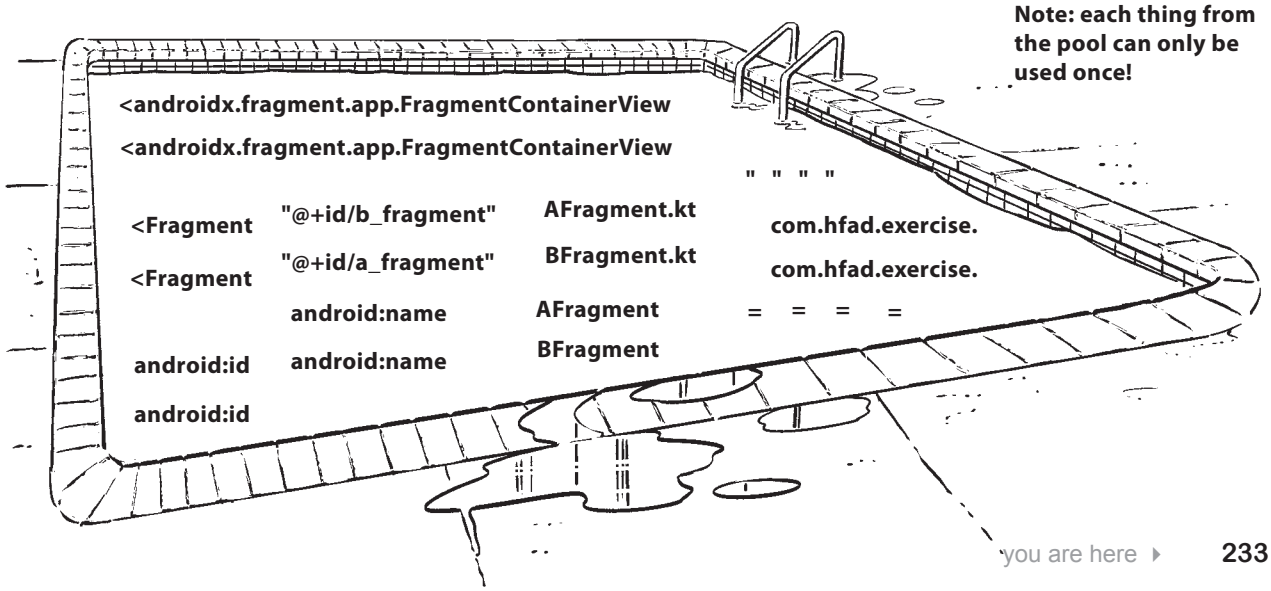
AFragment needs to fill the top half of the screen, while BFragment fills the bottom half.




We've left out some of the LinearLayout attributes.

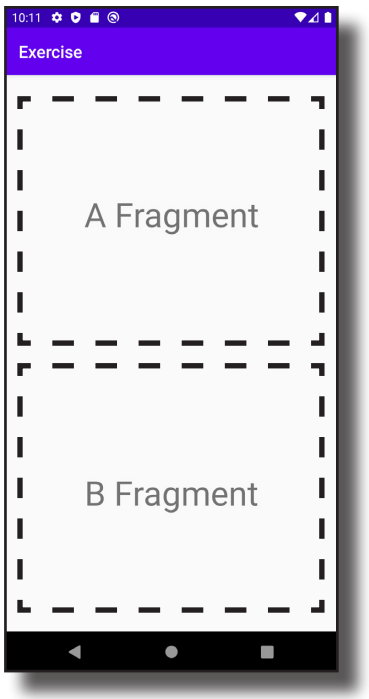
```
<LinearLayout ...
    android:orientation="vertical">

    .....

        .....

        .....
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1" />

    .....

        .....

        .....
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1" />
</LinearLayout>
```

**Note: each thing from the pool can only be used once!**



# Pool Puzzle Solution

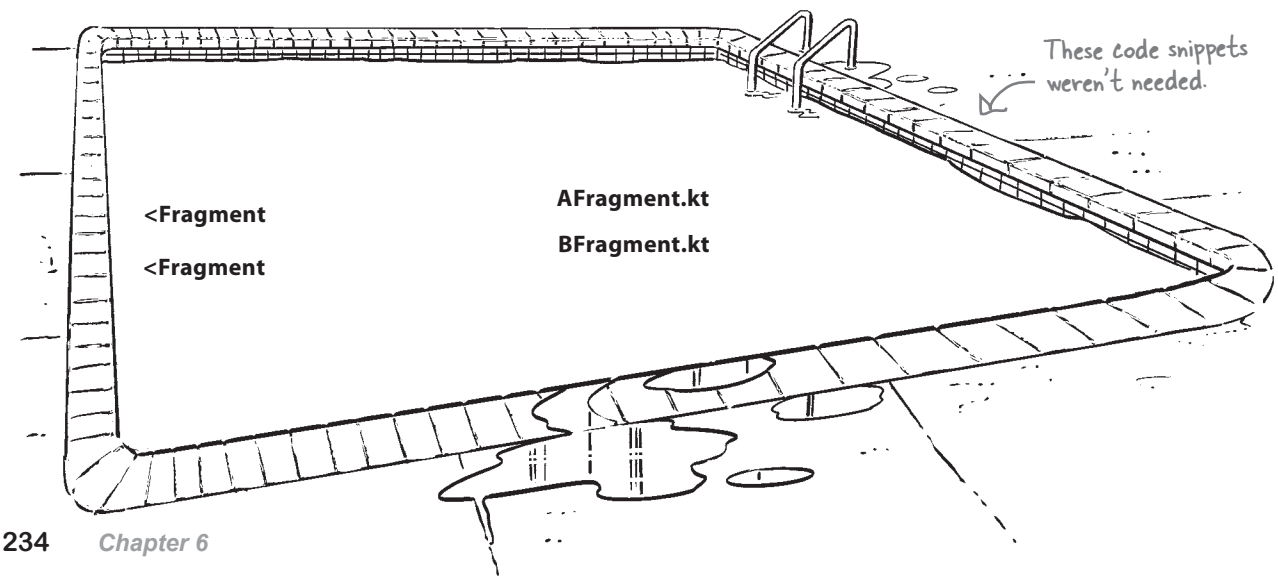
Your **job** is to take code snippets from the pool and place them into the blank lines in the linear layout. You may **not** use the same code snippet more than once, and you won't need to use all the snippets. Your **goal** is to make the linear layout display two fragments, `AFragment` and `BFragment`, so that it displays the screen shown on the right. Hint: both fragments are in a package named *com.hfad.exercise*.




```
<LinearLayout ...
    android:orientation="vertical">

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/a_fragment"
        android:name="com.hfad.exercise.AFragment"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1" />

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/b_fragment"
        android:name="com.hfad.exercise.BFragment"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1" />
</LinearLayout>
```

Use this element to add a fragment to the layout.

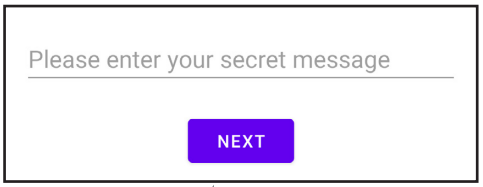
Use the android:name attribute to specify the fully qualified name of the fragment.



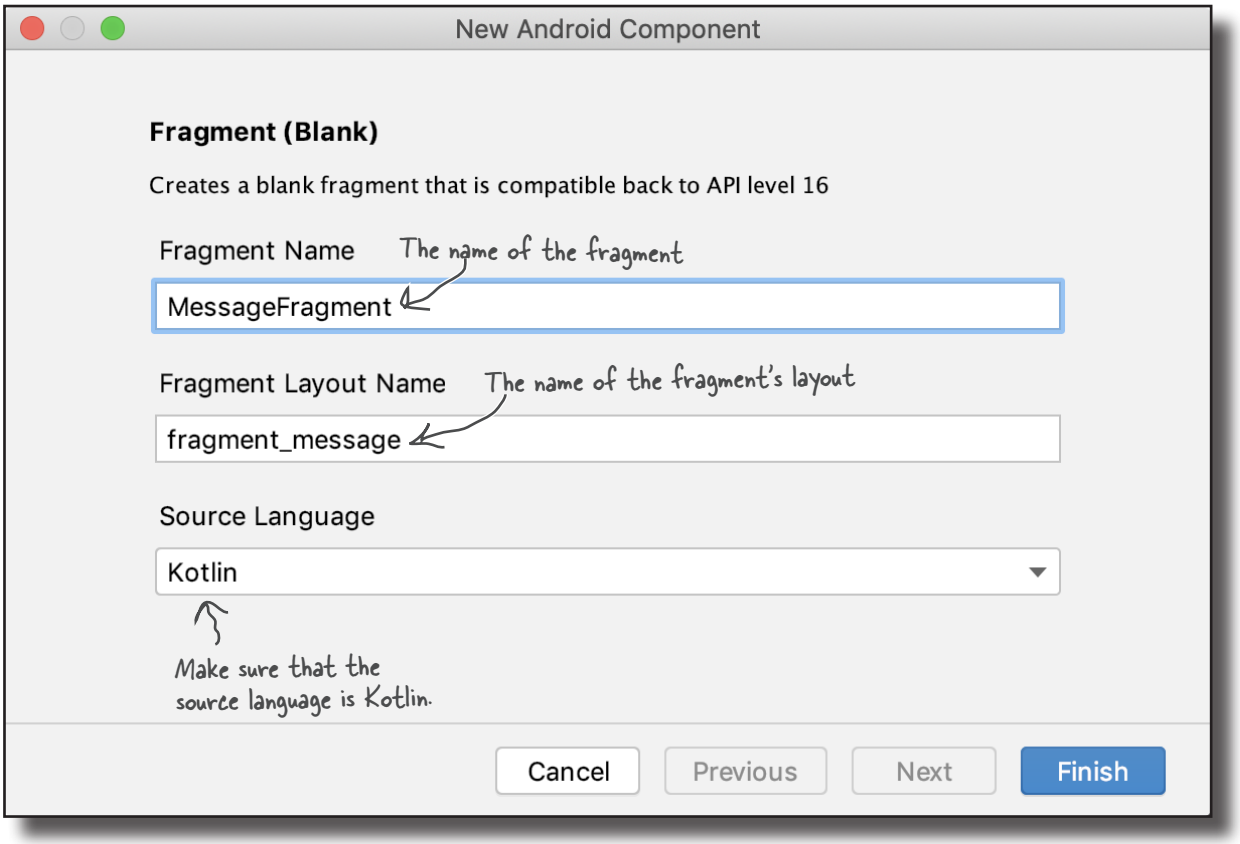
# Create MessageFragment

So far, we've created a fragment named `WelcomeFragment`, which is displayed in `MainActivity`'s layout. Next, we'll create a new fragment named `MessageFragment`, which we'll navigate to when the user clicks on `WelcomeFragment`'s Start button.

We'll add `MessageFragment` in the same way we added `WelcomeFragment`. Highlight the *com.hfad.secretmessage* package in the *app/src/main/java* folder in the project explorer, go to the File menu, and choose New→Fragment→Fragment (Blank). Name the fragment "MessageFragment" and its layout "fragment_message", and make sure the language is set to Kotlin. Then click on the Finish button to add the fragment and its layout to the project.




This is what MessageFragment will look like.



**WelcomeFragment**
**MessageFragment**

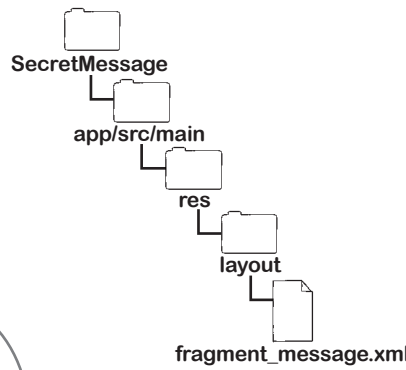
# Update MessageFragment's layout

When you create `MessageFragment`, Android Studio adds two new files to your project: *MessageFragment.kt* (which specifies the fragment's behavior), and *fragment_message.xml* (which defines its appearance). We'll update both of these files, starting with the layout.

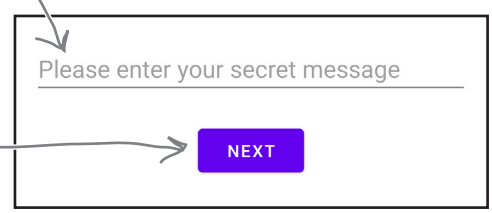
The fragment needs to have an edit text to let the user enter a message, and a button which, later on, will be used for navigation. You're already familiar with the code to add these views, so update the code in *fragment_message.xml* so it matches the code shown here:

```
<?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"
    android:gravity="center_horizontal"
    tools:context=".MessageFragment">

    <EditText
        android:id="@+id/message"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="20sp"
        android:hint="@string/message_hint"
        android:inputType="textMultiLine" />

    <Button
        android:id="@+id/next"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="20dp"
        android:text="@string/next" />
</LinearLayout>
```

We're using a LinearLayout for the fragment's layout.




The user will enter a message in the edit text.




In the next chapter, the user will use the button to navigate to another fragment.

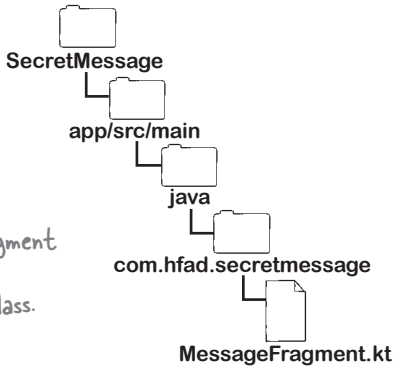
That's all the code we need for `MessageFragment`'s layout, so let's go and update its Kotlin code.

# Update MessageFragment.kt

`MessageFragment`'s Kotlin code defines the fragment's behavior. For now, all we need to do is make sure Android Studio hasn't added any unnecessary extra code to it that might stop it from working in the way we want.

Go to the *com.hfad.secretmessage* package in the *app/src/main/java* folder and open the file *MessageFragment.kt*. Then replace the code that Android Studio has generated with the code below:

```
package com.hfad.secretmessage

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup

class MessageFragment : Fragment() {

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        //Inflate the layout for this fragment
        return inflater.inflate(R.layout.fragment_message, container, false)
    }
}
```



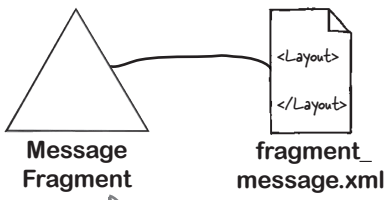

MessageFragment extends the Fragment class.

The fragment overrides onCreateView().

Use fragment_message.xml for the fragment's layout.

The above code is everything that *MessageFragment.kt* needs to define a basic fragment. Just like the code you saw for `WelcomeFragment`, it extends the `Fragment` class, and overrides its `onCreateView()` method. This method inflates the fragment's layout, and returns its root view. It gets called whenever the app needs to display the fragment.

We've now finished writing all the layout and Kotlin code that `MessageFragment` needs. Next, we need to make `WelcomeFragment` navigate to it. So how is this done?




MessageFragment uses fragment_message.xml for its layout.

**WelcomeFragment**
**MessageFragment**

# Use the Navigation component to navigate between fragments

As we said earlier in the chapter, the standard way of navigating between fragments is to use Android's Navigation component.

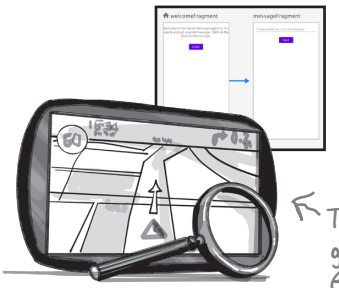
The Navigation component is part of Android Jetpack, and it's a suite of libraries, plug-ins, and tools that you add to your project. It's extremely flexible, and simplifies many of the complexities of fragment navigation—such as fragment transactions and back stack manipulation—which were previously much harder to implement.

Navigating between fragments is comprised of three main parts:

### A navigation graph

The navigation graph holds all of the navigation-related information that your app requires, and describes the possible paths the user can take when navigating the app.

The navigation graph is an XML resource, but you usually edit it using a visual design editor.



Navigation Graph

The navigation graph is like a GPS that tells you how to get to different destinations.

### A navigation host

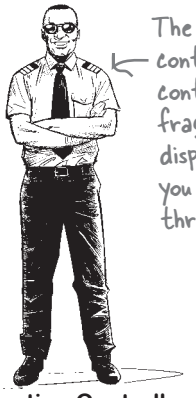
A navigation host is an empty container that's used to display the fragment you navigate to. You add the navigation host to your activity's layout.

Navigation Host

The navigation host is an empty container that you add to your activity's layout.

### A navigation controller

The navigation controller controls which fragment is displayed in the navigation host as the user navigates through the app. You interact with the navigation controller using Kotlin code.



Navigation Controller

The navigation controller controls which fragment is displayed when you navigate through the app.

We're going to use all three of these to implement navigation in the Secret Message app. First, let's add the Navigation component's library to the project.

**WelcomeFragment**
**MessageFragment**

# Use Gradle to add the Navigation component to your project

As you learned in Chapter 4, you add any extra libraries, tools, and plug-ins your app needs by making changes to the *build.gradle* files. When you create a new project, Android Studio automatically includes two of these files for you: one for the project, and another for the app.

To add the Navigation component, you need to edit *both* versions of *build.gradle*. Let's begin by updating the project version.

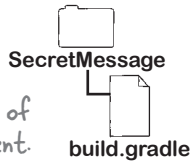
**Always click on the Sync Now option after updating the *build.gradle* files.**

*Any changes you make (such as adding libraries) don't come into effect until you sync them.*

## Add a version number to the project build.gradle file

We'll begin by adding a new variable to the project's *build.gradle* file that specifies which version of the Navigation component we're going to use. Using a variable for this version number means that if we add any extra Navigation component libraries (which we'll do in the next chapter), the version number for each library will stay nice and consistent.

To add the variable, open the file *SecretMessage/build.gradle*, and add the following line (in bold) to the `buildscript` section:



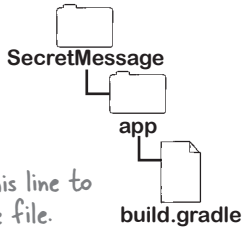

```
buildscript {
    ext.nav_version = "2.3.5"
    ...
}
```

We're using this version of the Navigation component.

## Add a dependency to the app build.gradle file

Next, you need to add a library dependency to the app's version of the *build.gradle* file.

Open the file *SecretMessage/app/build.gradle*, and add the following line (in bold) to the `dependencies` section:




Make sure you add this line to your app's build.gradle file.

```
dependencies {
    ...
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
    ...
}
```

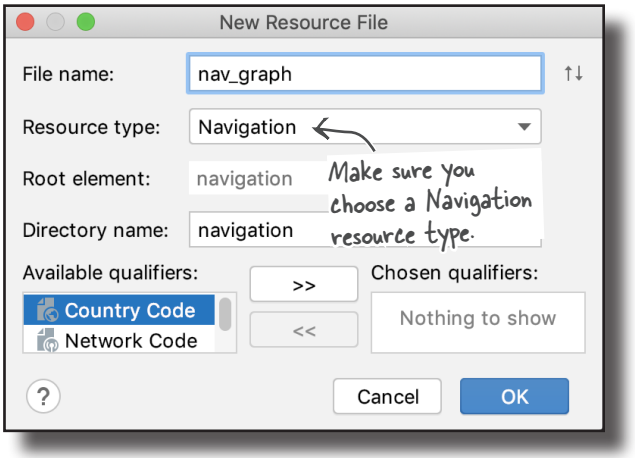
Once you've made these changes, click on the Sync Now option that appears at the top of the code editor. This syncs the changes you've made with the rest of your project, and adds the library.

**WelcomeFragment**
**MessageFragment**

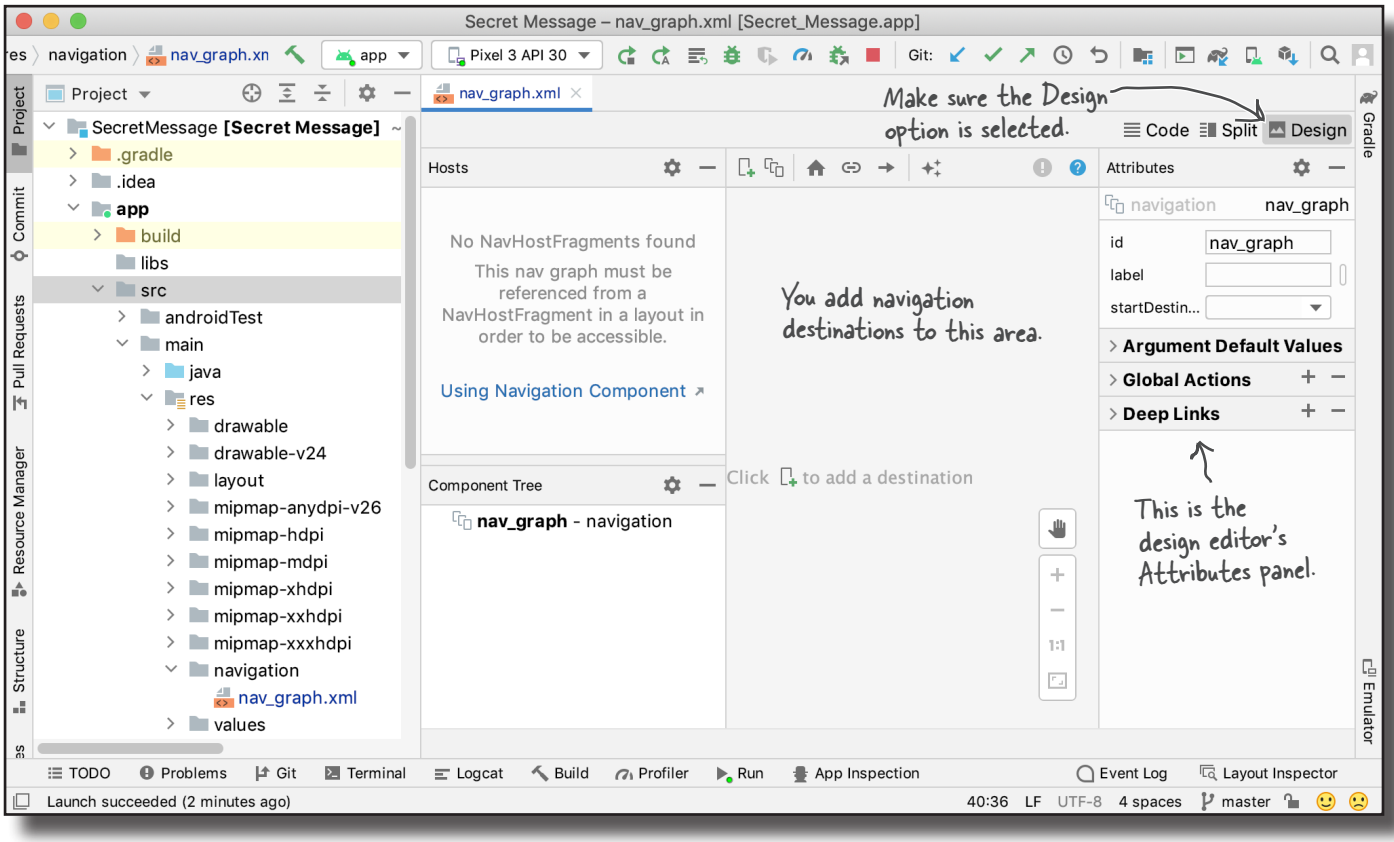
# Create a navigation graph

Now that we've added the Navigation component's main library to the Secret Message project, we can implement navigation.

We'll start by adding a navigation graph to the project. Select the *SecretMessage/app/src/main/res* folder in the project explorer, and then choose File→New→Android Resource File. When prompted, enter a file name of "nav_graph", choose a resource type of "Navigation", and click on the OK button. This adds an empty navigation graph file named *nav_graph.xml* to the *SecretMessage/app/src/main/res/navigation* folder.




Once you've created the new navigation graph, open it (if it's not open already) by double-clicking on the *nav_graph.xml* file in the project explorer. The file should open in the navigation graph design editor, which looks like this:

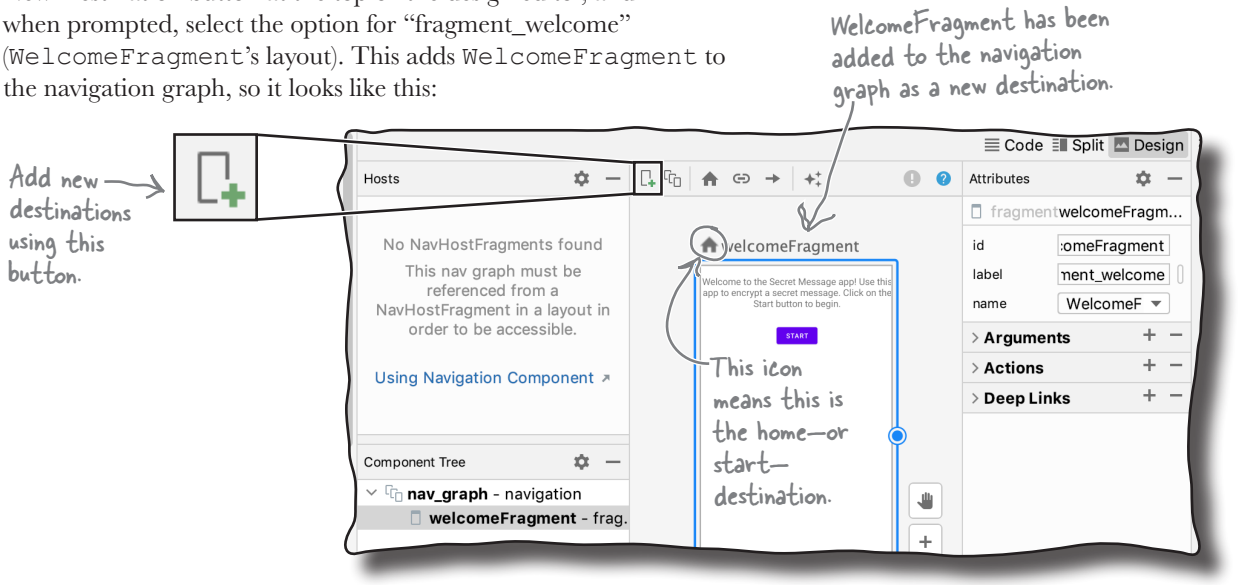


**WelcomeFragment**
**MessageFragment**

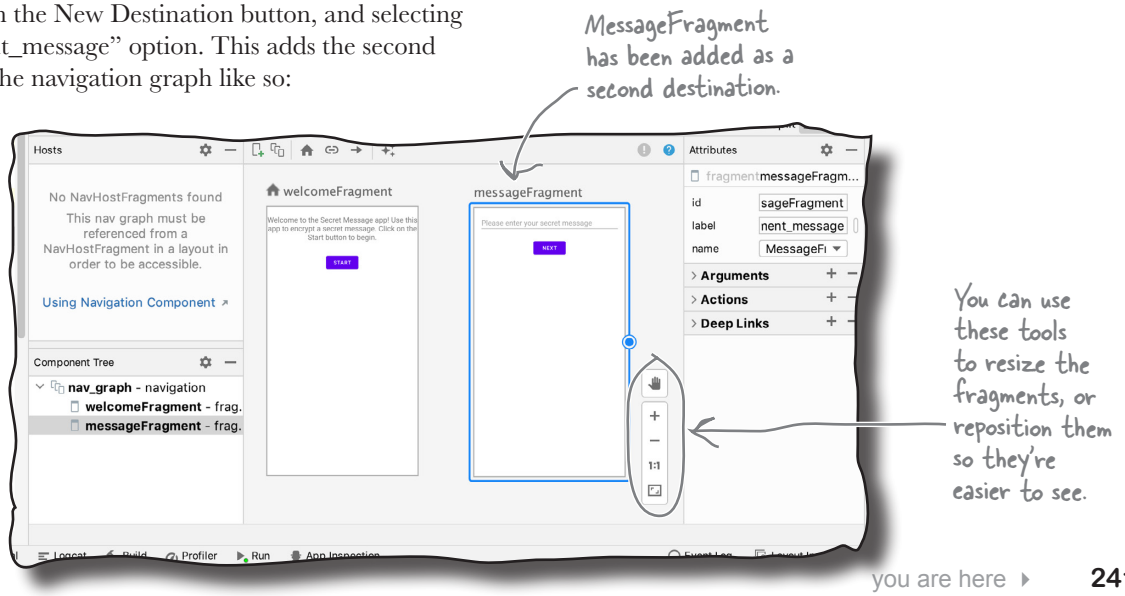
# Add fragments to the navigation graph

We want the user to be able to navigate from `WelcomeFragment` to `MessageFragment`, so we need to add these fragments to the navigation graph as **destinations**. A destination is a screen in the app—usually a fragment—which the user can navigate to.

We'll add `WelcomeFragment` first, as this is the first screen we want the user to see when the app launches. Click on the New Destination button at the top of the design editor, and when prompted, select the option for "fragment_welcome" (`WelcomeFragment`'s layout). This adds `WelcomeFragment` to the navigation graph, so it looks like this:




Next, add `MessageFragment` to the navigation graph by clicking on the New Destination button, and selecting the "fragment_message" option. This adds the second fragment to the navigation graph like so:

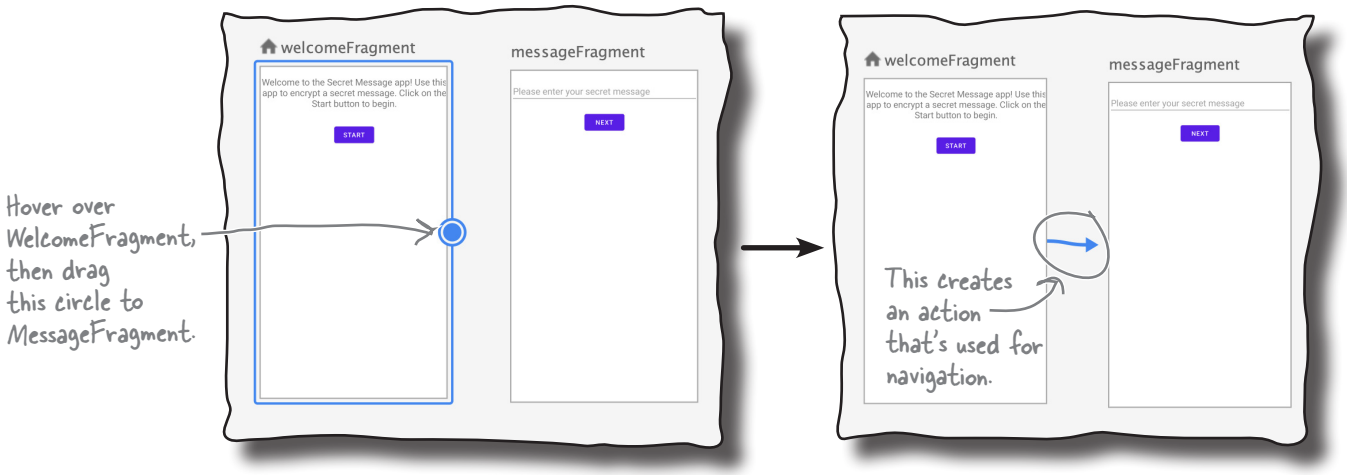


**WelcomeFragment**
**MessageFragment**

# Connect fragments using an action

We next need to specify that the user can navigate from `WelcomeFragment` to `MessageFragment`, and this is done with an **action**. Actions are used to connect destinations in the navigation graph, and they define possible paths the user can take when navigating through the app.

We're going to add an action that goes from `WelcomeFragment` to `MessageFragment`, as this is the direction in which we want the user to navigate through the app. Hover your mouse pointer over `WelcomeFragment` in the design editor, then click on the circle that appears on its right edge, and drag it to `MessageFragment`. This draws an arrow—the action—between the two fragments:




## Every action needs a unique ID

Every action must have a unique ID. Android uses this ID to determine which destination needs to be displayed as the user navigates through the app.

Each time you create an action, Android Studio assigns it a default ID. You can edit this ID—and any other of the action's properties—using the Attributes panel on the right of the navigation graph.

You want the action you just created to have an ID of "action_welcomeFragment_to_messageFragment" so that it matches the code in this chapter. Make sure this is the case by selecting the action (the arrow) in the design editor, and checking the value of its `id` attribute in the Attributes panel. You'll use this ID a few pages ahead.

# Navigation graphs are XML resources

Just like a layout, a navigation graph is really just a bunch of XML code. To view the code, click on the Code button at the top of the design editor.

Here's the underlying XML code for the Secret Message app's navigation graph *nav_graph.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/welcomeFragment">

    <fragment
        android:id="@+id/welcomeFragment"
        android:name="com.hfad.secretmessage.WelcomeFragment"
        android:label="fragment_welcome"
        tools:layout="@layout/fragment_welcome" >
        <action
            android:id="@+id/action_welcomeFragment_to_messageFragment"
            app:destination="@id/messageFragment" />
    </fragment>

    <fragment
        android:id="@+id/messageFragment"
        android:name="com.hfad.secretmessage.MessageFragment"
        android:label="fragment_message"
        tools:layout="@layout/fragment_message" />
</navigation>
```

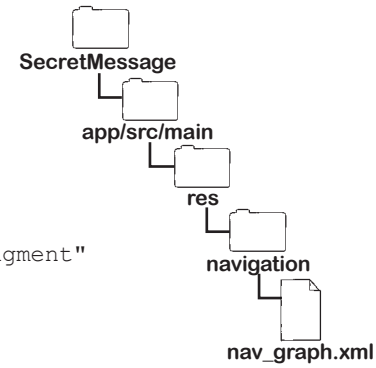
The navigation graph has a <navigation> root element.

Navigation starts with WelcomeFragment.

This section describes Welcome Fragment.

WelcomeFragment has an action that goes to MessageFragment.

This section describes Message Fragment.




As you can see, *nav_graph.xml* has a root element of `<navigation>`, and contains two `<fragment>` elements: one for `WelcomeFragment`, and another for `MessageFragment`. The `<fragment>` element for `WelcomeFragment` includes an extra `<action>` element, which denotes the action we just added.

Now that we've created the navigation graph, let's move onto the next part of the Navigation component.

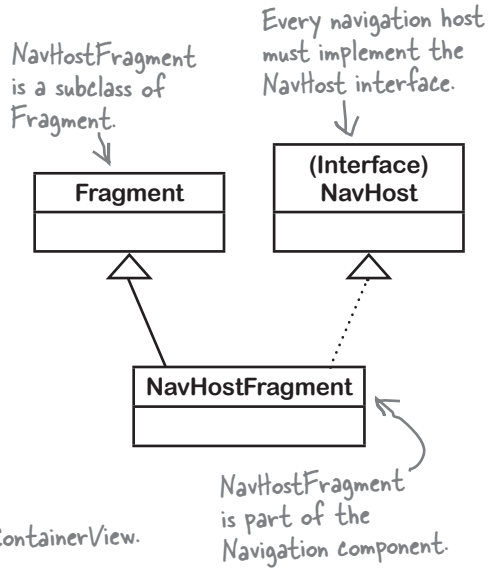
**Navigation graphs are usually edited using the design editor, but it can still be useful to check the XML.**

**WelcomeFragment**
**MessageFragment**

# Add a navigation host to the layout using a FragmentContainerView

As we said earlier, the Navigation component is comprised of three main parts: a navigation graph that defines the possible navigation paths, a navigation host that displays destinations, and a navigation controller that controls which destination is displayed. We've just created a navigation graph, so the next thing we'll add is the navigation host.

You add a navigation host by including one in the activity's layout. The great news is that the Navigation component comes with a built-in one named **`NavHostFragment`** so you don't have to write one yourself. It's a subclass of `Fragment` that implements the `NavHost` interface.




As `NavHostFragment` is a type of fragment, you add it to a layout file using a `FragmentContainerView`. The code looks like this:

This is the FragmentContainerView.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/nav_host_fragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:name="androidx.navigation.fragment.NavHostFragment"
    app:navGraph="@navigation/nav_graph"
    app:defaultNavHost="true" />
```

The FragmentContainerView holds a NavHostFragment.

There are two extra attributes.

The above code is like the `FragmentContainerView` code you've seen before, but it includes two extra attributes: **`app:navGraph`** and **`app:defaultNavHost`**.

The `app:navGraph` attribute tells the navigation host which navigation graph to use, in this case *nav_graph.xml*. The navigation graph specifies which fragment to display first (its start destination) and lets the user navigate between its destinations.

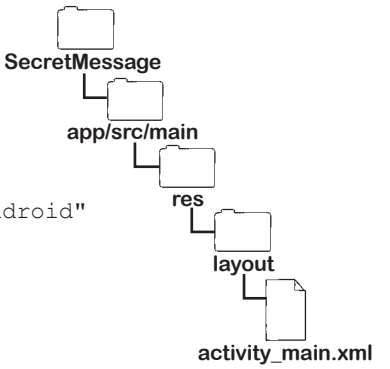
The `app:defaultNavHost` attribute lets the navigation host interact with the device back button: you'll find out more about this in the next chapter.

NavHostFragment is the navigation host. It displays the fragment the user has navigated to, starting with the navigation graph's start destination.

**WelcomeFragment**
**MessageFragment**

# Add a NavHostFragment to activity_main.xml

We're going to add a navigation host to `MainActivity`'s layout that uses the navigation graph we created. To do this, update the code in *activity_main.xml* so it includes the changes below (in bold).




```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/fragment_container_view"
    android:id="@+id/nav_host_fragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:name="com.hfad.secretmessage.WelcomeFragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    app:navGraph="@navigation/nav_graph"
    app:defaultNavHost="true"
    tools:context=".MainActivity" />
```

We're using extra attributes from this namespace.

Change the id.

Specify the navigation graph resource file.

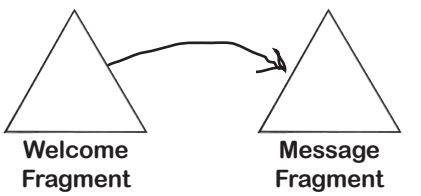
Update the name so we're using a NavHostFragment.

This lets the navigation host interact with the device back button. You'll learn more about this in Chapter 7.

## We need to navigate between the fragments

We've now created a navigation graph, and linked it to a navigation host that's held in a `FragmentContainerView` in `MainActivity`'s layout. When the app runs, `WelcomeFragment`—which is the navigation graph's start destination—will be displayed.

The final thing we need to do in this chapter is navigate from `WelcomeFragment` to `MessageFragment` when the user clicks on the Start button in `WelcomeFragment`'s layout. Let's find out how this is done.



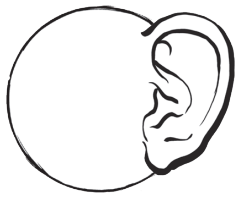
**WelcomeFragment**
**MessageFragment**

# Add an OnClickListener to the button

To navigate from `WelcomeFragment` to `MessageFragment`, we first need to make `WelcomeFragment`'s Start button respond to clicks. We'll do this by adding an `OnClickListener` to it.

Previously, you added `OnClickListener` to an *activity's* button by getting a reference to it with `findViewById()`, and then calling its `setOnClickListener` method. You included this code in the activity's `onCreate()` method as this is where it first has access to the views in its layout.

When you want to add an `OnClickListener` to a *fragment's* button, however, things are slightly different.

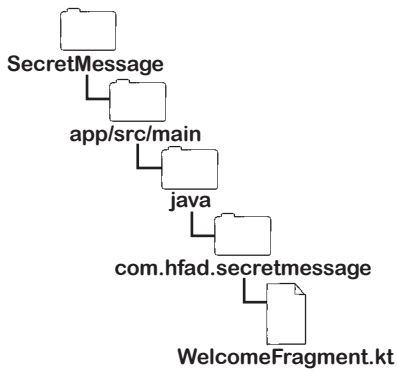



## Fragment OnClickListener code is a little different

The first difference is that you add an `OnClickListener` to a *fragment's* button in the fragment's **`onCreateView()`** method, not `onCreate()`. This is because a fragment first has access to its views in `onCreateView()`, so it's the best place to set any `OnClickListeners`.

The second difference is **the `Fragment` class doesn't include a `findViewById()` method**, so you can't directly call it to get a reference to any views. You can, however, call `findViewById()` on the fragment's root view instead.

Here's what the code looks like to add an `OnClickListener` to a view in fragment code: we'll add this to `WelcomeFragment` a couple pages ahead:



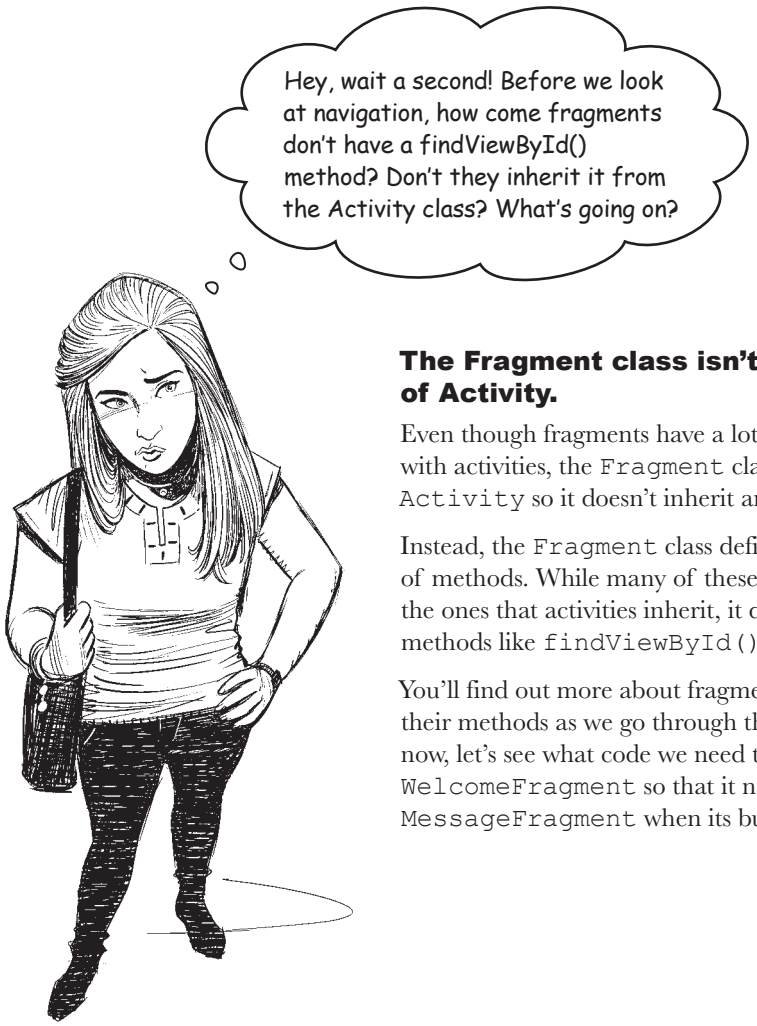

```
...
class WelcomeFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        val view = inflater.inflate(R.layout.fragment_welcome, container, false)
        val startButton = view.findViewById<Button>(R.id.start)

        startButton.setOnClickListener {
            //Code that runs when the button is clicked
        }
        return view
    }
}
```

This gets a reference to the fragment's root view.

This uses the root view to get a reference to the fragment's Start button.

Return the root view.

Now that you know how to add an `OnClickListener` to a fragment's button, let's make it navigate when clicked.




### The Fragment class isn't a subclass of Activity.

Even though fragments have a lot in common with activities, the `Fragment` class doesn't extend `Activity` so it doesn't inherit any of its methods.

Instead, the `Fragment` class defines its own set of methods. While many of these look identical to the ones that activities inherit, it doesn't include methods like `findViewById()`.

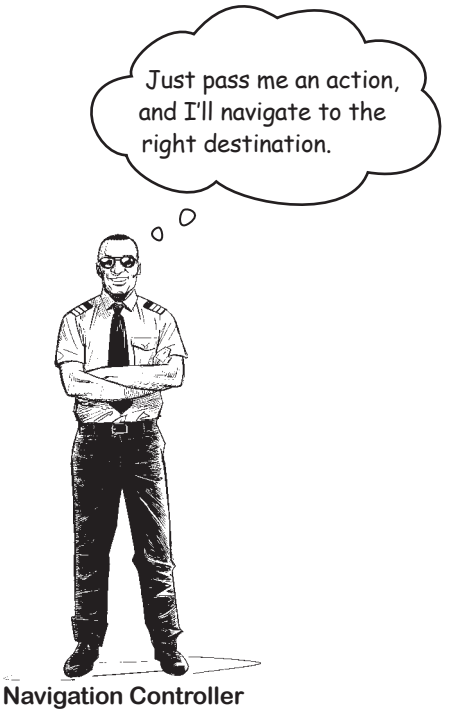
You'll find out more about fragments and their methods as we go through the book. For now, let's see what code we need to add to `WelcomeFragment` so that it navigates to `MessageFragment` when its button gets clicked.

**WelcomeFragment**
**MessageFragment**

# Get a navigation controller

Each time you want to navigate to a new fragment, you first need to get a reference to a navigation controller. You do this by calling the **`findNavController()`** method on its root `View` object. The following code, for example, gets a reference to the navigation controller associated with a root view object named `view`:

```
val navController = view.findNavController()
```




## Tell it where to navigate with an action

Once you have a navigation controller, you ask it to navigate to a new destination by calling its **`navigate()`** method. This method takes one parameter: a navigation action ID.

As you may recall, when we created the navigation graph, we included an action that went from `WelcomeFragment` to `MessageFragment`. We gave this action the ID "action_welcomeFragment_to_messageFragment".

If we pass this ID to the navigation controller's `navigate()` method, the controller will see that the action goes from `WelcomeFragment` to `MessageFragment`, and use it to navigate to the new fragment.

Here's what the code looks like:

```
view.findNavController().navigate(R.id.action_welcomeFragment_to_messageFragment)
```

Once you have a reference to the navigation controller, you ask it to navigate using an action ID.

We want to navigate to `MessageFragment` when the user clicks on `WelcomeFragment`'s Start button. We'll therefore add this code to the Start button's `OnClickListener` like this:

```
val view = inflater.inflate(R.layout.fragment_welcome, container,
false)
val startButton = view.findViewById<Button>(R.id.start)
startButton.setOnClickListener {
    view.findNavController()
        .navigate(R.id.action_welcomeFragment_to_messageFragment)
}
```

Get the navigation controller.

Navigate to MessageFragment.

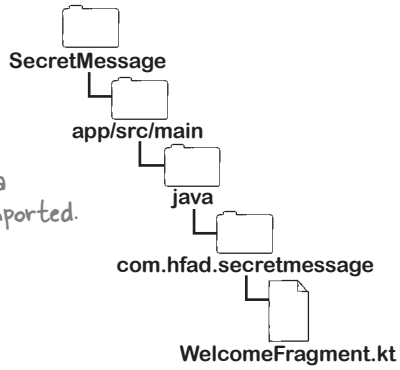
We'll show you the full `WelcomeFragment` code on the next page.

# The full code for WelcomeFragment.kt

Below is the complete code for `WelcomeFragment`; update *WelcomeFragment.kt* so it includes the changes (in bold):

```
package com.hfad.secretmessage

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Button
import androidx.navigation.findNavController

class WelcomeFragment : Fragment() {

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        val view = inflater.inflate(R.layout.fragment_welcome, container, false)
        val startButton = view.findViewById<Button>(R.id.start)

        startButton.setOnClickListener {
            view.findNavController()
                .navigate(R.id.action_welcomeFragment_to_messageFragment)
        }
        return view
    }
}
```

We're using these two extra classes, which need to be imported.




We need to get a reference to the fragment's root view so we can call its findViewById() method.

Navigate to MessageFragment when the user clicks the button.

Return the fragment's root view.

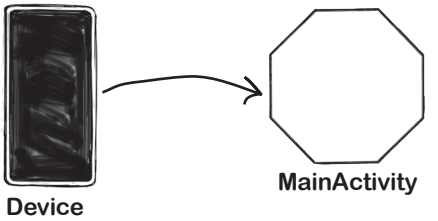
That's all the code we need to make `WelcomeFragment` navigate to `MessageFragment`. Let's go through what the code does when it runs, and then take the app for a test drive.

WelcomeFragment
**MessageFragment**

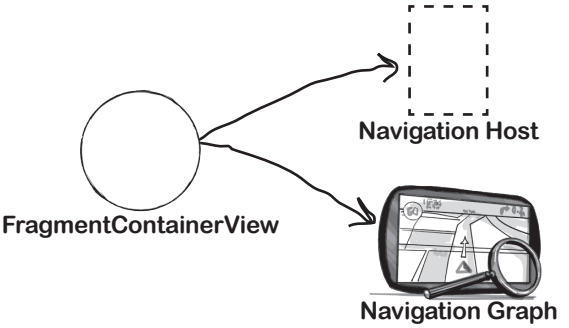
# What happens when the app runs

The following things happen when the app runs:

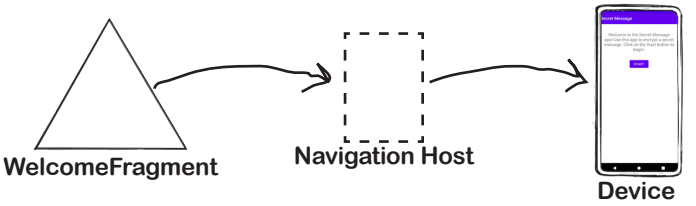
1. **The app is launched and MainActivity gets created.**



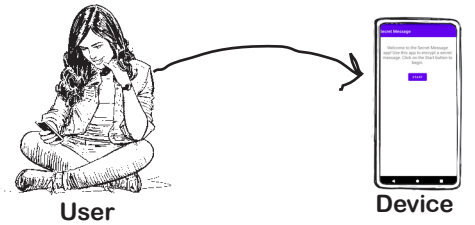

2. **MainActivity's layout, activity_main.xml, includes a FragmentContainerView that specifies a navigation host and a navigation graph.**




3. **The start destination in the navigation graph is WelcomeFragment, so this fragment is added to the navigation host and displayed on the device screen.**

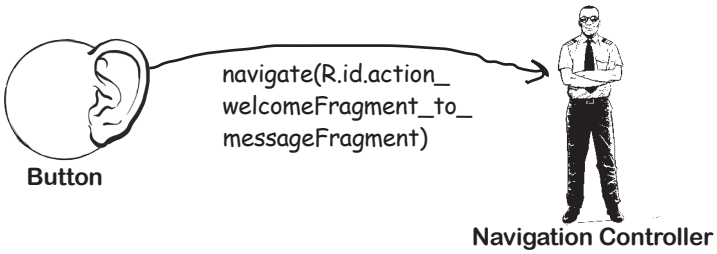



4. **The user clicks on the Start button in WelcomeFragment's layout.**

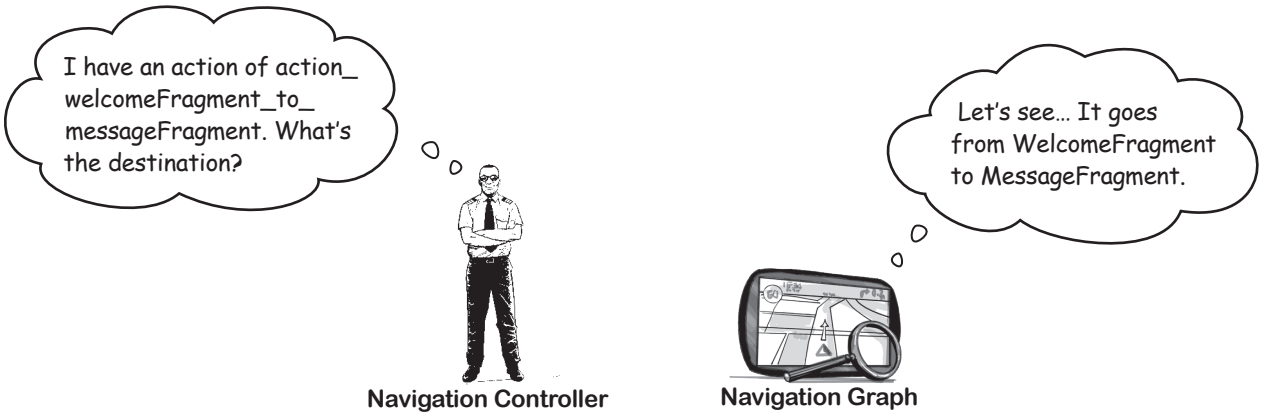


# The story continues

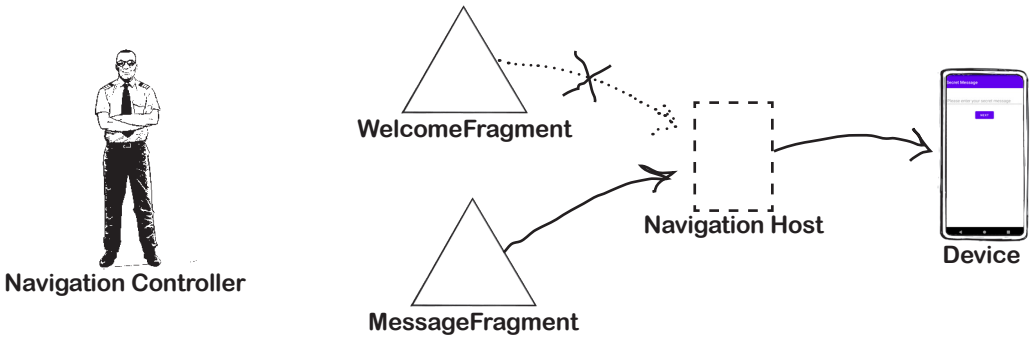
**5** **The code for the Start button's OnClickListener finds the navigation controller, and calls its navigate() method.**
It passes it the action used to navigate from `WelcomeFragment` to `MessageFragment`.




**6** **The navigation controller looks for the action with this ID in the navigation graph.**
It sees that the action goes from `WelcomeFragment` to `MessageFragment`.




**7** **The navigation controller replaces WelcomeFragment with MessageFragment in the navigation host, and MessageFragment is displayed on the device screen.**

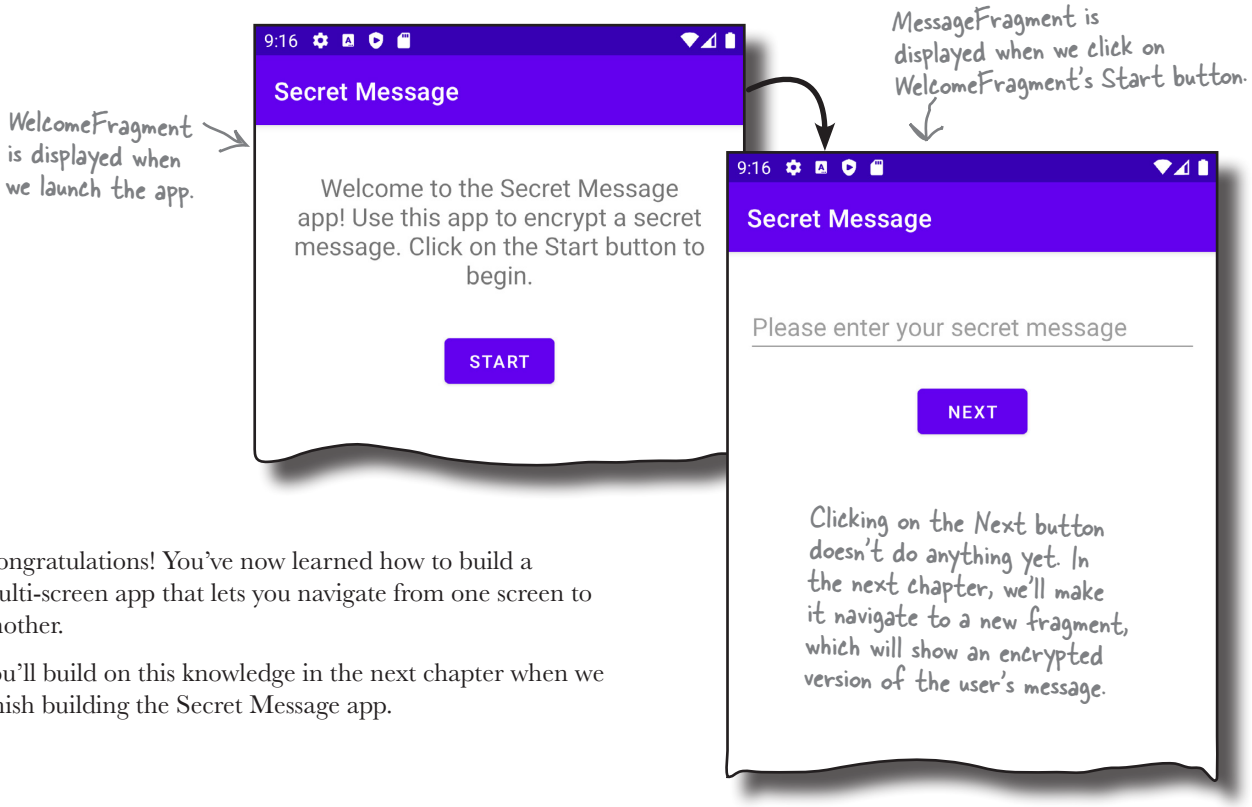


# Test Drive

WelcomeFragment
MessageFragment

When we run the app, `MainActivity` is launched, and `WelcomeFragment` is displayed as before.

When we click on the Start button, the app navigates to `MessageFragment`, and this fragment is displayed on the device.




Congratulations! You've now learned how to build a multi-screen app that lets you navigate from one screen to another.

You'll build on this knowledge in the next chapter when we finish building the Secret Message app.

## there are no Dumb Questions

**Q: I tried creating a navigation graph before editing the *build.gradle* files, and Android Studio offered to add the Navigation component library for me. Why are we adding the libraries manually?**

A: It's so you can see what libraries are needed, and how to include them, without relying on Android Studio.

Another reason is we know the code in this book works with the library version we've specified in the *build.gradle* files. Android Studio may include a different version, which might break the code.

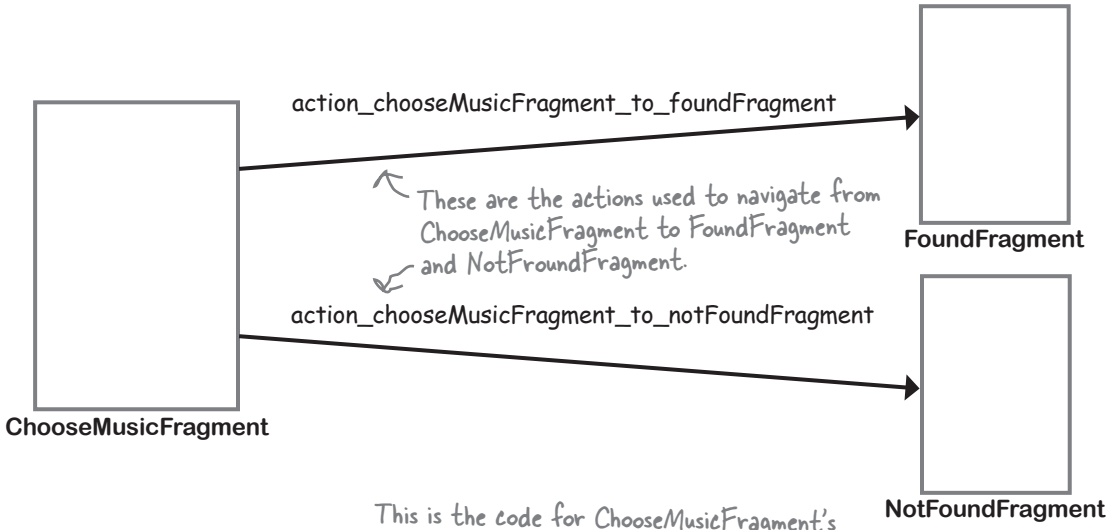
**Q: Do I have to implement navigation using the Navigation component?**

A: Strictly speaking, no, but it's far more complicated if you don't. Before the Navigation component was introduced, navigating between screens required a *lot* more code.

# Sharpen your pencil

The diagram below shows part of the navigation graph for a music app. It describes two navigation paths: one that goes from `ChooseMusicFragment` to `FoundFragment`, and another that goes from `ChooseMusicFragment` to `NotFoundFragment`.

Complete `ChooseMusicFragment`'s `onCreateView()` method so that if the `musicFound()` method returns *true*, the app displays `FoundFragment`, and if the method returns *false*, the app displays `NotFoundFragment` instead.



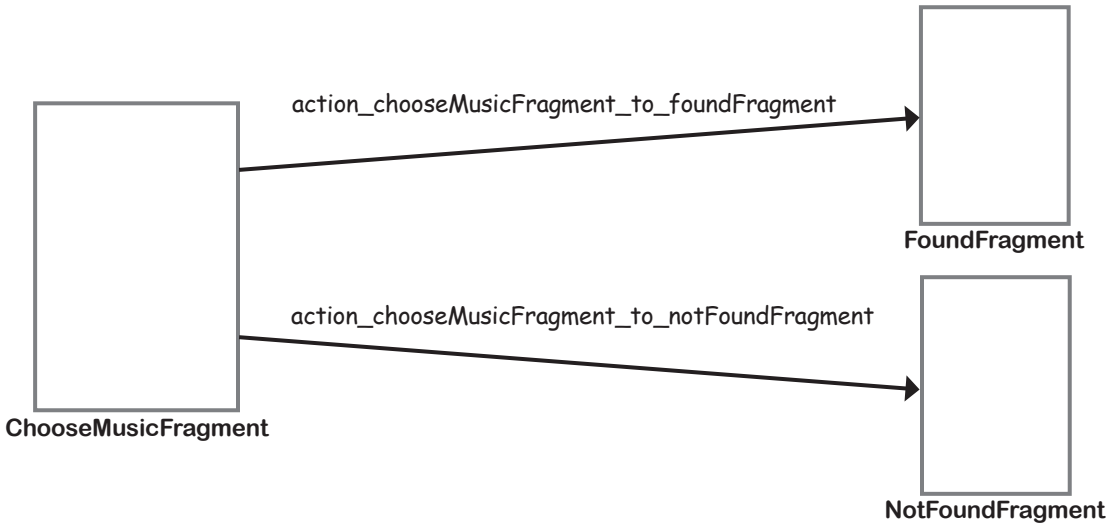

This is the code for ChooseMusicFragment's onCreateView method.

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                          savedInstanceState: Bundle?): View? {
    val view = inflater.inflate(R.layout.fragment_choose_music, container, false)
    val searchButton = view.findViewById<Button>(R.id.search)
    searchButton.setOnClickListener {
        if (musicFound()) {
            ..........
            ..........
        } else {
            ..........
            ..........
        }
    }
    return view
}
```

## Sharpen your pencil Solution

The diagram below shows part of the navigation graph for a music app. It describes two navigation paths: one that goes from `ChooseMusicFragment` to `FoundFragment`, and another that goes from `ChooseMusicFragment` to `NotFoundFragment`.

Complete `ChooseMusicFragment`'s `onCreateView()` method so that if the `musicFound()` method returns *true*, the app displays `FoundFragment`, and if the method returns *false*, the app displays `NotFoundFragment` instead.

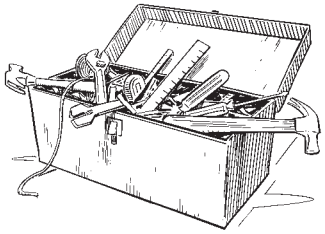



```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                          savedInstanceState: Bundle?): View? {
    val view = inflater.inflate(R.layout.fragment_choose_music, container, false)
    val searchButton = view.findViewById<Button>(R.id.search)
    searchButton.setOnClickListener {
        if (musicFound()) {
            view.findNavController()
                .navigate(R.id.action_chooseMusicFragment_to_foundFragment)
        } else {
            view.findNavController()
                .navigate(R.id.action_chooseMusicFragment_to_notFoundFragment)
        }
    }
    return view
}
```

Get a reference to the navigation controller.

Call the navigate method, and pass it an action ID that describes the navigation path.

# Your Android Toolbox



**You've got Chapter 6 under your belt and now you've added fragments and navigation to your toolbox.**

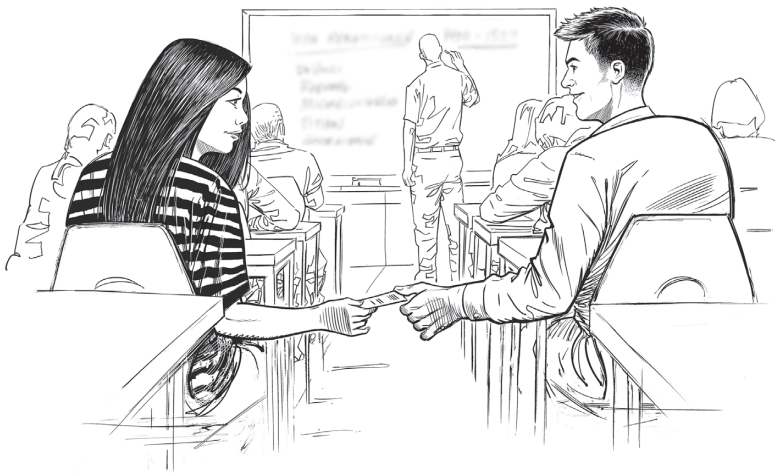
You can download the full code for the chapter from tinyurl.com/hfad3.

## Bullet Points

- A fragment has Kotlin code and a layout.
- Every fragment extends the `Fragment` class or one of its subclasses.
- Add a fragment to an activity's layout using a `FragmentContainerView`.
- `onCreateView()` gets called each time Android needs the fragment's layout.
- The `Fragment` class doesn't extend `Activity`.
- Fragments don't have a `findViewById()` method.
- The Navigation component is a suite of libraries, plug-ins, and tools which you add to your project using Gradle.
- A navigation graph describes possible destinations and navigation paths. It uses actions to describe these paths.
- A navigation host is an empty container used to display the fragment you navigate to. The Navigation component comes with a default navigation host named `NavHostFragment`, which extends the `Fragment` class and implements the `NavHost` interface.
- A navigation controller uses actions to control which fragment is displayed in the navigation host.

# 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. Read on—there's no going back…

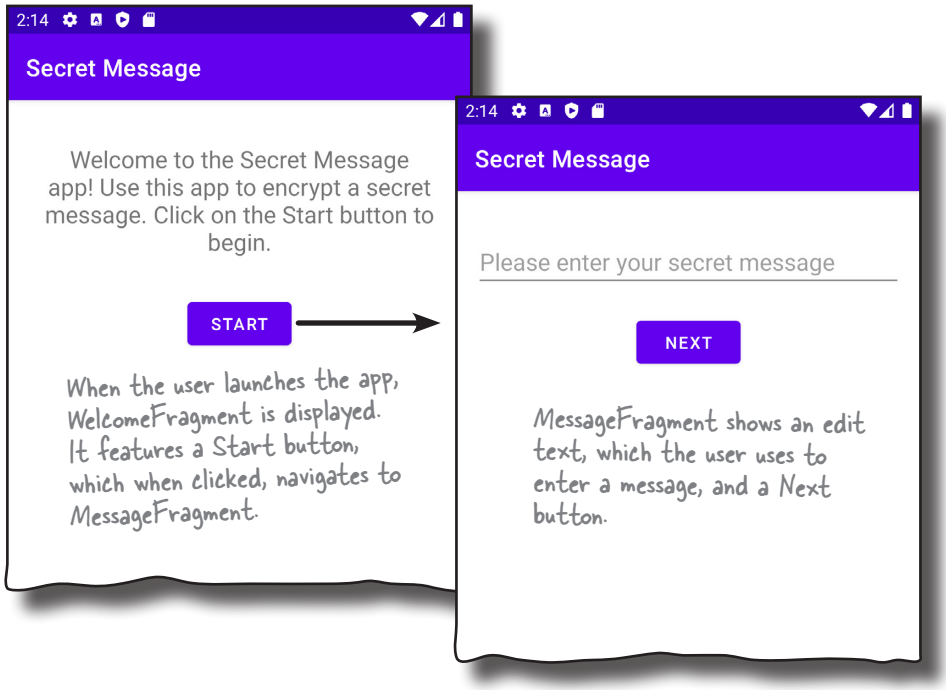
# The Secret Message app navigates between fragments

In the previous chapter, you learned how to use the Navigation component to navigate between two fragments. You used this knowledge to build the first half of the Secret Message app: an app that takes a message the user enters, and encrypts it.

The current version of the app uses a fragment named `WelcomeFragment` to display some introductory text. When the user clicks on its Start button, the fragment navigates to a second fragment named `MessageFragment`.

`MessageFragment` includes an edit text that lets the user enter her message. It also features a Next button, which the user clicks on to encrypt the message.

Here's a reminder of what the app currently looks like:

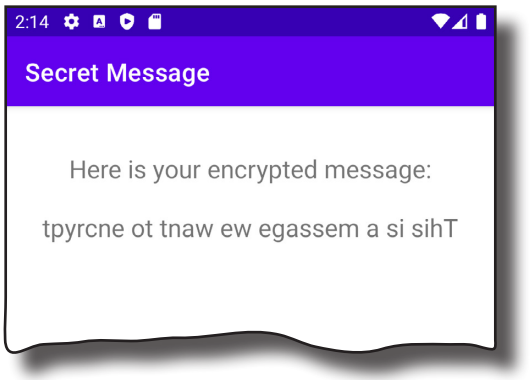



In the current version of the app, nothing happens when the user clicks on `MessageFragment`'s Next button. In this chapter, we're going to finish building the app so the user's encrypted message gets displayed in a new fragment.

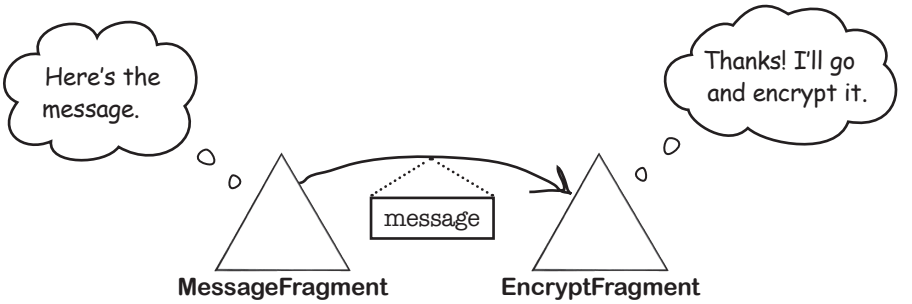
# MessageFragment needs to pass the message to a new fragment

We're going to add a new fragment to the app named `EncryptFragment`. This fragment will display the user's encrypted message, and will look like this:

This is a new fragment (which we need to create) named EncryptFragment. It displays an encrypted version of the user's message.




`EncryptFragment` will get the user's message from `MessageFragment`. When the user clicks on `MessageFragment`'s Next button, the app will navigate to `EncryptFragment`, and pass it the text. This fragment will then encrypt the text, and display the result:



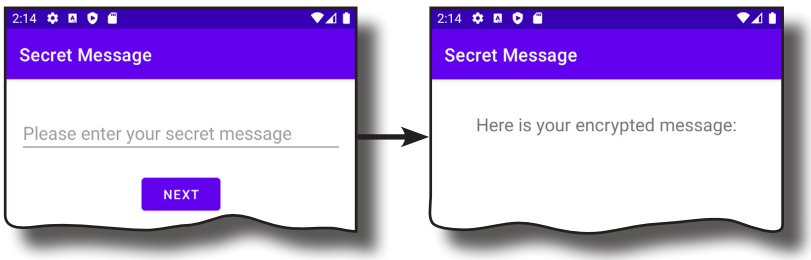

To make this work, we need to be able to pass data between fragments. The best way of doing this is with a Gradle plug-in called **Safe Args**, which is an extra part of the Navigation component. It gives you a way of passing data between fragments in a way that's type-safe. This prevents you from accidentally passing the wrong type of data, which might cause runtime errors.

You'll learn more about using Safe Args as we continue building the Secret Message app. For now, let's look at the steps we'll go through.

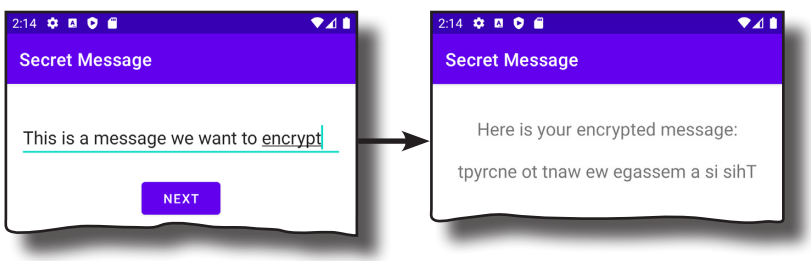
# Here's what we're going to do

Here are the steps we'll go through to build the second half of the Secret Message app:

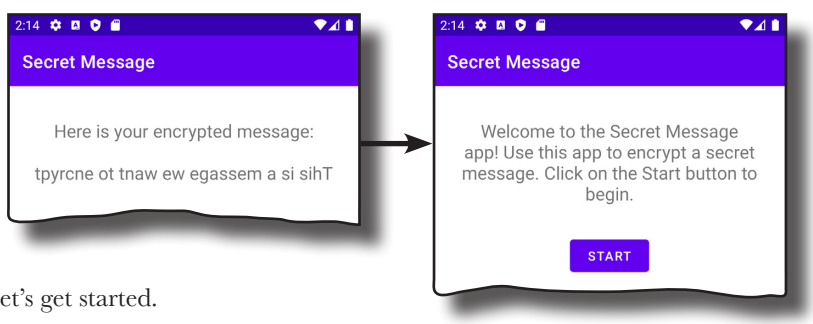
1. **Create and display EncryptFragment.**
We'll create a new fragment named `EncryptFragment`, and navigate to it when the user clicks on the Next button in `MessageFragment`'s layout.




2. **Pass the user's message to EncryptFragment.**
We'll use Safe Args to pass the user's message from `MessageFragment` to `EncryptFragment`. `EncryptFragment` will then display the encrypted message.



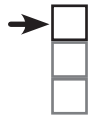

3. **Modify the app's back button behavior.**
Finally, we'll update the app so that when the user presses the device back button while `EncryptFragment` is displayed, the app goes back to `WelcomeFragment`.




Let's get started.

**Don't forget!**

You're going to modify the Secret Message app you created in Chapter 6, so make sure you open the project for this app.



EncryptFragment
Safe Args
Back button

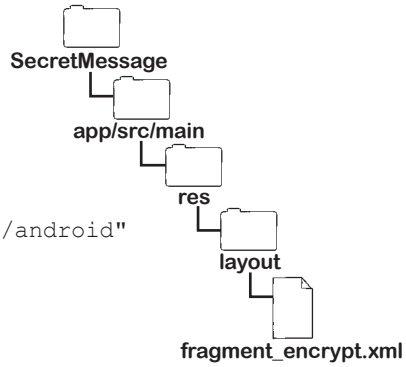
# Create EncryptFragment...

We're going to use `EncryptFragment` to display an encrypted version of the user's message. Create this fragment by highlighting the *com.hfad.secretmessage* package in the *app/src/main/java* folder, going to the File menu, and choosing New→Fragment→Fragment (Blank). Name the fragment "EncryptFragment" and give the layout a name of "fragment_encrypt". Make sure the language is set to Kotlin, then click on the Finish button.

## ...and update its layout

We're going to update `EncryptFragment`'s layout so that it includes two text views. The first will display a `String` resource named `encrypt_text` (we added this to *strings.xml* in the previous chapter), and the second will display the encrypted message.

Open the layout file *fragment_encrypt.xml* and update its contents so it matches the code below:




```
<?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:orientation="vertical"
    android:gravity="center_horizontal"
    tools:context=".EncryptFragment">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="20dp"
        android:textSize="20sp"
        android:text="@string/encrypt_text" />

    <TextView
        android:id="@+id/encrypted_message"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="20dp"
        android:textSize="20sp"/>
</LinearLayout>
```

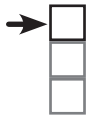
Define a vertical LinearLayout.

Display some text...

Here is your encrypted message:

tpyrcne ot tnaw ew egassem a si sihT

....and the top secret encrypted message.

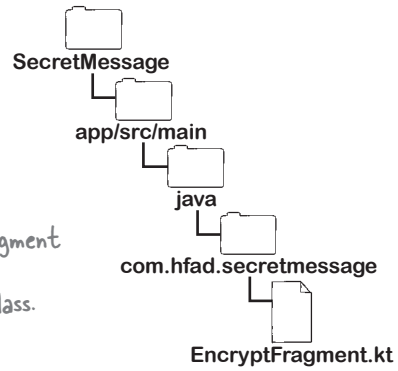


**EncryptFragment**
Safe Args
Back button

# Update EncryptFragment.kt

We also need to update `EncryptFragment`'s Kotlin code to make sure that Android Studio hasn't added any unnecessary extra code to it that might stop it from working in the way we want.

Go to the *com.hfad.secretmessage* package in the *app/src/main/java* folder and open the file *EncryptFragment.kt*. Then replace the code that Android Studio has generated with the code shown here:




```
package com.hfad.secretmessage

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup

class EncryptFragment : Fragment() {

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.fragment_encrypt, container, false)
    }
}
```

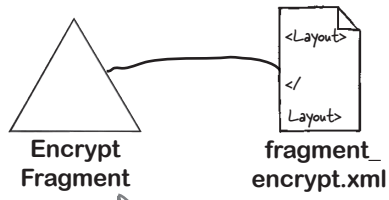
EncryptFragment extends the Fragment class.

The fragment overrides onCreateView().

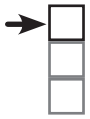
Use fragment_encrypt.xml for the fragment's layout..

The above code is everything that *EncryptFragment.kt* needs to define a basic fragment. Just like the other fragment code you've seen, it extends the `Fragment` class, and overrides its `onCreateView()` method. This method inflates the fragment's layout, and returns its root view.

We've now finished writing all the layout and Kotlin code that `EncryptFragment` needs. Next, we'll make sure that `MessageFragment` can navigate to it by adding it to the navigation graph.




EncryptFragment uses fragment_encrypt.xml for its layout.

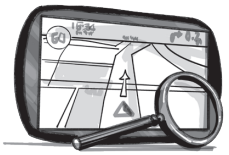


**EncryptFragment**
**Safe Args**
**Back button**

# Add EncryptFragment to the navigation graph

As you already know, the navigation graph holds details of the app's destinations, and the possible paths you can take to navigate to them.

To add `EncryptFragment` to the navigation graph, open *nav_graph.xml*, click on the New Destination button in the design editor, and when prompted, choose the "fragment_encrypt" option. This adds the fragment.

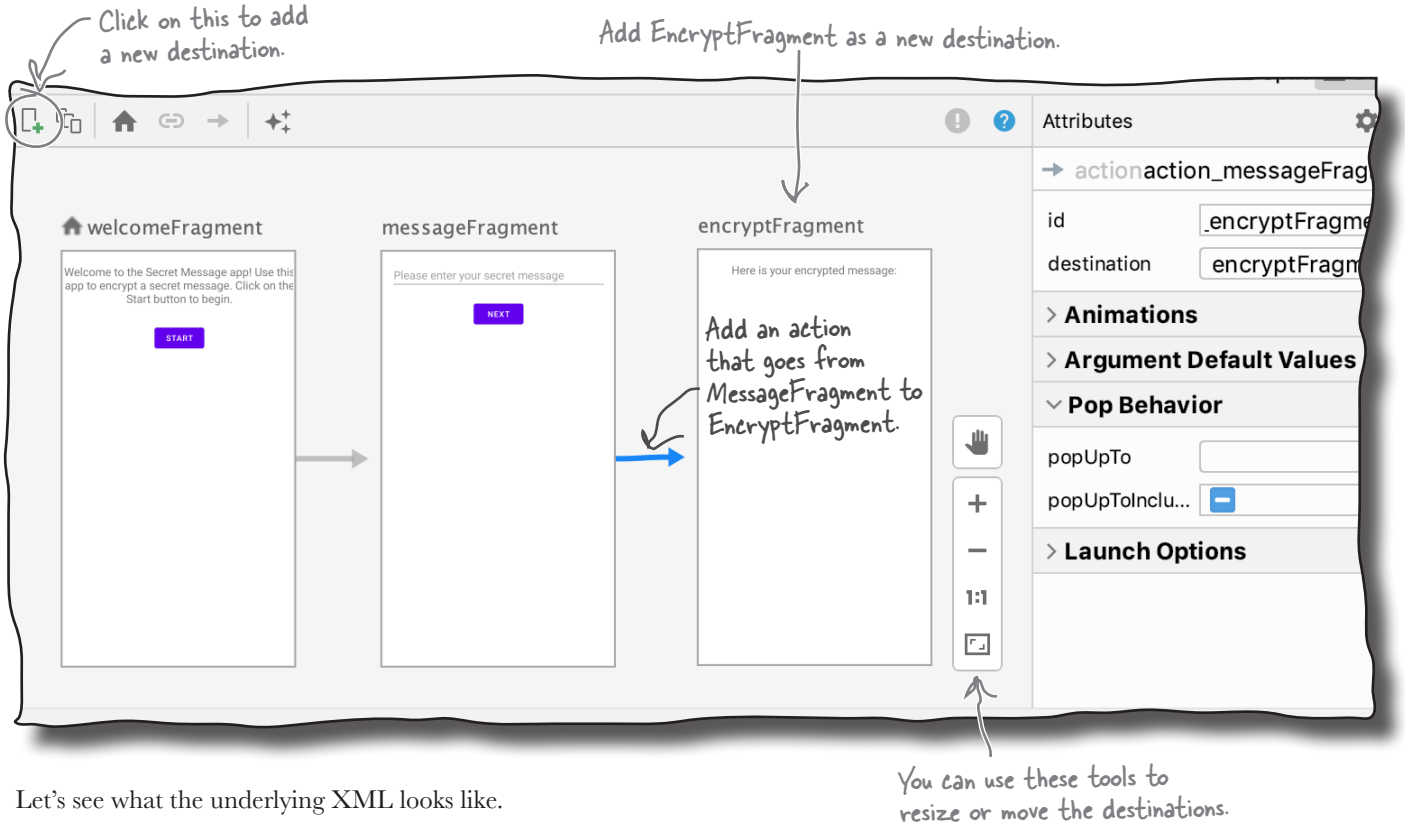


Navigation Graph

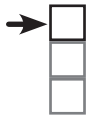
We also need to add a new action so that `MessageFragment` can navigate to `EncryptFragment`. Do this by hovering your mouse pointer over `MessageFragment` and drawing an action arrow from its right edge to `EncryptFragment`. Make sure the action has an ID of "action_messageFragment_to_encryptFragment" so that it matches the code in this book.

Don't worry if you find this a bit too fiddly. We'll show you the full code on the next page so if you prefer, you can update that instead.

After you've made these changes, the navigation graph should look something like this:




Let's see what the underlying XML looks like.

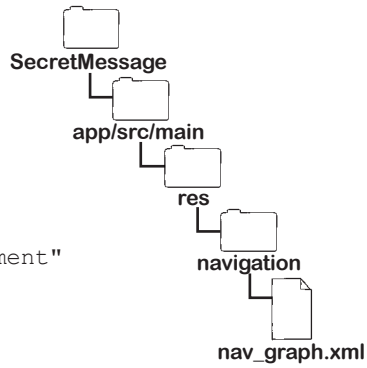


**EncryptFragment**
Safe Args
Back button

# The updated nav_graph.xml code

Each time you update the navigation graph, any changes are added to the underlying XML. The new code looks like this (the changes are in bold):

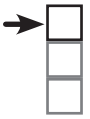
```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/welcomeFragment">

    <fragment
        android:id="@+id/welcomeFragment"
        android:name="com.hfad.secretmessage.WelcomeFragment"
        android:label="fragment_welcome"
        tools:layout="@layout/fragment_welcome" >
        <action
            android:id="@+id/action_welcomeFragment_to_messageFragment"
            app:destination="@id/messageFragment" />
    </fragment>

    <fragment
        android:id="@+id/messageFragment"
        android:name="com.hfad.secretmessage.MessageFragment"
        android:label="fragment_message"
        tools:layout="@layout/fragment_message" >
        <action
            android:id="@+id/action_messageFragment_to_encryptFragment"
            app:destination="@id/encryptFragment" />
    </fragment>

    <fragment
        android:id="@+id/encryptFragment"
        android:name="com.hfad.secretmessage.EncryptFragment"
        android:label="fragment_encrypt"
        tools:layout="@layout/fragment_encrypt" />
</navigation>
```




There's a new action that goes from MessageFragment to EncryptFragment.

EncryptFragment has been added.

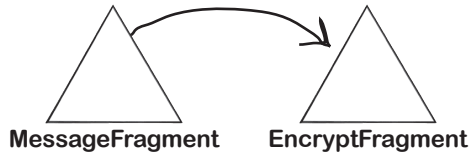
Let's use the new action to navigate to `EncryptFragment`.



**EncryptFragment**
Safe Args
Back button

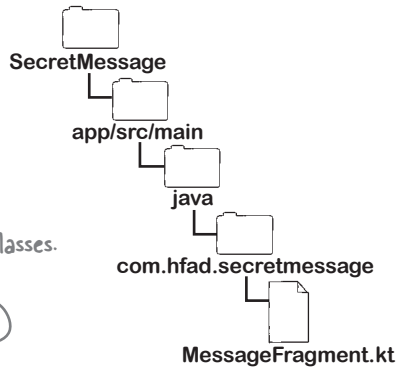
# MessageFragment needs to navigate to EncryptFragment

We need to make `MessageFragment` navigate to `EncryptFragment` when the user clicks its Next button. To enable this, we'll add an `OnClickListener` to the button, which will include the navigation code.




As you learned in the previous chapter, you navigate from one fragment to another by getting a navigation controller, and passing it a navigation action. The navigation controller uses this action to display the correct fragment.

You're already familiar with all the code needed to do this, so update *MessageFragment.kt* so that it includes the changes shown here (in bold):




```
package com.hfad.secretmessage

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Button
import androidx.navigation.findNavController

class MessageFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        ~~return~~ val view = inflater.inflate(R.layout.fragment_message, container, false)
        val nextButton = view.findViewById<Button>(R.id.next)

        nextButton.setOnClickListener {
            view.findNavController()
                    .navigate(R.id.action_messageFragment_to_encryptFragment)
        }
        return view
    }
}
```

Import these classes.

Get a reference to the root view.

Make the Next button navigate to EncryptFragment.

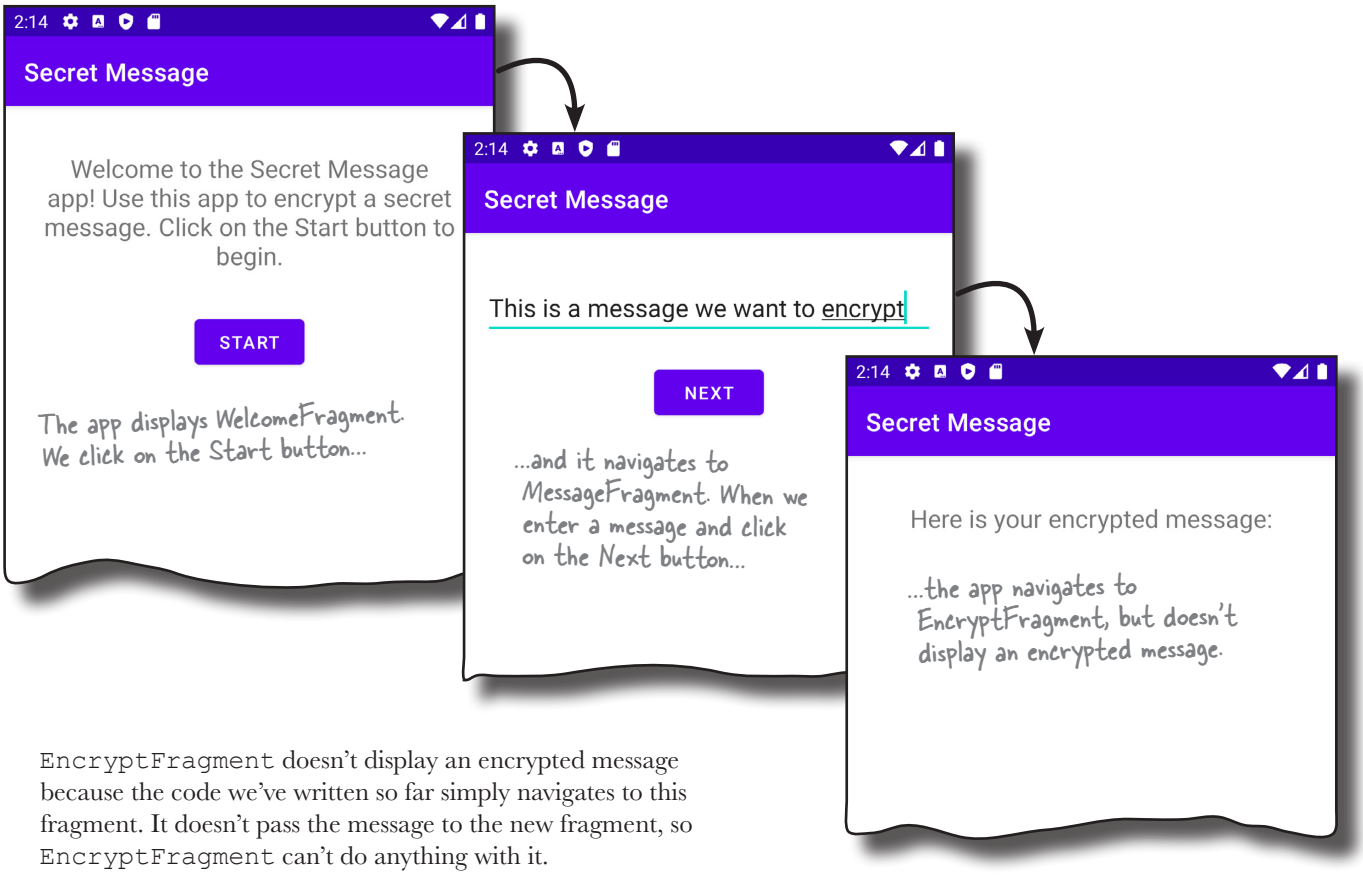
Let's take the app for a test drive and make sure this works.

# Test Drive

**EncryptFragment**
**Safe Args**
**Back button**

When we run the app, `MainActivity` is launched, and `WelcomeFragment` is displayed. When we click on its Start button, the app navigates to `MessageFragment` as before.

When we enter a message in `MessageFragment`'s edit text view and click on its Next button, the app navigates to `EncryptFragment`. It doesn't do anything with the message we entered.




`EncryptFragment` doesn't display an encrypted message because the code we've written so far simply navigates to this fragment. It doesn't pass the message to the new fragment, so `EncryptFragment` can't do anything with it.

To pass the message from `MessageFragment` to `EncryptFragment`, we're going to use Safe Args. As you learned earlier, this is an extra part of the Navigation component that lets you pass arguments to destinations in a type-safe way.

Let's find out how this is done.

EncryptFragment
**Safe Args**
Back button

# Add Safe Args to the build.gradle files

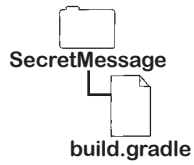
Before we can start using Safe Args, we need to update the project and app *build.gradle* files. Let's do this now.

## Add a classpath to the project build.gradle file

You first need to add a new classpath to the project's *build.gradle* file specifying that you want to use the Safe Args plug-in. The classpath includes a version number, which needs to match the one that you're using for the main Navigation component library.

In the Secret Message app, we're using a variable named `nav_version` to make sure that these version numbers match, so we'll reference this in the classpath.

Open the file *SecretMessage/build.gradle*, and add the classpath (in bold) to the `dependencies` section:

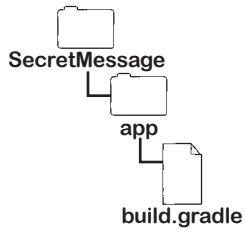



```
dependencies {
    ...
    classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version"
}
```

In the previous chapter, we included a nav_version variable to make sure all parts of the Navigation component use the same version number.

## Add the plug-in to the app build.gradle file

Next, you need to tell Gradle that you're using the Safe Args plug-in by adding a line to the app *build.gradle* file. Open the file *SecretMessage/app/build.gradle*, and add the line in bold to the `plugins` section:




```
plugins {
    ...
    id 'androidx.navigation.safeargs.kotlin'
}
```

This adds the Safe Args plug-in.

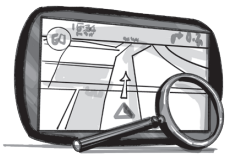
Once you've made these changes, click on the Sync Now option that appears at the top of the code editor. This syncs the changes you've made with the rest of your project.

Make sure that you sync your changes each time you update the build.gradle files.

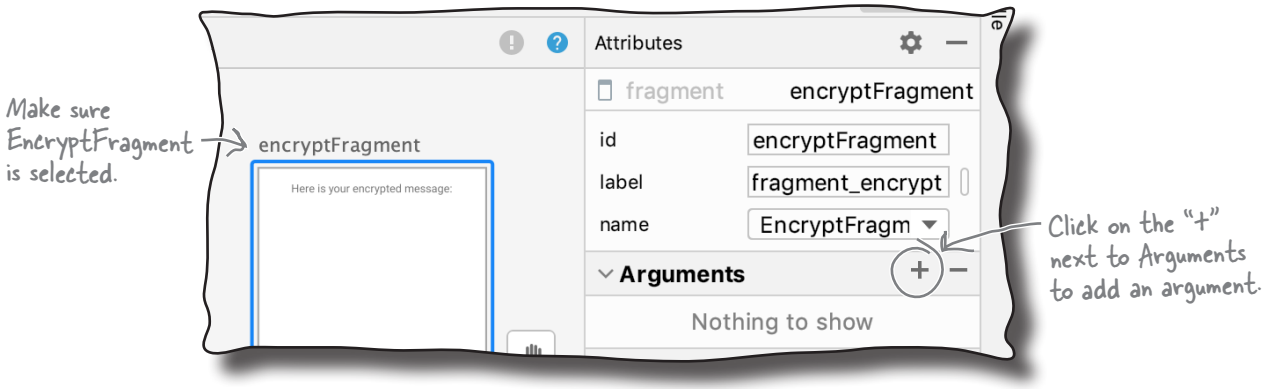
Now that the app includes Safe Args, we can use it to pass a message from `MessageFragment` to `EncryptFragment`. We'll do this next.

# EncryptFragment needs to accept a String argument

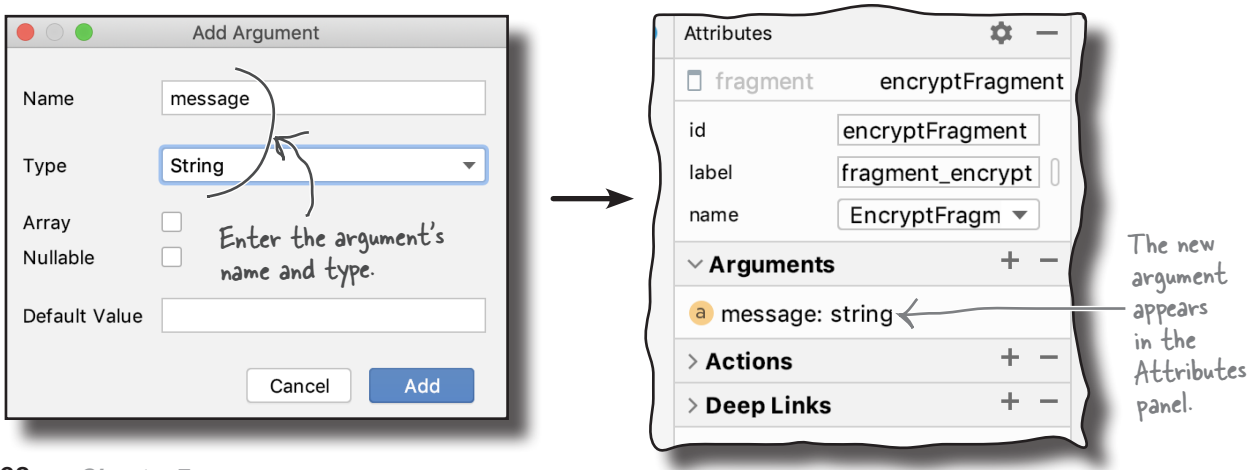
The first thing we need to do is specify that `EncryptFragment` can accept the user's message. We'll do this by adding a `String` argument to this fragment in the navigation graph; `MessageFragment` will then use this argument to pass the user's message (a `String`) to `EncryptFragment`.


Navigation Graph

To add the argument, open the file *nav_graph.xml* from the *app/src/main/res/nav_graph.xml*. Then select `EncryptFragment` in the navigation graph design editor, go to the Attributes panel, and click on the "+" button that appears next to the Arguments section:



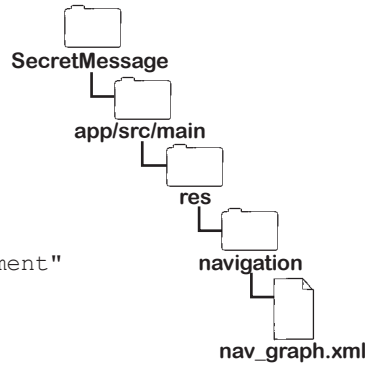

When you click on the "+" button, an Add Argument window appears, which you use to add details about the argument. Here, we want `EncryptFragment` to accept a `String` argument for the user's message, so name the argument "message", select "String" as its type, and click on the Add button. This creates the new argument, and adds it to the Arguments section of the Attributes panel:



EncryptFragment
**Safe Args**
Back button

# The updated nav_graph.xml code

When you add an argument to a navigation graph, a new `<argument>` element gets added to the underlying XML. Here's the updated code for *nav_graph.xml* (the new code is in bold):




```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/welcomeFragment">

    <fragment
        android:id="@+id/welcomeFragment"
        android:name="com.hfad.secretmessage.WelcomeFragment"
        android:label="fragment_welcome"
        tools:layout="@layout/fragment_welcome" >
        <action
            android:id="@+id/action_welcomeFragment_to_messageFragment"
            app:destination="@id/messageFragment" />
    </fragment>
    <fragment
        android:id="@+id/messageFragment"
        android:name="com.hfad.secretmessage.MessageFragment"
        android:label="fragment_message"
        tools:layout="@layout/fragment_message" >
        <action
            android:id="@+id/action_messageFragment_to_encryptFragment"
            app:destination="@id/encryptFragment" />
    </fragment>
    <fragment
        android:id="@+id/encryptFragment"
        android:name="com.hfad.secretmessage.EncryptFragment"
        android:label="fragment_encrypt"
        tools:layout="@layout/fragment_encrypt" >
        <argument
            android:name="message"
            app:argType="string" />
    </fragment>
</navigation>
```

Here's the new argument. It's a String named "message".

EncryptFragment
**Safe Args**
Back button

# MessageFragment needs to pass a message to EncryptFragment

Now that we've added a `String` argument to `EncryptFragment`, `MessageFragment` can use it to pass along the user's message when it navigates to this fragment.

As you already know, you navigate from one destination to another by passing a navigation action to the navigation controller. The navigation controller then uses this action to display the correct fragment. `MessageFragment`, for example, uses the following code to navigate to `EncryptFragment` when its button is clicked:

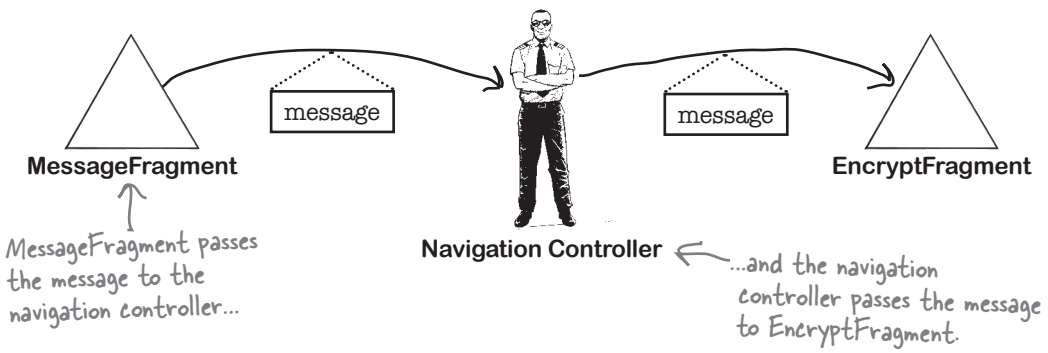
```
nextButton.setOnClickListener {
    view.findNavController()
            .navigate(R.id.action_messageFragment_to_encryptFragment)
}
```

This passes an action to the navigation controller, which it uses to display the correct fragment.

## You can pass arguments to navigation actions

If you want to pass an argument to a destination, you do so by passing its value to the navigation action.

When the navigation controller receives an action that includes an argument, it navigates to the appropriate fragment, and passes along the argument's value. In the Secret Message app, for example, we can get `MessageFragment` to pass the user's message to `EncryptMessage` (via the navigation controller) by including it in the navigation action. It will work like this:

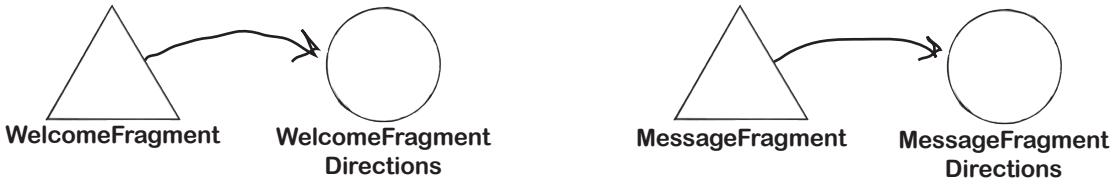



You can add arguments to navigation actions using a **`Directions`** class. Let's find out what this is.

EncryptFragment
**Safe Args**
Back button

# Safe Args generates Directions classes

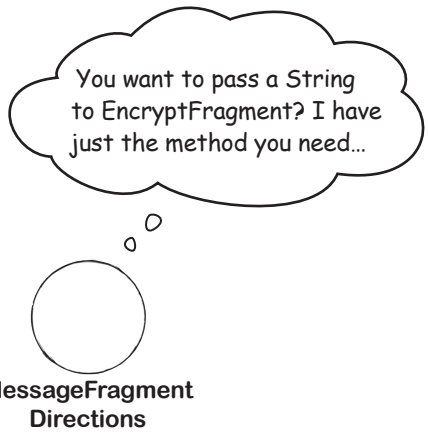
`Directions` classes are used to pass arguments to destinations. When you enable the Safe Args plug-in, Android Studio uses it to generate a `Directions` class **for each fragment you can navigate from**. In the Secret Message app, for example, you can navigate from `WelcomeFragment` and `MessageFragment` to other destinations, so the Safe Args plug-in generates a `Directions` class for each of these fragments; it generates a class named `WeclomeFragmentDirections` for `WelcomeFragment`, and one named `MessageFragmentDirections` gets generated for `MessageFragment`:




Each fragment uses its own `Directions` class to navigate with. `MessageFragment`, for example, needs to use the `MessageFragmentDirections` class if it wants to navigate elsewhere, and pass an argument.

## Use a Directions class to add arguments to actions

Each `Directions` class includes a generated method for each of the fragment's actions. `MessageFragment`, for example, has an action with the ID "action_messageFragment_to_encryptFragment", so the `MessageFragmentDirections` class includes a corresponding method named `actionMessageFragmentToEncryptFragment()`. And because `EncryptFragment` accepts a `String`, the generated method includes a `String` argument.




You use the generated methods to navigate to destinations. To navigate from `MessageFragment` to `EncryptFragment` and pass it a `String` message, for example, you'd add the following code to `MessageFragment`:

```
val action = MessageFragmentDirections
                .actionMessageFragmentToEncryptFragment(message)
view.findNavController().navigate(action)
```

Use MessageFragmentDirections to create the action.

Add the message String to the action.

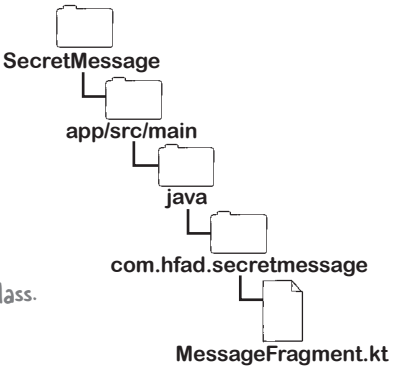
Let's add this code now.

# Update the MessageFragment.kt code

EncryptFragment
**Safe Args**
Back button

Here's the updated code for `MessageFragment`; make sure you add the changes (in bold) to *MessageFragment.kt*:

```
package com.hfad.secretmessage

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Button
import android.widget.EditText
import androidx.navigation.findNavController

class MessageFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        val view = inflater.inflate(R.layout.fragment_message, container, false)
        val nextButton = view.findViewById<Button>(R.id.next)
        val messageView = view.findViewById<EditText>(R.id.message)

        nextButton.setOnClickListener {
            val message = messageView.text.toString()
            val action = MessageFragmentDirections
                    .actionMessageFragmentToEncryptFragment(message)
            view.findNavController().navigate(action)
                    .navigate(R.id.action_messageFragment_to_encryptFragment)
        }
        return view
    }
}
```




Import this class.

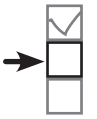
Get a reference to the message edit text.

Get the message String.

Navigate to EncryptMessage, passing it the message.

Delete this line.

That's everything `MessageFragment` needs to pass the user's message to `EncryptFragment`. Next, we need to make `EncryptFragment` pick it up and use it.

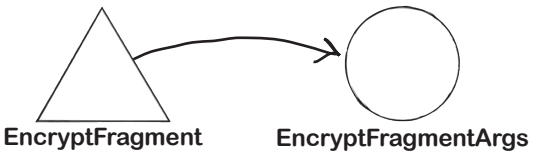


**EncryptFragment**
**Safe Args**
**Back button**

# EncryptFragment needs to get the argument's value

As you know, `MessageFragment` uses a `String` argument to pass the user's message to `EncryptFragment`. `EncryptFragment` needs to retrieve this value so that it can display an encrypted version of it.

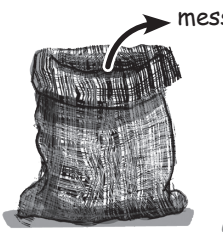
Fragments can retrieve arguments using an **`Args`** class. When you enable the Safe Args plug-in, Android Studio uses it to generate an `Args` class **for each fragment that accepts arguments**. In the Secret Message app, for example, `EncryptFragment` accepts a `String` argument, so the Safe Args plug-in generates an `Args` class for it named `EncryptFragmentArgs`:




**The Safe Args plug-in generates Directions and Args classes. Use the Directions class to pass arguments to a destination. Use the Args class to retrieve them.**

## Use an Args class to retrieve arguments

Each `Args` class includes a **`fromBundle()`** method that you use to retrieve any arguments that have been passed to the fragment. In the Secret Message app, for example, `EncryptFragment` accepts a `String` argument named `message`, so you can get the value of this argument using:




Get the value of the message argument.

```
val message = EncryptFragmentArgs.fromBundle(requireArguments()).message
```

This assigns a `String` object (the value of the argument) to `message`.

## We need to encrypt the message

Now that we know how to make `EncryptFragment` retrieve the user's message, we can encrypt it and display the result.

We'll encrypt the message using Kotlin's `reversed()` method, which simply reverses the order of a `String`'s letters. The code looks like this:

In the real world, you'd want to use a much more advanced encryption technique. This is just an example.

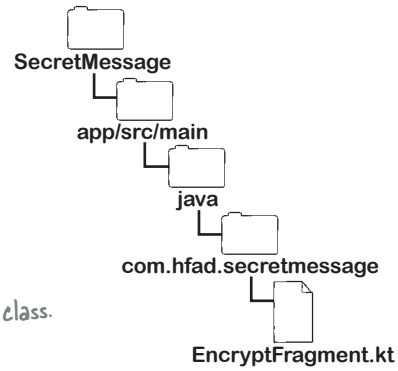
```
val encryptedView = view.findViewById<TextView>(R.id.encrypted_message)
encryptedView.text = message.reversed()
```

We'll show you the full `EncryptFragment` code on the next page.

# The full code for EncryptFragment.kt

EncryptFragment
Safe Args
Back button

Here's the complete code for *EncryptFragment.kt*; update its code so that it includes the changes below:




```
package com.hfad.secretmessage

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.TextView

class EncryptFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        return val view = inflater.inflate(R.layout.fragment_encrypt, container, false)
        val message = EncryptFragmentArgs.fromBundle(requireArguments()).message
        val encryptedView = view.findViewById<TextView>(R.id.encrypted_message)
        encryptedView.text = message.reversed()
        return view
    }
}
```

Import this class.

Get the root view.

Get the message.

Reverse the order of the letters, and display it.

That's everything that we need to display the encrypted message. We'll run through what happens when the app runs on the next page.

## there are no Dumb Questions

**Q: My project doesn't recognize the `Directions` and `Args` classes. Why is that?**

A: It sounds like the Safe Args plug-in hasn't generated them.

First, make sure you've added the Safe Args plug-in to the app's *build.gradle* file, and synced the changes. If it's missing, Android Studio won't be able to generate the `Directions` and `Args` classes.

Next, make sure you've added an argument to `EncryptFragment` in the navigation graph. Safe Args generates an `Args` class for every destination that accepts arguments, so if it's missing, the `Args` class won't get generated.

If you still get problems, try rebuilding your project by going to the Build menu and choosing the Rebuild Project option.

**Q: You said that Safe Args is added to the *build.gradle* files as a plug-in, while the rest of the Navigation component is included as a dependency. Why is this?**

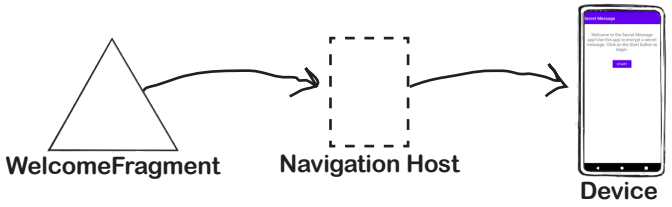
A: Safe Args is added as a plug-in because it needs to be able to generate the `Directions` and `Args` classes. Plug-ins can generate code, but dependencies can't.

**EncryptFragment**
**Safe Args**
**Back button**

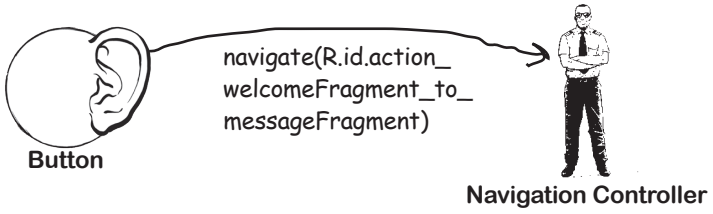
# What happens when the app runs

The following things happen when the app runs:

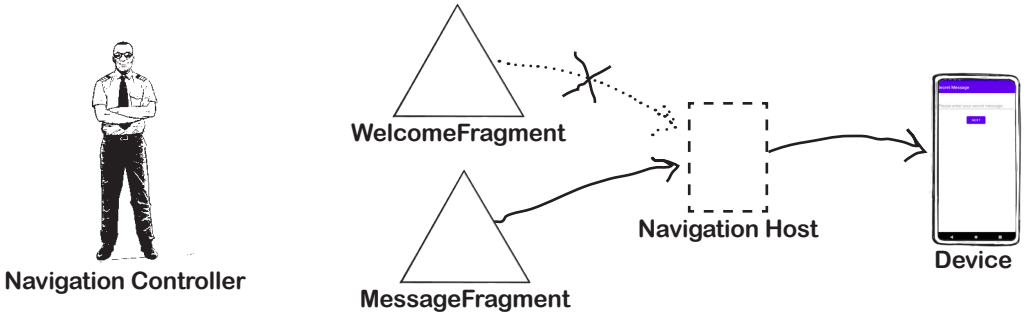
1. **The app is launched and MainActivity gets created.**
`WelcomeFragment` is added to the navigation host and displayed on the device screen.



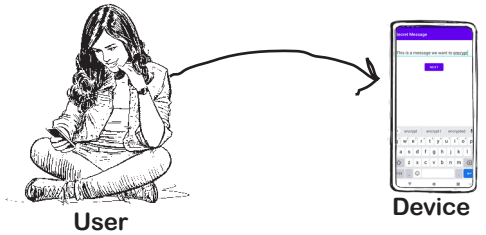

2. **When the user clicks on the Start button, its OnClickListener code passes an action to the navigation controller's navigate() method.**
The action describes a navigation path from `WelcomeFragment` to `MessageFragment`.




3. **The navigation controller puts MessageFragment in the navigation host so that it's displayed on the device screen.**



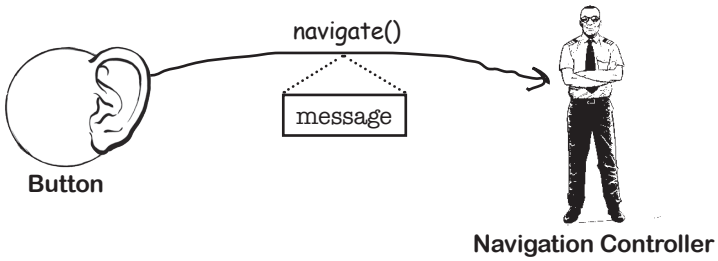

4. **The user enters a message and clicks the Next button.**



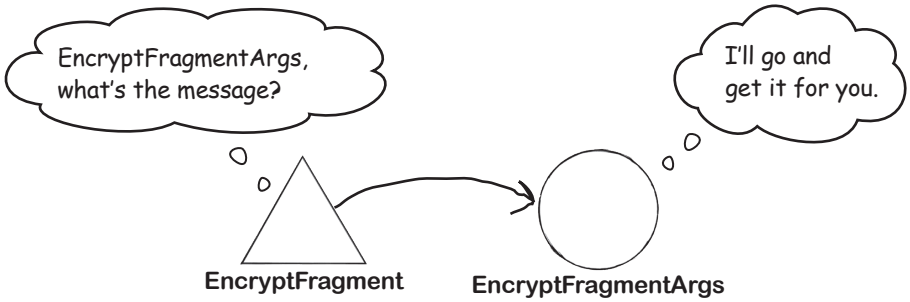
# The story continues

EncryptFragment
**Safe Args**
Back button

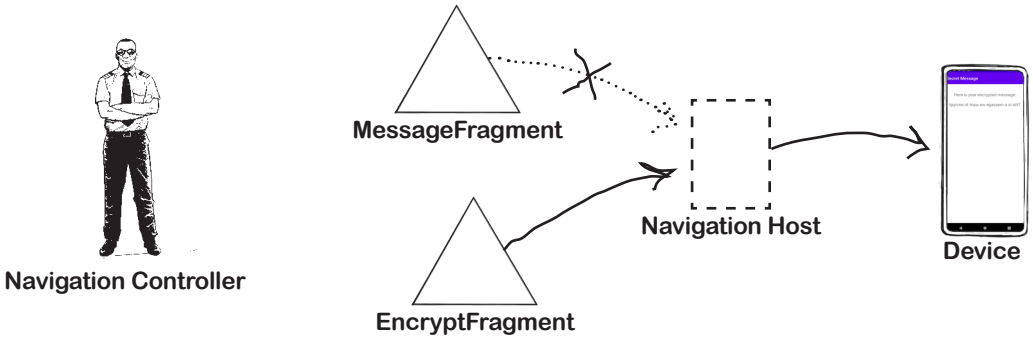
**5** **The Next button's OnClickListener uses the MessageFragmentDirections class to attach the message to an action.**
The action (including the message) is passed to the navigation controller.




**6** **The navigation controller passes the message to EncryptFragment.**
`EncryptFragment` uses the `EncryptFragmentArgs` class to retrieve its value.




**7** **The navigation controller replaces MessageFragment with EncryptFragment in MainActivity's navigation host.**
`EncryptFragment` is displayed on the device screen and shows the encrypted message.



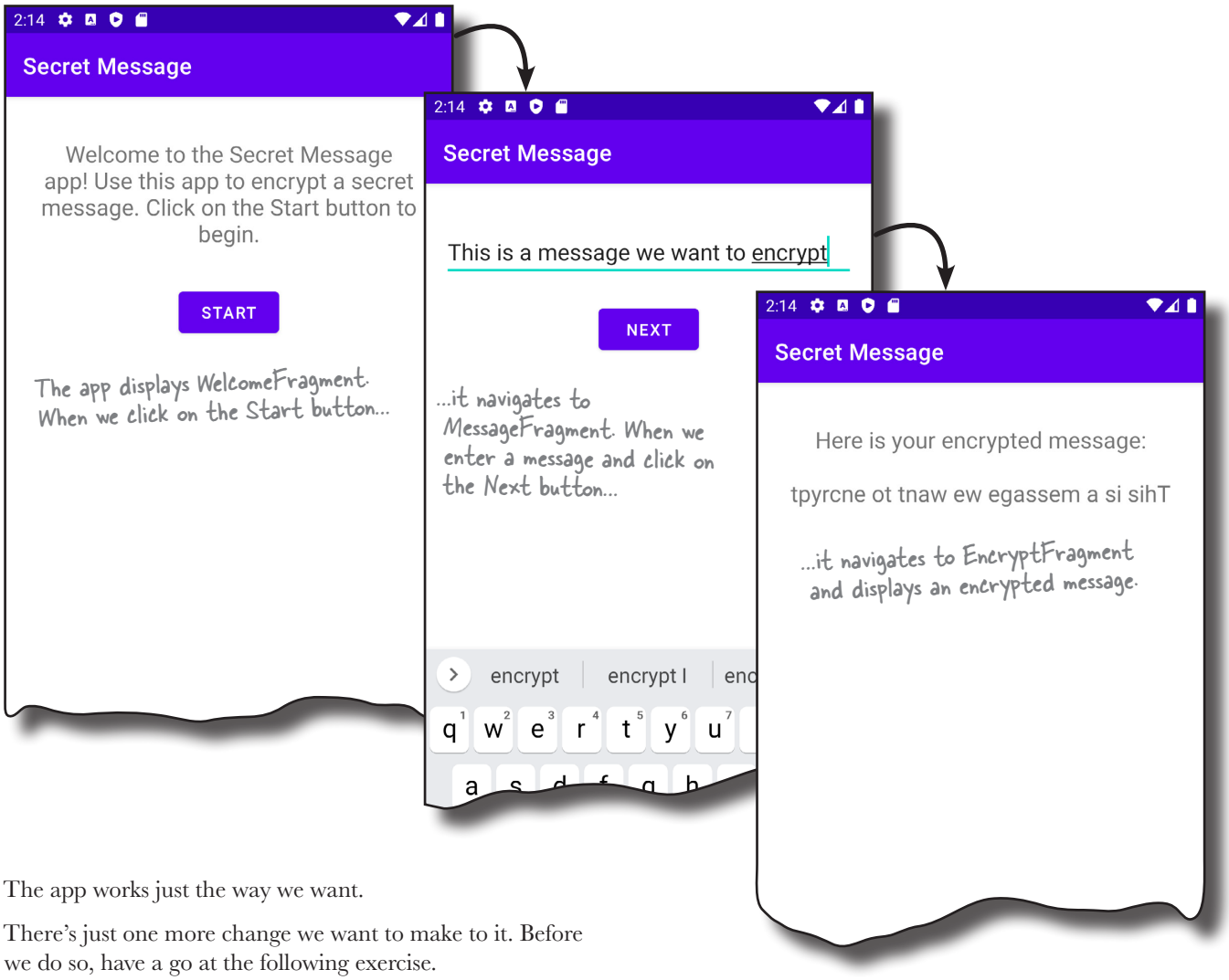

Let's take the app for a test drive.

**EncryptFragment**
**Safe Args**
**Back button**

# Test Drive

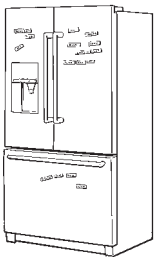
When we run the app, `WelcomeFragment` is displayed, and we can navigate to `MessageFragment` by clicking on the Start button as before.

When we enter a message and click on `MessageFragment`'s Next button, `EncryptFragment` is displayed. It shows the encrypted message.




The app works just the way we want.

There's just one more change we want to make to it. Before we do so, have a go at the following exercise.



# Navigation Magnets

The navigation graph code below defines the navigation path between two fragments: `ChooseTypeFragment` and `DrinksFragment`.

`ChooseTypeFragment` features a spinner with the ID `choose`, and a button with the ID `next`. When the user clicks on the button, `ChooseTypeFragment` needs to navigate to `DrinksFragment`, passing it the user's selected value from the spinner.

When `DrinksFragment` is displayed, it needs to display the selected value in a text view in its layout that has the ID `choice`.

Someone tried to write the code for `ChooseTypeFragment` and `DrinksFragment` using fridge magnets, but some of them fell off when we shut the fridge door too quickly. Can you piece the code back together again?

### Navigation graph:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/chooseTypeFragment">

    <fragment
        android:id="@+id/chooseTypeFragment"
        android:name="com.hfad.drinksapp.ChooseTypeFragment"
        android:label="fragment_choose_type"
        tools:layout="@layout/fragment_choose_type" >
        <action
            android:id="@+id/action_chooseTypeFragment_to_drinksFragment"
            app:destination="@id/drinksFragment" />
    </fragment>
    <fragment
        android:id="@+id/drinksFragment"
        android:name="com.hfad.drinksapp.DrinksFragment"
        android:label="fragment_drinks"
        tools:layout="@layout/fragment_drinks" >
        <argument
            android:name="drinkType"
            app:argType="string" />
    </fragment>
</navigation>
```

The fragment code is on the next page.

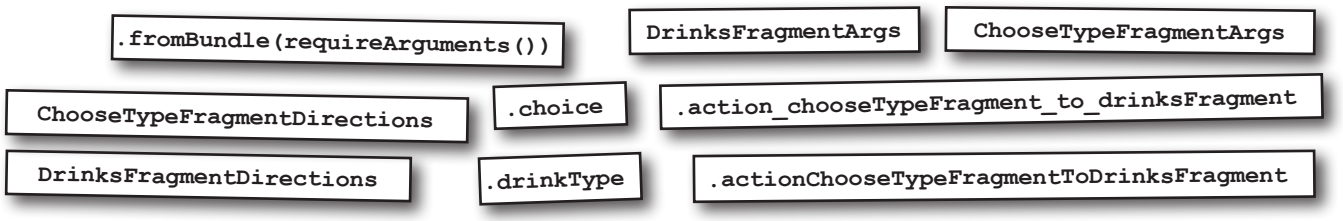
### ChooseTypeFragment:

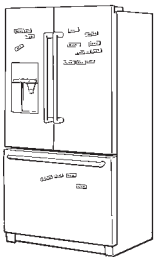
```
class ChooseTypeFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        val view = inflater.inflate(R.layout.fragment_choose_type, container, false)
        val choice = view.findViewById<Spinner>(R.id.choose).selectedItem.toString()
        val nextButton = view.findViewById<Button>(R.id.next)
        nextButton.setOnClickListener {

            val action = ..........................................................................

                         .......................................................................... (choice)
            view.findNavController().navigate(action)
        }
        return view
    }
}
```

### DrinksFragment:

```
class DrinksFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        val view = inflater.inflate(R.layout.fragment_drinks, container, false)

        val choice = ..........................................................................
        val choiceView = view.findViewById<TextView>(R.id.choice)
        choiceView.text = choice
        return view
    }
}
```

You won't need to use all the magnets.





# Navigation Magnets Solution

The navigation graph code below defines the navigation path between two fragments: `ChooseTypeFragment` and `DrinksFragment`.

`ChooseTypeFragment` features a spinner with the ID `choose`, and a button with the ID `next`. When the user clicks on the button, `ChooseTypeFragment` needs to navigate to `DrinksFragment`, passing it the user's selected value from the spinner.

When `DrinksFragment` is displayed, it needs to display the selected value in a text view in its layout that has the ID `choice`.

Someone tried to write the code for `ChooseTypeFragment` and `DrinksFragment` using fridge magnets, but some of them fell off when we shut the fridge door too quickly. Can you piece the code back together again?

### Navigation graph:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/chooseTypeFragment">

    <fragment
        android:id="@+id/chooseTypeFragment"
        android:name="com.hfad.drinksapp.ChooseTypeFragment"
        android:label="fragment_choose_type"
        tools:layout="@layout/fragment_choose_type" >
        <action
            android:id="@+id/action_chooseTypeFragment_to_drinksFragment"
            app:destination="@id/drinksFragment" />
    </fragment>
    <fragment
        android:id="@+id/drinksFragment"
        android:name="com.hfad.drinksapp.DrinksFragment"
        android:label="fragment_drinks"
        tools:layout="@layout/fragment_drinks" >
        <argument
            android:name="drinkType"
            app:argType="string" />
    </fragment>
</navigation>
```

The fragment code is on the next page.

### ChooseTypeFragment:

```
class ChooseTypeFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        val view = inflater.inflate(R.layout.fragment_choose_type, container, false)
        val choice = view.findViewById<Spinner>(R.id.choose).selectedItem.toString()
        val nextButton = view.findViewById<Button>(R.id.next)
        nextButton.setOnClickListener {
            val action = ChooseTypeFragmentDirections
                         .actionChooseTypeFragmentToDrinksFragment(choice)
            view.findNavController().navigate(action)
        }
        return view
    }
}
```

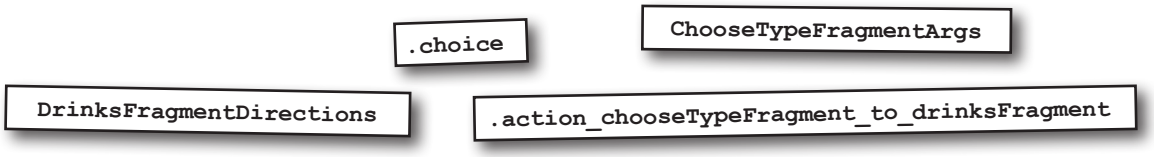
Use this class to navigate from ChooseTypeFragment.

### DrinksFragment:

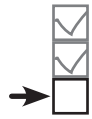
```
class DrinksFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        val view = inflater.inflate(R.layout.fragment_drinks, container, false)
        val choice = DrinksFragmentArgs.fromBundle(requireArguments()).drinkType
        val choiceView = view.findViewById<TextView>(R.id.choice)
        choiceView.text = choice
        return view
    }
}
```

This is the name of DrinksFragment's argument in the navigation graph.

You didn't need to use these magnets.



# What if the user wants to go back?

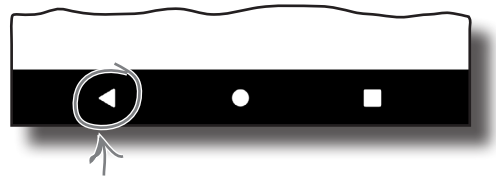


EncryptFragment
Safe Args
**Back button**

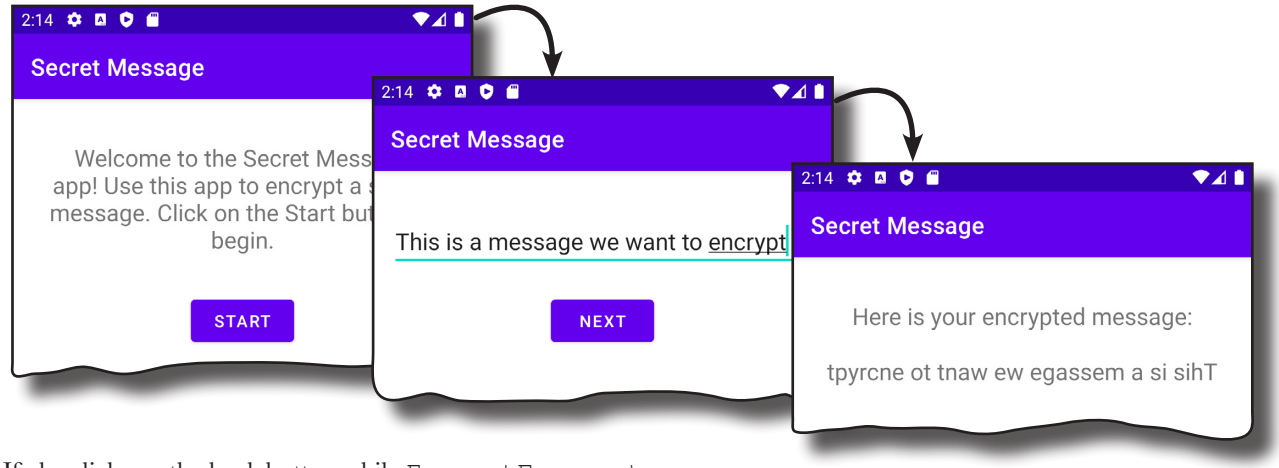
There's just one more thing we need to consider in the Secret Message app: what happens when the user tries to go back through the app's screens.

As we're sure you know, you go back in Androidville using the device back button or with a back gesture, and this takes you back through any screens that have been displayed.

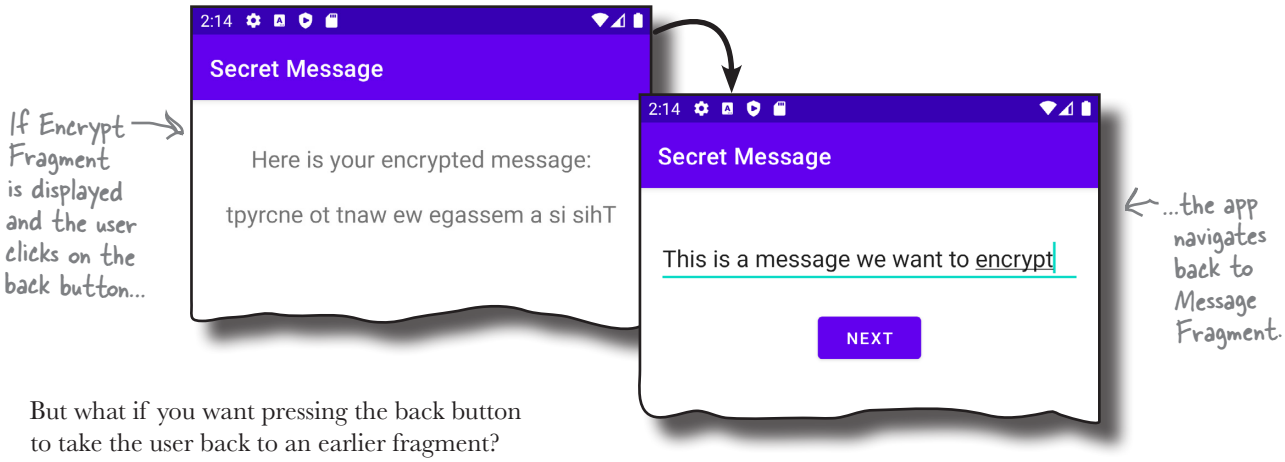
As an example, suppose the user launches the Secret Message app and navigates from `WelcomeFragment` to `MessageFragment` and then to `EncryptFragment` like this:



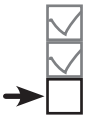
The back button appears at the bottom of the device screen. Some users or devices might use gestures instead, which serve the same purpose.




If she clicks on the back button while `EncryptFragment` is displayed, the app goes back to the previous fragment—`MessageFragment`—like so:



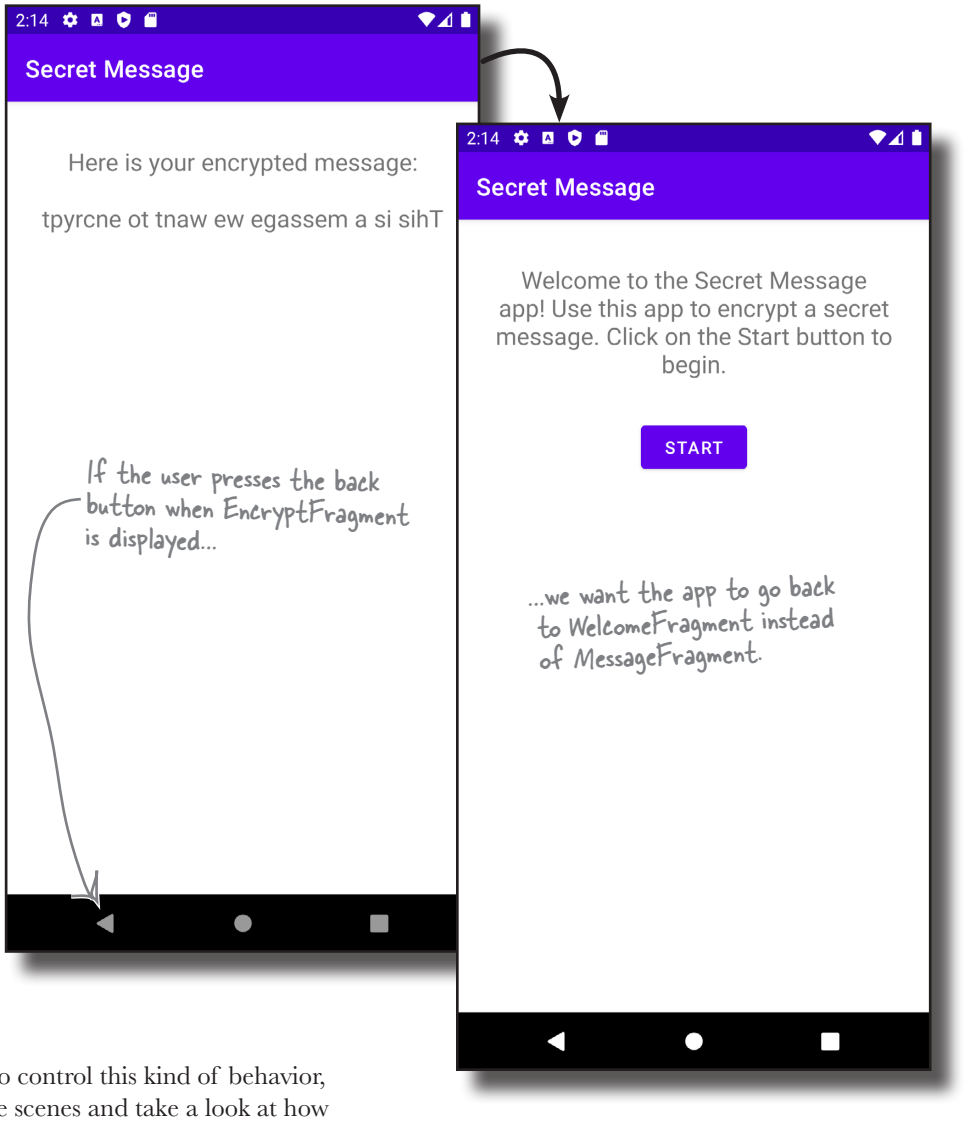

But what if you want pressing the back button to take the user back to an earlier fragment?



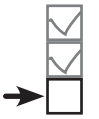
**EncryptFragment**
**Safe Args**
**Back button**

# We could change the back behavior

Instead of going back to `MessageFragment` when the user clicks on the back button, it might be better to go straight back to `WelcomeFragment`. This would let the user start again from the beginning of the app like this:




To find out how to control this kind of behavior, let's go behind the scenes and take a look at how Android's back stack works.



EncryptFragment
Safe Args
**Back button**

# Welcome to the back stack

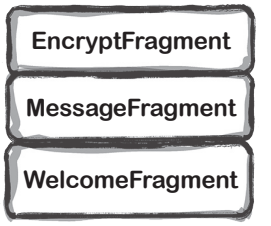
When you go from destination to destination in your app, Android keeps track of each place you've visited by adding it to the **back stack**. The back stack is a log of all the places you've visited in the app. Each time you navigate to a destination, Android adds it to the top of the back stack, and when you press the back button, it pops the most recent destination off the stack, and displays the one underneath.

## A back stack scenario

1. **When you start the Secret Message app, WelcomeFragment is displayed.**
   Android adds `WelcomeFragment` to the back stack.

2. **You navigate to MessageFragment.**
   This destination is added to the top of the back stack, above `WelcomeFragment`.

3. **You then navigate to EncryptFragment.**
   `EncryptFragment` is added to the top of the back stack.




4. **You click on the Back button, and EncryptFragment pops off the back stack.**
   `MessageFragment` is displayed, as this is now at the top of the back stack.

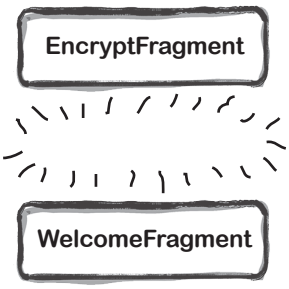
5. **You click on the Back button again.**
   `MessageFragment` pops off the top of the back stack, and `WelcomeFragment` is displayed.

EncryptFragment
Safe Args
**Back button**

# Use the navigation graph to pop fragments off the back stack

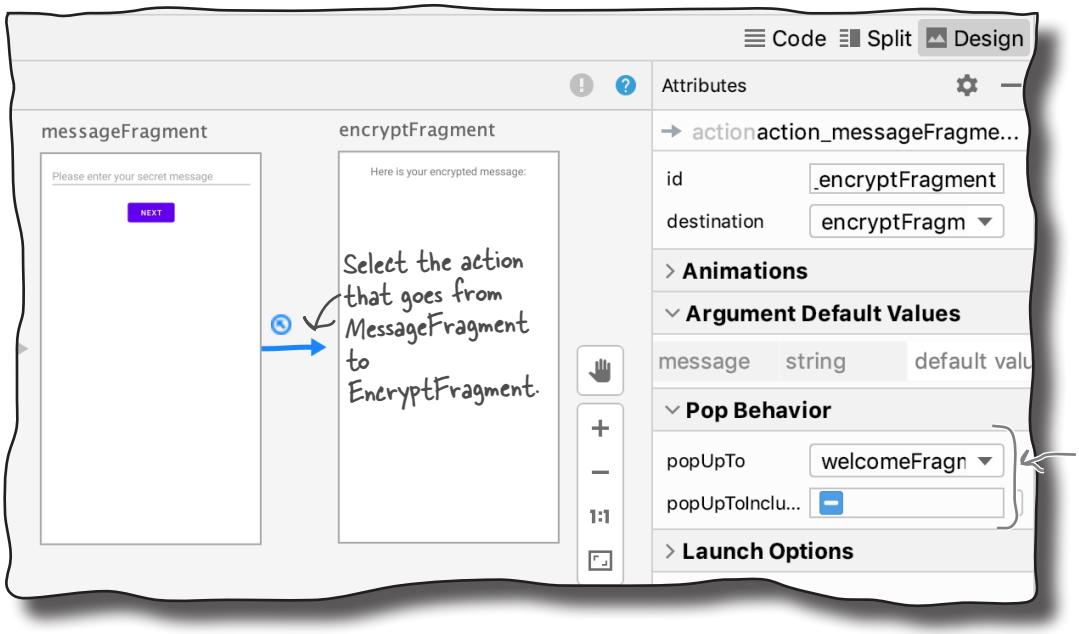
You've just seen how the back stack and back button operate by default, but if you want, you can pop destinations off the back stack as the user navigates through the app. You do this by specifying **pop behavior** in the navigation graph.

To see how this works, let's update the navigation graph in the Secret Message app so that when the app navigates from `MessageFragment` to `EncryptFragment`, `MessageFragment` is popped off the back stack. This means that when the user clicks on the back button while `EncryptFragment` is displayed, `WelcomeFragment` is displayed instead of `MessageFragment`.




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

Open the navigation graph *nav_graph.xml* (if it's not already open), and switch to the design editor. Select the action that connects `MessageFragment` to `EncryptFragment`. Then, in the Pop Behavior section of the Attributes panel, change the value of the `popUpTo` property to "welcomeFragment". This tells Android to pop fragments off the back stack until it reaches `WelcomeFragment`:




Update the pop behavior so that when the navigation controller uses this action, it pops all fragments up to WelcomeFragment off the back stack. If we were to set popUpInclusive to true, WelcomeFragment would be popped off as well.

Before we take the app for a test drive, let's see what the underlying XML looks like.

EncryptFragment
Safe Args
**Back button**

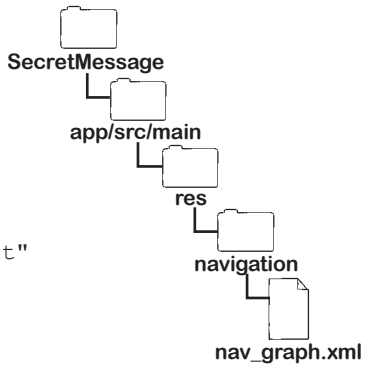
# The updated nav_graph.xml code

When you add pop behavior to an action, a <popUpTo> element is added to the underlying XML. This specifies how far back fragments should be popped off the back stack to.

Here's the updated code for *nav_graph.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/welcomeFragment">

    <fragment
        android:id="@+id/welcomeFragment"
        android:name="com.hfad.secretmessage.WelcomeFragment"
        android:label="fragment_welcome"
        tools:layout="@layout/fragment_welcome" >
        <action
            android:id="@+id/action_welcomeFragment_to_messageFragment"
            app:destination="@id/messageFragment" />
    </fragment>
    <fragment
        android:id="@+id/messageFragment"
        android:name="com.hfad.secretmessage.MessageFragment"
        android:label="fragment_message"
        tools:layout="@layout/fragment_message" >
        <action
            android:id="@+id/action_messageFragment_to_encryptFragment"
            app:destination="@id/encryptFragment"
            app:popUpTo="@id/welcomeFragment" />
    </fragment>
    <fragment
        android:id="@+id/encryptFragment"
        android:name="com.hfad.secretmessage.EncryptFragment"
        android:label="fragment_encrypt"
        tools:layout="@layout/fragment_encrypt" >
        <argument
            android:name="message"
            app:argType="string" />
    </fragment>
</navigation>
```

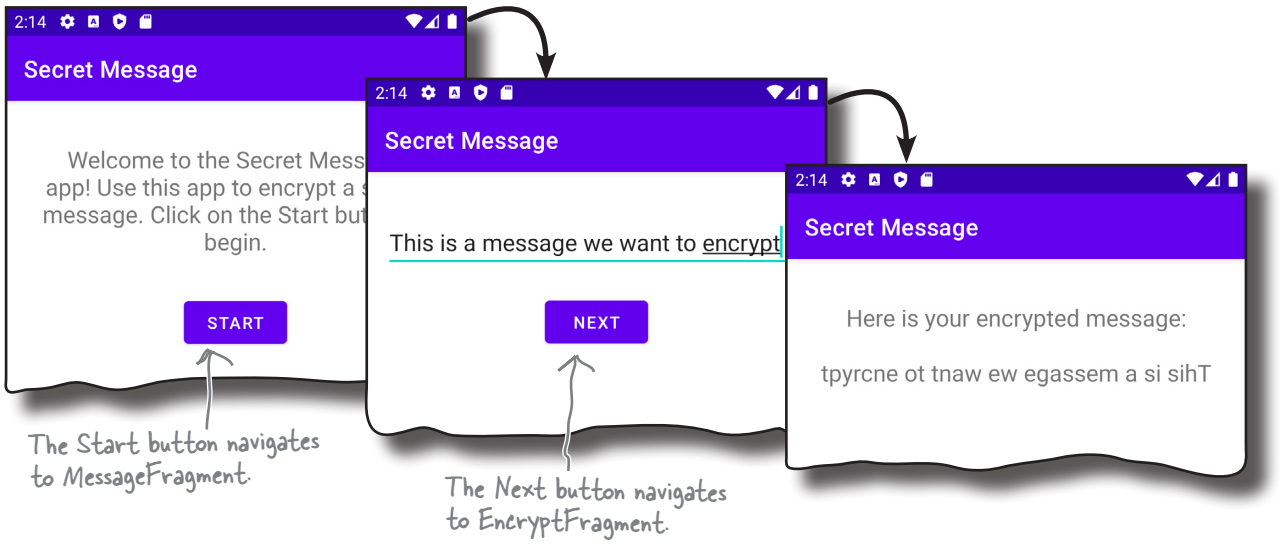
This new attribute pops fragments off the back stack to WelcomeFragment.



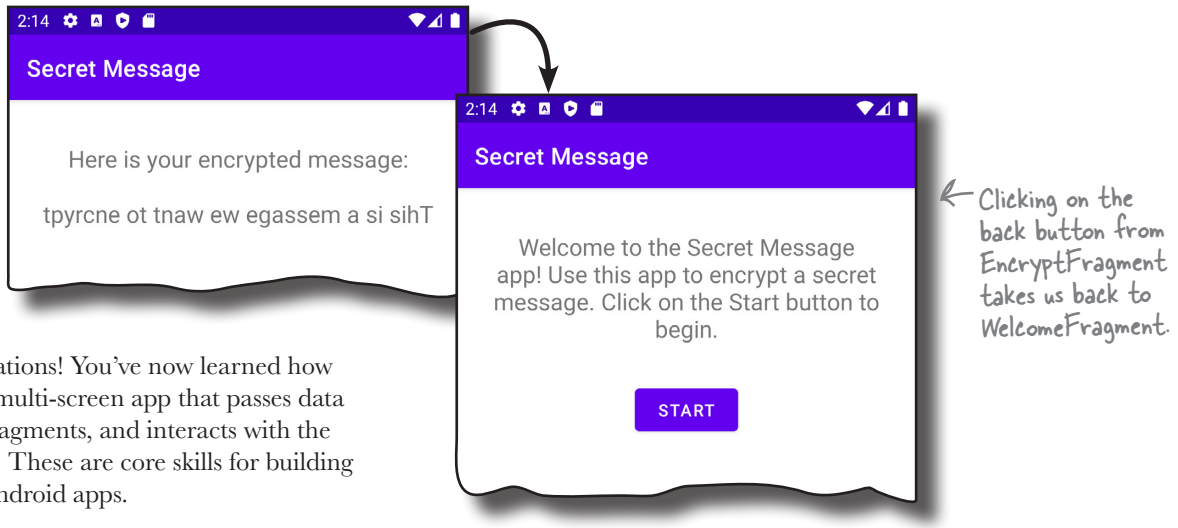
# Test Drive

EncryptFragment
Safe Args
**Back button**

When we run the app, `WelcomeFragment` is displayed. We can navigate to `MessageFragment` and `EncryptFragment` as before:




When we click on the back button, the app goes back to `WelcomeFragment`, and bypasses `MessageFragment`:




Congratulations! You've now learned how to build a multi-screen app that passes data between fragments, and interacts with the back stack. These are core skills for building modern Android apps.

You'll discover more ways in which you can use the Navigation component in the next chapter.

# BE the Safe Args Plug-in

The code below shows the navigation graph for a Starbuzz app. Your job is to play like you're the Safe Args plug-in, and say which Directions and Args classes will get generated from this code.

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/selectCoffeeFragment">

    <fragment
        android:id="@+id/selectCoffeeFragment"
        android:name="com.hfad.starbuzz.SelectCoffeeFragment"
        android:label="fragment_select_coffee"
        tools:layout="@layout/fragment_select_coffee" >
        <action
            android:id="@+id/action_selectCoffeeFragment_to_coffeeFragment"
            app:destination="@id/coffeeFragment" />
    </fragment>

    <fragment
        android:id="@+id/coffeeFragment"
        android:name="com.hfad.starbuzz.CoffeeFragment"
        android:label="fragment_coffee"
        tools:layout="@layout/fragment_coffee" >
        <argument
            android:name="coffee"
            app:argType="string" />
    </fragment>
</navigation>
```

→ Answers on page 290.

## Sharpen your pencil

The navigation graph below lets the user navigate from `TitleFragment` to `ChooseTypeFragment`, and from `ChooseTypeFragment` to `DrinksFragment`.

How would you update the code so that when `DrinksFragment` is displayed and the user presses the back button, the app goes straight back to `TitleFragment`?

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/titleFragment">

    <fragment
        android:id="@+id/titleFragment"
        android:name="com.hfad.drinksapp.TitleFragment"
        android:label="fragment_title"
        tools:layout="@layout/fragment_title" >
        <action
            android:id="@+id/action_titleFragment_to_chooseTypeFragment"
            app:destination="@id/chooseTypeFragment" />
    </fragment>

    <fragment
        android:id="@+id/chooseTypeFragment"
        android:name="com.hfad.drinksapp.ChooseTypeFragment"
        android:label="fragment_choose_type"
        tools:layout="@layout/fragment_choose_type" >
        <action
            android:id="@+id/action_chooseTypeFragment_to_drinksFragment"
            app:destination="@id/drinksFragment" />
    </fragment>

    <fragment
        android:id="@+id/drinksFragment"
        android:name="com.hfad.drinksapp.DrinksFragment"
        android:label="fragment_drinks"
        tools:layout="@layout/fragment_drinks" />
</navigation>
```

Answers on page 291.

# BE the Safe Args Plug-in Solution

The code below shows the navigation graph for a Starbuzz app. Your job is to play like you're the Safe Args plug-in, and say which Directions and Args classes will get generated from this code.

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/selectCoffeeFragment">

    <fragment
        android:id="@+id/selectCoffeeFragment"
        android:name="com.hfad.starbuzz.SelectCoffeeFragment"
        android:label="fragment_select_coffee"
        tools:layout="@layout/fragment_select_coffee" >
        <action
            android:id="@+id/action_selectCoffeeFragment_to_coffeeFragment"
            app:destination="@id/coffeeFragment" />
    </fragment>

    <fragment
        android:id="@+id/coffeeFragment"
        android:name="com.hfad.starbuzz.CoffeeFragment"
        android:label="fragment_coffee"
        tools:layout="@layout/fragment_coffee" >
        <argument
            android:name="coffee"
            app:argType="string" />
    </fragment>
</navigation>
```

You can navigate from SelectCoffeeFragment, so Safe Args generates a SelectCoffeeFragmentDirections class.

CoffeeFragment accepts an argument, so Safe Args generates a CoffeeFragmentArgs class.

## Sharpen your pencil Solution

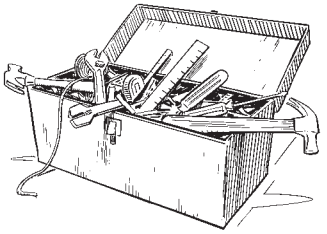
The navigation graph below lets the user navigate from `TitleFragment` to `ChooseTypeFragment`, and from `ChooseTypeFragment` to `DrinksFragment`.

How would you update the code so that when `DrinksFragment` is displayed and the user presses the back button, the app goes straight back to `TitleFragment`?

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/titleFragment">

    <fragment
        android:id="@+id/titleFragment"
        android:name="com.hfad.drinksapp.TitleFragment"
        android:label="fragment_title"
        tools:layout="@layout/fragment_title" >
        <action
            android:id="@+id/action_titleFragment_to_chooseTypeFragment"
            app:destination="@id/chooseTypeFragment" />
    </fragment>

    <fragment
        android:id="@+id/chooseTypeFragment"
        android:name="com.hfad.drinksapp.ChooseTypeFragment"
        android:label="fragment_choose_type"
        tools:layout="@layout/fragment_choose_type" >
        <action
            android:id="@+id/action_chooseTypeFragment_to_drinksFragment"
            app:destination="@id/drinksFragment"
            app:popUpTo="@id/titleFragment" />
    </fragment>

    <fragment
        android:id="@+id/drinksFragment"
        android:name="com.hfad.drinksapp.DrinksFragment"
        android:label="fragment_drinks"
        tools:layout="@layout/fragment_drinks" />
</navigation>
```

You need to add this line to pop ChooseTypeFragment off the back stack when the user navigates to DrinksFragment.

# Your Android Toolbox



**You've got Chapter 7 under your belt and now you've added the Safe Args plug-in and back stack manipulation to your toolbox.**

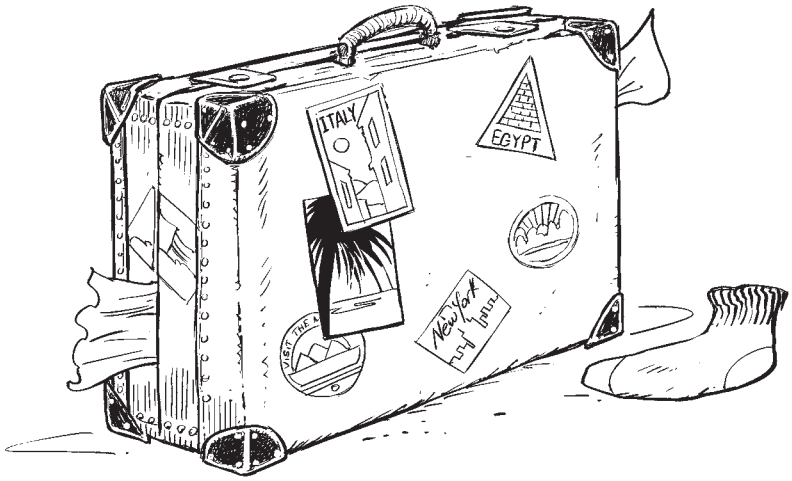
You can download the full code for the chapter from tinyurl.com/hfad3.

## Bullet Points

- Safe Args is an extra part of the Navigation component that's used to pass arguments in a type-safe way. It generates `Directions` and `Args` classes.
- `Directions` classes are used to pass arguments to destinations.
- Safe Args generates a `Directions` class for every destination you can navigate from.
- `Args` classes are used to retrieve arguments that have been passed to destinations.
- Safe Args generates an `Args` class for every destination that accepts arguments.
- Use pop behavior in the navigation graph to pop destinations off the back stack.

# 8 navigation ui

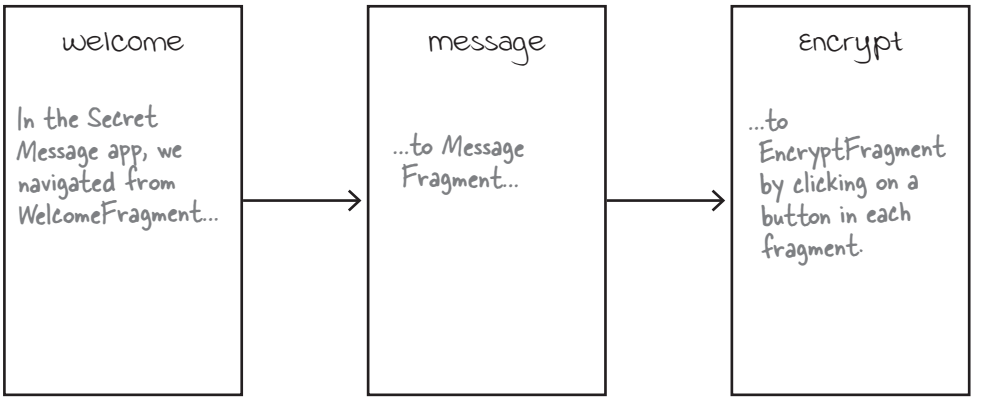
# Going Places



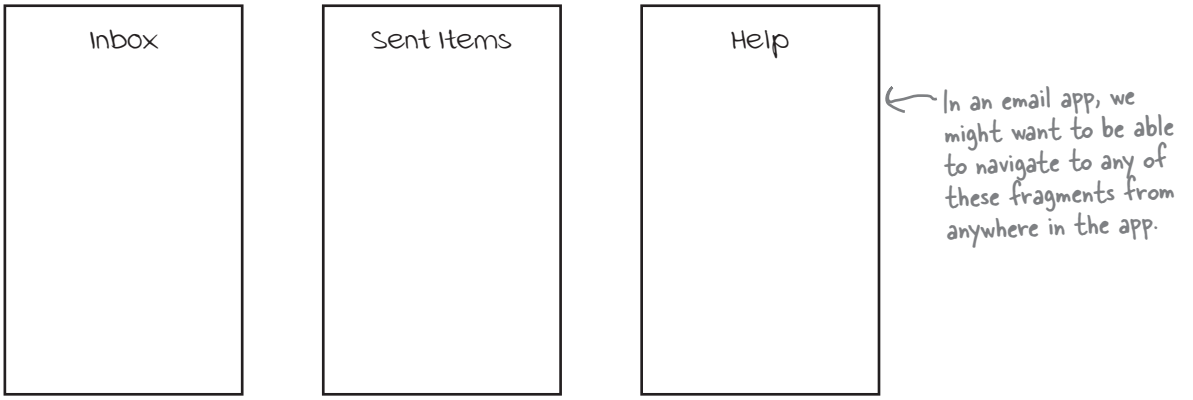

**Most apps need to be able to navigate between destinations.** And with Android's Navigation component, building this UI became much simpler. Here, you'll learn how to use some of Android's navigation UI components so that **your users can navigate your app more easily**. You'll see how to use **themes**, 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.

# Different apps, different structures

In the previous two chapters, you learned how to use Android's Navigation component to navigate from fragment to fragment by clicking on a button. This approach worked well in the Secret Message app because we needed to navigate from one destination to another in a linear way:




But not all apps follow this kind of structure. Many apps have screens you need to be able to navigate to from wherever you are in the app. An email app, for example, may have inbox, sent items, and help screens that must be available throughout the app:



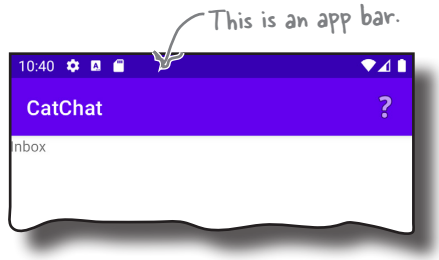

When you have an app with a looser navigation structure like this, how do you make sure you can always navigate to each screen?

# Android includes navigation UI components

When you have screens you want to be able to access from anywhere in your app, you might want to use one of Android's navigation UI components. They include:

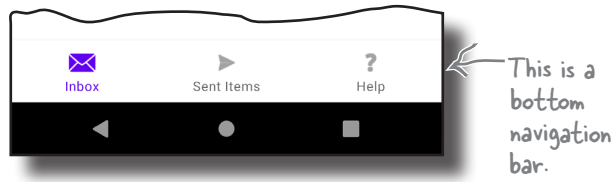
## An app bar

This is a bar that appears at the top of the screen, and Android Studio usually includes one for you by default. You can add items to the app bar which, when clicked, navigate to destinations.



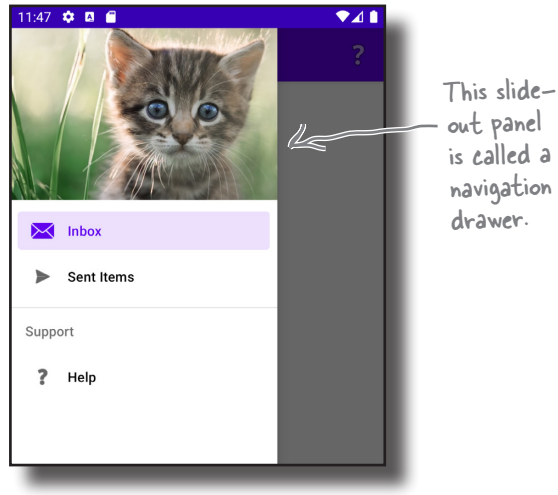

## The bottom navigation bar

This appears at the bottom of the screen. It features a small number of items you can use for navigation.

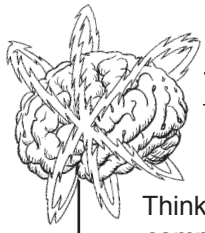



## A navigation drawer

This is a drawer that slides out from the edge of the screen. Many apps include one of these as it can be very flexible.




In this chapter, we're going to discover how to implement each of these three types of navigation UI by building a prototype email app named CatChat.

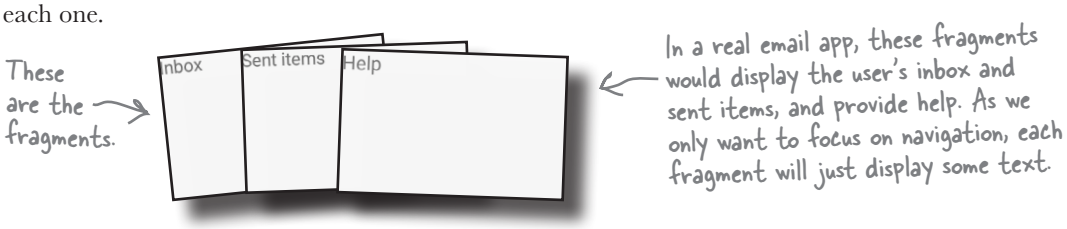


## Brain Power

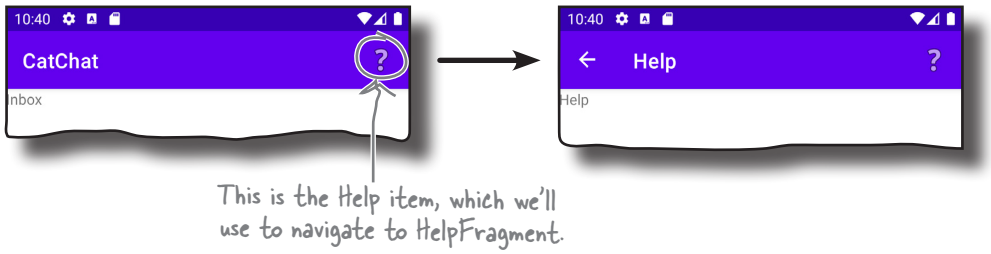
Think about where you may have seen each of these navigation UI components in existing Android apps. What situations do you think each one is best for? Why?

# How the CatChat app will work

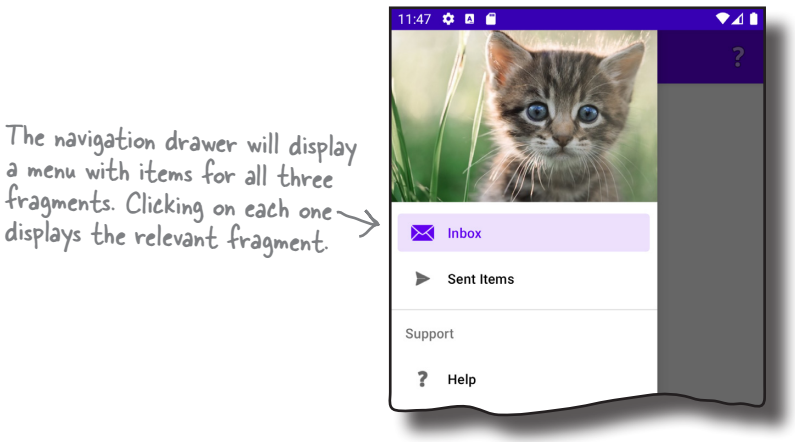
The CatChat app will feature a single activity named `MainActivity`, and three fragments: `InboxFragment`, `SentItemsFragment`, and `HelpFragment`. Each fragment will be displayed in `MainActivity` as the user navigates to each one.




The app will include an app bar that includes a Help menu item. When the user clicks on this, the app will navigate to `HelpFragment` by displaying it in `MainActivity`.




The app will also feature a navigation drawer that will be used to navigate between all three fragments. The drawer will include an item for each fragment, and clicking on each one will display the correct fragment:




Let's go through the steps we'll take to create the CatChat app.

# Here's what we're going to do

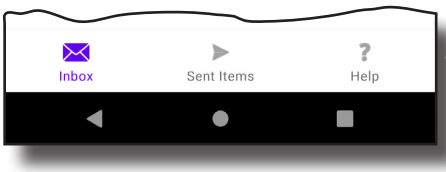
There are four main steps we'll go through to create the CatChat app:

1. **Replace the default app bar with a toolbar.**
A toolbar looks like an app's default app bar, but it provides you with more flexibility, and includes the latest app bar features. Along the way, you'll learn how to apply themes and styles.

2. **Add a Help menu item to the toolbar.**
When clicked, the Help menu item will navigate to `HelpFragment`.

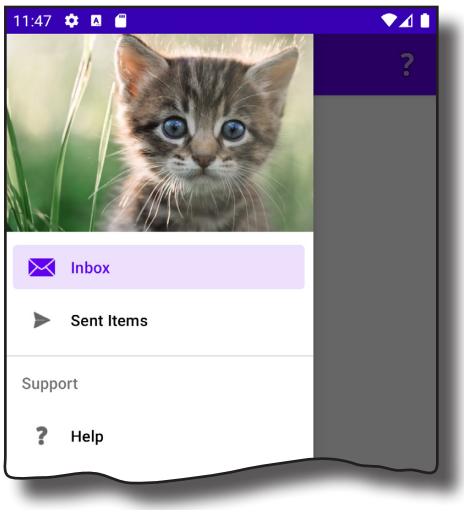
We'll add this item to the toolbar.

3. **Implement the bottom bar navigation.**
We'll add a bottom bar to the app so that we can navigate between each of the app's fragments.




The final version of the app won't include a bottom navigation bar, but we want to show you how to create one.

4. **Create the navigation drawer.**
Finally, we'll replace the bottom navigation bar with a navigation drawer. The drawer will include a header image, and items for each of the app's fragments, grouped into sections.




We'll start by creating a new project for the CatChat app.

**Toolbar**
Toolbar navigation
Bottom bar
Navigation drawer

# Create a new project

We're going to use a new project for the CatChat app, so create a new Android Studio project using the same steps you used in the previous chapters. Choose the Empty Activity option, enter a name of "CatChat" and a package name of "com.hfad.catchat", and accept the default save location. Make sure the language is set to Kotlin and the minimum SDK is API 21 so that it will run on most Android devices.

## A default app bar gets added...

When you create a new project with an empty activity, Android Studio usually includes an app bar that displays the app's name. You can see this app bar at the top of the screen when you run the app:

This is the default app bar that Android Studio includes for you. Don't worry if your version of Android Studio hasn't added an app bar because you'll find out how to include one over the next few pages.

Having an app bar is useful for a number of reasons:

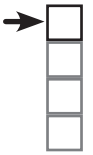
- **It makes key actions more prominent in a way that's predictable, such as sharing content or performing searches.**
- **It can help the user know where they are in the app by displaying the app name, or a label for the current screen.**
- **You can use it to navigate to different destinations.**

But how does the app bar get added?

## ...by applying a theme

The default app bar gets added to the app by applying a **theme**. A theme gives your app a consistent look and feel across multiple screens. It controls the app's general appearance, and whether it includes an app bar.

Android comes with a number of themes you can use in your apps, and by default, Android Studio applies one that includes an app bar.



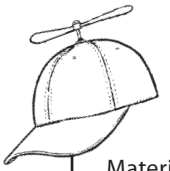


# The CatChat app will use a Material theme

We're going to apply a theme to the CatChat app that uses **Material Design**.

Material is a design system developed by Google that helps you build high-quality apps and websites with a consistent look and feel. The idea behind it is that a user can switch from a Google app like the Play Store to an app designed by a third-party developer and instantly feel comfortable and know what to do.

Material was originally inspired by print-based design principles and movement to reflect how real-world objects (such as index cards and pieces of paper) look and behave. Its latest evolution is known as **Material You**, which gives users a more fluid experience with a personalized color palette.



## Serious Coding

Material Design includes a whole host of components, themes, tools, and guidelines. You'll put some of these into practice over the next couple chapters, and you can learn more here:

*https://material.io*

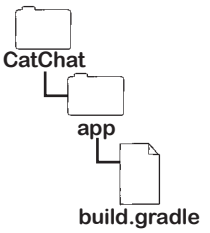
## The app build.gradle file needs a Material library dependency

The Material themes are held in a separate library that you need to include in your app. You do this by adding a dependency to the *com.google.android.material* library in the app's *build.gradle* file.

As we want to use a Material theme in the CatChat app, we need to make sure it includes this library. Open the file *CatChat/app/build.gradle*, and make sure it includes the following line (in bold):

```
dependencies {
    ...
    implementation 'com.google.android.material:material:1.4.0'
    ...
}
```

We're using this version of the Material library.




Android Studio may have already added this dependency to the file for you. If not, you'll need to add it yourself, and click on the Sync Now option that appears at the top of the code editor to sync the change with the rest of your project.

We want to apply a Material theme to the CatChat app to control its app bar. Let's find out how to do this.

**Toolbar**
**Toolbar navigation**
**Bottom bar**
**Navigation drawer**

# Apply a theme in AndroidManifest.xml

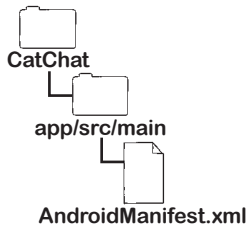
You apply a theme in the app's *AndroidManifest.xml* file. As you learned in Chapter 3, this file provides information about the app's configuration. It features attributes—including the theme—that have a direct impact on the app bar.

Here's the *AndroidManifest.xml* code Android Studio created for us in the CatChat project (we've bolded the key areas):

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.hfad.catchat">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.CatChat">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

This is the user-friendly name of the app.

This is the theme that's applied to the app.




The **android:label** attribute refers to the text that's displayed in the app bar, and it's the name you specify when you first create the app. In the code above, it uses a `String` resource named `app_name`, which Android Studio has added to *strings.xml*.

The **android:theme** attribute specifies the theme. In the code above, it's set using:

```
android:theme="@style/Theme.CatChat">
```

This specifies that the theme is defined as a style resource (denoted by `@style`) named `Theme.CatChat`.

**Toolbar**
**Toolbar navigation**
**Bottom bar**
**Navigation drawer**

# Define styles in style resource files

Style resources are used to describe any themes and styles the app needs to use, and they're held in one or more style resources files.

When we created the CatChat app, Android Studio created two style resource files for us. Both files are named *themes.xml*, and they're located in the *app/src/main/res/values* and *app/src/main/res/values-night* folders. The one in the *values* folder is the app's default style resource file, while the one in the *values-night* folder is used at night.

Some versions of Android Studio name this file styles.xml instead.

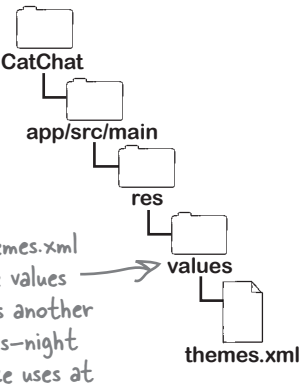
*themes.xml* in the *values* folder defines a style code like this:

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Theme.CatChat" parent="Theme.MaterialComponents.DayNight.DarkActionBar">
        <!-- Primary brand color. -->
        <item name="colorPrimary">@color/purple_500</item>
        <item name="colorPrimaryVariant">@color/purple_700</item>
        <item name="colorOnPrimary">@color/white</item>
        <!-- Secondary brand color. -->
        <item name="colorSecondary">@color/teal_200</item>
        <item name="colorSecondaryVariant">@color/teal_700</item>
        <item name="colorOnSecondary">@color/black</item>
        ...
    </style>
</resources>
```

The style is based on this theme.

This defines the style.

These lines override the parent theme's colors.




This is the themes.xml code from the values folder. There's another version in values-night that the device uses at night.

The **<style>** element tells Android that it's a style resource.

Every style must have a name to identify it by, and you define this using the **name** attribute like so:

```
name="Theme.CatChat"
```

This names the style Theme.CatChat.

*AndroidManifest.xml*'s `theme` attribute uses this name to set the app's theme, for example `"@style/Theme.CatChat"`.

The style also includes a **parent** attribute, which specifies which underlying theme the style should be based on. The above code uses:

```
parent="Theme.MaterialComponents.DayNight.DarkActionBar"
```

This theme displays a dark action bar, and lets the app transition between day and night styles.

which is a theme with a dark app bar that lets the app transition between day and night themes. During the day, it uses the style defined in the *values* folder's style resource file, and it switches to the one in the *values-night* folder at night:

The above style also includes `<item>` elements, which override some of the theme's colors. Let's look at this in more detail.

# Styles can override theme colors

**Toolbar**
**Toolbar navigation**
**Bottom bar**
**Navigation drawer**

If you want to override any properties in the parent theme, such as its color scheme, you can do so by adding `<item>` elements to the style. The CatChat app, for example, includes items that override the primary and secondary brand colors. The primary color is the app's main color, and it's used for things like the app bar. The secondary color provides a contrast, and it's used by some views.

You can use multiple style resource files to apply a different color scheme in different situations. The CatChat app, for example, includes one style resource file in the *values* folder, and another in *values-night*. This arrangement—along with the `DayNight` theme each style is based on—lets you apply different color schemes during the day and at night.

## Serious Coding

You can find out more about how colors are used here:

*https://material.io/develop/android/theming/color*

There's also a color tool to help you choose your app's color scheme:

*https://material.io/tools/color*

Here's the code the CatChat app uses to override the primary colors in its *themes.xml* file in the *values* folder:

```
<item name="colorPrimary">@color/purple_500</item>
<item name="colorPrimaryVariant">@color/purple_700</item>
<item name="colorOnPrimary">@color/white</item>
```

As you can see, each item has a name, and uses `@color` to refer to a color in a **color resource file**.

## Color resource files define a set of colors

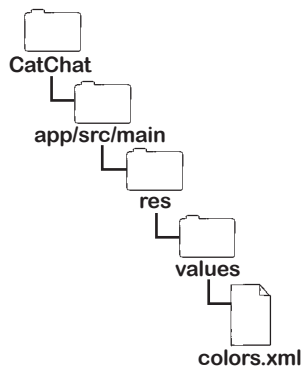
When you create a new project, Android Studio usually includes a default color resource file named *colors.xml*. It's located in the *app/src/main/res/values* folder, and its colors can be used throughout your app.

A typical color resource file looks like this:

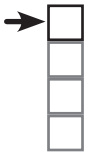
```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="purple_200">#FFBB86FC</color>
    <color name="purple_500">#FF6200EE</color>
    <color name="purple_700">#FF3700B3</color>
    ...
</resources>
```

Each color has a name and a hexadecimal color value.

We've left out some of the other colors.




To change your app's color scheme, simply add the colors you want to the color resource file, and refer to them in its style resource files.



**Toolbar**
Toolbar navigation
Bottom bar
Navigation drawer

# Replace the default app bar with a toolbar

So far, you've seen how Android Studio adds a default app bar to your app by applying a theme. Adding an app bar in this way is easy, but a more flexible approach is to **replace it with a toolbar**.

A basic toolbar looks just like the default app bar you've already seen, but it's much more flexible. You can change its height, for example, and in the next chapter, you'll learn how to make one expand or collapse when the user scrolls the device screen. It also includes all the latest app bar features, so applying a toolbar makes it easier to include these in your app.

You replace the default app bar by removing the original app bar, including a toolbar in your layout, and then telling the activity to use the toolbar as its app bar. We'll show you how to do this over the next few pages.

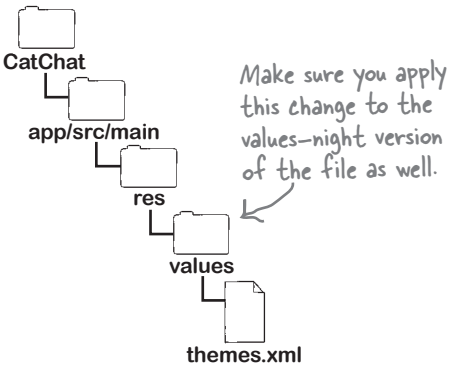
A toolbar looks just like the app bar you had previously, but it's more flexible.

## Remove the default app bar using a theme

You remove the default app bar by applying a theme to the app that doesn't include an app bar.

The current version of the CatChat app, for example, uses the theme `Theme.MaterialComponents.DayNight.DarkActionBar`. We're going to remove the default app bar by replacing it with `Theme.MaterialComponents.DayNight.NoActionBar`. These two themes have the same general appearance, except that the second has no app bar.

Change the theme now by updating the code in the *themes.xml* files in the *values* and *values-night* folders so that they include the change shown below (in bold):




Make sure you apply this change to the values-night version of the file as well.

```
<resources>
    <style name="Theme.CatChat"
           parent="Theme.MaterialComponents.DayNight.~~DarkActionBar~~NoActionBar">
        ...
    </style>
</resources>
```

Change the theme to remove the default app bar.

That's everything that we need to do to remove the default app bar. Next up, how to add the toolbar.

# A toolbar is a type of View

Unlike the default app bar, a toolbar is a type of view that you add to your layout. As it's a view, this means you have full control over its size and position.

Different types of toolbars are available, and in the CatChat app we're going to use a **Material toolbar**. This is a type of toolbar that works well with Material themes, such as the one we're using in this app.

You add a Material toolbar using code like this:

```
<com.google.android.material.appbar.MaterialToolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    style="@style/Widget.MaterialComponents.Toolbar.Primary" />
```

This defines the toolbar.

This gives the toolbar an ID so you can refer to it in your activity code.

These set the size.

This applies a style to it.

You start by defining the toolbar using:

```
<com.google.android.material.appbar.MaterialToolbar
    ... />
```

This is the full path of the MaterialToolbar class.

where `com.google.android.material.appbar.MaterialToolbar` is the fully qualified path of the `MaterialToolbar` class. You then use other view attributes to give it an ID, and specify its appearance. To make the toolbar as wide as its parent and as tall as the default app bar from the underlying theme, for example, you use:

```
android:layout_width="match_parent"
android:layout_height="?attr/actionBarSize"
```

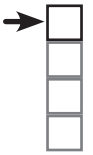
The **`?attr`** prefix in the above code means that you want to use an attribute from the current theme. In this particular case, we're using `?attr/actionBarSize` to refer to the height of the theme's default app bar.

You can also style the toolbar so that it uses the app's primary colors. This is done using the line:

```
style="@style/Widget.MaterialComponents.Toolbar.Primary"
```

This styles the toolbar so it has the same general appearance as the theme's default app bar.

Now that you've learned what toolbar code looks like, let's add one to `MainActivity`'s layout.



**Toolbar**
Toolbar navigation
Bottom bar
Navigation drawer

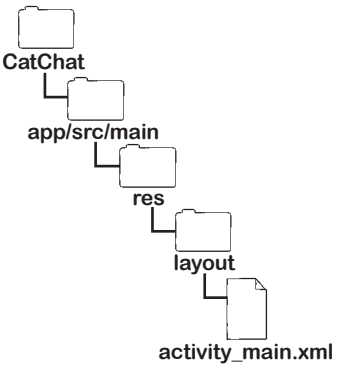
# Add the toolbar to activity_main.xml

We're going to add a toolbar to `MainActivity`'s layout so it appears at the top of the screen. Update the code in *activity_main.xml* so that it matches the code below:

We're using a linear layout.

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

    <com.google.android.material.appbar.MaterialToolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        style="@style/Widget.MaterialComponents.Toolbar.Primary" />
</LinearLayout>
```

The layout includes a Material toolbar and no other views.

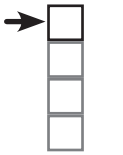



## Where did the app name go?

The above code adds a toolbar to `MainActivity`'s layout, but it doesn't yet include any app bar functionality. If you were to run the app at this point, you'd find, for example, that the app name isn't displayed in the toolbar as it was in the default app bar we had previously:

If you just add the toolbar to your layout and make no other changes, it will just look like an empty bar going across the screen. It won't include the app name or any other app bar functionality.

To get the toolbar to behave like a proper app bar, we need to make some changes to the activity's Kotlin code.



**Toolbar**
Toolbar navigation
Bottom bar
Navigation drawer

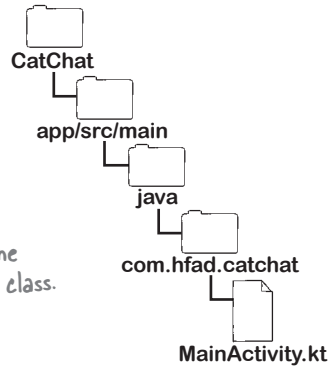
# Set the toolbar as MainActivity's app bar

To make the toolbar behave like a proper app bar, you need to tell `MainActivity` to treat it as one. You do this by calling the **`setSupportActionBar()`** method in the activity's Kotlin code, and passing it a reference to the toolbar like so:

```
val toolbar = findViewById<MaterialToolbar>(R.id.toolbar)
setSupportActionBar(toolbar)
```

This tells Android to treat the toolbar as though it's the default app bar and display the app's name.

We'll add this code to `MainActivity`'s `onCreate()` method so that it runs immediately after the activity gets created. Update the code for *MainActivity.kt* so that it includes the changes (in bold) shown below:




```
package com.hfad.catchat

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import com.google.android.material.appbar.MaterialToolbar

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val toolbar = findViewById<MaterialToolbar>(R.id.toolbar)
        setSupportActionBar(toolbar)
    }
}
```

The code uses the MaterialToolbar class.

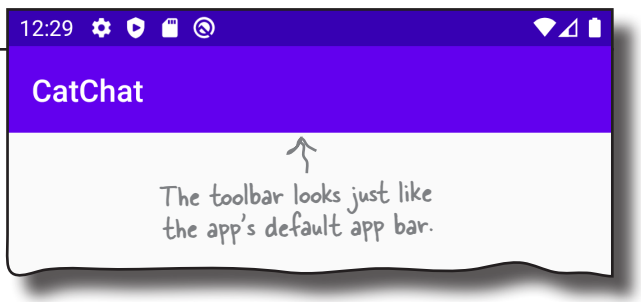
Use the toolbar like the app's default app bar.

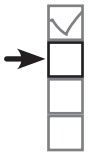
That's everything we need to replace the app's default app bar with a toolbar. Let's take the app for a test drive and make sure it appears correctly.

## Test Drive

When we run the app, `MainActivity` displays a toolbar at the top of the screen. The toolbar includes the name of the app.

Now that we've replaced the default app bar with a toolbar, let's add some more functionality to it.





Toolbar
**Toolbar navigation**
Bottom bar
Navigation drawer

# Let's use the toolbar to navigate

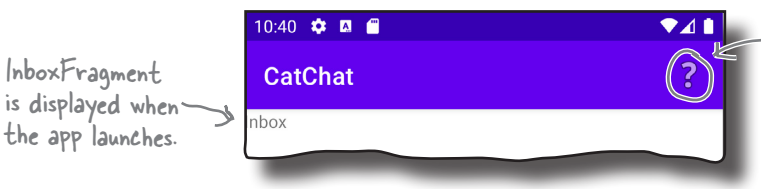
At the beginning of the chapter, we said we wanted to be able to navigate to different screens in the CatChat app using Android's navigation UI components. Now that we've added a toolbar to the app, let's create a couple of screens, and use the toolbar to navigate between them.

## How toolbar navigation will work

We'll begin by creating two new fragments—`InboxFragment` and `HelpFragment`—which will be displayed in `MainActivity` when the user navigates to each one. `InboxFragment` will be displayed when the app is launched, and `HelpFragment` will be displayed when we navigate to it.

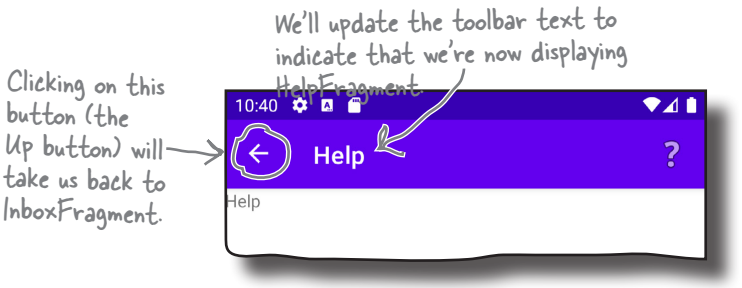
We're not going to add any email or help functionality to InboxFragment and HelpFragment. We'll just display a piece of text in each one so we know which fragment is being displayed.

We'll navigate to `HelpFragment` by adding a Help item to the toolbar so it looks like this:




We'll add the Help item using a menu; you'll find out how to do this a few pages ahead.

When we click on the Help item, the app will navigate to `HelpFragment`. We'll also update the toolbar's title to indicate what the current screen is, and provide an Up button to make it easy to navigate back to `InboxFragment`.




So that's how the toolbar navigation will work. Let's start implementing it by adding `InboxFragment` and `HelpFragment` to the app.

## there are no Dumb Questions

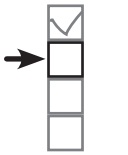
**Q: Why is the method you use to set an app bar called `setSupportActionBar()`? Shouldn't it be `setSupportAppBar()`?**

A: In early versions of Android, the app bar was called an action bar, and behind the scenes, the default app bar uses a class named `ActionBar`. This is why the method is called `setSupportActionBar`.

**Q: Can styles only be used to override the default colors?**

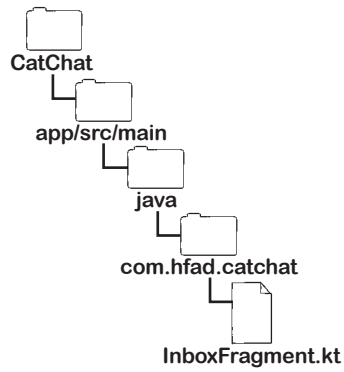
A: No, you can use them to override any of the theme's properties. To stop the text on all buttons from being displayed as uppercase, for example, you can add an item with the name `"android:textAllCaps"`, which has the value `false`.

# Create InboxFragment




We want to display `InboxFragment` in `MainActivity` when the app launches. Highlight the *com.hfad.catchat* package in the *app/src/main/java* folder, then go to File→New→Fragment→Fragment (Blank). Name the fragment "InboxFragment", name its layout "fragment_inbox", and make sure the language is set to Kotlin. Then update the code for *InboxFragment.kt* to match the code below:

This is what InboxFragment looks like.

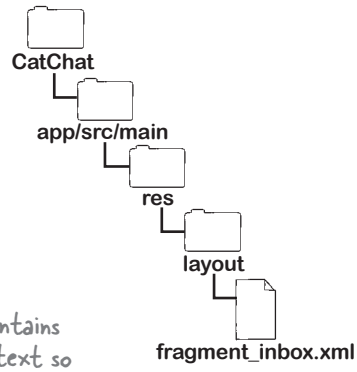


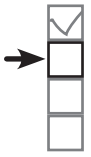

```
package com.hfad.catchat

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup

class InboxFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.fragment_inbox, container, false)
    }
}
```

And here's the code for *fragment_inbox.xml* (update the code for this file too):

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    tools:context=".InboxFragment">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Inbox" />
</FrameLayout>
```

InboxFragment's layout just contains a TextView. We're adding this text so we can easily tell when it's displayed.



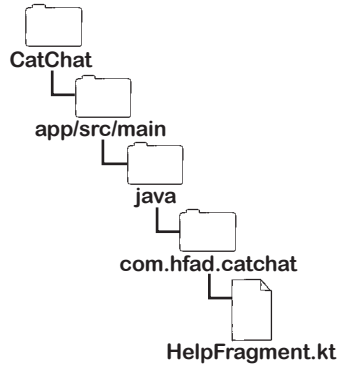


Toolbar
**Toolbar navigation**
Bottom bar
Navigation drawer

# Create HelpFragment

We'll display `HelpFragment` in `MainActivity` when the user clicks on the Help item in the toolbar. Highlight the *com.hfad.catchat* package in the *app/src/main/java* folder, then go to File→New→Fragment→Fragment (Blank). Name the fragment "HelpFragment", name its layout "fragment_help", and make sure the language is set to Kotlin. Then make sure the code for *HelpFragment.kt* matches the code below:

HelpFragment looks like this.

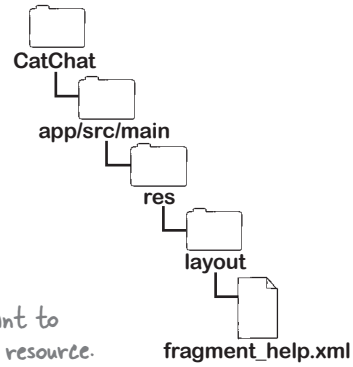


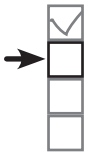

```
package com.hfad.catchat

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup

class HelpFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.fragment_help, container, false)
    }
}
```

Then make sure the code for *fragment_help.xml* matches the layout below:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    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"
    tools:context=".HelpFragment">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Help" />
</FrameLayout>
```

In the real world, you'd want to save this text as a String resource.





Toolbar
**Toolbar navigation**
Bottom bar
Navigation drawer

# We'll use the Navigation component to navigate to HelpFragment

In Chapter 6, you learned how to navigate between fragments using Android's Navigation component. Even though we're now going to use the toolbar to navigate to a new destination, we can still use the Navigation component for all our navigation needs.

We first need to add the Navigation component to the CatChat project using Gradle.

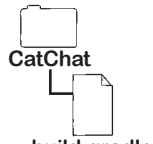
## Add a version number to the project build.gradle file

Just as before, you need to add a new variable to the project's *build.gradle* file that specifies the version of the Navigation component that should be added to the app.

Open the file *CatChat/build.gradle*, and add the following line (in bold) to the `buildscript` section:

```
buildscript {
    ext.nav_version = '2.3.5'
    ...
}
```

We'll use this version of the Navigation component.




## Add dependencies to the app build.gradle file

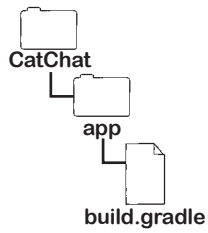
Next, you need to add two Navigation component dependencies to the app's version of the *build.gradle* file.

Open the file *CatChat/app/build.gradle*, and add the following two lines (in bold) to the `dependencies` section:

Make sure you add both lines to your app's build.gradle file.

```
dependencies {
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
    ...
}
```

We're adding an extra dependency for the navigation UI components.




Once you've made these changes, click on the Sync Now option that appears at the top of the code editor to sync the changes with the rest of your project.

Now that you've added the Navigation component to your project, let's use it to create a navigation graph after the following exercise.

## there are no Dumb Questions

**Q: Do I have to add the Navigation component to each app that includes navigation? Isn't there a shortcut?**

A: The Navigation component isn't included in the main Android libraries, so you need to add it to your project before you can use it.

Some versions of Android Studio automatically add Navigation component dependencies when you create a navigation graph, but they might use a different version. We're getting you to add them manually so that you know it's there, and what version it is.

# What's My Purpose?

We're going to use the Navigation component to navigate between fragments in the CatChat app, but can you can remember what each part of the Navigation component does?

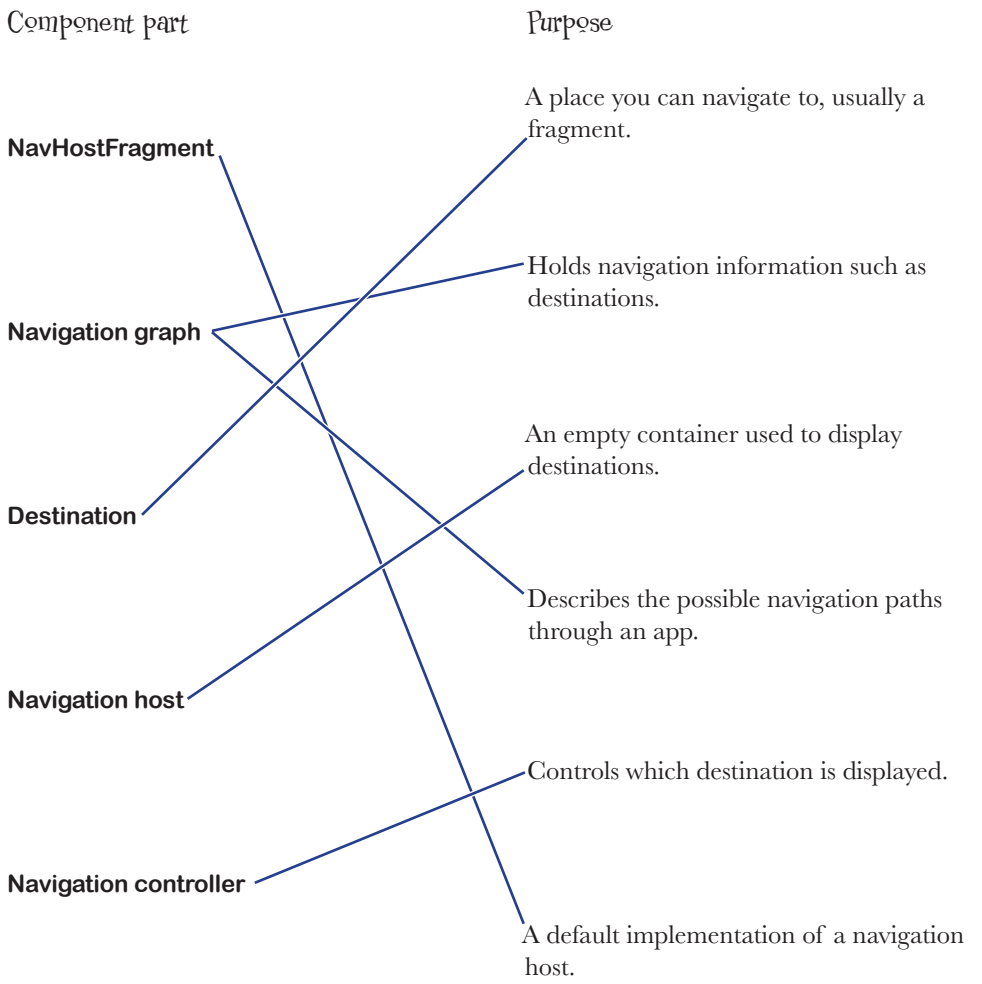
Draw a line from each part to its purpose. Hint: some things might have more than one purpose.

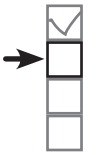
| Component part | Purpose |
| --- | --- |
| **NavHostFragment** | A place you can navigate to, usually a fragment. |
| **Navigation graph** | Holds navigation information such as destinations. |
| **Destination** | An empty container used to display destinations. |
| **Navigation host** | Describes the possible navigation paths through an app. |
| **Navigation controller** | Controls which destination is displayed. |
| | A default implementation of a navigation host. |

# What's My Purpose? Solution

We're going to use the Navigation component to navigate between fragments in the CatChat app, but can you can remember what each part of the Navigation component does?

Draw a line from each part to its purpose. Hint: some things might have more than one purpose.

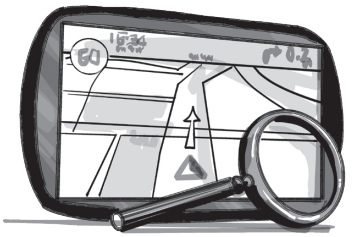




Toolbar
**Toolbar navigation**
Bottom bar
Navigation drawer

# Add the fragments to a navigation graph

As you already know, your app's navigation graph holds details of the destinations in your app and the possible paths you can take to navigate to them. As `InboxFragment` and `HelpFragment` are possible destinations in the CatChat app, we'll create a new navigation graph, and add the two fragments to it.



Navigation Graph

To create the navigation graph, select the *CatChat/app/src/main/res* folder in the project explorer, and then choose File→New→Android Resource File. When prompted, enter a file name of "nav_graph", choose a resource type of "Navigation", and click on the OK button.

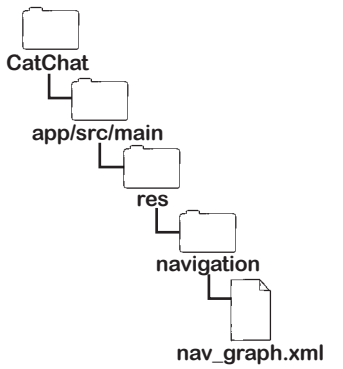
Next, open the navigation graph (file *nav_graph.xml*), switch to the Code view, and update the file so that it matches the code below:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/inboxFragment">

    <fragment
        android:id="@+id/inboxFragment"
        android:name="com.hfad.catchat.InboxFragment"
        android:label="Inbox"
        tools:layout="@layout/fragment_inbox" />
    <fragment
        android:id="@+id/helpFragment"
        android:name="com.hfad.catchat.HelpFragment"
        android:label="Help"
        tools:layout="@layout/fragment_help" />
</navigation>
```

This line means that InboxFragment is the first destination to be displayed.

InboxFragment has an ID of inboxFragment and is labeled Inbox.

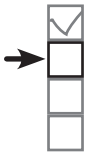
HelpFragment has an ID of helpFragment and is labeled Help.




The above code adds `InboxFragment` and `HelpFragment` to the navigation graph, and assigns a user-friendly label to each one.

Those are all the changes we need to make to the navigation graph. Next, let's update `MainActivity`'s layout so that it can display each fragment when we navigate to it.

# Add a navigation host to activity_main.xml

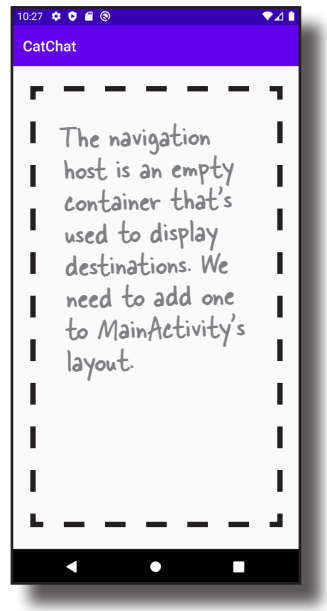


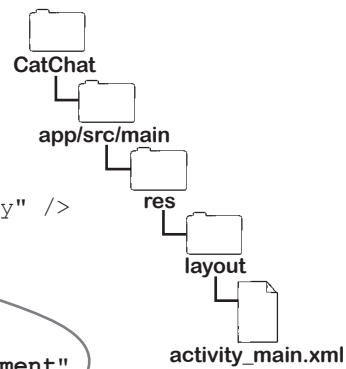
Toolbar
**Toolbar navigation**
Bottom bar
Navigation drawer

As you learned in Chapter 6, you display each destination you navigate to by adding a navigation host to the activity's layout. You do this using a `FragmentContainerView`, specifying the type of navigation host you want to use, and the name of the navigation graph.

The code to do this in the CatChat app is almost the same as the code that we added to the Secret Message app in Chapter 6. Update your code for *activity_main.xml* to include the changes below (in bold):

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

    <com.google.android.material.appbar.MaterialToolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        style="@style/Widget.MaterialComponents.Toolbar.Primary" />

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/nav_host_fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:defaultNavHost="true"
        app:navGraph="@navigation/nav_graph" />
</LinearLayout>
```




This is the code to add the navigation host. It uses nav_graph.xml as its navigation graph.

Now that we've added the navigation host, let's add an item to the toolbar, which we can use to navigate to `HelpFragment`.

# Specify items in the toolbar with a menu resource file

You tell Android which items should appear on the toolbar by defining a **menu**. Each menu is defined in an XML menu resource file which specifies the items that you want to appear.

We're going to create a new menu resource file named *menu_toolbar.xml* which we'll use to add the Help item to the toolbar. Select the *app/src/main/res* folder, go to the File menu, choose New, and then select the option to create a new Android resource file. When prompted, give it a name of "menu_toolbar", specify a resource type of "Menu", and make sure that the directory name is *menu*. When you click on the OK button, Android Studio will create the file for you, and add it to the *app/src/main/res/menu* folder.

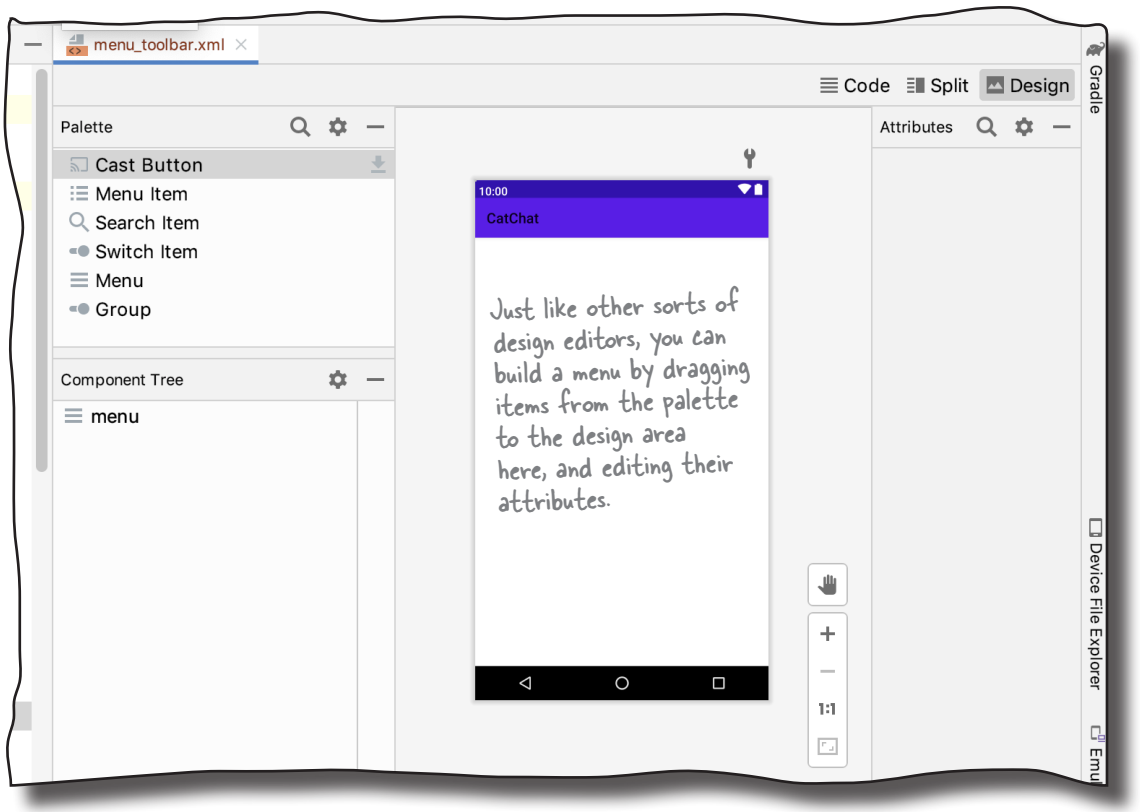
Just like navigation graphs and layout files, you can edit menu resource files by updating the XML code, or by using the built-in design editor. Here's what the design editor looks like:

## there are no Dumb Questions

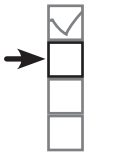
**Q: Is it just toolbars that use menu resource files?**

A: As you'll find out later in the chapter, menu resource files are also used for bottom navigation bars and navigation drawers. You can create pop-up and context menus with them too. Find out more here:

*https://developer.android.com/guide/topics/ui/menus*

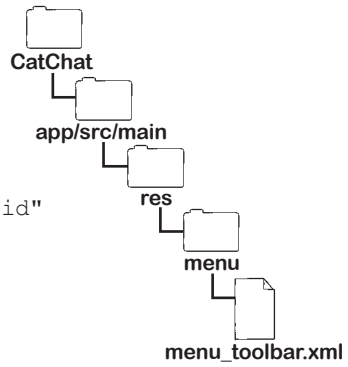


# Let's add a Help item to the menu




We want to be able to navigate to `HelpFragment` from the toolbar, so we'll add a Help item to the menu resource file. We'll do this by directly editing the XML code.

Switch to the Code view of *menu_toolbar.xml*, then update the code so it matches the code below (the lines you need to add are in bold):




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

    <item
        android:id="@+id/helpFragment"
        android:icon="@android:drawable/ic_menu_help"
        android:title="Help"
        app:showAsAction="always" />
</menu>
```

This element defines the menu.

This section defines the Help item, and displays it using the built-in Android Help icon.

Every menu resource file, including the one above, has a **<menu>** root element. This is what tells Android that it defines a menu.

Inside the `<menu>` element, you typically have a number of `<item>` elements, each one describing a separate item. In this particular case, there's a single Help item with the title "Help".

The `<item>` element has a number of attributes that you can use to control the item's appearance.

The `android:id` attribute assigns an ID to the item. This ID is used by the Navigation component to navigate to a destination, and **it must have the same ID as the destination you want to navigate to in the navigation graph**. You'll find out how this works a few pages ahead.

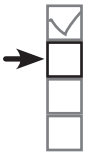
The `android:icon` attribute specifies which icon (if any) should be displayed for the item. Android has many built-in icons, and the IDE presents you with a list of available ones when you start typing the icon name.

The `android:title` attribute defines the item's text.

The `app:showAsAction` attribute specifies how you want the item to appear in the toolbar. Setting it to "always" means that it should always appear in the main area of the toolbar.

**The menu item's ID must match the destination ID in the navigation graph.**

*Navigation UI components—such as the toolbar—rely on these IDs being the same. If they're different, your toolbar navigation won't work properly.*



Toolbar
**Toolbar navigation**
Bottom bar
Navigation drawer

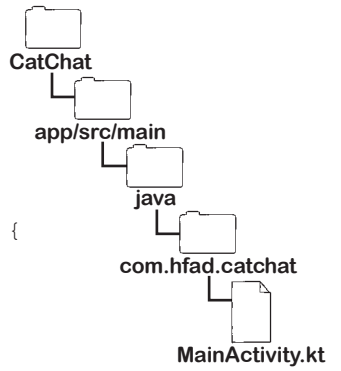
# onCreateOptionsMenu() adds menu items to the toolbar

Once you've defined the menu resource file, you add its items to the toolbar by implementing the **`onCreateOptionsMenu()`** method in your activity code. This method gets called when the activity is ready to add items to the toolbar. It inflates the menu resource file, and adds each item it describes to the toolbar.

In the CatChat app, we want to add the item defined in *menu_toolbar.xml* to `MainActivity`'s toolbar. The code to add this item is shown below in bold (you'll update your code a few pages ahead):

```
package com.hfad.catchat

...
import android.view.Menu

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
    }

    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        menuInflater.inflate(R.menu.menu_toolbar, menu)
        return super.onCreateOptionsMenu(menu)
    }
}
```

The onCreateOptionsMenu() method uses the Menu class.




Implementing this method adds any items in the menu resource file to the toolbar.

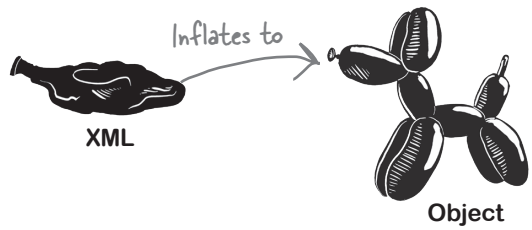
This is the name of the menu resource file.

The line:

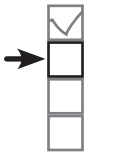
```
menuInflater.inflate(R.menu.menu_toolbar, menu)
```

inflates the menu resource file. Behind the scenes, it creates a `Menu` object that represents the menu resource file, and any items the menu resource file contains are translated to `MenuItem` objects. These are then added to the toolbar.

That's everything that's needed to add a menu to the toolbar. Next, we'll make `MainActivity` navigate to `HelpFragment` when the user clicks the Help item.



# Respond to menu item clicks with onOptionsItemSelected()



Toolbar
**Toolbar navigation**
Bottom bar
Navigation drawer

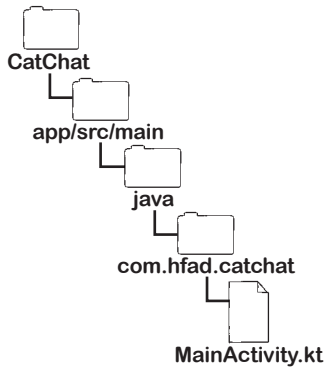
Once you've used the `onCreateOptionsMenu()` method to add menu items to the toolbar, you make them respond to clicks by implementing **`onOptionsItemSelected()`**. This method runs whenever an item in the toolbar is clicked:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    //Code that runs when an item is clicked
}
```

This refers to the item that's been clicked.

In the CatChat app, we want to navigate to `HelpFragment` when the user clicks the Help menu item, and we can achieve this with a little help from the Navigation component.

Here's what the code looks like; we'll add these changes to *MainActivity.kt* a few pages ahead:



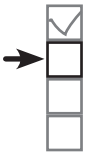

```
package com.hfad.catchat

...
import android.view.MenuItem
import androidx.navigation.findNavController
import androidx.navigation.ui.onNavDestinationSelected

class MainActivity : AppCompatActivity() {
    ...

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        val navController = findNavController(R.id.nav_host_fragment)
        return item.onNavDestinationSelected(navController)
                || super.onOptionsItemSelected(item)
    }
}
```

The code uses these extra classes.

When an item in the toolbar is clicked, the navigation controller navigates to the correct destination.

Most onOptionsItemSelected implementations look like this.

Each time the Help menu item is clicked, the navigation controller gets its ID and looks for a matching one in the navigation graph. It then passes the destination with this ID to the navigation host so that it's displayed on the device screen.



Toolbar
**Toolbar navigation**
Bottom bar
Navigation drawer

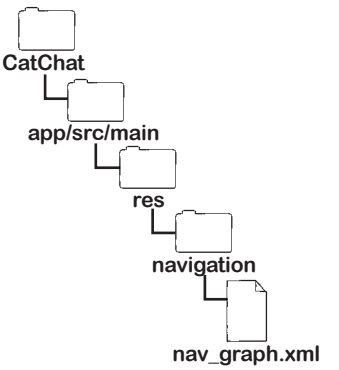
# We need to configure the toolbar

We've now covered everything that's needed to add the Help menu item to the toolbar and make it navigate to `HelpFragment`. Before we take the app for a test drive, however, there's one more change we're going to make.

When we defined the navigation graph *nav_graph.xml* earlier in the chapter, we added labels to each destination using code like this:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation ...>
    <fragment
        android:id="@+id/inboxFragment"
        android:name="com.hfad.catchat.InboxFragment"
        android:label="Inbox"
        tools:layout="@layout/fragment_inbox" />
    <fragment
        android:id="@+id/helpFragment"
        android:name="com.hfad.catchat.HelpFragment"
        android:label="Help"
        tools:layout="@layout/fragment_help" />
</navigation>
```

This is the navigation graph code you've already created. There's nothing new.




To make it clear which screen is being displayed, we're going to configure the toolbar so that each time we navigate to a new destination, its label is displayed in the toolbar. We'll also add an Up button to the toolbar so that when the user navigates to `HelpFragment`, it provides an easy way of navigating back to `InboxFragment`:

This is what the app bar looks like when we launch the app.

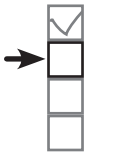
When we navigate to HelpFragment, we want the app bar text to say "Help" and to include an Up button.

This is the Up button.

Up navigation may sound the same as using the device Back button, but it's slightly different. The Back button allows users to work their way *back* through the back stack: the history of screens they've visited. The Up button, on the other hand, is based on the navigation graph hierarchy. It provides a quick way of going *up* this hierarchy.

We can configure the toolbar to update the text and include an Up button using the Navigation component. Let's find out how.

# Configure the toolbar using an AppBarConfiguration




We want to configure the toolbar so that the text it displays corresponds to the label of the current destination in the navigation graph, and so that it includes an Up button. We can do this by building an **`AppBarConfiguration`** object that's based on the navigation graph, and linking it to the toolbar. The `AppBarConfiguration` class is part of the Navigation component, and it's used to make app bars and toolbars play nicely with the navigation controller.

Here's the code to build an `AppBarConfiguration` and link it to a toolbar (in bold):

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    val toolbar = findViewById<MaterialToolbar>(R.id.toolbar)
    setSupportActionBar(toolbar)
    val navHostFragment = supportFragmentManager
        .findFragmentById(R.id.nav_host_fragment) as NavHostFragment
    val navController = navHostFragment.navController
    val builder = AppBarConfiguration.Builder(navController.graph)
    val appBarConfiguration = builder.build()
    toolbar.setupWithNavController(navController, appBarConfiguration)
}
```

These lines get a reference to the navigation controller from the navigation host.

This builds a configuration linking the toolbar to the navigation graph.

This line applies the configuration to the toolbar.

The above code first gets a reference to the navigation controller from the navigation host using the code:

```
val navHostFragment = supportFragmentManager
    .findFragmentById(R.id.nav_host_fragment) as NavHostFragment
val navController = navHostFragment.navController
```

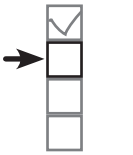
We're using this code because calling findNavController() from an activity's onCreate() method can fail.

Code like this is necessary whenever you need to get a reference to the navigation controller from an activity's `onCreate()` method.

The code then builds a configuration that links the toolbar to the navigation graph, and applies it to the toolbar.

When the code runs, it uses information in the navigation graph to display the current destination's label. It also adds an Up button to the toolbar for all destinations except the navigation graph's Start destination, which in this case is `InboxFragment`.

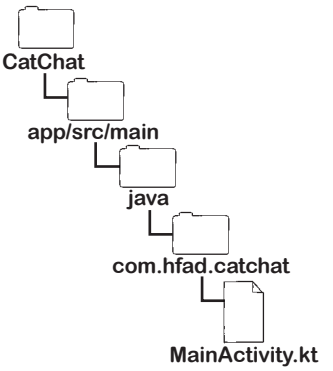
That's everything that you need to know to implement toolbar navigation. We'll show you the full `MainActivity` code on the next page.



Toolbar
**Toolbar navigation**
Bottom bar
Navigation drawer

# The full code for MainActivity.kt

Here's the complete code for *MainActivity.kt*; update your code to include the changes below (in bold):




```
package com.hfad.catchat

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.view.Menu
import android.view.MenuItem
import androidx.navigation.findNavController
import androidx.navigation.fragment.NavHostFragment
import androidx.navigation.ui.AppBarConfiguration
import androidx.navigation.ui.onNavDestinationSelected
import androidx.navigation.ui.setupWithNavController
import com.google.android.material.appbar.MaterialToolbar

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val toolbar = findViewById<MaterialToolbar>(R.id.toolbar)
        setSupportActionBar(toolbar)
        val navHostFragment = supportFragmentManager
            .findFragmentById(R.id.nav_host_fragment) as NavHostFragment
        val navController = navHostFragment.navController
        val builder = AppBarConfiguration.Builder(navController.graph)
        val appBarConfiguration = builder.build()
        toolbar.setupWithNavController(navController, appBarConfiguration)
    }

    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        menuInflater.inflate(R.menu.menu_toolbar, menu)
        return super.onCreateOptionsMenu(menu)
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        val navController = findNavController(R.id.nav_host_fragment)
        return item.onNavDestinationSelected(navController)
                || super.onOptionsItemSelected(item)
    }
}
```

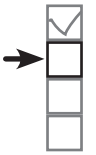
Import all of these extra classes.

Configure the toolbar to include an Up button and display which screen you've navigated to.

Add any menu items to the toolbar (in this case the Help item).

Navigate to a destination when an item's clicked.

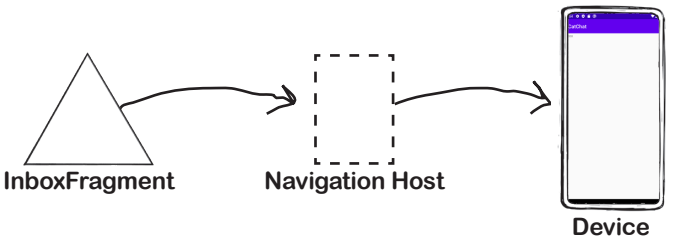
# What happens when the app runs



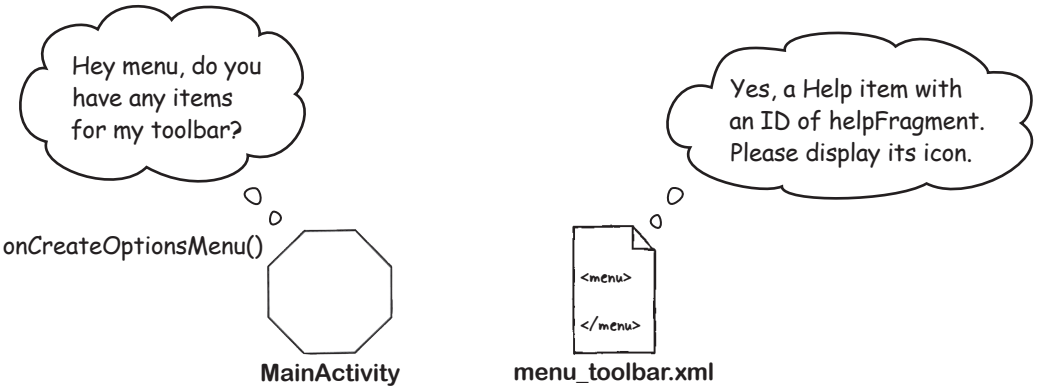
**Toolbar**
**Toolbar navigation**
**Bottom bar**
**Navigation drawer**

The following things happen when the app runs:

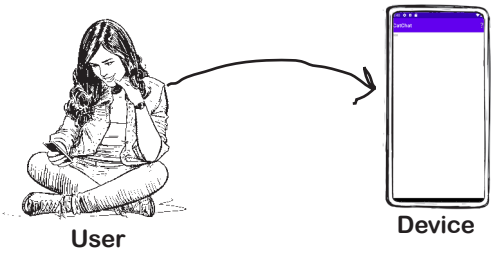
1. **The app is launched and MainActivity gets created.**
`InboxFragment` is added to the navigation host and displayed on the device screen.

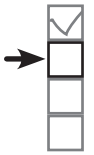



2. **MainActivity's onCreateOptionsMenu method runs.**
It adds the Help menu item defined in *menu_toolbar.xml* to the toolbar.



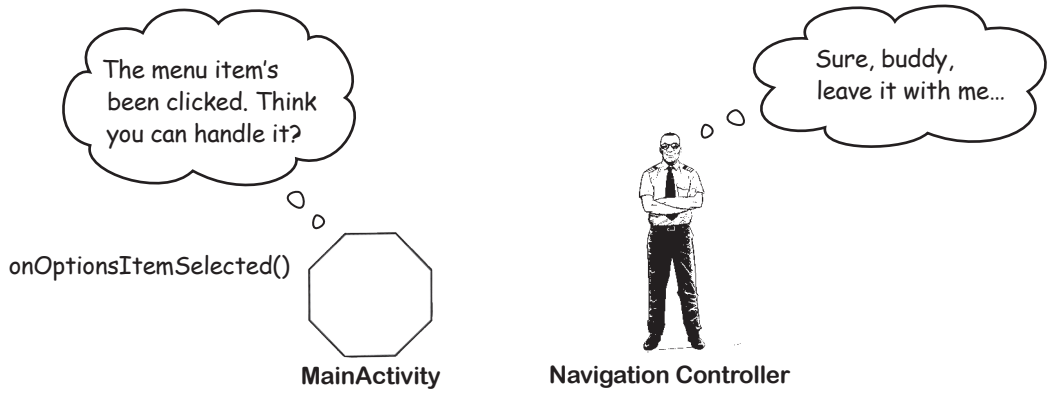

3. **The user clicks the Help item in the toolbar.**



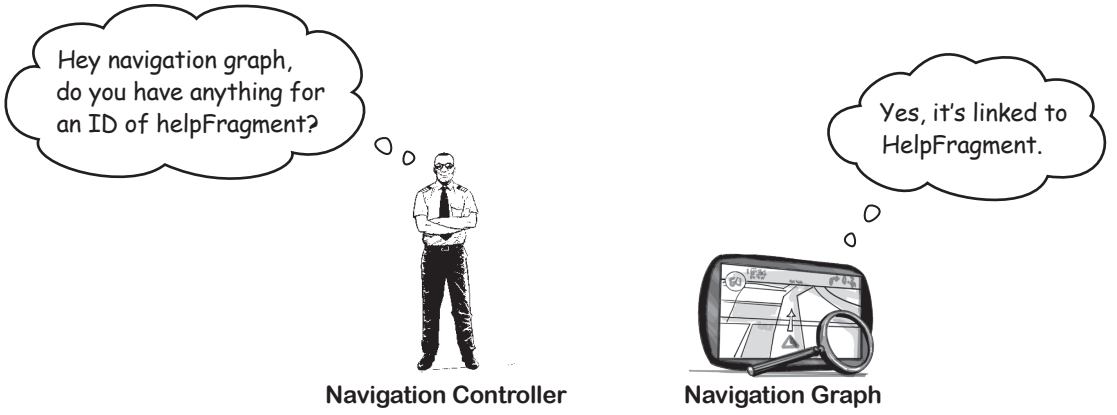


Toolbar
**Toolbar navigation**
Bottom bar
Navigation drawer

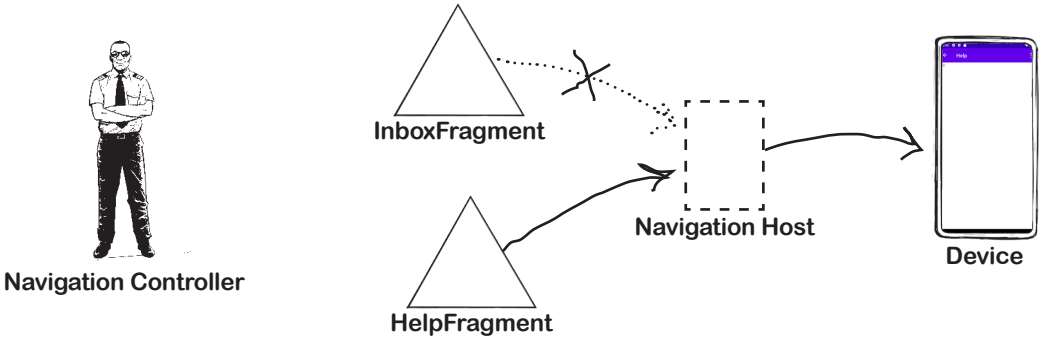
**4** **MainActivity's onOptionsItemSelected() method runs.**
It passes navigation for the Help item to the navigation controller.



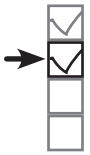

**5** **The navigation controller looks for the Help item's ID in the navigation graph.**




**6** **The navigation controller replaces InboxFragment with HelpFragment in the navigation host.**



# Test Drive

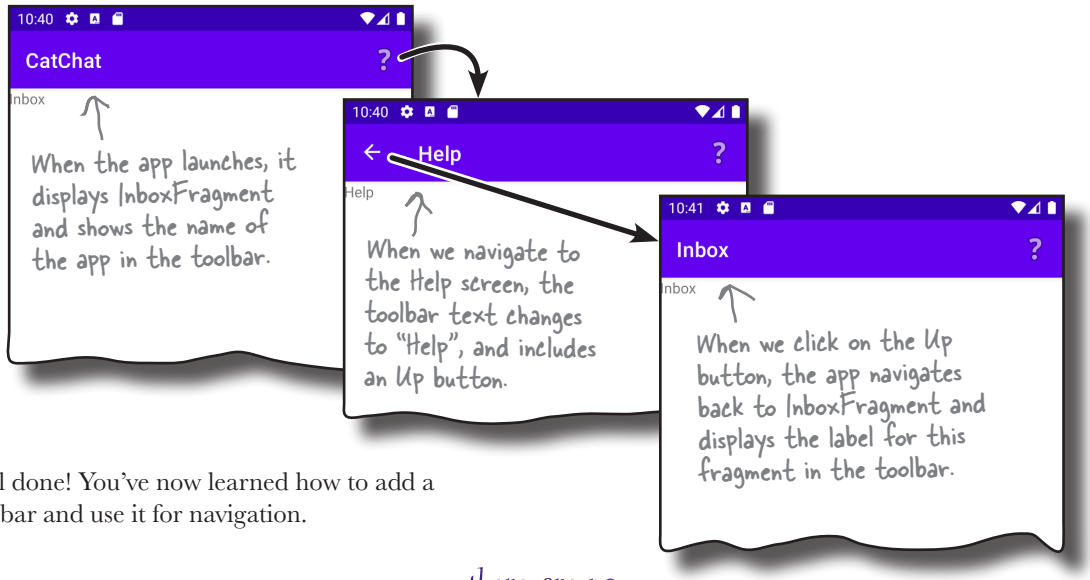


- Toolbar
- **Toolbar navigation**
- Bottom bar
- Navigation drawer

When we run the CatChat app, `MainActivity` is launched, and `InboxFragment` is displayed in `MainActivity`'s layout.

A Help item is displayed in `MainActivity`'s toolbar. When we click on it, the app navigates to `HelpFragment`. The toolbar text is changed to "Help" and an Up button appears.

When we click on the Up button, the app navigates to `InboxFragment`. The Up button disappears and the text "Inbox" is displayed in the toolbar.




Well done! You've now learned how to add a toolbar and use it for navigation.

## there are no Dumb Questions

**Q: Could we have added menu items and added an Up button using the default app bar?**

A: Yes, but it would have required more Kotlin code to do so. To implement the Up button, for example, you'd need to override a separate method named `onSupportNavigateUp()`.

**Q: When does the Up button appear?**

A: When any fragment is displayed in the navigation host that's not the start destination in the navigation graph. In this example, it appears for any fragment except `InboxFragment`.

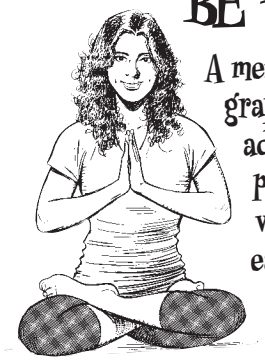
**Q: Can you add a toolbar to a fragment?**

A: Yes. A toolbar is a type of `View`, so you can add it to activities or fragments.

**Q: How do I know whether to add it to an activity or a fragment?**

A: If you want the toolbar's layout to be the same for all fragments, add it to the activity that displays these fragments. If you want the toolbar's layout to be significantly different, consider adding it to the fragment instead.

# BE the Menu



A menu resource file and navigation graph is shown below. The menu is added to a toolbar. Your job is to play like you're the menu, and say which fragment is navigated to when each item in the menu is clicked.

This is the menu resource file.

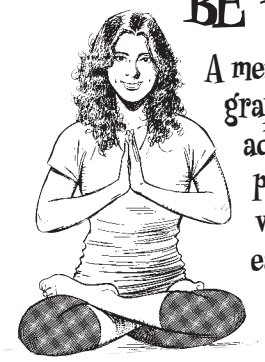
```
<menu ...>
A   <item
        android:id="@+id/homeFragment"
        android:title="Home" />
B   <item
        android:id="@+id/ordersFragment"
        android:title="Orders" />
C   <item
        android:id="@+id/accountFragment"
        android:title="Account" />
</menu>
```

Which fragment does the app navigate to when each of these items is clicked?

This is the navigation graph.

```
<navigation ...>
    <fragment
        android:id="@+id/startFragment"
        android:name="com.hfad.store.HomeFragment"
        android:label="Home" />
    <fragment
        android:id="@+id/homeFragment"
        android:name="com.hfad.store.StartFragment"
        android:label="Hello" />
    <fragment
        android:id="@+id/orderFragment"
        android:name="com.hfad.store.OrdersFragment"
        android:label="Orders" />
    <fragment
        android:id="@+id/accountFragment"
        android:name="com.hfad.store.YourAccountFragment"
        android:label="Your Account" />
</navigation>
```

# BE the Menu Solution



A menu resource file and navigation graph is shown below. The menu is added to a toolbar. Your job is to play like you're the menu, and say which fragment is navigated to when each item in the menu is clicked.

This is the menu resource file.

```
<menu ...>
A   <item
        android:id="@+id/homeFragment"
        android:title="Home" />
B   <item
        android:id="@+id/ordersFragment"
        android:title="Orders" />
C   <item
        android:id="@+id/accountFragment"
        android:title="Account" />
</menu>
```

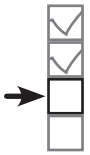
This navigates to StartFragment, as this destination has an ID of homeFragment.

There's no destination with a matching ID, so this item doesn't navigate anywhere.

This navigates to YourAccountFragment, as this destination has an ID of accountFragment.

This is the navigation graph.

```
<navigation ...>
    <fragment
        android:id="@+id/startFragment"
        android:name="com.hfad.store.HomeFragment"
        android:label="Home" />
    <fragment
        android:id="@+id/homeFragment"
        android:name="com.hfad.store.StartFragment"
        android:label="Hello" />
    <fragment
        android:id="@+id/orderFragment"
        android:name="com.hfad.store.OrdersFragment"
        android:label="Orders" />
    <fragment
        android:id="@+id/accountFragment"
        android:name="com.hfad.store.YourAccountFragment"
        android:label="Your Account" />
</navigation>
```



- Toolbar
- Toolbar navigation
- **Bottom bar**
- Navigation drawer

# Most types of UI navigation work with the Navigation component

In the chapter so far, you've learned how to enable toolbar navigation by defining a menu resource file, using the Navigation component to navigate between fragments, and configuring the toolbar to change its appearance as you navigate through the app.

The great news is that other types of UI navigation—such as bottom navigation bars and navigation drawers—work in a similar way. Even though they look different from one another, you can take the techniques you've learned for toolbar navigation and apply them to other types of navigation.

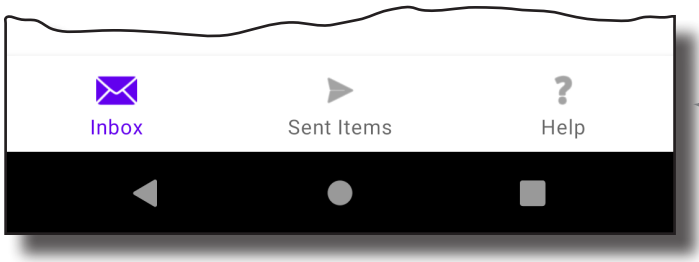
To see how this works, we're going to add a bottom navigation bar to the CatChat app's `MainActivity`.

**A bottom navigation bar can hold up to five items. It's displayed at the bottom of the screen.**

## How the bottom navigation bar will work

As its name suggests, a bottom navigation bar is a type of navigation bar that's positioned at the bottom of the device screen. You can use it to navigate to up to five destinations.

The CatChat app's bottom navigation bar will look like this:

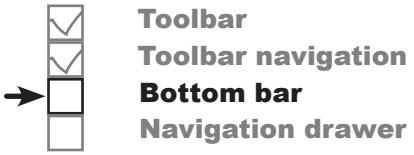



This is the bottom navigation bar that we'll add to the CatChat app. We'll add three items to it, one for each destination we want to navigate to.

As you can see, the bar contains three items: Inbox, Sent Items, and Help. When we click on each item, the app will navigate to the fragment associated with it. When we click on the Help item, for example, the app will navigate to `HelpFragment`, and when we click on Sent Items, it will navigate to a new fragment (which we need to create) named `SentItemsFragment`.

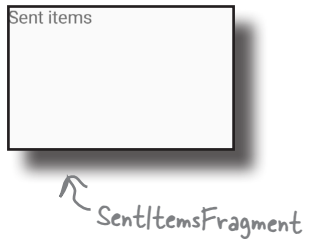
We'll implement bottom bar navigation over the next few pages, and we'll start by creating the new fragment: `SentItemsFragment`.

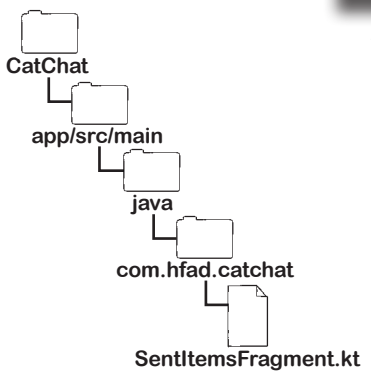
# Create SentItemsFragment




We need to create a new fragment named `SentItemsFragment`.

Highlight the *com.hfad.catchat* package in the *app/src/main/java* folder, then go to File→New→Fragment→Fragment (Blank). Name the fragment "SentItemsFragment", name its layout "fragment_sent_items", and make sure the language is set to Kotlin. Then update the code for *SentItemsFragment.kt* to match the code below:

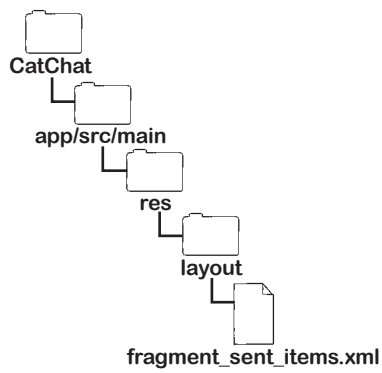


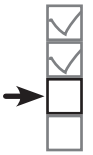




```
package com.hfad.catchat

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup

class SentItemsFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.fragment_sent_items, container, false)
    }
}
```

And here's the code for *fragment_sent_items.xml* (update your version of this code too):

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    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"
    tools:context=".SentItemsFragment">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Sent Items" />
</FrameLayout>
```





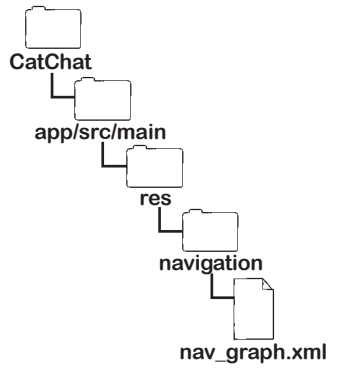
- Toolbar
- Toolbar navigation
- **Bottom bar**
- Navigation drawer

# Add SentItemsFragment to the Navigation graph

We want to be able to navigate to `SentItemsFragment` from the bottom navigation bar using the Navigation component. To do this, we need to add the fragment to the navigation graph as a new destination.

Open the navigation graph file *nav_graph.xml* (if it's not already open), then update its code so that it matches the code below (our changes are in bold):

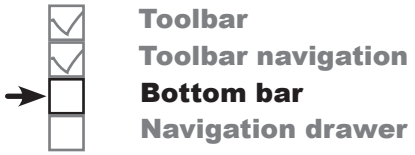
```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/inboxFragment">

    <fragment
        android:id="@+id/inboxFragment"
        android:name="com.hfad.catchat.InboxFragment"
        android:label="Inbox"
        tools:layout="@layout/fragment_inbox" />
    <fragment
        android:id="@+id/helpFragment"
        android:name="com.hfad.catchat.HelpFragment"
        android:label="Help"
        tools:layout="@layout/fragment_help" />
    <fragment
        android:id="@+id/sentItemsFragment"
        android:name="com.hfad.catchat.SentItemsFragment"
        android:label="Sent Items"
        tools:layout="@layout/fragment_sent_items" />
</navigation>
```




We want to be able to navigate to SentItemsFragment, so it needs to be added to the navigation graph.

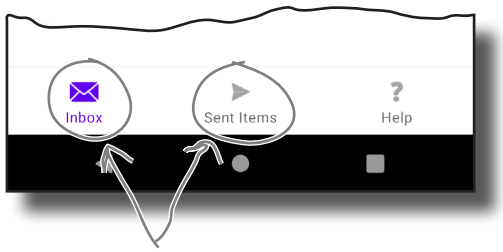
As you can see, the new destination has an ID of `sentItemsFragment`, and a label of "Sent Items". We'll use the ID in the bottom navigation bar's menu resource file, which we'll create next.

# The bottom navigation bar needs a new menu resource file




Earlier in the chapter, we created a menu resource file named *menu_toolbar.xml* to add a Help item to the toolbar. While it's possible for different navigation UI components to share the same menu, we need to create a new one for the bottom navigation bar because it shows two extra items: Inbox and Sent Items.

To create the new menu resource file, select the *app/src/main/res* folder, go to the File menu, choose New, and then select the option to create a new Android resource file. When prompted, give it a name of "menu_main", specify a resource type of "Menu", and make sure the directory name is *menu*. When you click on the OK button, Android Studio will create the file for you, and add it to the *app/src/main/res/menu* folder.

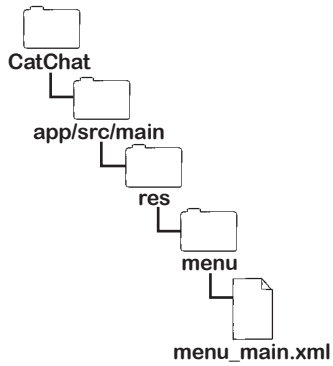



The bottom navigation bar needs to include these extra items, so we're creating a new menu file for it.

Next, open the file *menu_main.xml* (if it's not already open), and update its code to add menu items for `InboxFragment`, `SentItemsFragment`, and `HelpFragment` (shown below in bold):

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

    <item
        android:id="@+id/inboxFragment"
        android:icon="@android:drawable/ic_dialog_email"
        android:title="Inbox" />
    <item
        android:id="@+id/sentItemsFragment"
        android:icon="@android:drawable/ic_menu_send"
        android:title="Sent Items" />
    <item
        android:id="@+id/helpFragment"
        android:icon="@android:drawable/ic_menu_help"
        android:title="Help" />
</menu>
```

These are the three items that we want to add to the bottom navigation bar. Their IDs need to match the corresponding IDs in the navigation graph.




Now that we've created the new menu resource file, let's add a bottom navigation bar to `MainActivity`.

Toolbar
Toolbar navigation
**Bottom bar**
Navigation drawer

# A bottom navigation bar is a type of View

Just like a toolbar, a bottom navigation bar is a type of view that you add to your layout. The code to add a bottom navigation bar looks like this:

```
<com.google.android.material.bottomnavigation.BottomNavigationView
    android:id="@+id/bottom_nav"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:menu="@menu/menu_main" />
```

This defines a bottom navigation bar that uses menu_main.xml for its items.

You start by defining the bottom navigation bar using:

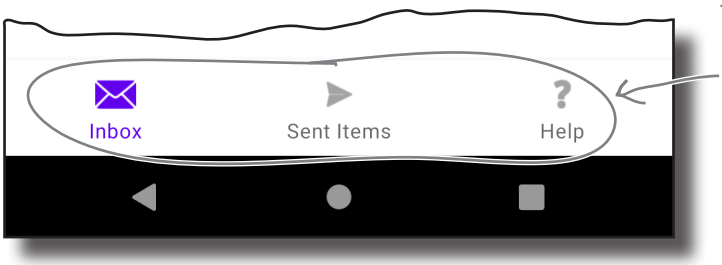
```
<com.google.android.material.bottomnavigation.BottomNavigationView
    ... />
```

where `com.google.android.material.bottomnavigation.BottomNavigationView` is the fully qualified path of the `BottomNavigationView` class: the class that defines the bar. You then use extra attributes to give it an ID, and specify its appearance.

Unlike a toolbar, you don't need to write Kotlin code to add menu items to a bottom navigation bar. You simply specify which menu resource file to add to the bar using the `app:menu` attribute like this:

```
app:menu="@menu/menu_main"
```

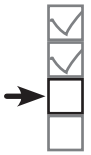
The above code attaches the *menu_main.xml* menu resource file to the bottom navigation bar, and its items are added to the bar at runtime like this:




These are the three items we added to menu_main.xml. The bottom navigation bar knows to include items from this menu because we set the value of the app:menu attribute to "@menu/menu_main".

Now that you've seen what bottom navigation bar code looks like, let's add one to `MainActivity`'s layout.

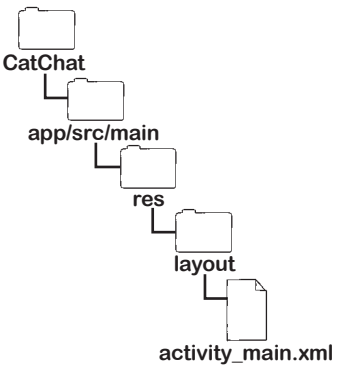
# The full code for activity_main.xml



Toolbar
Toolbar navigation
**Bottom bar**
Navigation drawer

Here's the full code for *activity_main.xml*: update your code to include the changes below (in bold):

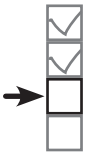
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"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

    <com.google.android.material.appbar.MaterialToolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        style="@style/Widget.MaterialComponents.Toolbar.Primary" />

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/nav_host_fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="match_parent"
        android:layout_height="~~"match_parent~~"0dp"
        android:layout_weight="1"
        app:defaultNavHost="true"
        app:navGraph="@navigation/nav_graph" />

    <com.google.android.material.bottomnavigation.BottomNavigationView
        android:id="@+id/bottom_nav"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:menu="@menu/menu_main" />
</LinearLayout>
```




We're adding a layout_weight attribute to the navigation host to display the navigation bar at the bottom of the screen.

Add the bottom navigation bar to the layout.

Now that we've added the bottom navigation bar to `MainActivity`'s layout, let's get it to navigate between destinations.



Toolbar
Toolbar navigation
**Bottom bar**
Navigation drawer

# Link the bottom navigation bar to the navigation controller

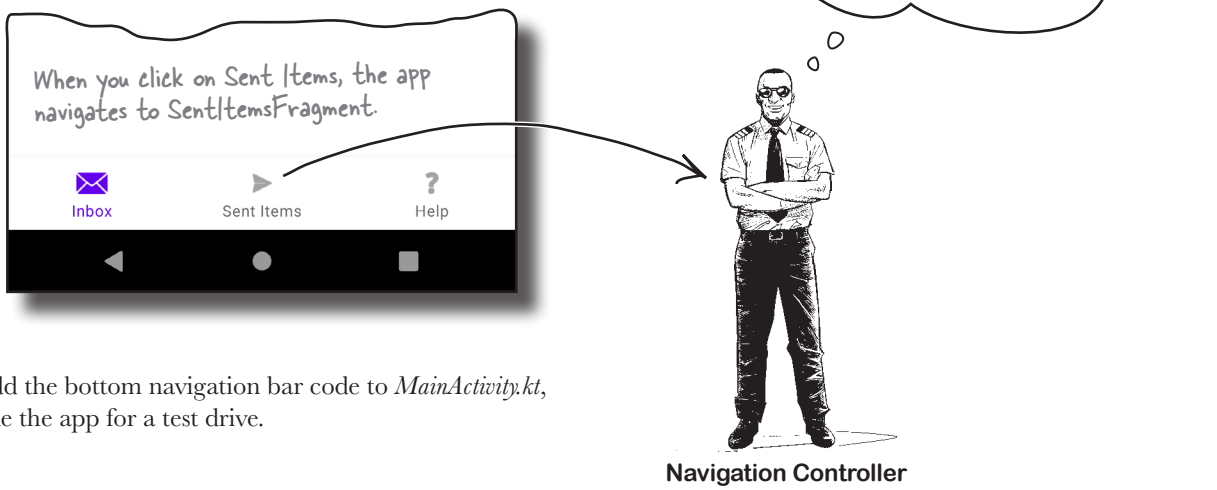
The code to make a bottom navigation bar navigate between destinations is simpler than the code needed to implement toolbar navigation. All you need to do is get a reference to the `BottomNavigationView`, which defines the navigation bar, and call its `setupWithNavController()` method.

Here's what the code looks like:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    ...
    val navHostFragment = supportFragmentManager
        .findFragmentById(R.id.nav_host_fragment) as NavHostFragment
    val navController = navHostFragment.navController
    val bottomNavView = findViewById<BottomNavigationView>(R.id.bottom_nav)
    bottomNavView.setupWithNavController(navController)
}
```

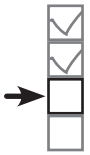
This is all the activity code you need to implement a bottom navigation bar. It links the bar to the navigation controller.

Each time you click on an item in the navigation bar, the navigation controller fetches its ID and looks for a destination with a matching ID in the navigation graph. It then passes this destination to the navigation host so that it's displayed on the device screen.




Let's add the bottom navigation bar code to *MainActivity.kt*, and take the app for a test drive.

# The updated code for MainActivity.kt



Toolbar
Toolbar navigation
**Bottom bar**
Navigation drawer

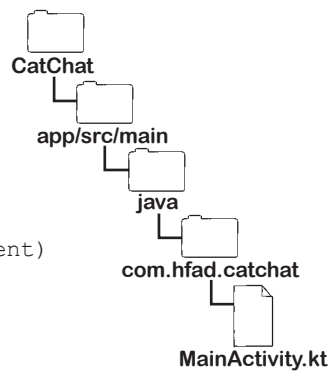
Here's the updated code for *MainActivity.kt*; update the code to include the changes below (in bold):

```
package com.hfad.catchat

...
import com.google.android.material.bottomnavigation.BottomNavigationView

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val toolbar = findViewById<MaterialToolbar>(R.id.toolbar)
        setSupportActionBar(toolbar)
        val navHostFragment = supportFragmentManager
            .findFragmentById(R.id.nav_host_fragment) as NavHostFragment
        val navController = navHostFragment.navController
        val builder = AppBarConfiguration.Builder(navController.graph)
        val appBarConfiguration = builder.build()
        toolbar.setupWithNavController(navController, appBarConfiguration)
        val bottomNavView = findViewById<BottomNavigationView>(R.id.bottom_nav)
        bottomNavView.setupWithNavController(navController)
    }

    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        menuInflater.inflate(R.menu.menu_toolbar, menu)
        return super.onCreateOptionsMenu(menu)
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        val navController = findNavController(R.id.nav_host_fragment)
        return item.onNavDestinationSelected(navController)
                || super.onOptionsItemSelected(item)
    }
}
```

Import this class.

Link the bottom navigation bar to the navigation controller.

These two methods are only required for the toolbar menu. They're not needed for the bottom navigation bar.



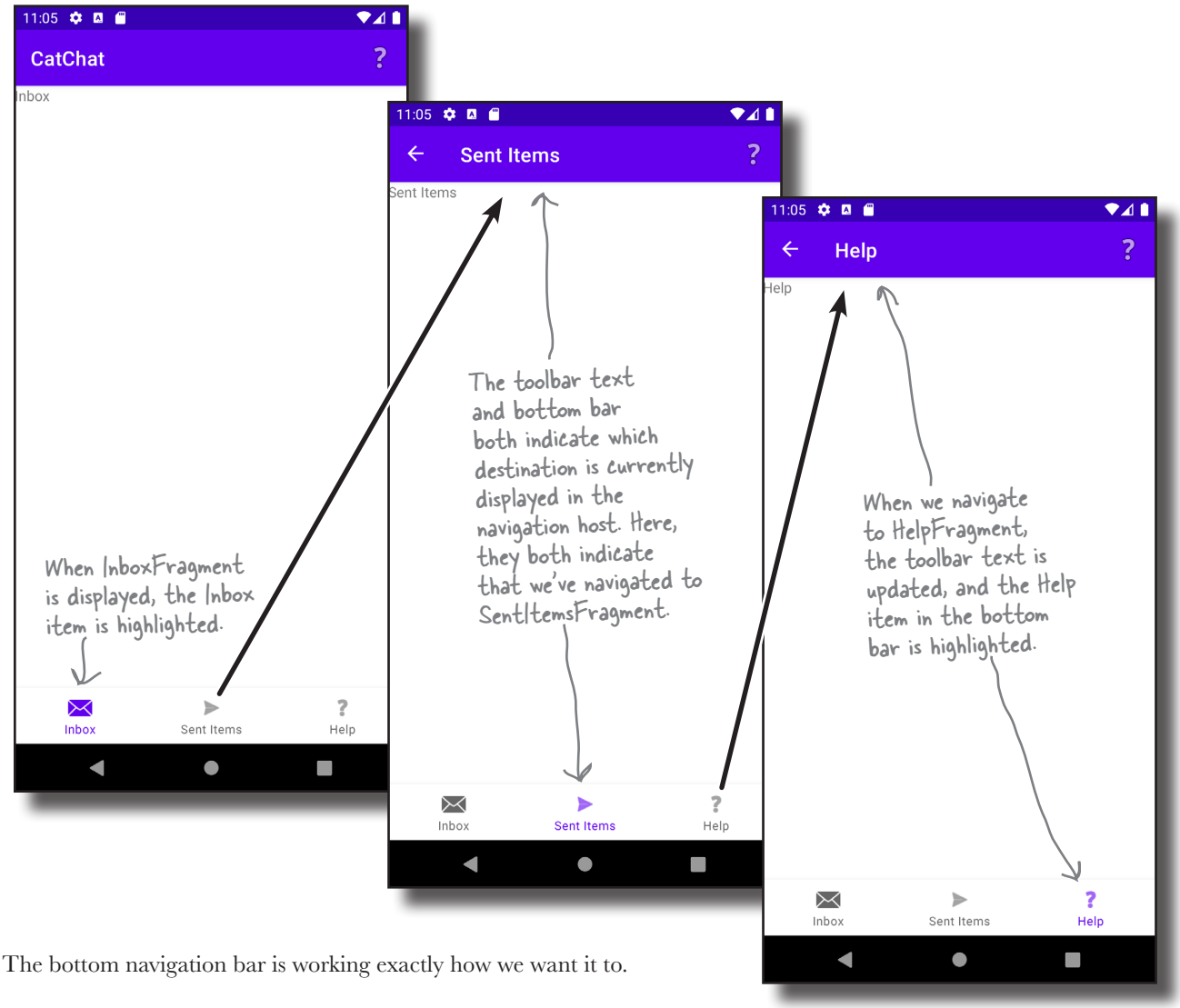

Let's take the app for a test drive.

# Test Drive

- Toolbar
- Toolbar navigation
- **Bottom bar**
- Navigation drawer

When we run the CatChat app, `MainActivity` is launched, and `InboxFragment` is displayed in `MainActivity`'s layout. A bottom navigtion bar is displayed that contains three items.

When we click on each item in the bottom navigation bar, the app navigates to the appropriate screen. When we click on Sent Items, for example, `SentItemsFragment` is displayed in `MainActivity`, and when we click on the Help item, `HelpFragment` is displayed instead.




The bottom navigation bar is working exactly how we want it to.

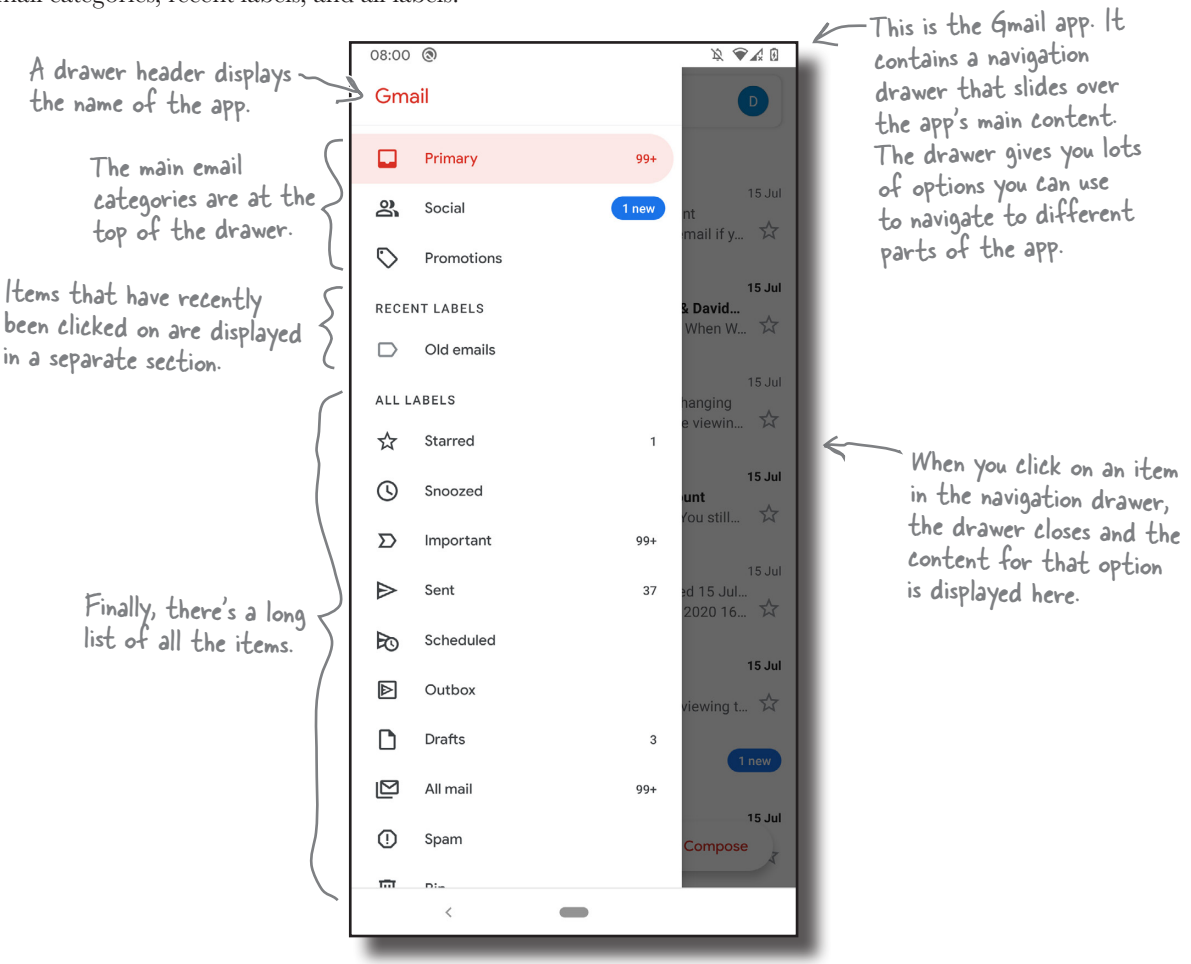
- Toolbar
- Toolbar navigation
- Bottom bar
- **Navigation drawer**

# A navigation drawer lets you display many navigation items

As you've learned, a bottom navigation bar is a good option if you have a small number of navigation items because it allows you to display up to five such items. But what if you have a larger number of menu items?

If you want users to be able to navigate through a large number of options, a better choice might be a **navigation drawer**. This is a slide-out, scrollable panel that contains links to other parts of the app that you can optionally group into different sections.

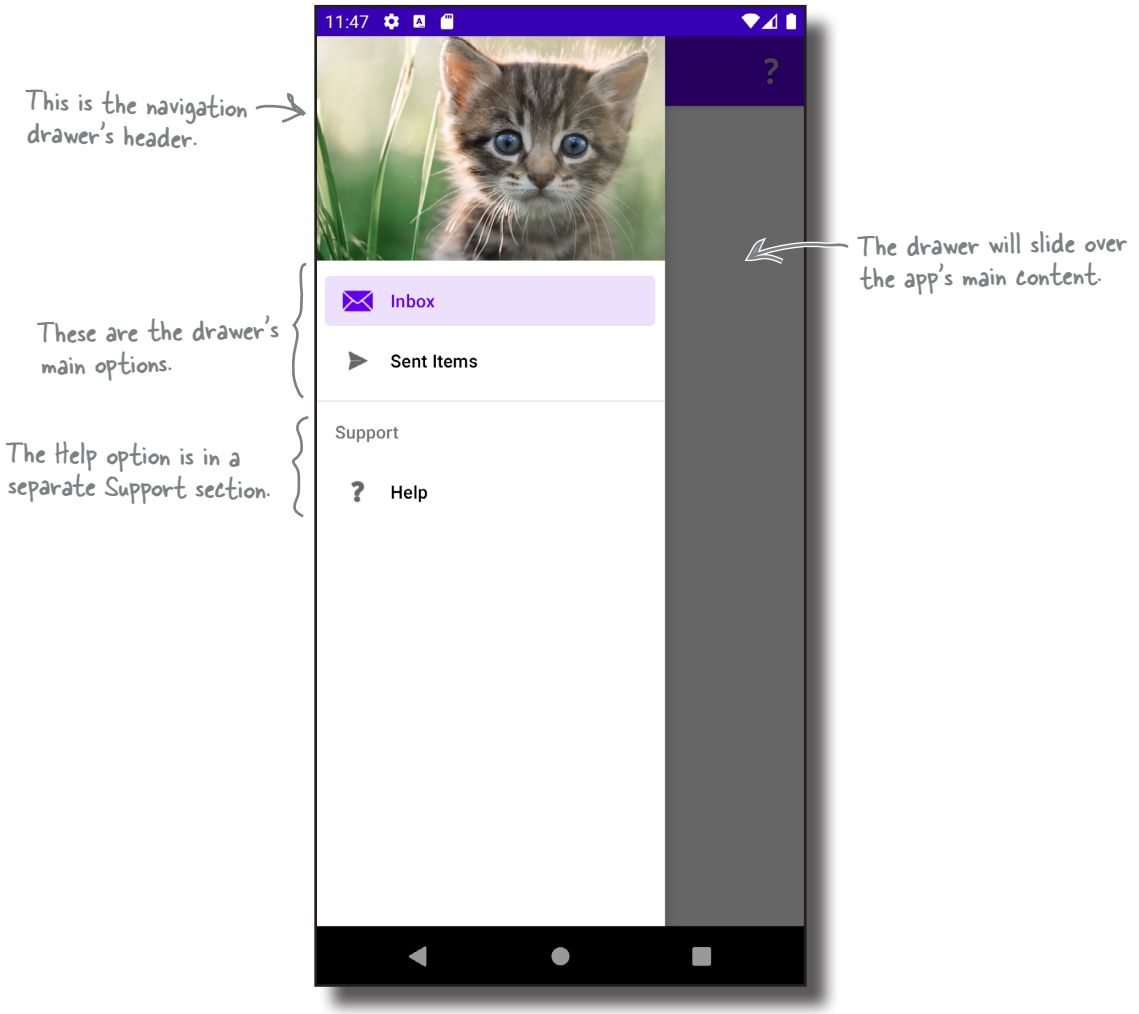
Navigation drawers are widely used across Android apps. The Gmail app, for example, uses a navigation drawer that lets you navigate to different screens in the app, and contains sections such as email categories, recent labels, and all labels:



- Toolbar
- Toolbar navigation
- Bottom bar
- **Navigation drawer**

# Let's replace the bottom navigation bar with a navigation drawer

We're going to replace the bottom navigation bar we added to the CatChat app with a navigation drawer. The navigation drawer will contain a header image and a set of options. The main options will let the user navigate to `InboxFragment` and `SentItemsFragment`, and we'll put a `HelpFragment` item in a separate section labeled Support. Here's what the drawer will look like:




The navigation drawer is composed of several different components. We'll go through these on the next page.

- Toolbar
- Toolbar navigation
- Bottom bar
- **Navigation drawer**

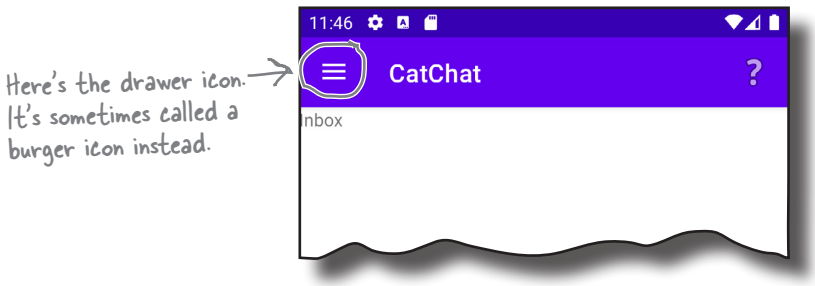
# Navigation drawers deconstructed

You implement a navigation drawer by adding a **drawer layout** to the root of your activity's layout. The drawer layout contains two views:

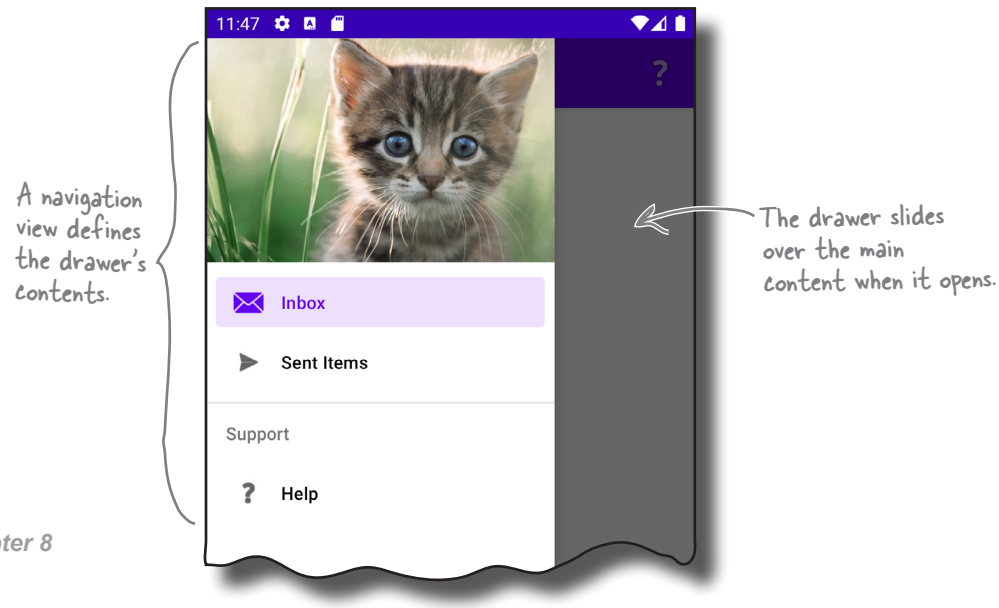
1. **A view for the screen's main content**
   This is usually a layout containing a toolbar and a navigation host, which you use to display fragments.

2. **A navigation view for the drawer's contents**
   A navigation view is a type of frame layout that's used to display a navigation menu. In this app, it will display a drawer header too.

When the drawer is closed, the drawer layout looks just like a normal activity, except that the toolbar includes a drawer icon that's used to open the drawer:




When you open the drawer, the navigation view slides over the activity's main content to display the drawer's contents. When you click on an item, it uses the Navigation component to display the relevant destination, and the drawer closes:



- Toolbar
- Toolbar navigation
- Bottom bar
- **Navigation drawer**

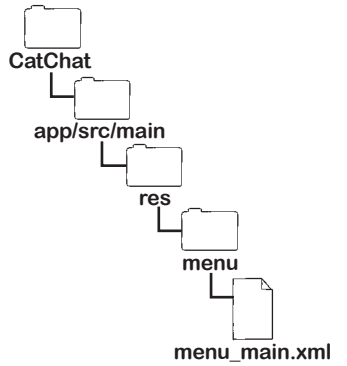
# The drawer gets its items from a menu

Just like a toolbar and bottom navigation bar, a navigation drawer gets the items it needs to display from a menu resource file.

Instead of creating a new menu resource file for the drawer, we're going to reuse *menu_main.xml*: the file that we used for the bottom navigation bar. Here's a reminder of the current code in *menu_main.xml*:

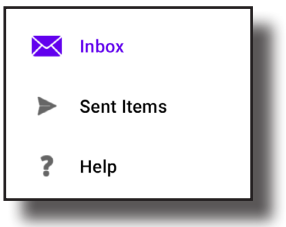
This is the same menu file that we used for the bottom navigation bar.

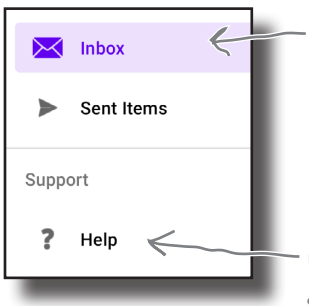
```
<?xml version="1.0" encoding="utf-8"?>
<menu
    xmlns:android="http://schemas.android.com/apk/res/android">

    <item
        android:id="@+id/inboxFragment"
        android:icon="@android:drawable/ic_dialog_email"
        android:title="Inbox" />
    <item
        android:id="@+id/sentItemsFragment"
        android:icon="@android:drawable/ic_menu_send"
        android:title="Sent Items" />
    <item
        android:id="@+id/helpFragment"
        android:icon="@android:drawable/ic_menu_help"
        android:title="Help" />
</menu>
```




The above menu resource file, when added to the navigation drawer, produces a list of items with an icon against each one. We're going to tweak the menu to add an extra highlight to the item the user selects, and split the menu into sections:

The above code produces this drawer menu.




We'll add an extra highlight to the item the user selects.

We'll put the Help item in a separate Support section.

Let's find out how.

Toolbar
Toolbar navigation
Bottom bar
**Navigation drawer**

# Add the support section...

We'll first add the Support heading, which is done by defining a new item. Because it's just a heading, we only need to give it a title: it doesn't have an icon, and no ID is required as we won't use it to navigate anywhere.

Here's the code to add the Support heading:

```
...
    <item android:title="Support">
    </item>
...
```

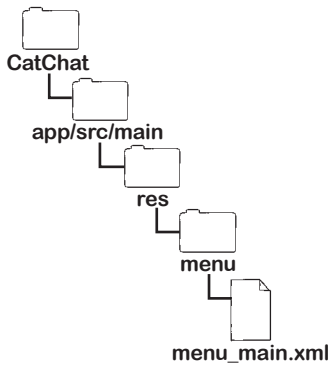
This adds a Support heading to the navigation drawer. We'll update the menu_main.xml code a couple pages ahead.

## ...as a separate submenu

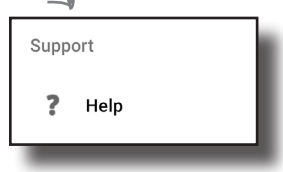
We want the Help item to appear under the Support heading so that it forms a separate section. To do this, we'll define a submenu inside the Support item, specified by a `<menu>` element. We'll add the Help item to this submenu.

Here's the code to add a submenu, which includes a Help item; we'll update *menu_main.xml* with this code a couple pages ahead:

```
...
    <item android:title="Support">
        <menu>
            <item
                android:id="@+id/helpFragment"
                android:icon="@android:drawable/ic_menu_help"
                android:title="Help" />
        </menu>
    </item>
...
```




The above code displays the Help item in a Support section.




The above submenu just contains a single item. If you wanted it to contain multiple items, you'd simply include each item in the submenu.

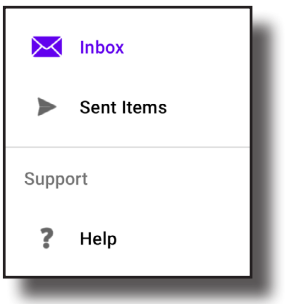
That's all the code we need to add a Support section to the drawer's menu. Next, let's find out how to add an extra highlight to the item the user selects.

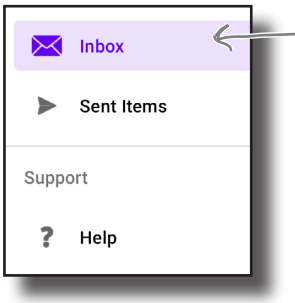
Toolbar
Toolbar navigation
Bottom bar
**Navigation drawer**

# Highlight the selected item with groups

The current menu produces a navigation drawer that changes the text color of the current selected item. You can make it extra clear to the user which item has been selected by adding an extra highlight like this:

This is what the menu looks like without the extra highlight.




This is the extra highlight. It makes it more obvious which item has been selected.

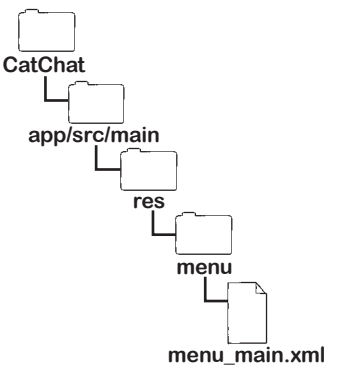
You add this extra highlight by placing items in a group using the **<group>** element. You then use an attribute named `android:checkableBehavior` to specify the group's behavior when an item is selected.

Here's the code to do this (we'll update *menu_main.xml* on the next page):

This option means that a single item in the group will be highlighted when the user clicks on it (the item the user selects).

```
...
    <group android:checkableBehavior="single">
        <item
            android:id="@+id/inboxFragment"
            android:icon="@android:drawable/ic_dialog_email"
            android:title="Inbox" />
        <item
            android:id="@+id/sentItemsFragment"
            android:icon="@android:drawable/ic_menu_send"
            android:title="Sent Items" />
    </group>
...
```

These items are included in the group. When one of these is selected, it will be highlighted in the navigation drawer.




As you can see, the above code sets the `android:checkableBehavior` attribute to `"single"`. This option means that only a single item in the group at a time will be highlighted—the option the user selects.

That's everything we need to make the menu look and behave the way we want when it's used by a navigation drawer. We'll show you the full code on the next page.

- Toolbar
- Toolbar navigation
- Bottom bar
- **Navigation drawer**

# The full code for menu_main.xml

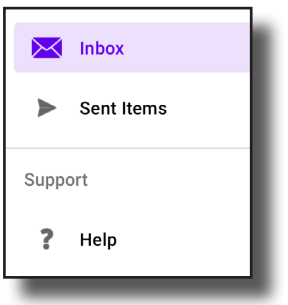
We're going to update the navigation drawer's menu so that the Help menu is shown in a Support section. We'll also use groups to add an extra highlight to the item the user selects.

Here's the complete code for *menu_main.xml*; update its code to include all the changes (in bold):

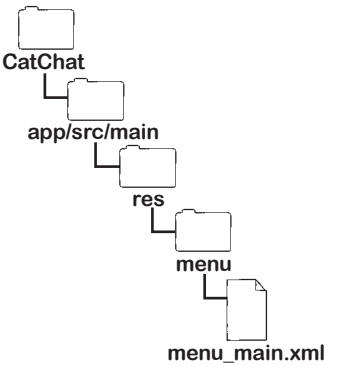
```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <group android:checkableBehavior="single">
        <item
            android:id="@+id/inboxFragment"
            android:icon="@android:drawable/ic_dialog_email"
            android:title="Inbox" />
        <item
            android:id="@+id/sentItemsFragment"
            android:icon="@android:drawable/ic_menu_send"
            android:title="Sent Items" />
    </group>

    <item android:title="Support">
        <menu>
            <group android:checkableBehavior="single">
                <item
                    android:id="@+id/helpFragment"
                    android:icon="@android:drawable/ic_menu_help"
                    android:title="Help" />
            </group>
        </menu>
    </item>
</menu>
```

Put the first two items in a group.

Add a Support section.




We've put the Help item in a group so that when it's selected, it will get an extra highlight.




That's the menu sorted. Next, we'll create the navigation drawer's header.

Toolbar
Toolbar navigation
Bottom bar
**Navigation drawer**

# Create the navigation drawer's header

The navigation drawer's header is a simple layout that we'll add to a new layout file named *nav_header.xml*.

Create this file now by selecting the *app/src/main/res/layout* folder in Android Studio, and choosing File→New→Layout Resource File. When prompted, name the layout "nav_header", and if prompted, choose a resource type of Layout.

## Add the image file...

The layout is composed of a single image, which needs to be added to the *app/src/main/res/drawable* folder. Android Studio probably created this folder for you when you created the project. If it's not there, create it by selecting the *app/src/main/res* folder, going to the File menu, choose the New option, and then clicking on the option to create a new Android resource directory. When prompted, choose a resource type of Drawable, name it "drawable", and click on OK.

Once you've created the *drawable* folder, download the file *kitten_small.webp* from *tinyurl.com/hfad3*, and add it to the *drawable* folder.

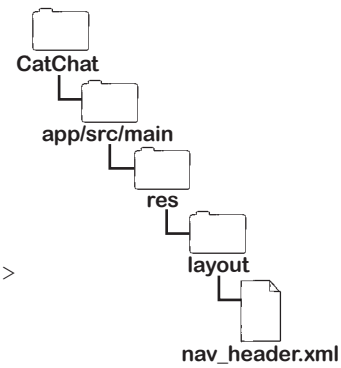
The header displays this image in an ImageView.

## ...and update the nav_header.xml code

We'll add the image to *nav_header.xml* using an `<ImageView>` element. You already know how to use this element, so update the code for *nav_header.xml* to match the code below:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="180dp" >

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:scaleType="centerCrop"
        android:src="@drawable/kitten_small" />
</FrameLayout>
```

The layout only needs to display an image, so we've used an ImageView inside a FrameLayout.




Now that we have a header for the navigation drawer, let's add the drawer itself.

- Toolbar
- Toolbar navigation
- Bottom bar
- **Navigation drawer**

# How to create a navigation drawer

You create a navigation drawer by adding a **`DrawerLayout`** to the activity's layout as its root element. The `DrawerLayout` needs to contain two things: a view or view group for the activity's content as its first element, and a navigation view that defines the drawer's contents as its second.

Typical `DrawerLayout` code looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.drawerlayout.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">
        ...
    </LinearLayout>

    <com.google.android.material.navigation.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        app:headerLayout="@layout/nav_header"
        app:menu="@menu/menu_main"/>
</androidx.drawerlayout.widget.DrawerLayout>
```

The DrawerLayout defines a layout that includes a drawer.

You give it an ID so you can refer to it in your activity code.

The DrawerLayout's first view is a layout for the activity's main content. You see this when the drawer is closed.

A NavigationView is a type of FrameLayout that provides a standard navigation menu.

This attaches the drawer to the start edge of the activity (the left edge for left-to-right languages).

This is the layout for the drawer's header.

This is the menu resource file containing the drawer's items.

There are two key `<NavigationView>` attributes that you use to control the drawer's appearance: **`app:headerLayout`** and **`app:menu`**.

The `app:headerLayout` attribute specifies the layout that should be used for the navigation drawer's header (in this case *nav_header.xml*). This attribute is optional.

The `app:menu` attribute specifies which menu resource file contains the drawer's options (in this case *menu_main.xml*). If you don't include this attribute, your navigation drawer won't include any items.

- Toolbar
- Toolbar navigation
- Bottom bar
- **Navigation drawer**

# The full code for activity_main.xml

We want to replace the `MainActivity`'s bottom navigation bar with a navigation drawer. The code to do this is shown below; update file *activity_main.xml* to include the changes (in bold):

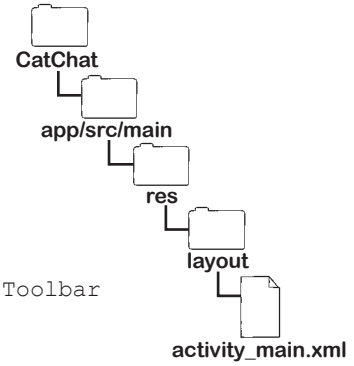
```
<?xml version="1.0" encoding="utf-8"?>
~~<LinearLayout~~
<androidx.drawerlayout.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    ~~android:orientation="vertical"~~
    tools:context=".MainActivity">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <com.google.android.material.appbar.MaterialToolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            style="@style/Widget.MaterialComponents.Toolbar.Primary" />

        <androidx.fragment.app.FragmentContainerView
            android:id="@+id/nav_host_fragment"
            android:name="androidx.navigation.fragment.NavHostFragment"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1"
            app:defaultNavHost="true"
            app:navGraph="@navigation/nav_graph" />
```

Change the layout's root element to a DrawerLayout.

Give the drawer an ID because we need it for the activity code.

Delete this line.

The activity's main content is defined using a linear layout. It includes a toolbar and navigation host as before.



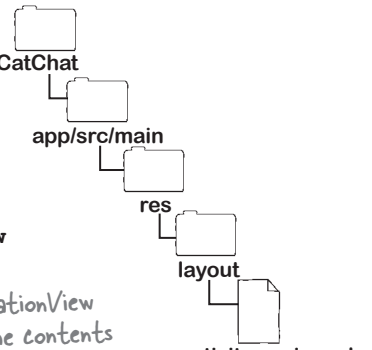

The code continues on the next page.

# activity_main.xml continued

Remove the bottom navigation bar.

```
<com.google.android.material.bottomnavigation.BottomNavigationView
    android:id="@+id/bottom_nav"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:menu="@menu/menu_main" />
</LinearLayout>
```

Close the linear layout.




```
<com.google.android.material.navigation.NavigationView
    android:id="@+id/nav_view"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:layout_gravity="start"
    app:headerLayout="@layout/nav_header"
    app:menu="@menu/menu_main"/>
</androidx.drawerlayout.widget.DrawerLayout>
```

The NavigationView defines the contents of the drawer.

The drawer includes a header layout.

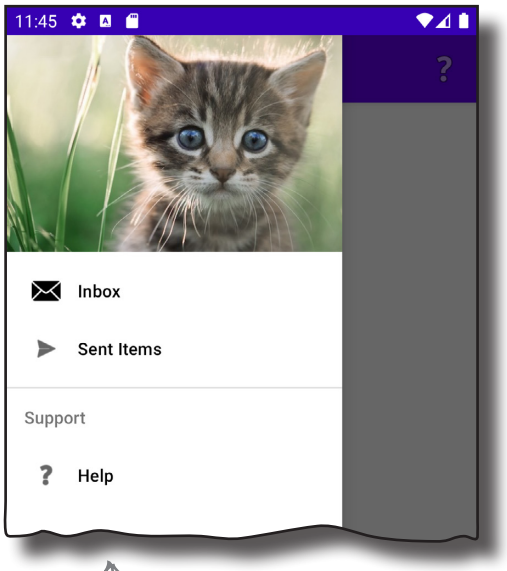
## We've added a navigation drawer to the layout

We've now replaced the bottom navigation bar in `MainActivity`'s layout with a navigation drawer. It displays an image in its header, and includes all the items specified in the menu resource file *menu_main.xml*.

Before we run the app, however, we need to link the navigation drawer to the navigation controller so that clicking on an item navigates to the correct fragment.

We also need to configure the toolbar so that it includes a drawer icon that, when clicked, will open or close the navigation drawer:

We need to add a drawer icon to the toolbar.




If we don't link the navigation drawer to the navigation controller, the items will appear in the navigation drawer, but won't do anything when we click on them.

Both of these changes are made by tweaking the code in *MainActivity.kt*. Let's find out how.

- Toolbar
- Toolbar navigation
- Bottom bar
- **Navigation drawer**

# Configure the toolbar's drawer icon...

Earlier in the chapter, we configured the toolbar to display the current destination's label and include an Up button. We did this by building an `AppBarConfiguration` object, and linking it to the toolbar.

We now want the toolbar to include a drawer icon, and we can add one by adding the navigation drawer to the `AppBarConfiguration`. Here's the code to do this:

Get a reference to the DrawerLayout.

This line means the toolbar will display a drawer icon with which to open the drawer. The drawer icon will be displayed on all screens with no Up button.

```
val drawer = findViewById<DrawerLayout>(R.id.drawer_layout)
val builder = AppBarConfiguration.Builder(navController.graph)
builder.setOpenableLayout(drawer)
val appBarConfiguration = builder.build()
toolbar.setupWithNavController(navController, appBarConfiguration)
```

The above code adds the drawer layout to the `AppBarConfiguration` object. This allows the toolbar to interact with the drawer by including a drawer icon on every screen where there's no Up button.

## ...and link the drawer to the navigation controller

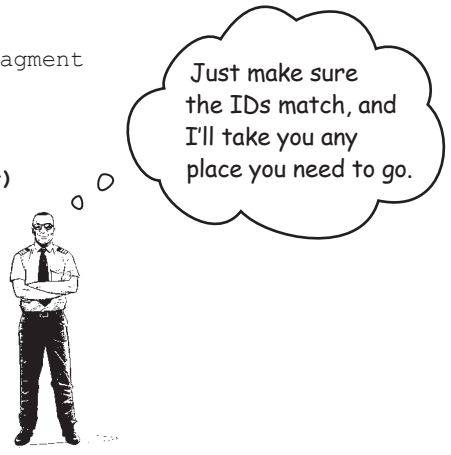
Finally, we need to make the drawer navigate to the correct destination each time the user clicks on one of its items. Just like with the bottom navigation bar, we do this by setting the navigation drawer up with the navigation controller.

Here's the code to do this:

```
val navHostFragment = supportFragmentManager
    .findFragmentById(R.id.nav_host_fragment) as NavHostFragment
val navController = navHostFragment.navController
val navView = findViewById<NavigationView>(R.id.nav_view)
NavigationUI.setupWithNavController(navView, navController)
```

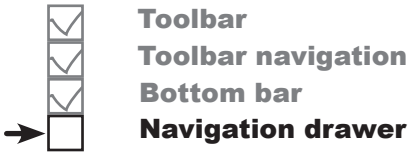
Each time the user clicks on an item in the navigation drawer, the navigation controller gets its ID from the menu resource file, and looks for a matching ID in the navigation graph. It then navigates to the destination with that ID.

That's all the code we need to control the navigation drawer's behavior. Let's add it to *MainActivity.kt* and take the app for a test drive.

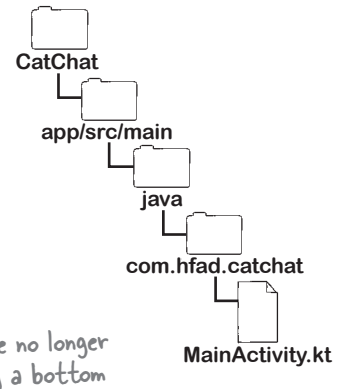



Navigation Controller

# The full code for MainActivity.kt




We need to update `MainActivity` to replace the bottom navigation code with the code for the navigation drawer. Update *MainActivity.kt* with the changes below (in bold):




```
package com.hfad.catchat

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.view.Menu
import android.view.MenuItem
import androidx.navigation.findNavController
import androidx.navigation.fragment.NavHostFragment
import androidx.navigation.ui.AppBarConfiguration
import androidx.navigation.ui.onNavDestinationSelected
import androidx.navigation.ui.setupWithNavController
~~import com.google.android.material.bottomnavigation.BottomNavigationView~~
import androidx.drawerlayout.widget.DrawerLayout
import androidx.navigation.ui.NavigationUI
import com.google.android.material.navigation.NavigationView
import com.google.android.material.appbar.MaterialToolbar

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val toolbar = findViewById<MaterialToolbar>(R.id.toolbar)
        setSupportActionBar(toolbar)
        val navHostFragment = supportFragmentManager
            .findFragmentById(R.id.nav_host_fragment) as NavHostFragment
        val navController = navHostFragment.navController
        val drawer = findViewById<DrawerLayout>(R.id.drawer_layout)
        val builder = AppBarConfiguration.Builder(navController.graph)
        builder.setOpenableLayout(drawer)
        val appBarConfiguration = builder.build()
        toolbar.setupWithNavController(navController, appBarConfiguration)
        ~~val bottomNavView = findViewById<BottomNavigationView>(R.id.bottom_nav)~~
        ~~bottomNavView.setupWithNavController(navController)~~
        val navView = findViewById<NavigationView>(R.id.nav_view)
        NavigationUI.setupWithNavController(navView, navController)
    }
```

We're no longer using a bottom navigation bar, so delete this line.

You need to import these extra classes as they're used by the navigation drawer code.

Get a reference to the drawer.

Add the drawer to the AppBarConfiguration.

Delete these lines as we're no longer using a bottom navigation bar.

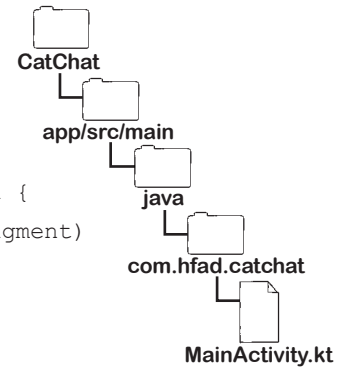
Enable navigation when an item's clicked by linking the drawer to the navigation controller.

The code continues on the next page.

# MainActivity.kt continued

```
    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        menuInflater.inflate(R.menu.menu_toolbar, menu)
        return super.onCreateOptionsMenu(menu)
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        val navController = findNavController(R.id.nav_host_fragment)
        return item.onNavDestinationSelected(navController)
                || super.onOptionsItemSelected(item)
    }
}
```

These two methods are only required for the toolbar. They're not needed for the navigation drawer.




That's all the code we need for the navigation drawer.
Let's take it for a test drive.

## there are no Dumb Questions

**Q: Why did we remove the bottom navigation bar? Can't an app have both a bottom navigation bar and a navigation drawer?**

A: Apps can have bottom navigation bars and navigation drawers. We decided to replace the bottom navigation bar with a navigation drawer in order to make it clearer what code was needed just for the navigation drawer.

**Q: I'd heard that it took lots of activity code to create a navigation drawer. Did I hear wrong?**

A: In the old days, adding a navigation drawer to your app took lots of effort.

Including a navigation drawer is now much easier thanks to the Navigation component. It handles most of the hard work for you, making your activity code much simpler.

**Q: How does the navigation drawer know which fragment to navigate to?**

A: The activity code includes a call to the `setupWithNavController()` method, and this links the drawer's navigation view to the navigation controller.

When an item in the drawer is clicked, the ID of this item is passed to the navigation controller. The navigation controller looks up this ID in the navigation graph, and navigates to the destination with a matching ID.

**Q: Can you remind me what a navigation view is?**

A: It's a type of frame layout that is used to display a standard navigation menu. It's usually used inside a navigation drawer.

**Q: I tried clicking on one of the items, but the drawer didn't navigate to the right fragment. Why not?**

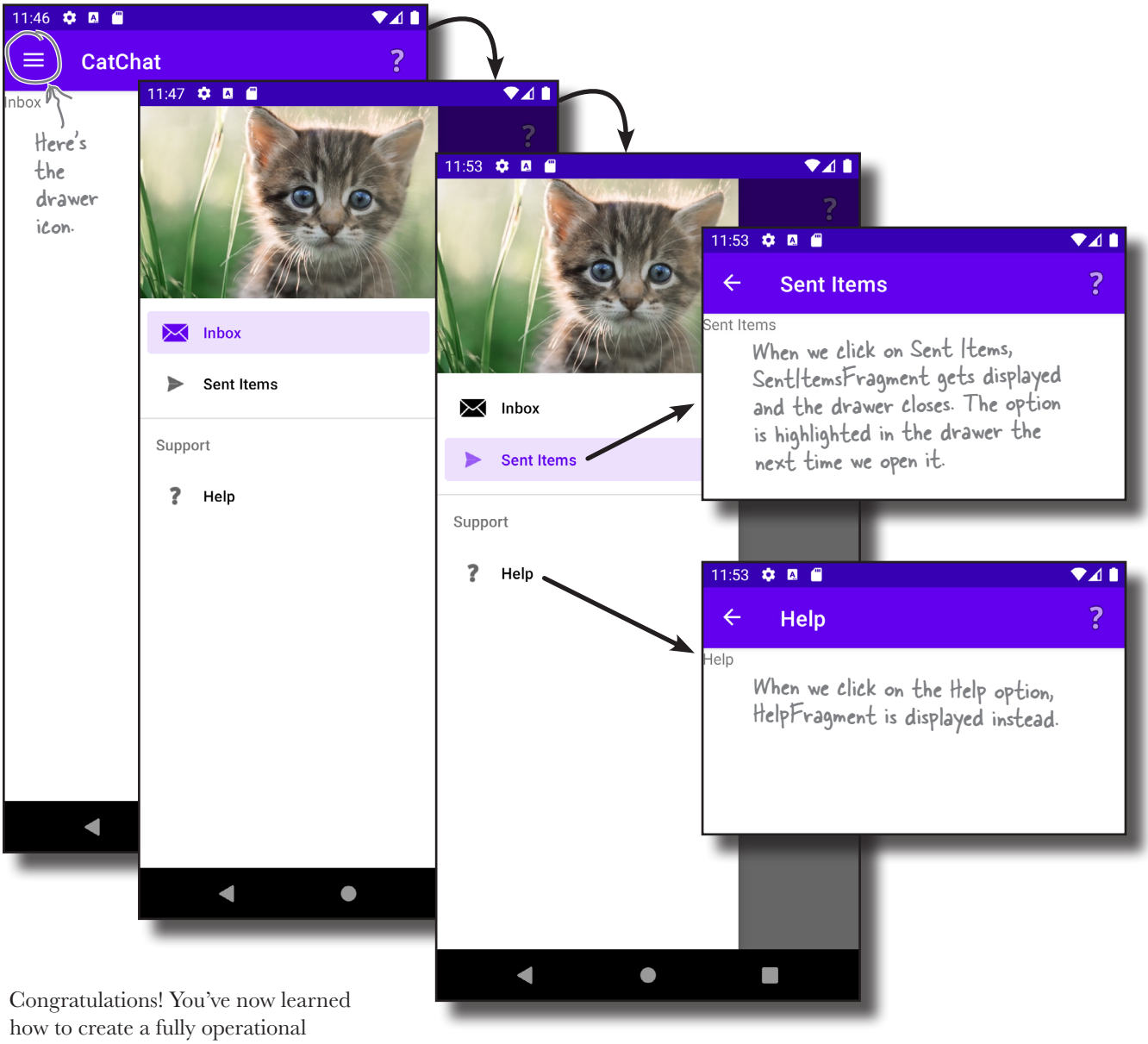
A: First, make sure that your activity code calls the `setupWithNavController()` method. This links the drawer to the navigation controller, so if it's missing, your drawer won't be able to use the Navigation component.

Next, make sure that you have matching IDs in the drawer's menu resource file and the navigation graph. If the navigation controller can't find a matching ID, it won't be able to use it to navigate.

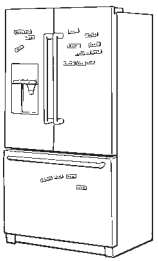
# Test Drive

- [x] Toolbar
- [x] Toolbar navigation
- [x] Bottom bar
- [x] Navigation drawer

When we run the app, a drawer icon is displayed in the toolbar. Clicking on this icon opens the navigation drawer. When we click on one of the items in the drawer, the fragment for that option is displayed in `MainActivity` and the drawer closes.




Congratulations! You've now learned how to create a fully operational navigation drawer.



# Layout Magnets

Somebody used fridge magnets to write the layout code for a navigation drawer, but some of them fell off overnight. Can you put the code back together again?

The activity uses a linear layout for its main content (we've left out most of the code for this). The drawer needs to display a header, defined in a layout file named *header.xml*. Its menu items are specified in the menu resource file *menu_drawer.xml*.

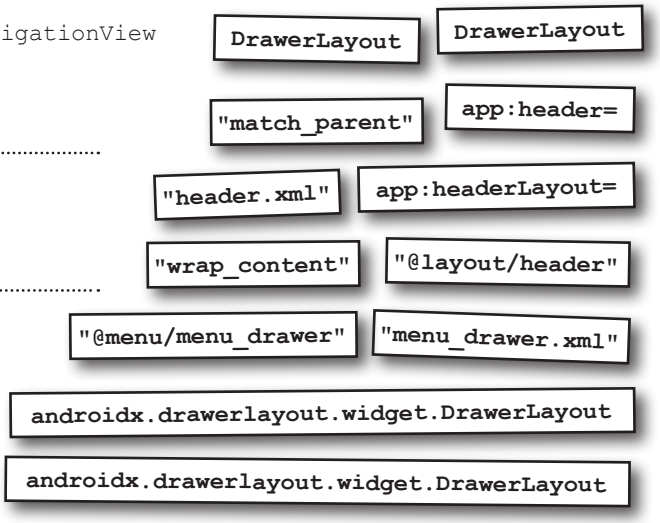
```
<?xml version="1.0" encoding="utf-8"?>

<  .....

    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <com.google.android.material.navigation.NavigationView
        android:id="@+id/nav_view"

        android:layout_width=  .....
        android:layout_height="match_parent"
        android:layout_gravity="start"

        .....

        app:menu= ...../>

</  .....>
```

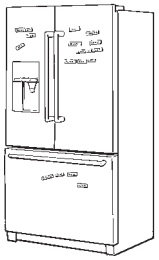
We've put blank lines where all of the missing magnets need to go except for the LinearLayout. You need to decide where this magnet should be placed.

We've left out most of the LinearLayout code because the fridge door wasn't big enough.

```
<LinearLayout ...>
    ...
</LinearLayout>
```

You won't need to use all the magnets.





# Layout Magnets Solution

Somebody used fridge magnets to write the layout code for a navigation drawer, but some of them fell off overnight. Can you put the code back together again?

The activity uses a linear layout for its main content (we've left out most of the code for this). The drawer needs to display a header, defined in a layout file named *header.xml*. Its menu items are specified in the menu resource file *menu_drawer.xml*.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.drawerlayout.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <LinearLayout ...>
        ...
    </LinearLayout>

    <com.google.android.material.navigation.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        app:headerLayout="@layout/header"
        app:menu="@menu/menu_drawer" />

</androidx.drawerlayout.widget.DrawerLayout>
```

You need to specify the fully qualified class name of the DrawerLayout.

The activity's main content needs to be the DrawerLayout's first element.

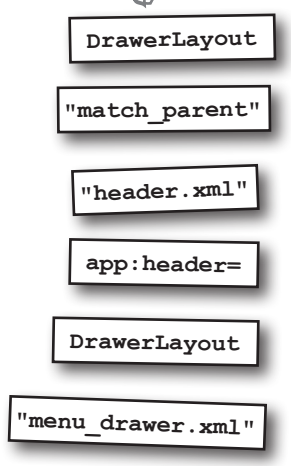
This makes the drawer just wide enough for its contents.

Specify the drawer's header.

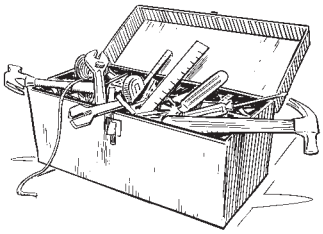
Use menu_drawer.xml for the drawer's menu.

The closing tag for the DrawerLayout.

You didn't need to use these magnets.



# Your Android Toolbox



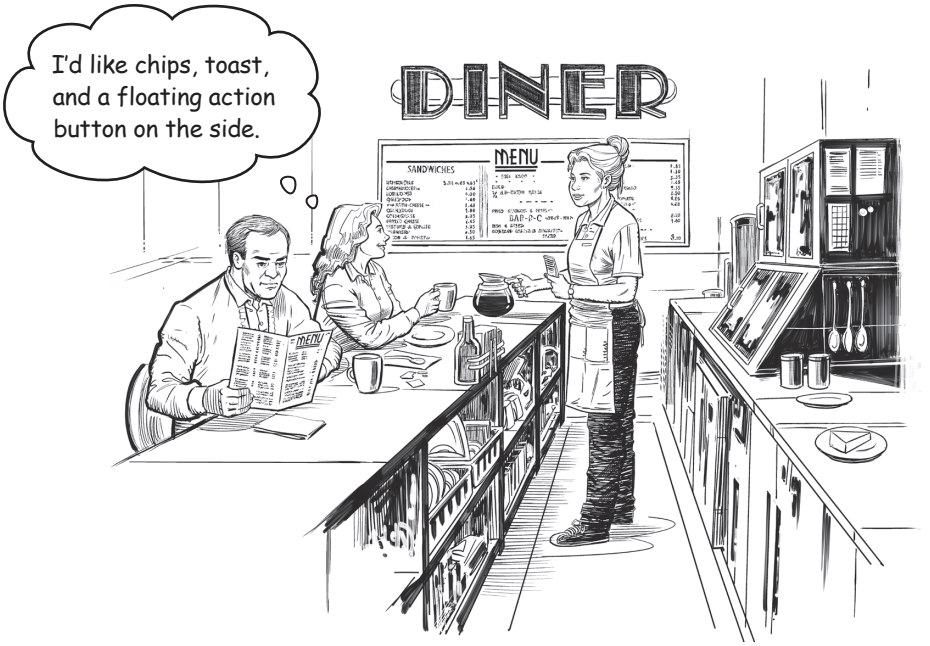
**You've got Chapter 8 under your belt and now you've added navigation UI to your toolbox.**

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

## Bullet Points

- The default app bar is added using a theme.
- The `android:theme` attribute in *AndroidManifest.xml* specifies which theme to apply.
- You define styles as `<style>` elements in one or more resource files.
- You can replace the default app bar with a toolbar that includes the latest Android features.
- A Material toolbar is a toolbar that works well with Material themes.
- A menu resource file adds items to toolbars, bottom navigation bars, and navigation drawers.
- Make sure there are matching item and destination IDs in the menu resource file and navigation graph. The Navigation component depends on matching IDs for navigation.
- Add items to a toolbar using `onCreateOptionsMenu()`.
- Implement toolbar navigation using `onOptionsItemSelected()`.
- Use an `AppBarConfiguration` object to configure the toolbar to work with the Navigation component.
- A bottom navigation bar can hold up to five items.
- A navigation drawer can hold many items, grouped into sections.
- Create a navigation drawer by adding a drawer layout to your activity's layout. The drawer layout's first element needs to be a view that defines the activity's main content. Its second element defines the contents of the drawer.

# 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**. Read on to find out more.

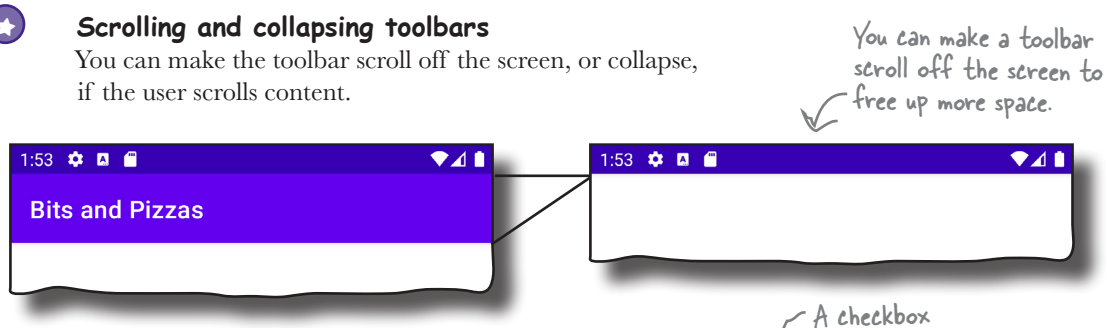
# Material is used throughout Androidville

In the previous chapter you learned how to use toolbars, bottom navigation bars, and navigation drawers to help users navigate your app, and styled them using a theme from the Material library. As you may recall, Material is a design system that helps you build apps with a consistent look and feel across all screens.

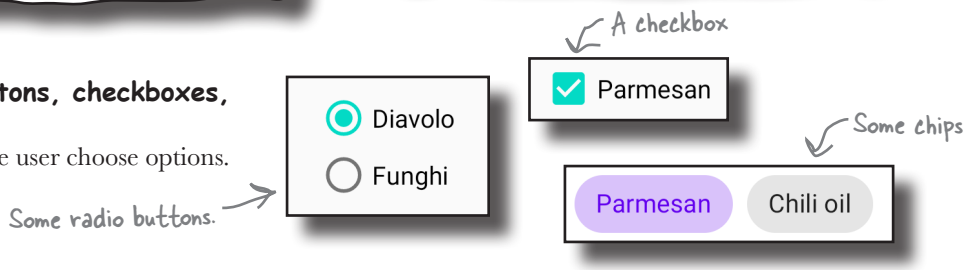
Material isn't just limited to toolbars, navigation drawers, and bottom navigation bars; it styles every view in your app, from buttons to text views.

Here are some more examples of components and features that use Material:

**Scrolling and collapsing toolbars**
You can make the toolbar scroll off the screen, or collapse, if the user scrolls content.

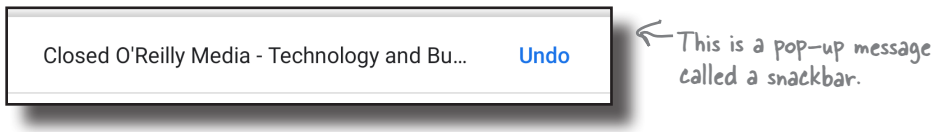



**Radio buttons, checkboxes, and chips**
These let the user choose options.




**Floating action buttons (FABs)**
FABs are special buttons that float above the main screen.

**Snackbars**
These are pop-up messages you can interact with.

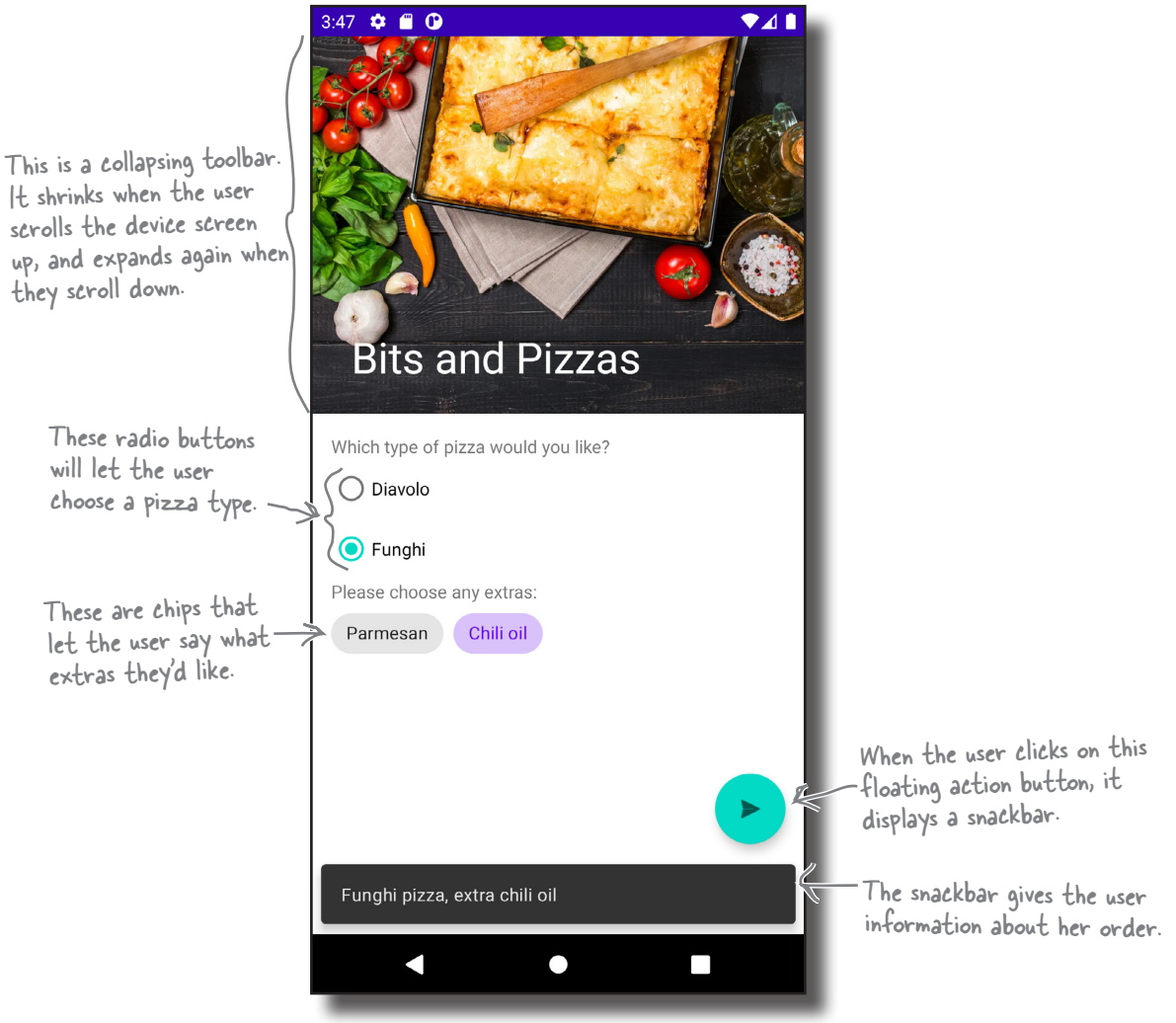



We're going to show you how to use these views and features by building a new app.

# The Bits and Pizzas app

We're going to build a new app named Bits and Pizzas. We're going to focus on its Create Order screen, which lets the user place an order for a pizza.

The screen looks like this:

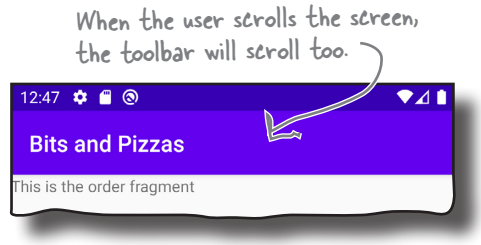



The app is comprised of an activity—`MainActivity`—that displays a fragment named `OrderFragment`. The fragment defines the Create Order screen's appearance and functionality.

Let's go through the steps we need to take to build the app.

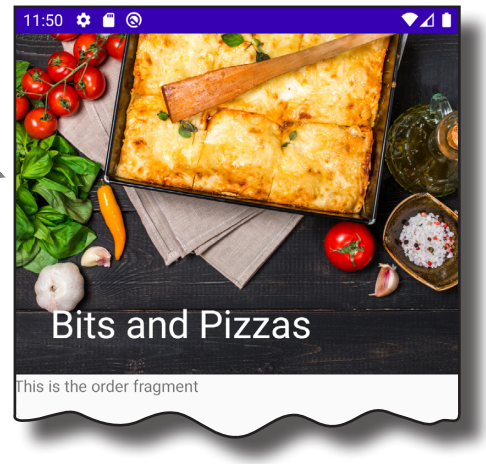
# Here's what we'll do

We'll build the app by going through the following steps:

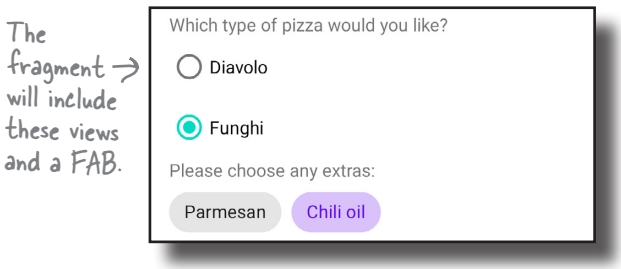



1. **Add a toolbar that can scroll.**
We'll create `OrderFragment`, and add a toolbar to its layout that will scroll off the screen when the user scrolls the screen up, and reappear when they scroll down.

2. **Implement a collapsing toolbar.**
Once we've made the toolbar scroll, we'll add an image to it, and get it to collapse and expand when the user scrolls the screen.




3. **Add views.**
The user needs to be able to place a pizza order. We'll enable this by adding radio buttons, chips, and a floating action button to `OrderFragment`'s layout.




4. **Make the FAB respond to clicks.**
When the user clicks the FAB, we'll display a pop-up message that gives details of her order.

# Create the Bits and Pizzas project

We're going to create a new project for the Bits and Pizzas app, so you need to create one using the same steps from previous chapters. Choose the Empty Activity option, enter a name of "Bits and Pizzas" and a package name of "com.hfad.bitsandpizzas", and accept the default save location. Make sure the language is set to Kotlin and the minimum SDK is API 21 so it will run on most Android devices.

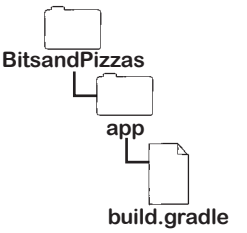
## Add a Material library dependency to the app build.gradle file

In this chapter, we're going to use themes, views, and features from the Material library, so we need to make sure that the app's *build.gradle* file includes it as a dependency.

Open the file *BitsandPizzas/app/build.gradle*, and make sure that its `dependencies` section includes the following line (in bold):

```
dependencies {
    ...
    implementation 'com.google.android.material:material:1.4.0'
    ...
}
```

We're using this version of the Material library.




It's likely that Android Studio has already added this dependency to the file for you. If not, you'll need to add it yourself, and click on the Sync Now option that appears at the top of the code editor to sync the change with the rest of your project.

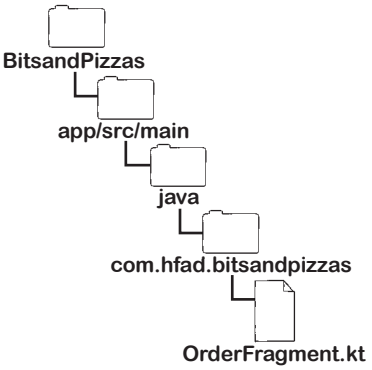
Now that you've made sure that the app includes the Material library, let's go and create `OrderFragment`.

**Scrolling toolbar**
Collapsing toolbar
Views
Respond to clicks

# Create OrderFragment

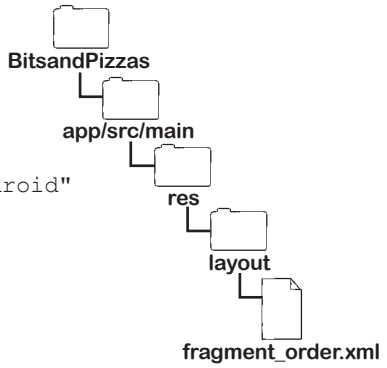
`OrderFragment` is the Bits and Pizzas app's main screen, and it's what the user will use to place a pizza order.

To create the fragment, highlight the *com.hfad.bitsandpizzas* package in the *app/src/main/java* folder, then go to File→New→Fragment→Fragment (Blank). Name the fragment "OrderFragment" and its layout "fragment_order", and make sure the language is set to Kotlin. Then update the code for *OrderFragment.kt* to match the code below:




```
package com.hfad.bitsandpizzas

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup

class OrderFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.fragment_order, container, false)
    }
}
```

Then update the code in its layout file *fragment_order.xml* so that it includes a frame layout like this:



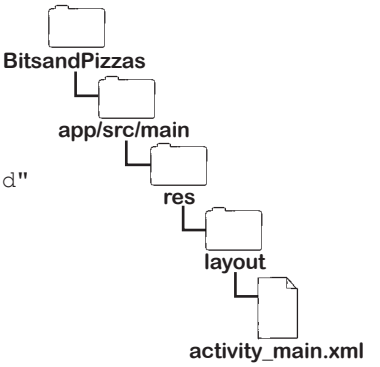

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".OrderFragment">
</FrameLayout>
```

We'll update `OrderFragment` through the rest of this chapter. First, let's display it in `MainActivity`'s layout.

# Display OrderFragment in MainActivity's layout

We'll add `OrderFragment` to `MainActivity`'s layout using a `FragmentContainerView`, specifying the name of the fragment.

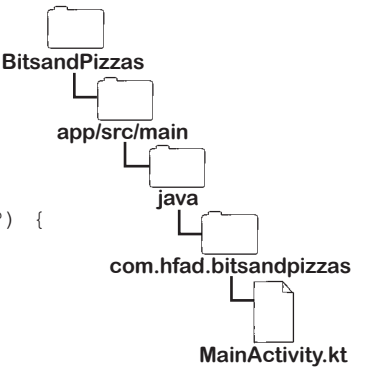
Here's the code to do this; open *activity_main.xml* and update its code to include the changes below:




```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/fragment_container_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:name="com.hfad.bitsandpizzas.OrderFragment"
    tools:context=".MainActivity" />
```

We're not using navigation in this chapter, so MainActivity's layout just needs to display OrderFragment.

Finally, open *MainActivity.kt* and make sure that its code matches the code shown below:




```
package com.hfad.bitsandpizzas

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)
    }
}
```

Now that we've got `MainActivity` to display `OrderFragment`, let's find out how to make the app bar respond to scrolls.

# Replace the default app bar with a toolbar

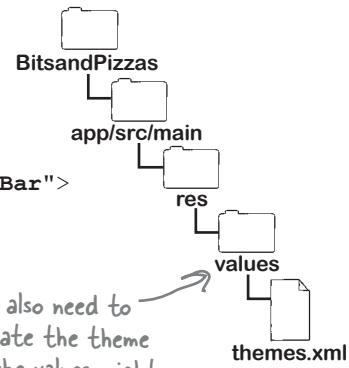
We're going to make the Bits and Pizzas app bar respond when the user scrolls. For now, we'll make it scroll off the screen when the user scrolls the screen up, and make it reappear when she scrolls down again.

To do this, we first need to replace the default app bar with a toolbar. This is because the default app bar is fixed to the top of the screen, and can't be made to scroll. A toolbar, however, is much more flexible.

To do this, we first need to change the app's theme to one that has no app bar. Open the file *themes.xml* in the *app/src/main/res/values* folder, and update its code to include the style below in bold:

```
<resources>
    <style name="Theme.BitsAndPizzas"
           parent="Theme.MaterialComponents.DayNight.NoActionBar">
        ...
    </style>
</resources>
```

We're going to use a toolbar in place of the default app bar, so we need to apply a "NoActionBar" style.



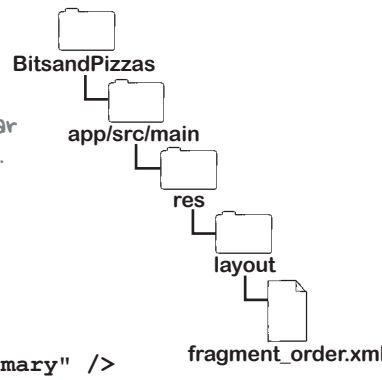

You also need to update the theme in the values-night version of this file (if one exists).

If your project includes a *themes.xml* file in the *values-night* folder, you'll need to apply the above change to this file as well.

Once you've made this change, add a toolbar to `FragmentOrder` by updating the *fragment_order.xml* code so it matches the code shown here:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout 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"
    tools:context=".OrderFragment">

    <com.google.android.material.appbar.MaterialToolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        style="@style/Widget.MaterialComponents.Toolbar.Primary" />

</FrameLayout>
```

This adds a Material toolbar to OrderFragment's layout.



# Fragments don't have a setSupportActionBar() method

Now that we've added the toolbar, we need to make it behave like a proper app bar that displays the app's name. As you learned in the previous chapter, this is done by calling the activity's `setSupportActionBar()` method.

Here, we've added the toolbar to a fragment, and **fragments don't include a `setSupportActionBar()` method**. To get around this, we'll get a reference to the activity that's displaying the fragment (using **`activity`**), cast it to an `AppCompatActivity` to reflect its type, and call its `setSupportActionBar()` method.

Here's the code to do this:

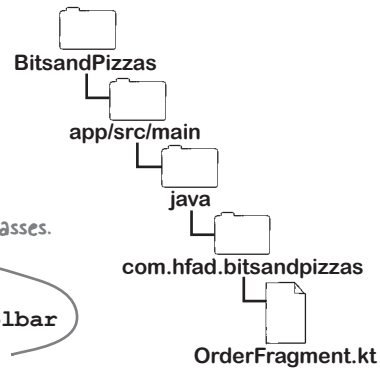
```
val toolbar = view.findViewById<MaterialToolbar>(R.id.toolbar)
(activity as AppCompatActivity).setSupportActionBar(toolbar)
```

This casts the activity that's displaying the fragment to an AppCompatActivity, and calls its setSupportActionBar method.

Here's the full code for *OrderFragment.kt*; update your code to include the changes (in bold):

```
package com.hfad.bitsandpizzas

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.appcompat.app.AppCompatActivity
import com.google.android.material.appbar.MaterialToolbar

class OrderFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        return val view = inflater.inflate(R.layout.fragment_order, container, false)
        val toolbar = view.findViewById<MaterialToolbar>(R.id.toolbar)
        (activity as AppCompatActivity).setSupportActionBar(toolbar)
        return view
    }
}
```

Import these classes.




This makes the activity use the toolbar as its app bar so that it displays the app's name.

**Scrolling toolbar**
Collapsing toolbar
Views
Respond to clicks

# We've added the toolbar...now what?

We've now added a toolbar to `OrderFragment`'s layout so that when the app runs, the toolbar is displayed at the top of the screen:

If we try and scroll the screen, however, the toolbar doesn't move. To make it respond to scrolls, we need to make a few more changes.

## We want the toolbar to respond to scrolls

To make the toolbar move, we need to add more views to the fragment's layout. The layout needs to follow the following structure:

The layout must have a coordinator layout at its root.

```
<androidx.coordinatorlayout.widget.CoordinatorLayout ...>

    <com.google.android.material.appbar.AppBarLayout ...>

        <com.google.android.material.appbar.MaterialToolbar .../>

    </com.google.android.material.appbar.AppBarLayout>

    <androidx.core.widget.NestedScrollView ...>

        ...

    </androidx.core.widget.NestedScrollView>

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

The toolbar needs to be inside an app bar layout.

The fragment's main content needs to be placed inside a nested scroll view.

The layout needs to include three things: a **coordinator layout**, an **app bar layout,** and a **nested scroll view**. Together, they let the toolbar respond when the user scrolls the screen.

Let's find out what each one does, starting with the coordinator layout.

# The coordinator layout coordinates animations between views

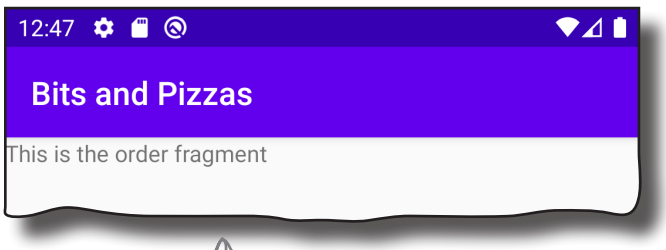
A coordinator layout is like a souped-up frame layout that's used to coordinate animations between different views. It can, for example, coordinate the user scrolling the layout's main content with the toolbar moving off the screen.

You add a coordinator layout to your layout code like this:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    ...>

    ...

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Put any views whose behavior you want to coordinate here.

You need to include any views whose animations you want to coordinate in the coordinator layout. In the Bits and Pizzas app, for example, we want to coordinate two things: the user scrolling the layout's main contents, and the toolbar scrolling off the screen. This means the toolbar and the screen's main contents need to be included in the coordinator layout:




The toolbar needs to scroll off the screen when the user scrolls the layout's main contents. As these two animations need to be coordinated, everything that's displayed in the layout needs to be included in the coordinator layout.

Now that we know what the coordinator layout does, let's move on to the app bar layout.

A CoordinatorLayout allows the behavior of one view to affect the behavior of another.

The CoordinatorLayout is usually the root element of your layout.

**Scrolling toolbar**
Collapsing toolbar
Views
Respond to clicks

# The app bar layout enables toolbar animation

An app bar layout is a type of vertical linear layout that's designed to work with app bars. It works with the coordinator layout to enable toolbar animation.

You add an app bar layout to your code like this:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    ...>

    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/ThemeOverlay.MaterialComponents.Dark.ActionBar">

        <com.google.android.material.appbar.MaterialToolbar .../>

    </com.google.android.material.appbar.AppBarLayout>

    ...

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

You add the AppBarLayout to the CoordinatorLayout.

The toolbar goes inside the AppBarLayout.

This applies a style to all of the app bar layout's views.

As you can see, the above app bar layout code includes an `android:theme` attribute like this:

```
android:theme="@style/ThemeOverlay.MaterialComponents.Dark.ActionBar"
```

This applies the specified style to the app bar layout and all of its views. In this example, it means that the toolbar—and anything else we add to the app bar layout—will be styled to use the Material theme's app bar properties, including the color of its background and text.

Great! Now your toolbar is animated and can respond to scroll events. But that's not the end of the story: you also need to specify how it should respond. Let's see how this is done.

# Tell the toolbar how to respond to scroll events

Once you've added an app bar layout, you tell the toolbar how to respond to scrolls by adding an **`app:layout_scrollFlags`** attribute to the toolbar and assigning a value to it. The value specifies how the toolbar should respond to scroll events.

In the Bits and Pizzas app, we want the toolbar to scroll upward off the screen when the user scrolls up, and quickly return to its original position when the user scrolls down. We can achieve this by setting the toolbar's `app:layout_scrollFlags` attribute to `"scroll|enterAlways"` using code like this:

**The toolbar MUST be inside an app bar layout in order for it to scroll. The app bar layout and coordinator layout work together to enable scrolling.**

```
<androidx.coordinatorlayout.widget.CoordinatorLayout ...>
    <com.google.android.material.appbar.AppBarLayout...>

        <com.google.android.material.appbar.MaterialToolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            app:layout_scrollFlags="scroll|enterAlways"/>

    </com.google.android.material.appbar.AppBarLayout>
    ...
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

This line tells the CoordinatorLayout (and AppBarLayout) how you want the toolbar to react to the user scrolling content.

The line:

```
app:layout_scrollFlags="scroll|enterAlways"
```

specifies two behaviors: `scroll` and `enterAlways`.

The `scroll` value means that the view can scroll off the top of the screen when the user scrolls up. Without this value, the toolbar would stay pinned to the top of the screen and wouldn't be able to scroll.

The `enterAlways` value means that the toolbar quickly scrolls down to its original position when the user scrolls down. The toolbar would still scroll down without this value, but it would be much slower.

You're almost there! There's just one more step you need to do before we can get the `OrderFragment` toolbar to scroll: the **nested scroll view**.

# A nested scroll view makes layout content scrollable

You now need to add a **nested scroll view** so the layout's main content can scroll. This kind of view works just like a normal scroll view, except that it enables *nested* scrolling. This is important, because the coordinator layout *only listens for nested scroll events*. If you use a normal scroll view in your layout, the toolbar won't be able to respond when the user scrolls the screen.

Another view that enables nested scrolling is the recycler view. You'll learn about this view in Chapter 15.

You add a nested scroll view to your layout using code like this:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout...>
    <com.google.android.material.appbar.AppBarLayout...>
        ...
    </com.google.android.material.appbar.AppBarLayout>

    <androidx.core.widget.NestedScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_behavior="@string/appbar_scrolling_view_behavior">

        <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:text="This is the order fragment" />

    </androidx.core.widget.NestedScrollView>
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

This defines the NestedScrollView.

This line makes sure that the NestedScroll view isn't hidden by the AppBarLayout

You add any views you want the user to be able to scroll to the NestedScrollView.

In the above example, the nested scroll view includes an extra attribute named `app:layout_behavior`, which is set to the built-in `String` value `"@string/appbar_scrolling_view_behavior"`. This makes sure that the contents of the nested scroll view are arranged below the app bar layout, and move when it scrolls.

Note that **a nested scroll view can only have one direct child**, which, in the example above, is a text view. If you want to add more than one view to a nested scroll view, you must first add them to a view group—such as a linear layout—and then add the view group to the nested scroll view.

You'll see an example of this later in the chapter.

That's everything that we need to know in order to get the Bits and Pizzas toolbar to scroll, so let's go ahead and update `OrderFragment`'s layout.

**Scrolling toolbar**
Collapsing toolbar
Views
Respond to clicks

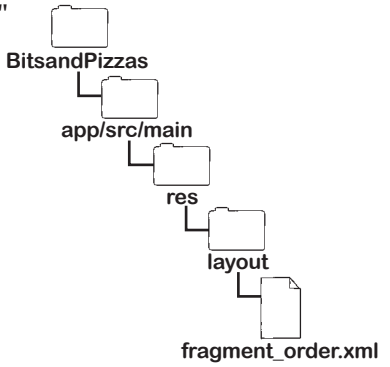
# The full code for fragment_order.xml

Here's the full code for *fragment_order.xml*; update the code to include the changes below (in bold):

```
<?xml version="1.0" encoding="utf-8"?>
<~~FrameLayout~~ androidx.coordinatorlayout.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".OrderFragment">

    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/ThemeOverlay.MaterialComponents.Dark.ActionBar">

        <com.google.android.material.appbar.MaterialToolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            ~~style="@style/Widget.MaterialComponents.Toolbar.Primary"~~
            app:layout_scrollFlags="scroll|enterAlways" />
    </com.google.android.material.appbar.AppBarLayout>

    <androidx.core.widget.NestedScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_behavior="@string/appbar_scrolling_view_behavior">

        <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:text="This is the order fragment" />
    </androidx.core.widget.NestedScrollView>
</~~FrameLayout~~ androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Use a CoordinatorLayout in place of a FrameLayout.

Add an extra name space.




Enclose the toolbar inside an AppBarLayout so it can respond to scrolls.

Remove this line, as the toolbar will be styled by the AppBarLayout's theme.

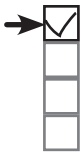
This line specifies how the toolbar should respond to scrolls.

Add a NestedScrollView, so that when its contents are scrolled, the toolbar can respond.

We're adding a TextView here so that we've got something to scroll when we take the app for a test drive.

Replace the FrameLayout with a CoordinatorLayout.

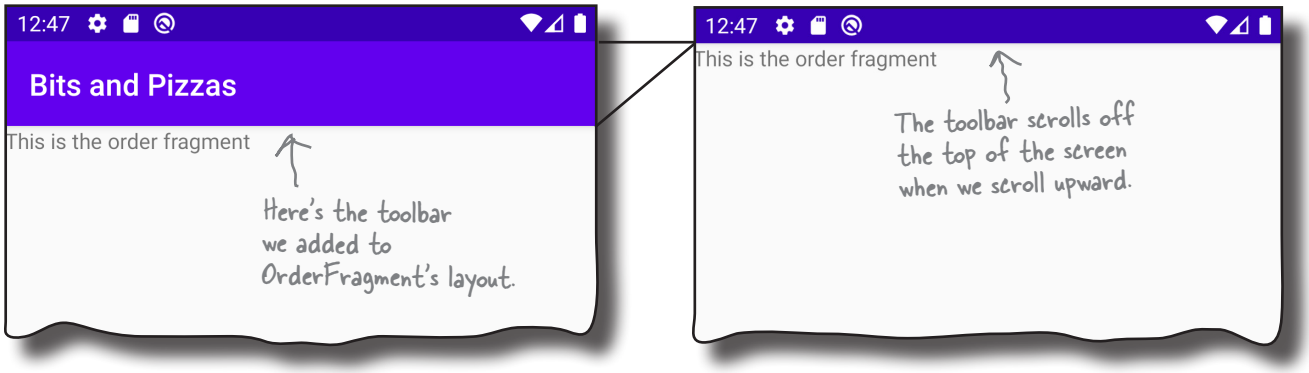
# Test Drive



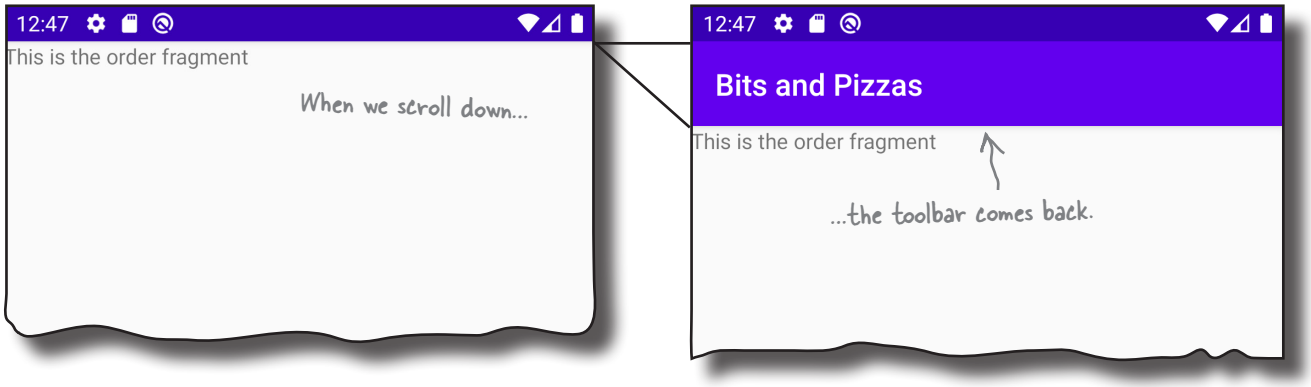
**Scrolling toolbar**
Collapsing toolbar
Views
Respond to clicks

When we run the app, `OrderFragment` is displayed in `MainActivity`'s layout. A toolbar is displayed at the top of the screen.

When we scroll the screen up, the toolbar scrolls upward off the top of the screen.




When we scroll the main content down, the toolbar reappears.




Congratulations! You've now learned how to create a toolbar that responds to scrolls.

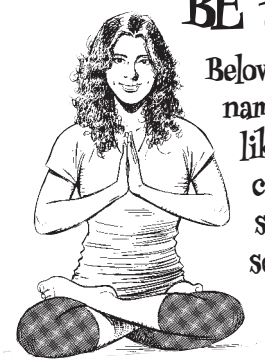
After the following exercise, we're going to learn how to turn the scrolling toolbar into one that collapses and expands as the user scrolls the screen.

# BE the Layout

Below is the layout file for a fragment named MyFragment. Your job is to play like you're the layout, and change the code so that when the user scrolls the screen up, the toolbar scrolls off the screen, and when they scroll down, it quickly reappears.

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MyFragment">

    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/ThemeOverlay.MaterialComponents.Dark.ActionBar">

        <com.google.android.material.appbar.MaterialToolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize" />
    </com.google.android.material.appbar.AppBarLayout>

    <ScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        ... ← There are more views here, but we've left them out.

    </ScrollView>
</FrameLayout>
```

# BE the Layout Solution



Below is the layout file for a fragment named MyFragment. Your job is to play like you're the layout, and change the code so that when the user scrolls the screen up, the toolbar scrolls off the screen, and when they scroll down, it quickly reappears.

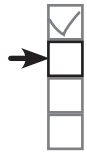
You need to use a CoordinatorLayout.

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout androidx.coordinatorlayout.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MyFragment">

    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/ThemeOverlay.MaterialComponents.Dark.ActionBar">

        <com.google.android.material.appbar.MaterialToolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            app:layout_scrollFlags="scroll|enterAlways" />
    </com.google.android.material.appbar.AppBarLayout>

    <androidx.core.widget.NestedScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_behavior="@string/appbar_scrolling_view_behavior">

        ...

    </androidx.core.widget.NestedScrollView>
</FrameLayout androidx.coordinatorlayout.widget.CoordinatorLayout>
```

This makes the toolbar scroll.

You need to use a NestedScrollView, not a ScrollView.

This arranges the NestedScrollView so that it's not hidden by the AppBarLayout.

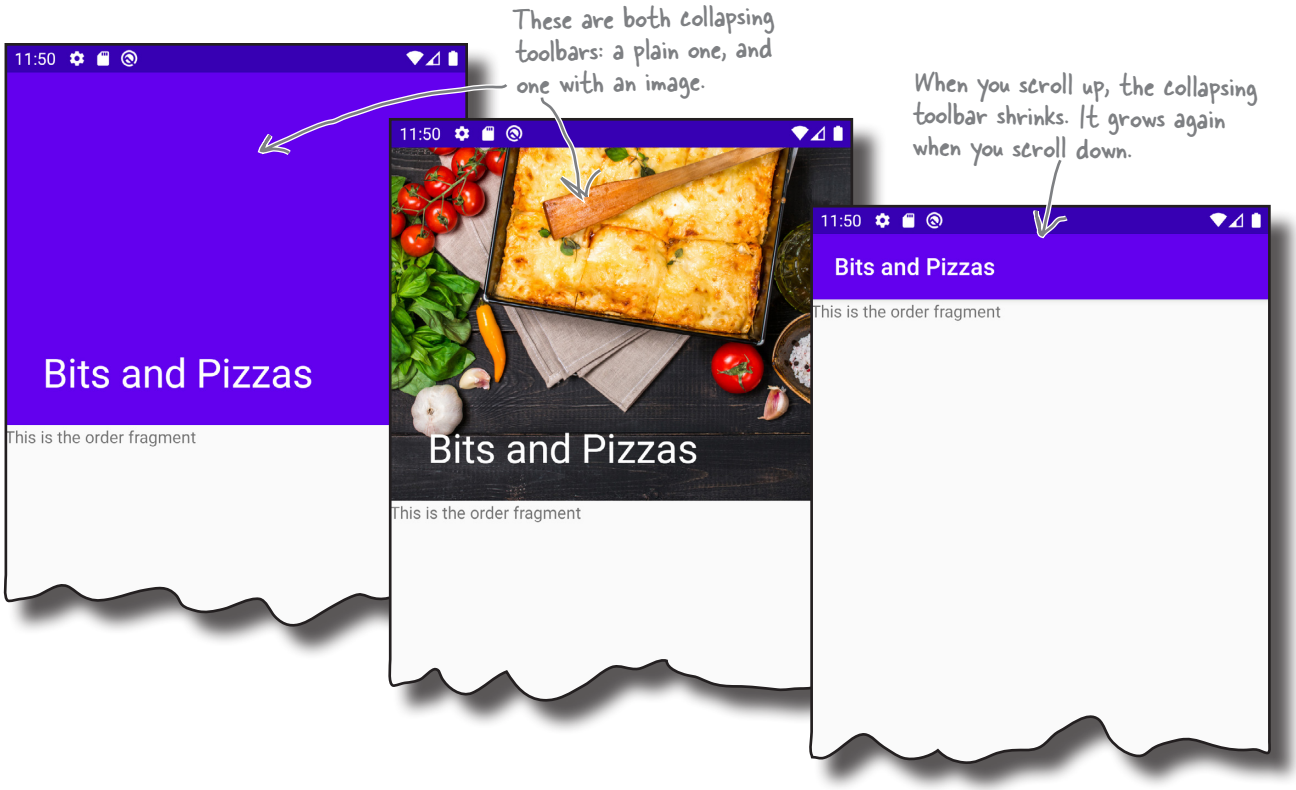


Scrolling toolbar
**Collapsing toolbar**
Views
Respond to clicks

# Let's create a collapsing toolbar

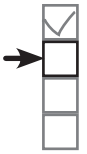
Now that you know how to make a toolbar scroll off the screen, let's replace it with a slightly different type of toolbar: a **collapsing toolbar**.

A collapsing toolbar is a toolbar that starts off large, shrinks when the user scrolls the screen up, and expands again when the user scrolls the screen back down. You can even add an image to it, which disappears when the toolbar reaches its minimum height, and becomes visible again as the toolbar expands:




Over the next few pages, we're going to find out how to turn a simple toolbar into a collapsing toolbar by adding one to `OrderFragment`'s layout. We'll start with a plain collapsing toolbar first, and then create one that includes an image.

Let's get started.



**Scrolling toolbar**
**Collapsing toolbar**
**Views**
**Respond to clicks**

# How to create a plain collapsing toolbar

It's relatively straightforward to turn a scrolling toolbar into a collapsing toolbar. You simply wrap the toolbar in a **collapsing toolbar layout**, and tweak the toolbar attributes. The basic code structure looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout ...>
    <com.google.android.material.appbar.AppBarLayout ...>

        <com.google.android.material.appbar.CollapsingToolbarLayout
            android:layout_width="match_parent"
            android:layout_height="300dp"
            app:layout_scrollFlags="scroll|exitUntilCollapsed">

            <com.google.android.material.appbar.MaterialToolbar
                android:id="@+id/toolbar"
                android:layout_width="match_parent"
                android:layout_height="?attr/actionBarSize"
                app:layout_collapseMode="pin"/>

        </com.google.android.material.appbar.CollapsingToolbarLayout>
    </com.google.android.material.appbar.AppBarLayout>
    ...
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

This adds the collapsing toolbar layout. It sits inside the app bar layout.

The collapsing toolbar's maximum height.

The toolbar goes inside the collapsing toolbar layout.

This line stops any items such as Up buttons from scrolling off the screen when the toolbar collapses.

The collapsing toolbar's closing element

As you can see, the collapsing toolbar layout is defined using the `<...CollapsingToolbarLayout>` element, part of the *com.google.android.material* library. You specify its maximum height using its `layout_height` attribute, and the line:

```
app:layout_scrollFlags="scroll|exitUntilCollapsed"
```

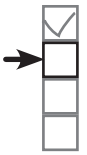
This means we want the toolbar to collapse until it's done collapsing.

tells it to collapse when the user scrolls up until it has nothing left to collapse, and expand when the user scrolls down until it reaches its full height.

We also need to make sure that when the toolbar collapses, anything that's displayed on the toolbar—like the Up button and any menu items—stay on the screen. This is achieved by adding the following attribute to the `Toolbar` element:

```
app:layout_collapseMode="pin"
```

Without this line, the Up button and any menu items would scroll off the screen.





# How to add an image to a collapsing toolbar

Once you've created a plain collapsing toolbar, you can add an image to it by adding an `<ImageView>` to the collapsing toolbar layout, specifying the image you want to use. The code follows this structure:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout ...>
    <com.google.android.material.appbar.AppBarLayout ...>

        <com.google.android.material.appbar.CollapsingToolbarLayout
            android:layout_width="match_parent"
            android:layout_height="300dp"
            app:layout_scrollFlags="scroll|exitUntilCollapsed"
            app:contentScrim="?attr/colorPrimary">

            <ImageView
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:scaleType="centerCrop"
                android:src="@drawable/image"
                app:layout_collapseMode="parallax"/>

            < com.google.android.material.appbar.MaterialToolbar .../>

        </com.google.android.material.appbar.CollapsingToolbarLayout>
    </com.google.android.material.appbar.AppBarLayout>
...
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

This line is optional. It gives the toolbar a plain background color when it's collapsed.

This adds an image to the collapsing toolbar layout.

This adds a parallax animation so that when the toolbar collapses, the image scrolls at a different rate to the rest of the toolbar.

The line:

```
app:contentScrim="?attr/colorPrimary"
```

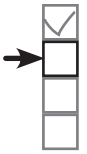
which has been added to the `<...CollapsingToolbarLayout>` gives the toolbar a plain background color when it's collapsed. We've also added a parallax animation to the image using this line:

```
app:layout_collapseMode="parallax"
```

This attribute is optional: it makes the image scroll at a different rate to the rest of the toolbar.

Now that you've learned about collapsing toolbars, let's add one to `OrderFragment`'s layout.

# Add a restaurant image drawable



Scrolling toolbar
**Collapsing toolbar**
Views
Respond to clicks

We want `OrderFragment`'s collapsing toolbar to include a restaurant image, so let's start by adding it to the project.

Make sure that your project includes a folder named *app/src/main/res/drawable*, then download the file *restaurant.webp* from *tinyurl.com/hfad3*, and add it to the *drawable* folder. This includes the image in your project as a drawable resource.

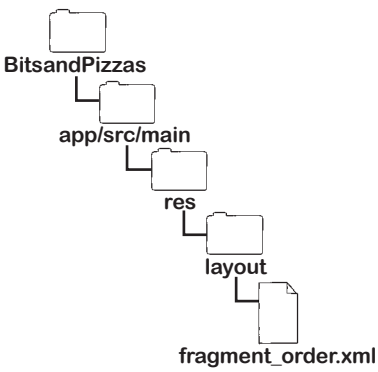
Next, let's add the collapsing toolbar.

You need to add this image to the drawable folder.

## The full code for fragment_order.xml

The code below adds a collapsing toolbar to `OrderFragment`'s layout. Update your code for *fragment_order.xml* to include the changes (in bold):

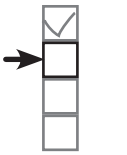
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".OrderFragment">

    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/ThemeOverlay.MaterialComponents.Dark.ActionBar">

        <com.google.android.material.appbar.CollapsingToolbarLayout
            android:layout_width="match_parent"
            android:layout_height="300dp"
            app:layout_scrollFlags="scroll|exitUntilCollapsed"
            app:contentScrim="?attr/colorPrimary">
```



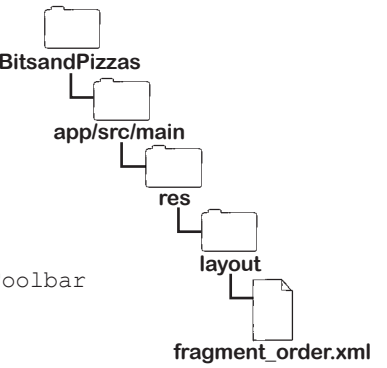

Add the CollapsingToolbarLayout.

The code continues on the next page.

# fragment_order.xml continued



Scrolling toolbar
**Collapsing toolbar**
Views
Respond to clicks




Add an image to the toolbar.

```
<ImageView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:scaleType="centerCrop"
    android:src="@drawable/restaurant"
    app:layout_collapseMode="parallax"/>

<com.google.android.material.appbar.MaterialToolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    app:layout_scrollFlags="scroll|enterAlways"
    app:layout_collapseMode="pin" />
```

Remove this line so that the basic toolbar doesn't scroll off the screen.

If the toolbar included an Up button or menu items, this would stop them from scrolling off the screen.

```
    </com.google.android.material.appbar.CollapsingToolbarLayout>
</com.google.android.material.appbar.AppBarLayout>

<androidx.core.widget.NestedScrollView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="This is the order fragment" />
</androidx.core.widget.NestedScrollView>
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

The CollapsingToolbarLayout's closing element

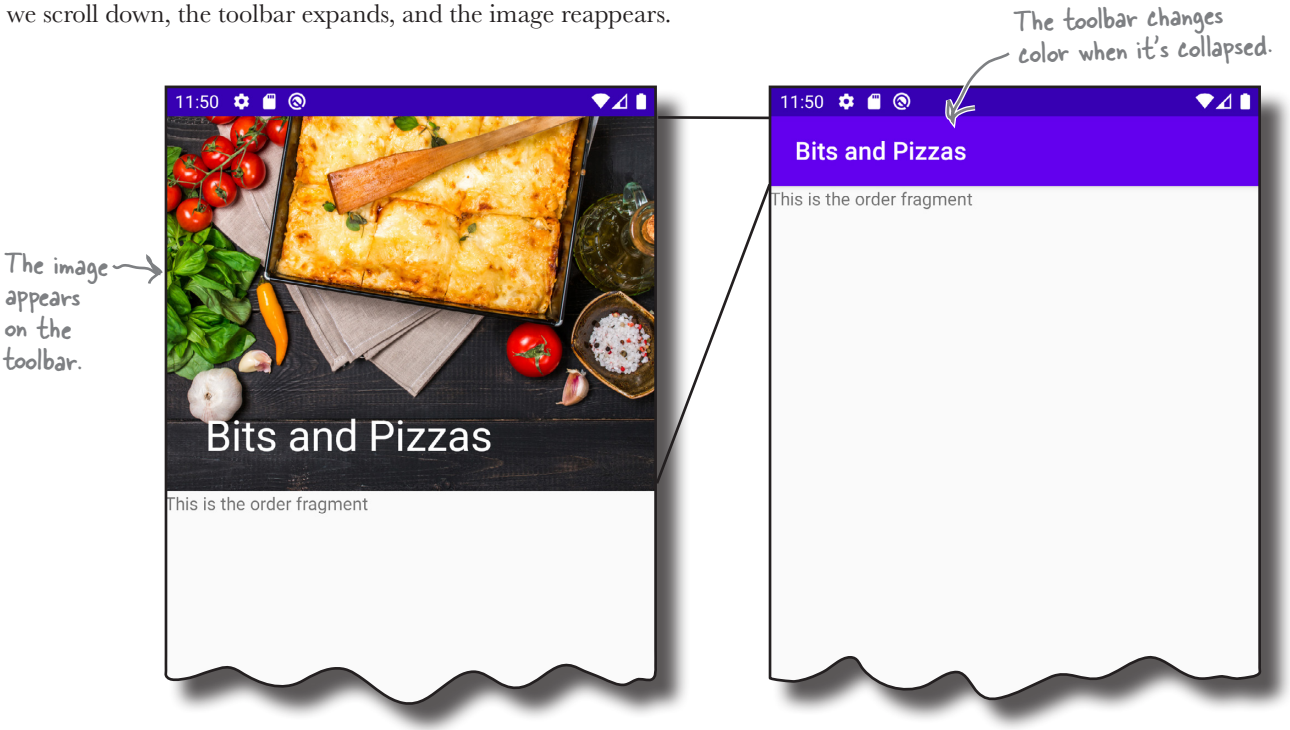
Those are all the changes that we need to create the collapsing toolbar. Let's take the app for a test drive, and see what it looks like.

# Test Drive

- Scrolling toolbar
- **Collapsing toolbar**
- Views
- Respond to clicks

When we run the app, `OrderFragment` is displayed. It includes a collapsing toolbar with an image.

When we scroll up, the toolbar collapses, the image fades, and the toolbar's background changes to the app's primary color. When we scroll down, the toolbar expands, and the image reappears.




## there are no Dumb Questions

**Q: Is the collapsing toolbar layout a type of view group?**

A: Yes. It's actually a subclass of `FrameLayout`, so you can use any of `FrameLayout`'s attributes with it.

**Q: Do I have to write any Kotlin code to get the toolbar to collapse?**

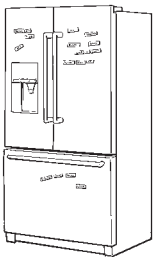
A: No, it's all done using layout code.

**Q: You said that the coordinator layout coordinates animations between its views. Are there any alternatives?**

A: Yes, there's a newer kind of layout you can use called a motion layout. This layout is part of the constraint layout library, from version 2.0 upward.

You can find out more about motion layouts here:

*https://developer.android.com/training/constraint-layout/motionlayout*



# Layout Magnets

Someone arranged some magnets on the fridge door to show how to structure a layout file that implements a collapsing toolbar with an image. Unfortunately, the magnets fell off when a large pterodactyl flew by in search of food.

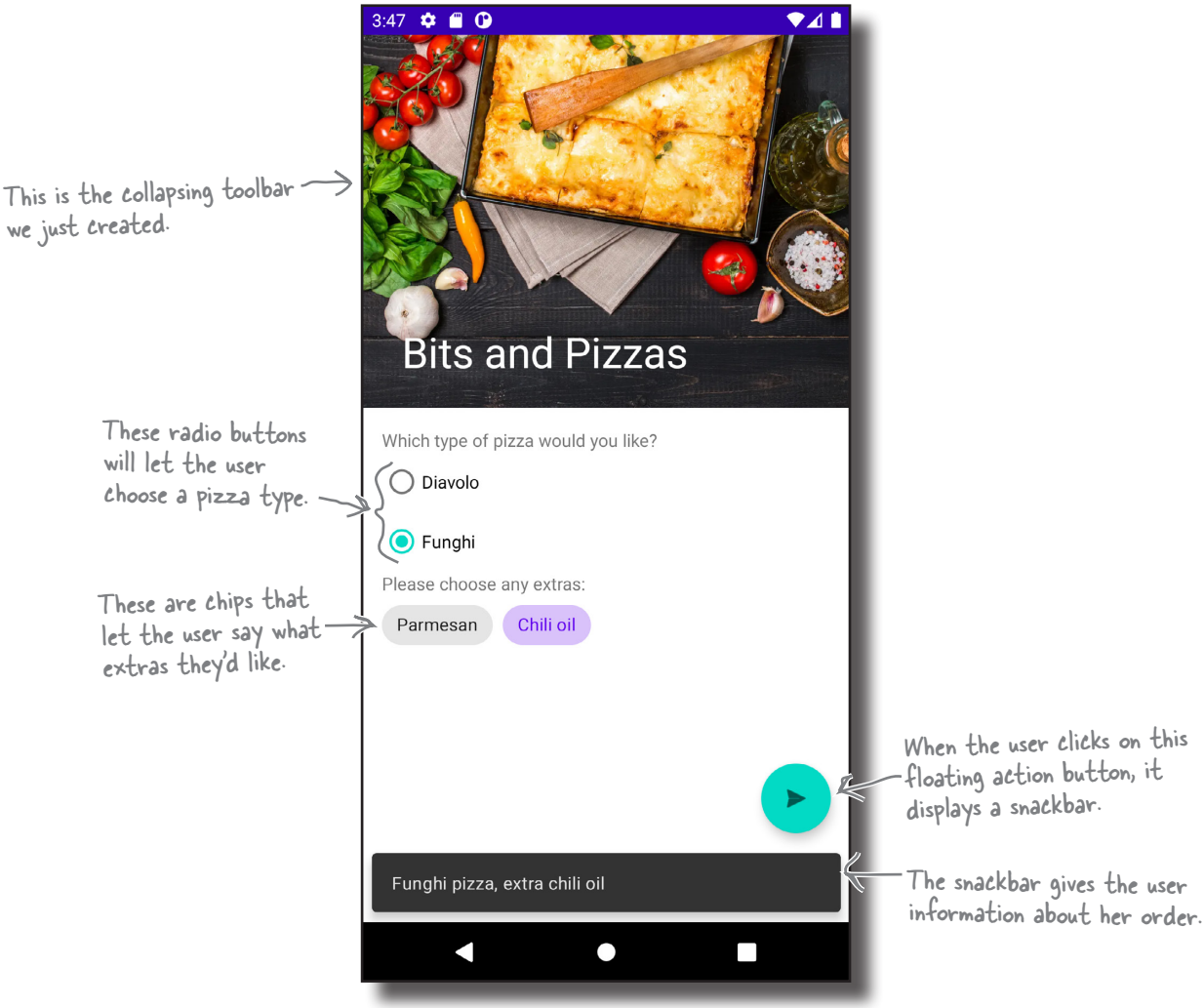
See if you can put the magnets back in the right order.

```
</com.google.android.material.appbar.MaterialToolbar>

<com.google.android.material.appbar.MaterialToolbar...>

<androidx.coordinatorlayout.widget.CoordinatorLayout...>

</androidx.coordinatorlayout.widget.CoordinatorLayout>

</androidx.core.widget.NestedScrollView>

<androidx.core.widget.NestedScrollView...>

</com.google.android.material.appbar.CollapsingToolbarLayout>

</ImageView>

<com.google.android.material.appbar.CollapsingToolbarLayout...>

<ImageView...>

</com.google.android.material.appbar.AppBarLayout>

<com.google.android.material.appbar.AppBarLayout...>
```

Answers on page 401.

# We need to build OrderFragment's main content

Now that we've added a collapsing toolbar to `OrderFragment`'s layout, we need to add some more views. These will let the user choose which type of pizza she wants to order, add any extra options such as parmesan or chili oil, and display a message when she clicks a button. The screen needs to look something like this:

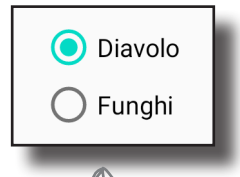



As you can see, `OrderFragment`'s layout includes some extra views that we haven't yet learned how to use. Before we build the layout, let's find out more about these views.

# Choose a pizza type using a radio button

The first view we'll use is a group of **radio buttons** so that the user can choose which type of pizza she wants. Radio buttons let you display multiple options from which you make a single choice, so they're a good choice for this situation.

You add radio buttons to a layout using two elements: `<RadioButton>` and `<RadioGroup>`. You use the `<RadioButton>` element to define each radio button, and you group them by putting them inside a `<RadioGroup>` element. Placing the radio buttons in a radio group in this way means that only a single radio button can be selected at a time.




This is a radio group with two radio buttons. Only one radio button can be selected at a time.

In the Bits and Pizzas app, we want to display radio buttons for the Diavolo and Funghi buttons. The code looks like this:

```
<RadioGroup
    android:id="@+id/pizza_group"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">

    <RadioButton android:id="@+id/radio_diavolo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Diavolo" />

    <RadioButton android:id="@+id/radio_funghi"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Funghi" />
</RadioGroup>
```

Once you've defined the radio group and radio buttons, you can write Kotlin code to find out which radio button has been selected using the radio group's `checkedRadioButtonId` property. Its value is the ID of the selected radio button, or -1 if no radio button has been selected:

```
val pizzaGroup = view.findViewById<RadioGroup>(R.id.pizza_group)
val pizzaType = pizzaGroup.checkedRadioButtonId
if (pizzaType == -1) {
    //no item selected
} else {
    val radio = view.findViewById<RadioButton>(id)
    //Do something using the radio button
}
```

This property lets you find out which radio button (if any) has been selected.

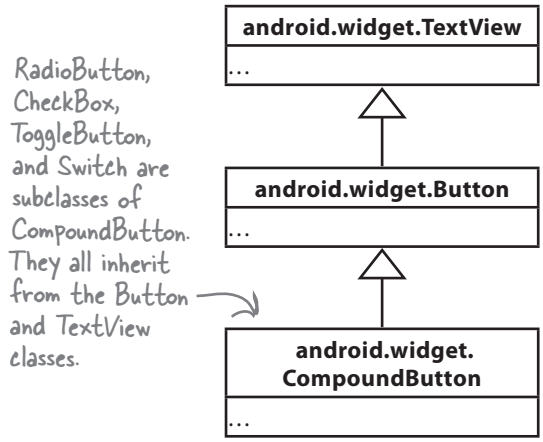
**RadioGroup is a subclass of LinearLayout, so you can use the same attributes with a radio group as you can with a linear layout.**

# Radio buttons are a type of compound button

Behind the scenes, radio buttons inherit from a class named `CompoundButton`: a subclass of `Button`. A compound button is a button that has two states: checked and unchecked, or on and off.

Android includes other types of compound button (in addition to radio buttons), such as **checkboxes**, **switches,** and **toggle buttons**. These views are useful if you want to offer the user yes/no choices, such as "Do you want chili oil?" or "Would you like extra parmesan?"

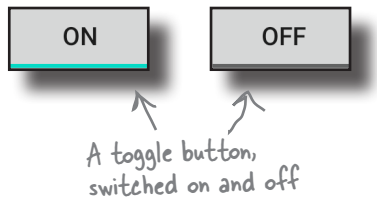
You add checkboxes, switches, and toggle buttons to your layout using code like this:




```
<CheckBox
    android:id="@+id/parmesan"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Parmesan" />
```

```
<Switch
    android:id="@+id/switch_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

```
<ToggleButton
    android:id="@+id/toggle_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textOn="On"
    android:textOff="Off" />
```



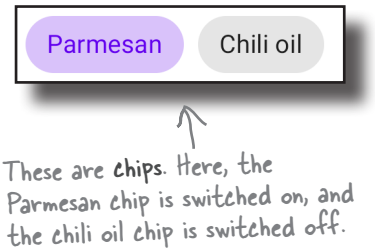

You can then use each view's `isChecked` property in your Kotlin code to find out if it has been selected, as in the example code here:

```
val parmesan = view.findViewById<CheckBox>(R.id.parmesan)
if (parmesan.isChecked) {
    //Do something
} else {
    //Do something else
}
```

The isChecked property lets you find out if a given CompoundButton has been checked, or switched on.

# A chip is a type of flexible compound button

So far, you've seen how to use different types of compound buttons such as radio buttons, checkboxes and switches. A more flexible type of compound button is a **chip**. This is a material view that's available so long as you're using a theme from the Material library, for example `Theme.MaterialComponents.DayNight.NoActionBar`. It's used for making yes/no choices like other types of compound button, but it has other uses too: it can also be used for user input, filtering data and performing actions.




In the Bits and Pizzas app, we're going to use chips to let the user say whether she wants extra parmesan or chili oil on her pizza.

You add a chip to your layout using code like this:

```
<com.google.android.material.chip.Chip
    android:id="@+id/parmesan"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Parmesan"
    style="@style/Widget.MaterialComponents.Chip.Choice"/>
```

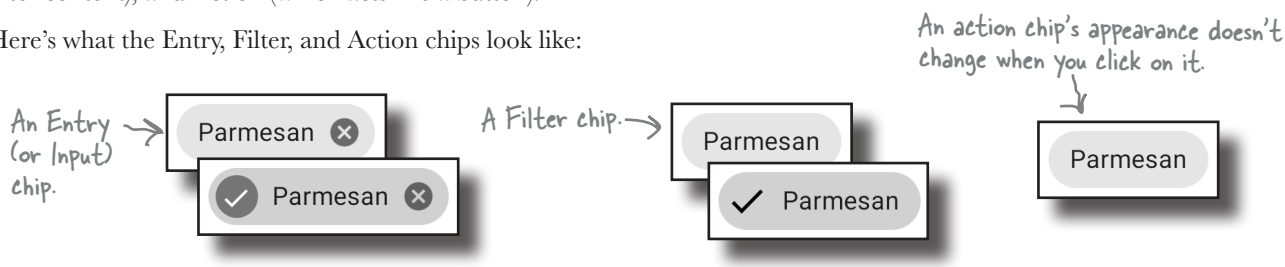
The Parmesan chip is styled using Choice so that it changes color when it's selected.

The key part of the chip code is the `style` attribute, because this controls the chip's appearance. The code:

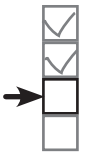
```
style="@style/Widget.MaterialComponents.Chip.Choice"
```

in the above example styles the chip as a Choice so that its color changes when its selected. Other options are Entry (which lets you use chips for data input), Filter (which is used for chips that filter content), and Action (which acts like a button).

Here's what the Entry, Filter, and Action chips look like:




As well as adding single chips to your layout, you can group several chips together. Let's find out how this is done.



Scrolling toolbar
Collapsing toolbar
**Views**
Respond to clicks

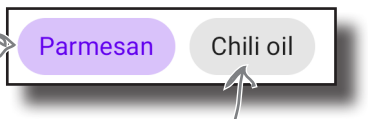
# Add multiple chips to a chip group

If you want your layout to include multiple chips that are grouped together, you can add them to a **chip group**. A chip group is a type of view group that's designed to neatly arrange multiple chips.

In the Bits and Pizzas app, we want to use two chips: one for parmesan and another for chili oil. We'll therefore group them in a chip group using code like this:

```
<com.google.android.material.chip.ChipGroup
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">

    <com.google.android.material.chip.Chip
        android:id="@+id/parmesan"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Parmesan"
        style="@style/Widget.MaterialComponents.Chip.Choice"/>

    <com.google.android.material.chip.Chip
        android:id="@+id/chili_oil"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Chili oil"
        style="@style/Widget.MaterialComponents.Chip.Choice"/>
</com.google.android.material.chip.ChipGroup>
```

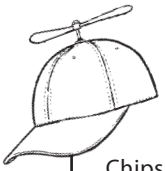
The parmesan and chili oil chips are neatly arranged inside a chip group.




## Use isChecked to find out if a chip is checked

Once you've added chips to your layout, you can find out if a chip has been selected using each chip's `isChecked` property, just as you can with other types of compound button such as switches, toggle buttons, and checkboxes. The following code, for example, checks whether the *parmesan* chip has been selected:

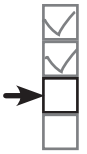
```
val parmesan = view.findViewById<Chip>(R.id.parmesan)
if (parmesan.isChecked) {
    //do something
}
```



## Serious Coding

Chips are very flexible and have many different options. Find out more about these options here:

*https://material.io/develop/android/components/chips*

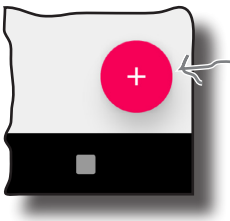




# A FAB is a floating action button

There's one final view we need to learn about before we can complete `OrderFragment`'s layout: a **FAB**.

A FAB—or **floating action button**—is a circular button that floats above the user interface. It's used to draw attention to common or important actions, and just like a normal button, you can make a FAB respond to clicks by assigning an `OnClickListener` to it in your Kotlin code.

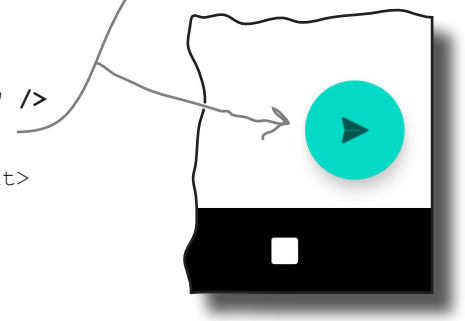


This is a FAB that's used in Google's Contacts app to add a contact.

You add a FAB to your layout using code like this:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout...>
    ...
    <androidx.core.widget.NestedScrollView...>
        ...
    </androidx.core.widget.NestedScrollView>

    <com.google.android.material.floatingactionbutton.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_margin="16dp"
        android:src="@android:drawable/ic_menu_send" />

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

This adds a round FAB with an icon. There's also an ExtendedFloatingActionButton type that lets you display an icon with text.



The above code uses the `layout_gravity` attribute to anchor a FAB to the bottom-end corner of the device screen with a margin of 16dp.

The line:

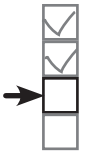
```
android:src="@android:drawable/ic_menu_send"
```

adds an icon to the FAB. In the above example, it displays one of Android's built-in icons named *ic_menu_send*, but you can use any kind of drawable so long as it fits on the FAB.

When you add a src attribute to a FAB, the code editor lets you browse Android's built-in icons. If you don't like any of them, you can find lots more here: https://material.io/resources/icons

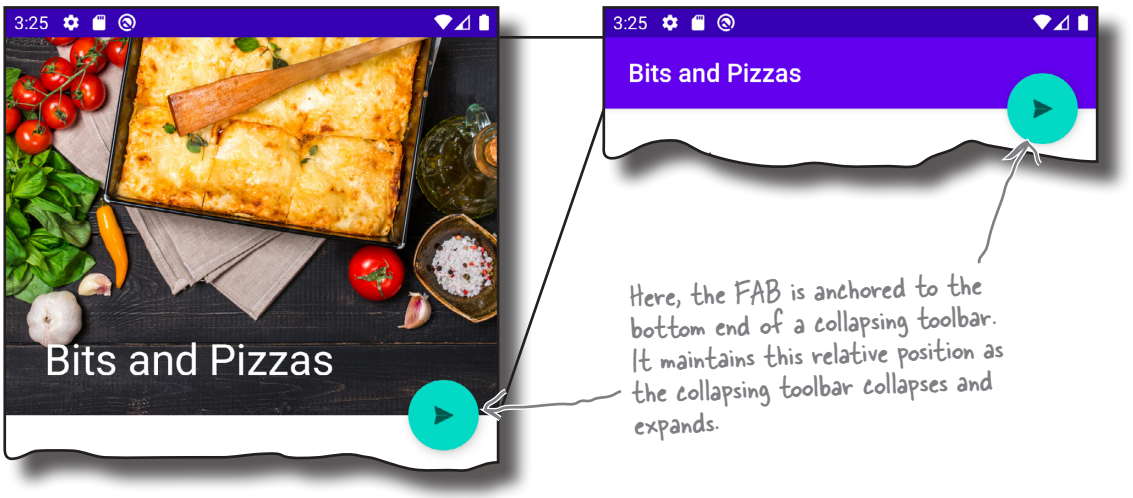
You usually use a FAB inside a coordinator layout so that you can coordinate movement between the different views in your layout. Let's look at an example.

# You can anchor a FAB to a collapsing toolbar



**Scrolling toolbar**
**Collapsing toolbar**
**Views**
**Respond to clicks**

FABs are often positioned at the bottom-end corner of the screen, but you can also anchor one to another view such as a collapsing toolbar. When you do this, the FAB moves with the collapsing toolbar as it collapses and expands:




The layout code for this is shown below. As you can see, it uses the FAB's `app:layout_anchor` and `app:layout_anchorGravity` attributes to anchor the FAB to the bottom end of the collapsing toolbar:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout...>
    ...
    <com.google.android.material.floatingactionbutton.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_anchor="@id/collapsing_toolbar"
        app:layout_anchorGravity="bottom|end"
        android:layout_margin="16dp"
        android:src="@android:drawable/ic_menu_send" />
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

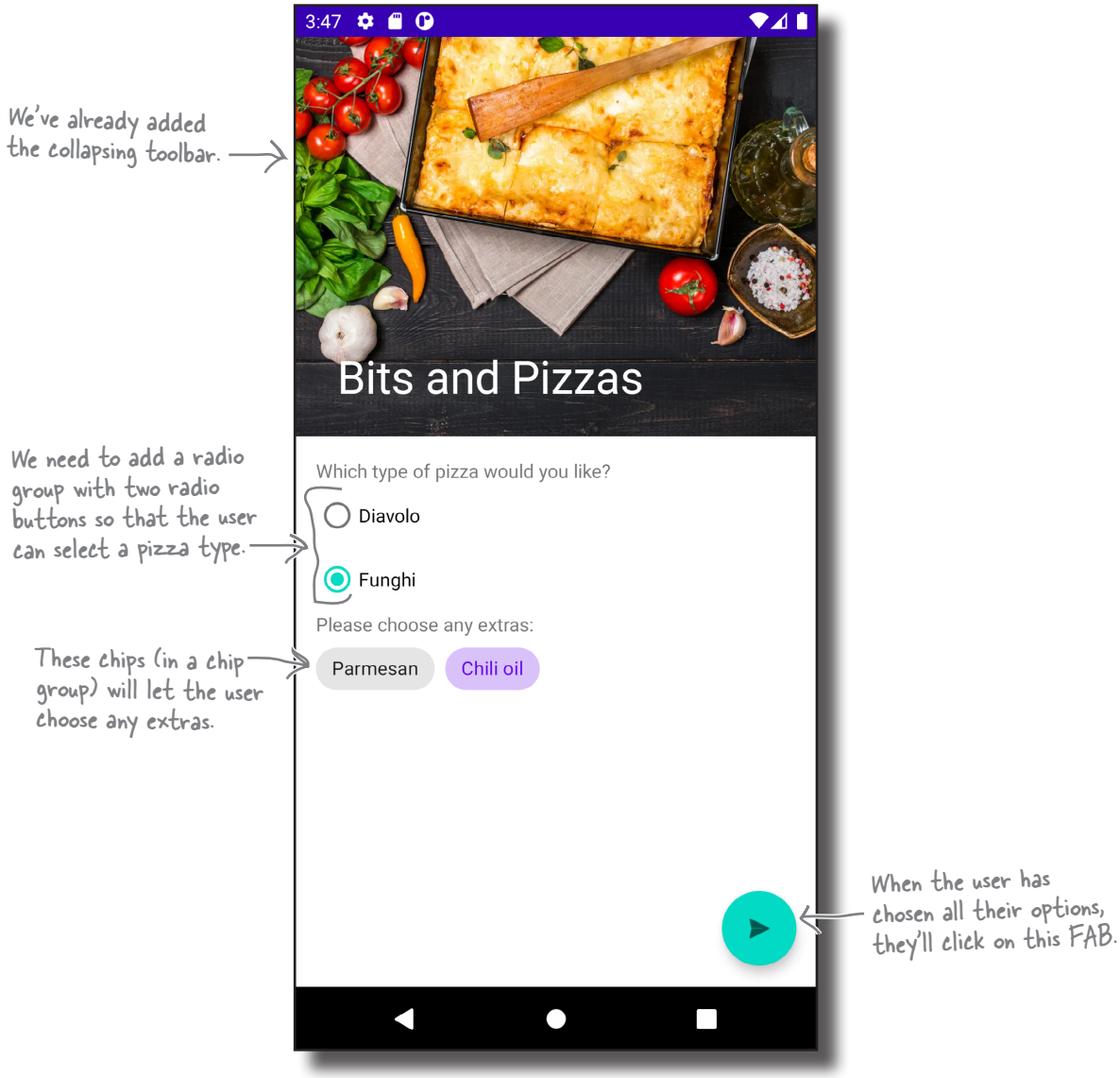
These lines anchor the FAB to the bottom end of a collapsing toolbar with the ID collapsing_toolbar.

Now that you've learned about radio buttons, chips, FABs, and other views, let's build the main content for `OrderFragment`'s layout.

**Scrolling toolbar**
**Collapsing toolbar**
**Views**
**Respond to clicks**

# We need to build OrderFragment's layout

We need to add views to `OrderFragment`'s layout so that the user can order a pizza type, and request any extras. The layout needs to look like this:




You're already familiar with all the code that's needed to create this layout, so let's go and update *fragment_order.xml*. We'll show you the full code over the next few pages.

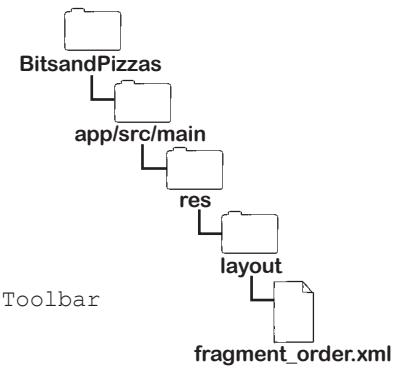
# The full code for fragment_order.xml

**Scrolling toolbar**
**Collapsing toolbar**
**Views**
**Respond to clicks**

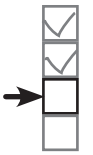
Here's the complete code for `OrderFragment`'s layout. Update the code for file *fragment_order.xml* so that it includes the changes below (in bold):

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".OrderFragment">

    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/ThemeOverlay.MaterialComponents.Dark.ActionBar">

        <com.google.android.material.appbar.CollapsingToolbarLayout
            android:layout_width="match_parent"
            android:layout_height="300dp"
            app:layout_scrollFlags="scroll|exitUntilCollapsed"
            app:contentScrim="?attr/colorPrimary">

            <ImageView
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:scaleType="centerCrop"
                android:src="@drawable/restaurant"
                app:layout_collapseMode="parallax"/>

            <com.google.android.material.appbar.MaterialToolbar
                android:id="@+id/toolbar"
                android:layout_width="match_parent"
                android:layout_height="?attr/actionBarSize"
                app:layout_collapseMode="pin" />
        </com.google.android.material.appbar.CollapsingToolbarLayout>
    </com.google.android.material.appbar.AppBarLayout>
```

You don't need to change any of the code on this page.




The code continues on the next page.

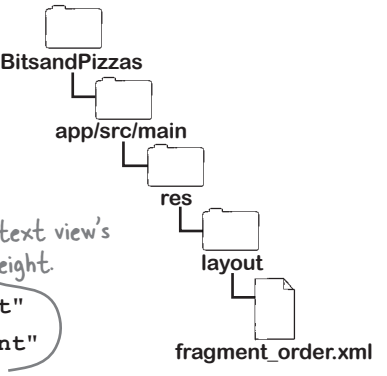




# fragment_order.xml continued

```
<androidx.core.widget.NestedScrollView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior">

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="16dp"
        android:orientation="vertical">

        <TextView
            android:layout_width="match_parentwrap_content"
            android:layout_height="match_parentwrap_content"
            android:text="This is the order fragment"
            android:text="Which type of pizza would you like?" />

        <RadioGroup
            android:id="@+id/pizza_group"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content">

            <RadioButton android:id="@+id/radio_diavolo"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Diavolo" />

            <RadioButton android:id="@+id/radio_funghi"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Funghi" />
        </RadioGroup>

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Please choose any extras:" />
```

A nested scroll view can only have one direct child, so we're putting all of the views in a linear layout.




Change the text view's width and height.

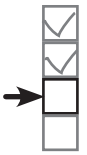
Replace the text that's displayed. In the real world, you'd want to put any Strings in a String resource file.

Add two pizza type radio buttons in a radio group.

Add a second text view asking the user to choose any extras.

The code continues on the next page.

# fragment_order.xml continued



**Scrolling toolbar**
**Collapsing toolbar**
**Views**
**Respond to clicks**

```
<com.google.android.material.chip.ChipGroup
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">

    <com.google.android.material.chip.Chip
        android:id="@+id/parmesan"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Parmesan"
        style="@style/Widget.MaterialComponents.Chip.Choice"/>

    <com.google.android.material.chip.Chip
        android:id="@+id/chili_oil"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Chili oil"
        style="@style/Widget.MaterialComponents.Chip.Choice"/>
</com.google.android.material.chip.ChipGroup>
```

Add two chips in a chip group.

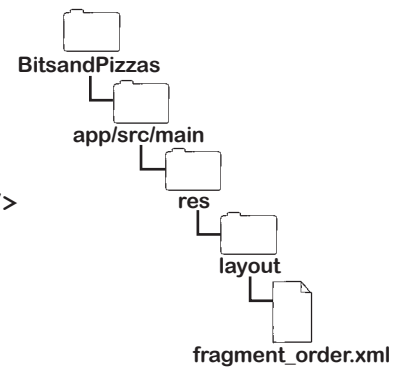
The chips are formatted as choice chips so that they change color when selected.

```
    </LinearLayout>
</androidx.core.widget.NestedScrollView>
```

This is the end of the linear layout.

```
<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|end"
    android:layout_margin="16dp"
    android:src="@android:drawable/ic_menu_send" />
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Add a FAB to the bottom end of the screen.

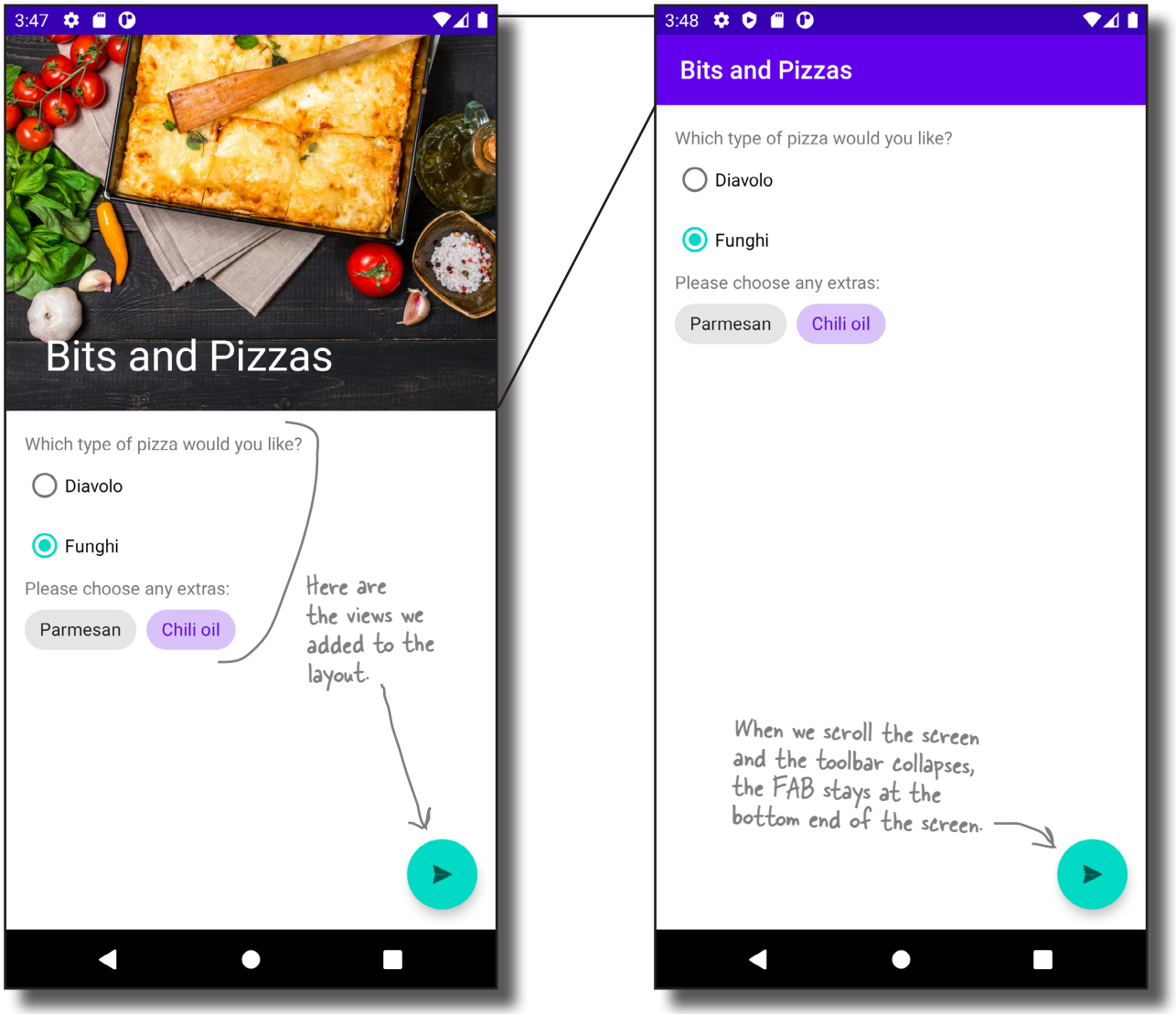



That's all the layout code we need, so let's take the app for a test drive and see what it looks like.

# Test Drive

When we run the app, `OrderFragment` is displayed in `MainActivity`. It includes a collapsing toolbar as before, but this time the main content features text views, radio buttons, chips, and a FAB.

When we scroll the device screen, the main content scrolls upward and the toolbar collapses. The FAB remains fixed in the bottom-end corner of the screen.



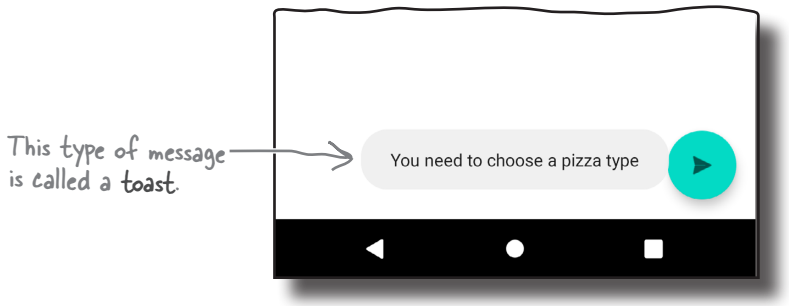
- Scrolling toolbar
- Collapsing toolbar
- Views
- **Respond to clicks**

# Let's make the FAB respond to clicks

We've now built `OrderFragment`'s layout, but when the user clicks on the FAB, nothing happens. Let's update the fragment's Kotlin code so that the FAB responds to clicks.

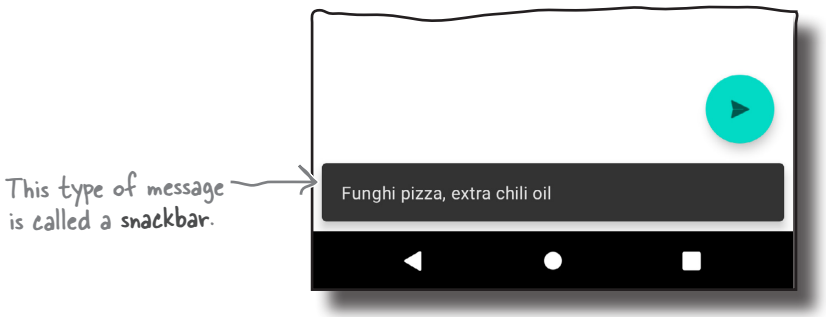
We'll make the FAB do two things:

1. **Display a message if no pizza has been chosen.**
We need the user to select which type of pizza she wants. If she clicks the FAB *without* choosing one, we'll display a pop-up message (called a **toast**) telling them to choose a pizza.




2. **Display their order in a separate message.**
If the user has selected a type of pizza, we'll display a message telling them what she has ordered. We'll use a different type of pop-up message for this called a **snackbar**.

In the real world, you'd want the app to place the pizza order when the user clicks on the FAB. Here, we just want to make the FAB do something (and it's a good excuse for teaching you about pop-up messages).




Let's begin by making the FAB respond to clicks.

Scrolling toolbar
Collapsing toolbar
Views
**Respond to clicks**

# Add an OnClickListener to the FAB

As we said earlier, you make a FAB respond to clicks in the same way that you make any other type of button respond to clicks: by attaching an `OnClickListener` to the FAB.

Here's the code to add an `OnClickListener` to the FAB in `OrderFragment`; as you can see, it looks the same as the code you've used with plain old normal buttons:

```
val fab = view.findViewById<FloatingActionButton>(R.id.fab)
fab.setOnClickListener {
    //Code that runs when the FAB is clicked
}
```

Make the FAB respond to clicks by getting a reference to it, and calling its setOnClickListener method.

Next, let's get the `OnClickListener` to display a message if the user hasn't selected a type of pizza.

## Find out if a pizza type has been selected

We can find out whether the user has selected a pizza type using the `checkedRadioButtonId` property of the `pizzas_group` radio group. The property's value is the ID of the selected radio button if one has been selected, and -1 if the user hasn't made a choice.

Here's the code to check whether the user has clicked the FAB without choosing which type of pizza she wants:

```
val fab = view.findViewById<FloatingActionButton>(R.id.fab)
fab.setOnClickListener {
    val pizzaGroup = view.findViewById<RadioGroup>(R.id.pizza_group)
    val pizzaType = pizzaGroup.checkedRadioButtonId
    if (pizzaType == -1) {
        //No pizza selected, so display a message
    } else {
        //Display the order in another message
    }
}
```

Get the value of the radio group's checkedRadioButtonId property.

If pizzaType is –1, no radio button in the group has been selected, so display a message.

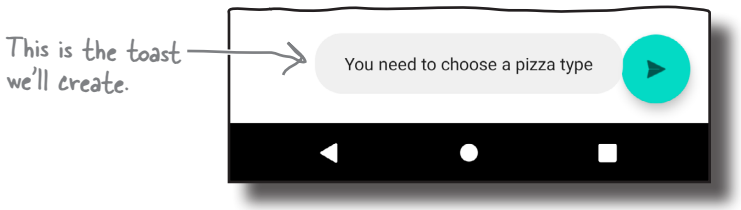
Display a different message if the user has selected one of the radio buttons.

If the user hasn't selected a type of pizza, we want to display a message in a pop-up message called a **toast**. Let's find out how to do this.

- Scrolling toolbar
- Collapsing toolbar
- Views
- **Respond to clicks**

# A toast is a simple pop-up message

If the user clicks on the FAB without first choosing a pizza type, we're going to display a toast on the device screen. A toast is a simple pop-up message that gives the user information, and automatically disappears when it times out:




You create a toast by calling `Toast.makeText()`. The `makeText` method takes three parameters: a `Context` (usually `this` or `activity`, depending on whether you're calling the toast from an activity or a fragment), a `CharSequence`! that's the message you want to display, and a duration. You then call the toast's `show()` method to display it.

Here's some example code that displays a message in a toast that appears on screen for a short duration:

```
val text = "Hello, I'm a toast!"
Toast.makeText(activity, text, Toast.LENGTH_SHORT).show()
```

## there are no Dumb Questions

**Q: My toasts don't appear when I run the app in the emulator. Any ideas?**

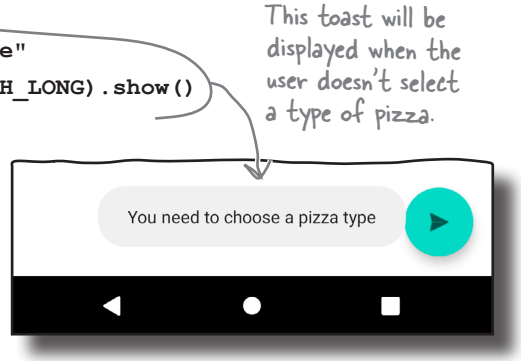
A: First, make sure you've called the toast's `show()` method. Without this, the toast won't appear.

If this doesn't help, you may have encountered a known emulator issue, which means toasts don't always pop up. Try closing the emulator, and open the AVD Manager from the Tools menu. Then select the virtual device, and choose the Wipe Data option from its list of actions.

## Add a toast to the FAB OnClickListener

We want to display a toast when the user clicks on the FAB without choosing a pizza type. The code to do this is shown below:

```
val fab = view.findViewById<FloatingActionButton>(R.id.fab)
fab.setOnClickListener {
    val pizzaGroup = view.findViewById<RadioGroup>(R.id.pizza_group)
    val pizzaType = pizzaGroup.checkedRadioButtonId
    if (pizzaType == -1) {
        val text = "You need to choose a pizza type"
        Toast.makeText(activity, text, Toast.LENGTH_LONG).show()
    } else {
        //Display the order in another message
    }
}
```

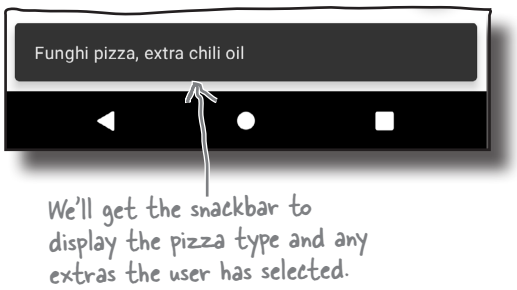



That's everything we need to do if the user hasn't selected a pizza type. Next, let's write the code to display their order.

**Scrolling toolbar**
**Collapsing toolbar**
**Views**
**Respond to clicks**

# Display the pizza order in a snackbar

If the user has chosen a pizza type, we're going to display their order in a pop-up message called a **snackbar** when she clicks on the FAB. A snackbar is like a toast, except that it's more interactive. You can swipe a snackbar away, for example, or make it do something if it's clicked.



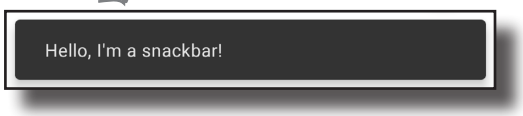

We'll get the snackbar to display the pizza type and any extras the user has selected.

You create a snackbar by calling `Snackbar.make()`. The `make` method takes three parameters: the `View` that triggered the snackbar (in this case a FAB), a `CharSequence`! that's the text you want to display, and a duration. You then display the snackbar by calling its `show()` method.

Here's some example snackbar code that displays a message on the screen for a short time:

```
val text = "Hello, I'm a snackbar!"
Snackbar.make(fab, text, Snackbar.LENGTH_SHORT).show()
```

The code displays this snackbar.




In the above code, we've used `LENGTH_SHORT` to show the snackbar for a short period of time. Other options are `LENGTH_LONG` (which shows it for a long duration) and `LENGTH_INDEFINITE` (which shows it indefinitely).

## Snackbars can have actions

If you want, you can add an action to the snackbar so that the user can, say, undo an action she's just performed. You do this by calling the snackbar's `setAction()` method before calling `show()`. `setAction` takes two parameters: the text that should appear for the action, and a lambda that runs when the user clicks on the action.

Here's an example of some snackbar code that includes an action:

```
Snackbar.make(fab, text, Snackbar.LENGTH_SHORT)
        .setAction("Undo") {
            //Code that runs when the user clicks the action
        }
        .show()
```

You need to specify what should happen if the user clicks on the Undo action.

Snackbars are usually displayed at the bottom of the screen, but you can override this using the snackbar's `setAnchorView()` method. This anchors the snackbar to a particular view so that the snackbar appears above it. This is useful if, for example, you want the snackbar to appear above a bottom navigation bar:

```
Snackbar.make(fab, text, Snackbar.LENGTH_SHORT)
    .setAnchorView(bottomNavBar)
    .show()
```

This anchors the snackbar so that it appears above the bottomNavBar view.

# The snackbar code for the pizza order

Now that you know how to create a snackbar, let's write the code to display the user's pizza order. We'll show a snackbar that displays the pizza type the user has selected, along with any extras such as parmesan or chili oil.

Here's the code to display the snackbar:

```
val fab = view.findViewById<FloatingActionButton>(R.id.fab)
fab.setOnClickListener {
    val pizzaGroup = view.findViewById<RadioGroup>(R.id.pizza_group)
    val pizzaType = pizzaGroup.checkedRadioButtonId
    if (pizzaType == -1) {
        val text = "You need to choose a pizza type"
        Toast.makeText(activity, text, Toast.LENGTH_LONG).show()
    } else {
        var text = (when (pizzaType) {
            R.id.radio_diavolo -> "Diavolo pizza"
            else -> "Funghi pizza"
        })
        val parmesan = view.findViewById<Chip>(R.id.parmesan)
        text += if (parmesan.isChecked) ", extra parmesan" else ""
        val chiliOil = view.findViewById<Chip>(R.id.chili_oil)
        text += if (chiliOil.isChecked) ", extra chili oil" else ""
        Snackbar.make(fab, text, Snackbar.LENGTH_LONG).show()
    }
}
```

Say which pizza has been selected.

Include any extras that have been selected.

Display the text in a snackbar.

That's all the Kotlin code we need for the Bits and Pizzas app. Let's look at the full code for *OrderFragment.kt* and take the app for a test drive.

## there are no Dumb Questions

**Q: Are toasts and snackbars types of view? Can I include them in my layouts?**

A: No, they're not views, so you can only make them appear on your screen using Kotlin code.

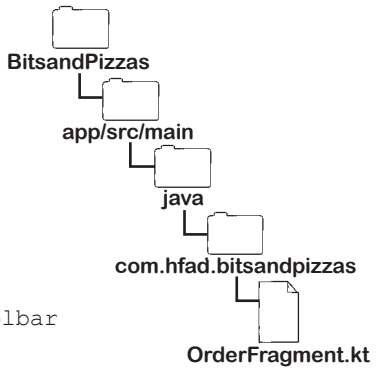
**Q: Can I use multiple FABs in a single layout?**

A: It's best if you don't use more than one FAB per screen. They're used for important actions, so if you use too many, they lose their impact.

Scrolling toolbar
Collapsing toolbar
Views
**Respond to clicks**

# The full code for OrderFragment.kt

Here's the full code for *OrderFragment.kt*; update the code to include the changes below (in bold):




```
package com.hfad.bitsandpizzas

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.appcompat.app.AppCompatActivity
import com.google.android.material.appbar.MaterialToolbar
import android.widget.RadioGroup
import com.google.android.material.chip.Chip
import com.google.android.material.floatingactionbutton.FloatingActionButton
import android.widget.Toast
import com.google.android.material.snackbar.Snackbar

class OrderFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        val view = inflater.inflate(R.layout.fragment_order, container, false)
        val toolbar = view.findViewById<MaterialToolbar>(R.id.toolbar)
        (activity as AppCompatActivity).setSupportActionBar(toolbar)

        val fab = view.findViewById<FloatingActionButton>(R.id.fab)
        fab.setOnClickListener {
            val pizzaGroup = view.findViewById<RadioGroup>(R.id.pizza_group)
            val pizzaType = pizzaGroup.checkedRadioButtonId
            if (pizzaType == -1) {
                val text = "You need to choose a pizza type"
                Toast.makeText(activity, text, Toast.LENGTH_LONG).show()
            } else {
                var text = (when (pizzaType) {
                    R.id.radio_diavolo -> "Diavolo pizza"
                    else -> "Funghi pizza"
                })
```

We're using these extra classes, so import them.

Add an OnClickListener to the FAB.

Display a toast if the user has clicked the FAB without choosing a pizza type.

Start building the pizza order text.

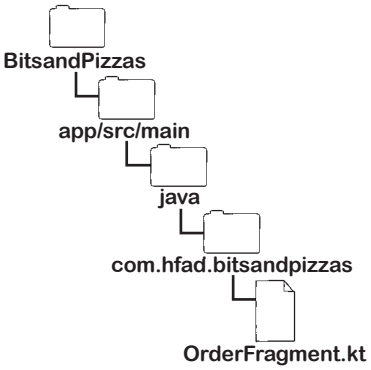
The code continues on the next page.

# OrderFragment.kt continued

Add any extras to the text.

```
            val parmesan = view.findViewById<Chip>(R.id.parmesan)
            text += if (parmesan.isChecked) ", extra parmesan" else ""
            val chiliOil = view.findViewById<Chip>(R.id.chili_oil)
            text += if (chiliOil.isChecked) ", extra chili oil" else ""
            Snackbar.make(fab, text, Snackbar.LENGTH_LONG).show()
        }
    }
    return view
  }
}
```

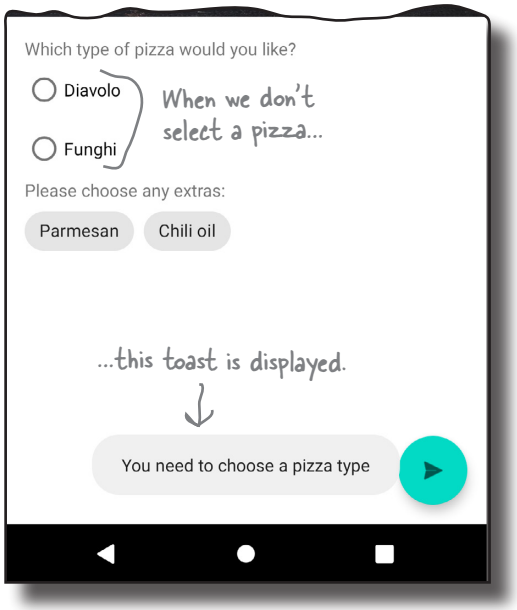
Make and display the snackbar.

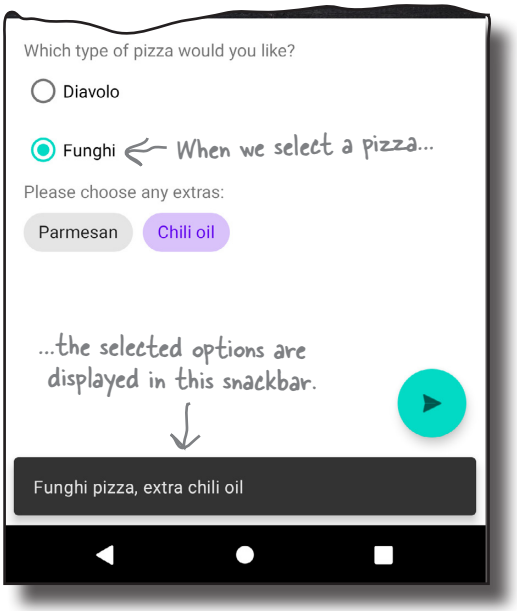


## Test Drive

When we click on the FAB without selecting a pizza type, a toast is displayed asking us to choose a pizza.

When we select a pizza type and click on the FAB again, a snackbar appears at the bottom of the screen giving details of our order.



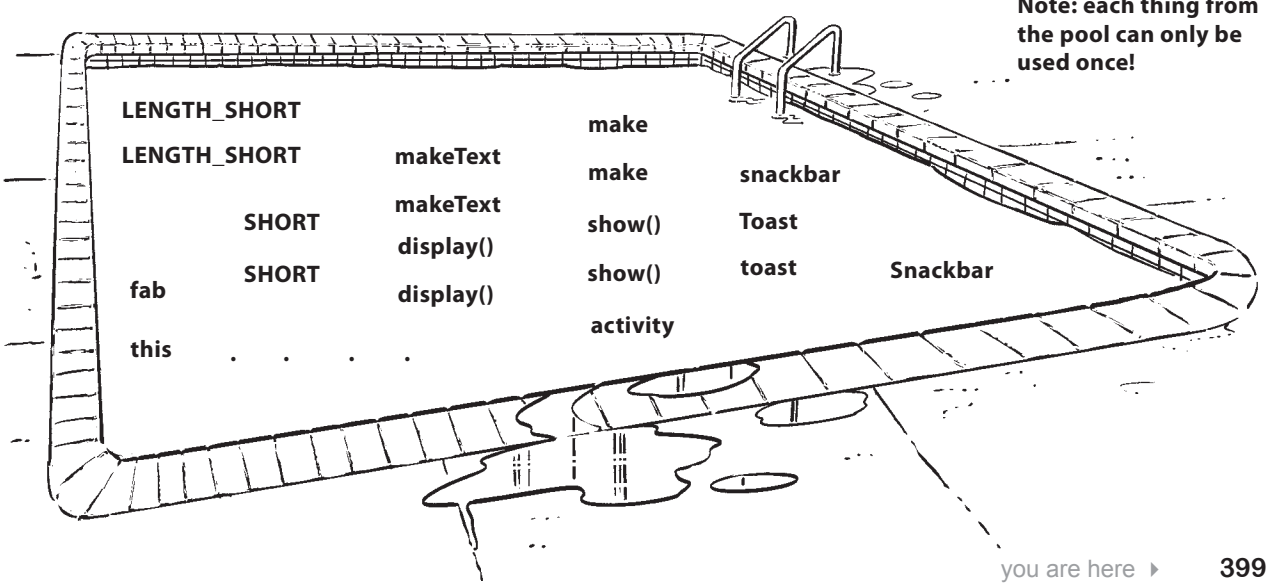

Congratulations! You've now learned how to make the FAB respond to clicks by displaying a pop-up message.

# Pool Puzzle

Your **goal** is to complete the fragment code below so that when a FAB with an ID of `fab` is clicked, it displays a snackbar. The snackbar should include an action, "Undo", which shows a toast when clicked. Take code snippets from the pool and place them into the blank lines in the code. You may **not** use the same snippet more than once, and you won't need to use all the snippets.

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
): View? {
    val view = inflater.inflate(R.layout.fragment_order, container, false)
    val fab = view.findViewById<FloatingActionButton>(R.id.fab)
    fab.setOnClickListener {
        Snackbar.……(fab, "Your order has been updated",……)
                .setAction("Undo") {
                    Toast.……(…… , "Undone!",……)
                            ……
                }
                ……
    }
    return view
}
```

**Note: each thing from the pool can only be used once!**



# Pool Puzzle Solution

Your **goal** is to complete the fragment code below so that when a FAB with an ID of `fab` is clicked, it displays a snackbar. The snackbar should include an action, "Undo", which shows a toast when clicked. Take code snippets from the pool and place them into the blank lines in the code. You may **not** use the same snippet more than once, and you won't need to use all the snippets.

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
): View? {
    val view = inflater.inflate(R.layout.fragment_order, container, false)
    val fab = view.findViewById<FloatingActionButton>(R.id.fab)
    fab.setOnClickListener {
        Snackbar.  make  (fab, "Your order has been updated", Snackbar.LENGTH_SHORT)
            .setAction("Undo") {
                Toast.makeText( activity , "Undone!", Toast.LENGTH_SHORT )
                    .show()
            }
            .show()
    }
    return view
}
```

Display the snackbar for a short period.

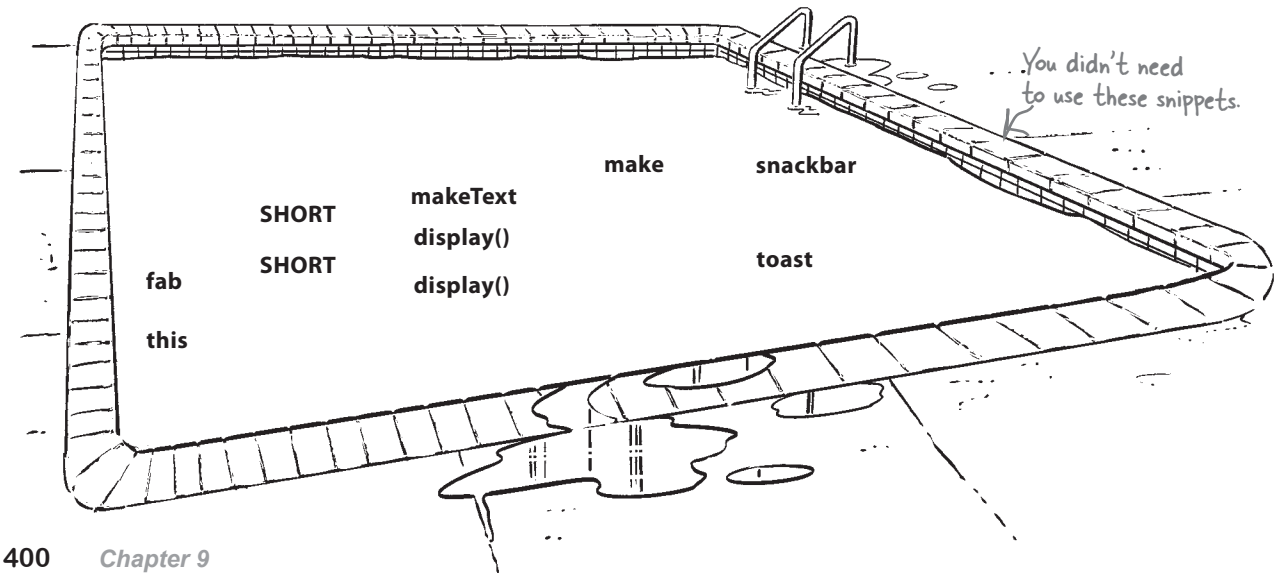
Use the make() method to build the snackbar.

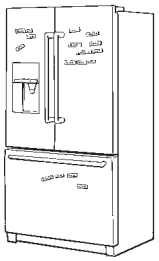
Use the makeText method to build the toast.

Show the toast.

Display the toast for a short period using the context of the activity that the fragment is displayed in.

Show the snackbar.





# Layout Magnets Solution

Someone arranged some magnets on the fridge door to show how to structure a layout file that implements a collapsing toolbar with an image. Unfortunately, the magnets fell off when a large pterodactyl flew by in search of food.

See if you can put the magnets back in the right order.

```
<androidx.coordinatorlayout.widget.CoordinatorLayout...>
```

The CoordinatorLayout goes at the root.

```
<com.google.android.material.appbar.AppBarLayout...>
```

An AppBarLayout goes inside the CoordinatorLayout, and contains a CollapsingToolbarLayout.

```
<com.google.android.material.appbar.CollapsingToolbarLayout...>
```

```
<ImageView...>
```

An ImageView and a MaterialToolbar go inside the CollapsingToolbarLayout.

```
</ImageView>
```

```
<com.google.android.material.appbar.MaterialToolbar...>
```

```
</com.google.android.material.appbar.MaterialToolbar>
```

```
</com.google.android.material.appbar.CollapsingToolbarLayout>
```

```
</com.google.android.material.appbar.AppBarLayout>
```

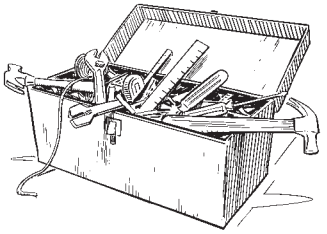
```
<androidx.core.widget.NestedScrollView...>
```

The NestedScrollView goes outside the AppBarLayout.

```
</androidx.core.widget.NestedScrollView>
```

```
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

# Your Android Toolbox



**You've got Chapter 9 under your belt and now you've added more views and components to your toolbox.**

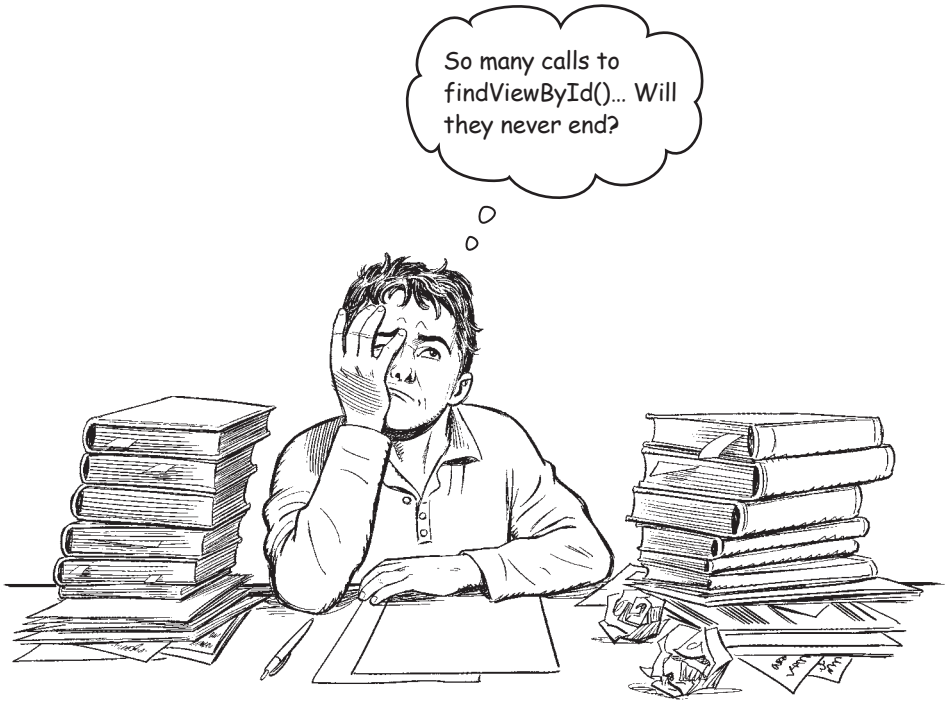
You can download the full code for the chapter from tinyurl.com/hfad3.

## Bullet Points

- Use a `CoordinatorLayout` to coordinate animations between views.
- Animate toolbars using an `AppBarLayout`.
- Use a `NestedScrollView` if you want views to respond to the user scrolling the device screen.
- Use a `CollapsingToolbarLayout` to add a toolbar that collapses and grows in response to user scroll actions.
- To define a group of radio buttons, first use `<RadioGroup>` to define the radio group. Then put individual radio buttons in the radio group using `<RadioButton>`.
- Radio buttons, toggle buttons, switches, and checkboxes are all types of compound button. A compound button is a button that can be switched on or off.
- A chip is a more flexible compound button that can be used for making choices, user input, filtering data and performing actions.
- You can put multiple chips in a chip group.
- Use a FAB (floating action button) to display common or important user actions.
- A toast is a type of pop-up message.
- A snackbar is another type of pop-up message that the user can interact with.

# 10 view binding

# 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, *you'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. Let's get started…

# Behind the scenes of findViewById()

As you already know, each time you want to interact with a view in your activity or fragment code, you first call `findViewById()` to get a reference to it. The following activity code, for example, gets a reference to a `Button` with the ID `start_button` so that it can respond to clicks:

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

        val startButton = findViewById<Button>(R.id.start_button)
        startButton.setOnClickListener {
            //Code that does something
        }
    }
}
```

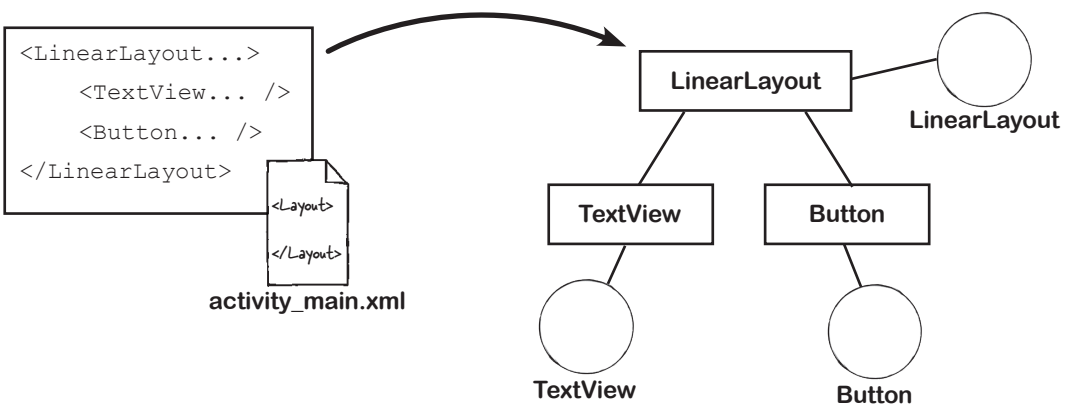
findViewById() gets a reference to the button so that we can call its methods.

But what actually happens when `findViewById()` gets called?

## findViewById() looks for a view in the view hierarchy

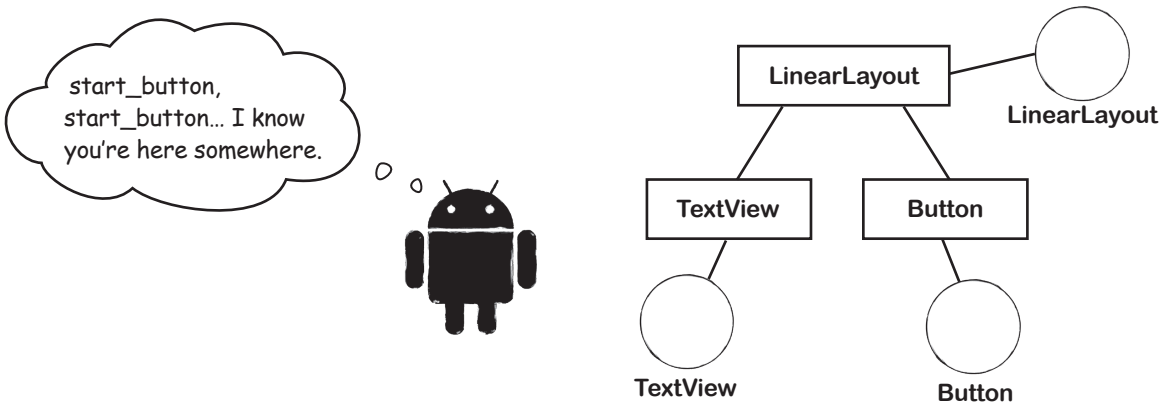
The following things happen when the above code runs:

1. **The MainActivity's layout file (activity_main.xml) is inflated into a hierarchy of View objects.**
   If the file describes a linear layout that contains a text view and a button, the layout gets inflated into `LinearLayout`, `TextView`, and `Button` objects. The `LinearLayout` is the root view of this hierarchy.

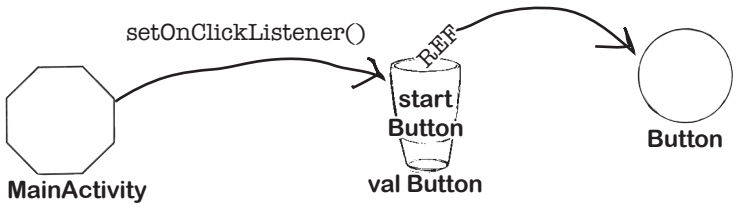


# The story continues

2 **Android searches the View hierarchy for a View with a matching ID.** We're using `findViewById<Button>(R.id.start_button)`, so Android searches the view hierarchy for a `View` whose ID is `start_button`.

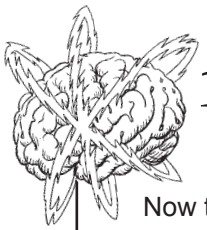



3 **Android returns the topmost View with this ID in the hierarchy, and casts it to the type specified in the call to findViewById().** Here, `findViewById<Button>(R.id.start_button)` finds the first `View` in the hierarchy with an ID of `start_button`, and casts it as a `Button`. `MainActivity` can now interact with it.




This is the startButton variable, which holds a reference to the Button returned by findViewById().

So each time you use `findViewById()`, Android searches the layout's hiearchy for a `View` with a matching ID, and casts it to the specified type.



## Brain Power

Now that you've seen how `findViewById()` works, what disadvantages can you see to its approach? What do you think could be done differently?

# There's a downside to findViewById()

Even though `findViewById()` is a useful way of getting references to views, it has several disadvantages.

- **It makes your code longer.**
  The more views you need to interact with, the more calls you need to make. This can lead to longer code that's harder to read.

- **It's inefficient.**
  Each time you call `findViewById()`, Android has to search the layout's hierarchy for a view with a matching ID. This is inefficient, particularly if your layout has many views in a deep hierarchy.

- **It's not null-safe.**
  `findViewById()` is used to search for a view at runtime, which means the compiler can't check for common errors.

  It's possible, for instance, to pass `findViewById()` an invalid ID: one that doesn't exist in the layout. If you try using:

```
val message = view.findViewById<EditText>(R.id.message)
```

  and the layout doesn't include a `View` with an ID of `message`, a null pointer exception will get thrown, and the app will crash.

- **It's not type-safe.**
  Another problem is that the compiler can't check that you've specified the `View`'s type correctly, which can lead to a class cast exception.

  Imagine you have a radio group named `pizza_group`, and you try to get a reference to it using this code:

```
val pizzaGroup = view.findViewById<ChipGroup>(R.id.pizza_group)
```

  Even though the `ChipGroup` type is specified instead of `RadioGroup`, the code still compiles. The compiler doesn't check whether the type is correct when it builds the code. When the app runs, it throws a class cast exception, and the app crashes.

So if `findViewById()` has these disadvantages, what's the alternative?

# View binding to the rescue

An alternative to calling `findViewById()` each time you need a `View` reference is to use **view binding**. With view binding, you set up a binding object (which you'll find out more about later) and use this to access each view.

View binding is part of Android Jetpack.

As an example, suppose you have a layout that includes a button with an ID of `start_button`. If you wanted to make it do something when clicked, you could get a reference to it using `findViewById()` like this:

```
val startButton = findViewById<Button>(R.id.start_button)
startButton.setOnClickListener {
    //Code that does something
}
```

This is the approach you've been using in the book so far.

With view binding, you no longer need to call `findViewById()` to get a reference to the button. Instead, you simply use the following:

```
binding.startButton.setOnClickListener {
    //Code that does something
}
```

With view binding, you can interact with the button using the binding object's startButton property. This property holds a reference to the view with an ID of start_button.

This code does the same thing, but it's simpler to write, and can make your code shorter and easier to read.

## View binding is safer and more efficient than findViewById()

When you use view binding, Android no longer needs to search your layout's view hierarchy for a matching `View`: it simply uses the binding object to access it. This makes it much more efficient than using `findViewById()`.

Another advantage is **the compiler prevents null pointer and class cast exceptions at compile time**. As you access views using a binding object, the compiler knows which views are available and their types. It won't let you reference a view that doesn't exist, and you no longer have to cast it to a specific type because the compiler already knows what this is. Your code is much safer as a result.

Now that you know about the benefits of using view binding, let's find out how to use it.

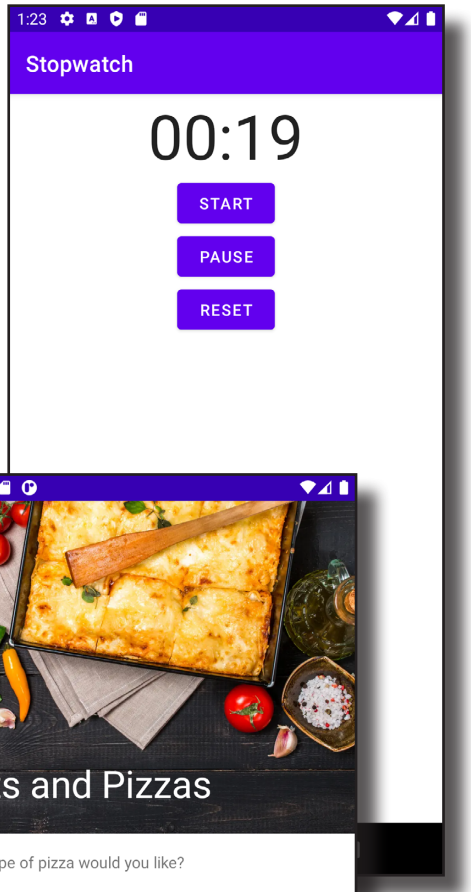
**View binding is a type-safe, more efficient alternative to calling findViewById().**

# Here's how we'll use view binding

View binding code is slightly different for activities and fragments, so in this chapter we're going to show you the code for both. Here are the steps we'll go through:

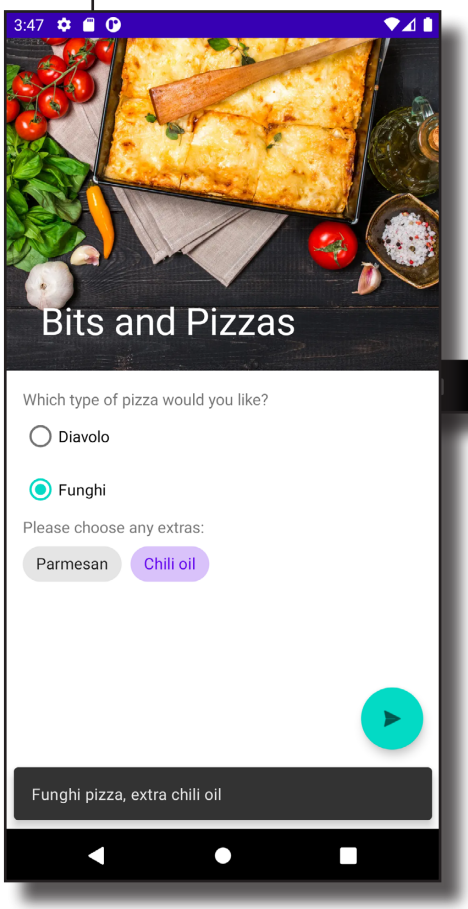
1. **Add view binding to the Stopwatch app's activity code.**
In Chapter 5, we built a Stopwatch app to teach you about Android's activity lifecycle methods. We'll revisit this app, and update its activity code so it uses view binding.

The Stopwatch app currently uses a single activity with no fragments. We'll change up the activity code so it uses view binding.




2. **Add view binding to the Bits and Pizzas app's fragment.**
We'll then go back to the Bits and Pizzas app we created in the previous chapter and show you how to implement view binding in its fragment code.

The Bits and Pizzas app uses fragment code to interact with its views. We'll change this code so it uses view binding.

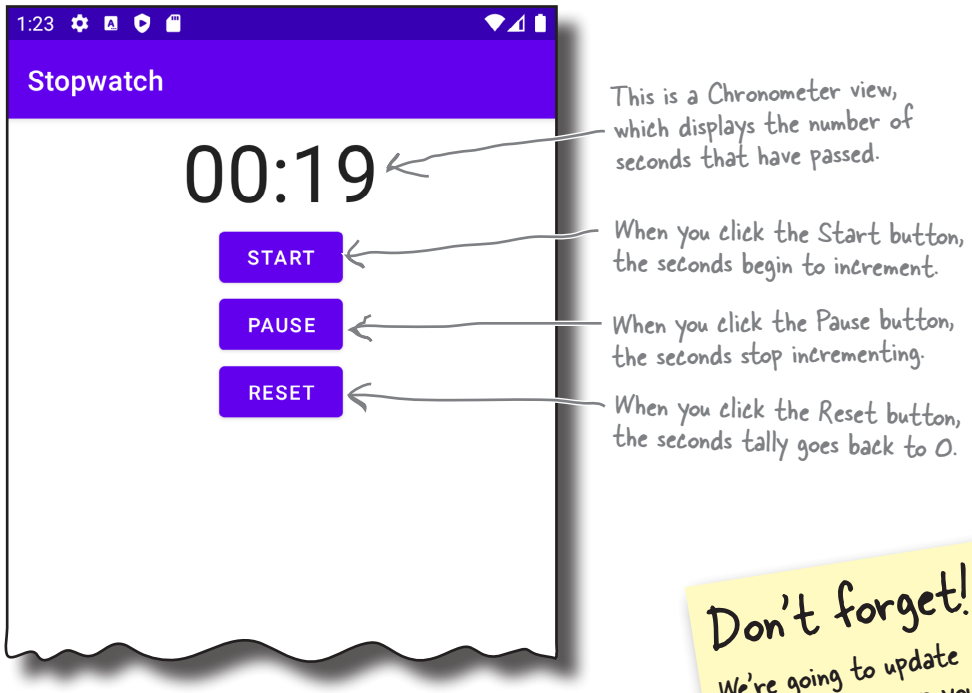



Let's get started.

# The Stopwatch app revisited

We'll begin by changing up the Stopwatch app you created in Chapter 5, so open the project for this app now.

As you may recall, the Stopwatch app displays a simple stopwatch that you can start, pause, and reset with the help of three buttons. It looks like this:




**Don't forget!**

We're going to update the Stopwatch app you created in Chapter 5, so make sure you open the project for this app.

The app uses a single activity—`MainActivity`—which has a layout file named *activity_main.xml*.

Each time it needs to interact with one of its views, it calls `findViewById()` to get a reference to it. To make its Start button respond to clicks, for example, it uses code like this:

```
val startButton = findViewById<Button>(R.id.start_button)
startButton.setOnClickListener {
    //Code that runs when the button's clicked
}
```

Let's find out how to update the app so that it uses view binding instead.

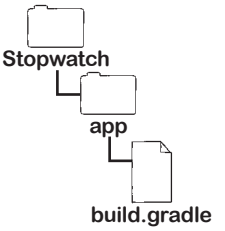
**Activity code**
**Fragment code**

# Enable view binding in the app build.gradle file

To use view binding, you first need to enable it in the `android` section of the app's *build.gradle* file. The code to enable view binding looks like this:

```
android {
    ...
    buildFeatures {
        viewBinding true
    }
}
```

Add these lines to the android section of the app's build.gradle file.

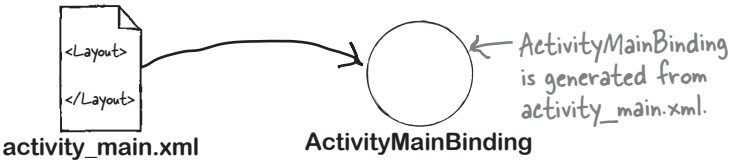



We're going to use view binding in the Stopwatch app, so make sure that you add the above change to the file *Stopwatch/app/build.gradle*. Then choose the Sync Now option to sync this change with the rest of your project.

Remember to sync this change, or you'll get errors in your code when you try to update the activity code.

## Enabling view binding generates code for each layout

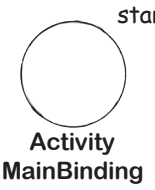
When you enable view binding, it automatically creates a binding class for each of the layout files in your app. The Stopwatch app, for example, contains a layout file named *activity_main.xml*, so when you enable view binding, it automatically generates a binding class named `ActivityMainBinding`:




Each binding class includes a property for each view in the layout that has an ID. The layout *activity_main.xml*, for example, includes a button with an ID of `start_button`, so the binding class `ActivityMainBinding` includes a property named `startButton` whose type is `Button`.

The binding classes are important because **the layout's views are bound to the binding class properties**. Instead of calling `findViewById()` each time you need a view reference, you simply interact with that view's property in the binding class.

Now that we've enabled view binding in the Stopwatch app, let's see how to use it in the code for `MainActivity`.




activity_main.xml includes a button whose ID is start_button, so ActivityMainBinding includes a Button property named startButton that references the Button object.

# How to add view binding to an activity

The code to make an activity use view binding is virtually identical for every activity you create. It looks like this:

```
package com.hfad.stopwatch

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import com.hfad.stopwatch.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        binding = ActivityMainBinding.inflate(layoutInflater)
        val view = binding.root
        setContentView(view)
    }
}
```

Import the binding class, in this example ActivityMainBinding.

Add this binding property.

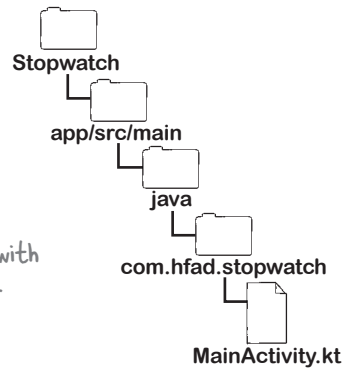
Replace this line with the code beneath.

Set the property.

This creates an ActivityMainBinding object that's linked to the layout.

Set view to the root view.

Pass the root view to setContentView().




The above code declares a property named `binding` whose type is `ActivityMainBinding`. This property gets set in the activity's `onCreate()` method using the following code:

```
binding = ActivityMainBinding.inflate(layoutInflater)
```

You need this line in order to get an ActivityMainBinding object.

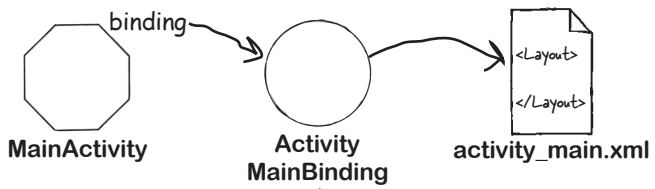
This calls `ActivityMainBinding`'s `inflate()` method, which creates an `ActivityMainBinding` object that's linked to the activity's layout.

The code:

```
val view = binding.root
setContentView(view)
```

gets a reference to the `binding` object's root view, and uses the `setContentView()` method to display it.

Once you've added view binding to an activity in this way, you can use the `binding` property to interact with the layout's views. Let's do this now.




MainActivity's binding property is set to an ActivityMainBinding object. The activity's views are bound to this object.

**Activity code**
**Fragment code**

# Use the binding property to interact with views

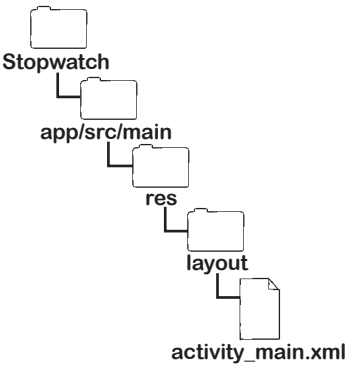
The current code for `MainActivity` interacts with its views in order to control the stopwatch and make its buttons respond to clicks. We can update this code so that instead of making calls to `findViewById()`, it accesses the views using the activity's `binding` property.

To see how this works, let's use `MainActivity`'s Start button as an example.

## The layout code

The Start button is defined in *activity_main.xml* using this code:

```
...
<Button
    android:id="@+id/start_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/start" />
...
```




As you can see, it has an ID of `start_button`.

## The activity code

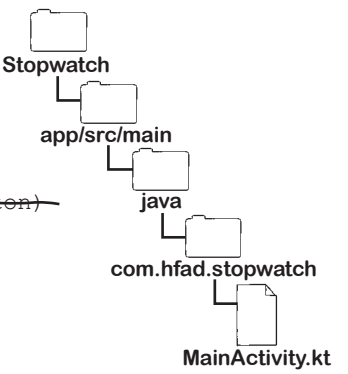
`MainActivity` makes the button respond to clicks using `findViewById()` like so:

```
val startButton = findViewById<Button>(R.id.start_button)
startButton.setOnClickListener {
    //Code that runs when the button's clicked
}
```

With view binding, we can change the code to:

We can use the binding property to access the start_button view.

```
~~val startButton = findViewById<Button>(R.id.start_button)~~
binding.startButton.setOnClickListener {
    //Code that runs when the button's clicked
}
```




The code does the same thing as the original code, but it uses `MainActivity`'s `binding` property to interact with the button.

Let's update the full code for `MainActivity` so it uses view binding.

# The full code for MainActivity.kt

Here's the updated code for `MainActivity`; update the code for *MainActivity.kt* so that it includes the changes below (in bold):

```
package com.hfad.stopwatch

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.os.SystemClock
import android.widget.Button
import android.widget.Chronometer
import com.hfad.stopwatch.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding
    lateinit var stopwatch: Chronometer  //The chronometer
    var running = false  //Is the chronometer running?
    var offset: Long = 0  //The base offset for the chronometer

    val OFFSET_KEY = "offset"
    val RUNNING_KEY = "running"
    val BASE_KEY = "base"

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        binding = ActivityMainBinding.inflate(layoutInflater)
        val view = binding.root
        setContentView(view)
        stopwatch = findViewById<Chronometer>(R.id.stopwatch)

        //Restore the previous state
        if (savedInstanceState != null) {
            offset = savedInstanceState.getLong(OFFSET_KEY)
            running = savedInstanceState.getBoolean(RUNNING_KEY)
            if (running) {
                binding.stopwatch.base = savedInstanceState.getLong(BASE_KEY)
                binding.stopwatch.start()
            } else setBaseTime()
        }
```

We no longer need these imports, so delete them.

Import the binding class.

Add the binding property.

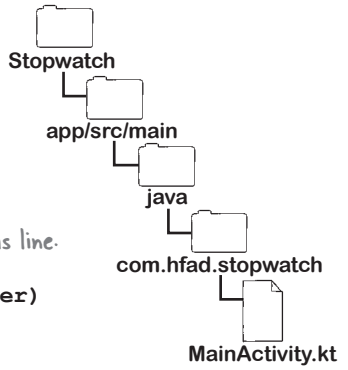
We originally added a stopwatch property, but thanks to view binding, we no longer need it.

Delete this line.

Add the view binding code.

We've removed the stopwatch property, so we don't need to set it here.

Use the binding property to access the stopwatch.




The code continues on the next page.

# MainActivity.kt continued

**Activity code**
Fragment code

```
        //The start button starts the stopwatch if it's not running
        val startButton = findViewById<Button>(R.id.start_button)
        binding.startButton.setOnClickListener {
            if (!running) {
                setBaseTime()
                binding.stopwatch.start()
                running = true
            }
        }

        //The pause button pauses the stopwatch
        val pauseButton = findViewById<Button>(R.id.pause_button)
        binding.pauseButton.setOnClickListener {
            if (running) {
                saveOffset()
                binding.stopwatch.stop()
                running = false
            }
        }

        //The reset button sets the offset and stopwatch to 0
        val resetButton = findViewById<Button>(R.id.reset_button)
        binding.resetButton.setOnClickListener {
            offset = 0
            setBaseTime()
        }
    }

    override fun onPause() {
        super.onPause()
        if (running) {
            saveOffset()
            binding.stopwatch.stop()
        }
    }
```

Remove this line.

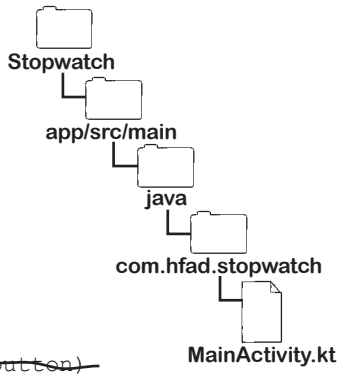
Use the binding property to access views.

Use the binding property here too.

Remove this line.

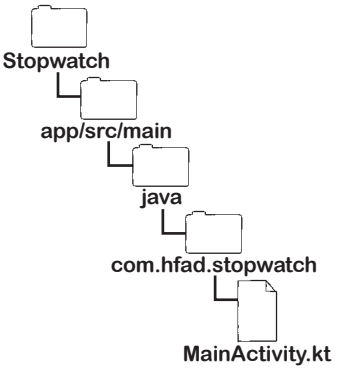
Use view binding here instead of calling findViewById().

Use view binding here as well.




The code continues on the next page.

# MainActivity.kt continued




```
        override fun onResume() {
            super.onResume()
            if (running) {
                setBaseTime()
                binding.stopwatch.start()
                offset = 0
            }
        }

        override fun onSaveInstanceState(savedInstanceState: Bundle) {
            savedInstanceState.putLong(OFFSET_KEY, offset)
            savedInstanceState.putBoolean(RUNNING_KEY, running)
            savedInstanceState.putLong(BASE_KEY, binding.stopwatch.base)
            super.onSaveInstanceState(savedInstanceState)
        }

        //Update the stopwatch base time, allowing for any offset
        fun setBaseTime() {
            binding.stopwatch.base = SystemClock.elapsedRealtime() - offset
        }

        //Record the offset
        fun saveOffset() {
            offset = SystemClock.elapsedRealtime() - binding.stopwatch.base
        }
    }
```

Use view binding here.

Use view binding here too.

We need to use the binding property here as well.

And here.

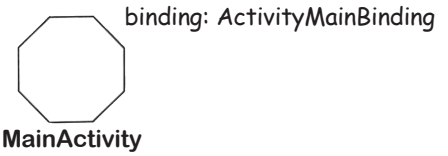
Those are all the changes we need to make to the Stopwatch app to get it to use view binding. Let's run through what happens when the code runs, and take it for a test drive.

**Activity code**
Fragment code

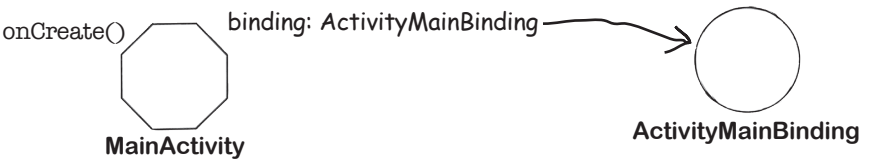
# What the code does

The following things happen when the app runs:

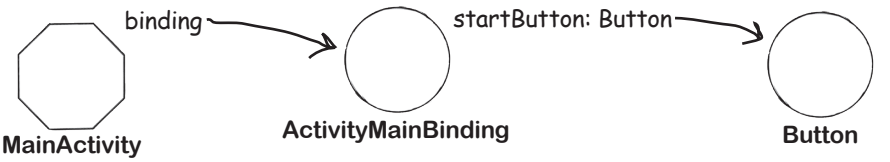
1. **When the app launches, MainActivity gets created.**
It includes an `ActivityMainBinding` property named `binding`.



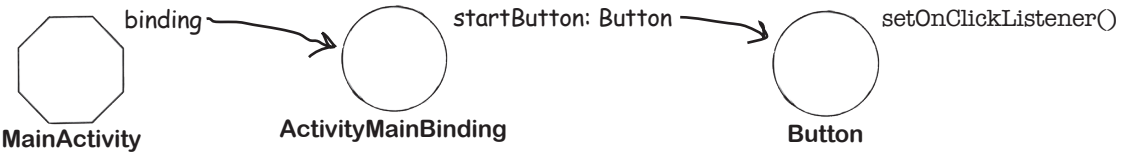

2. **When MainActivity's onCreate() method runs, it assigns an ActivityMainBinding object to the binding property.**
The `binding` property gets set in `onCreate()` because this is when `MainActivity` first has access to views.




3. **The ActivityMainBinding object includes a property for each view in the layout with an ID.**
The layout file *activity_main.xml* includes a button with an ID of `start_button`, for example, so the `ActivityMainBinding` object includes a `Button` property named `startButton`, which provides access to this view.



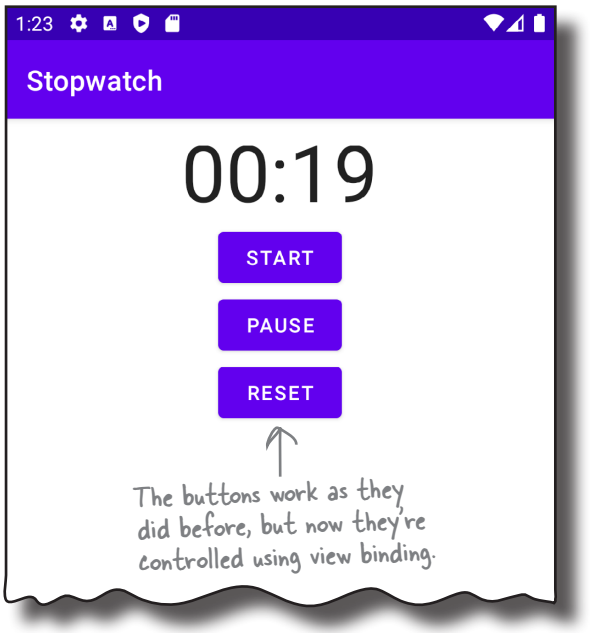

4. **MainActivity uses the binding property to access its views.**
It specifies how the Start button should respond to clicks, for example, by calling the `startButton` property's `setOnClickListener()` method.




Let's take the app for a test drive.

# Test Drive

When we run the app, it works in the same way as it did before. When we click on the Start button, the stopwatch starts, and it pauses and gets reset when we click on the Pause and Reset buttons.




The difference, however, is that the `MainActivity` now uses view binding to intract with its views instead of making calls to `findViewById()`.

## there are no Dumb Questions

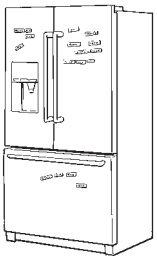
**Q: Why didn't you cover view binding earlier in the book?**

A: We wanted you to get more activity and fragment experience under your belt before introducing you to view binding. View binding makes some parts of the code a bit more complicated, and we felt it was important for you to get used to the basics first.

**Q: Is it possible to stop a binding class from being generated for a particular layout file?**

A: Yes. If you know that you won't use view binding with a particular layout, you can stop its binding class from being generated by adding to its root view:

```
tools:viewBindingIgnore="true"
```



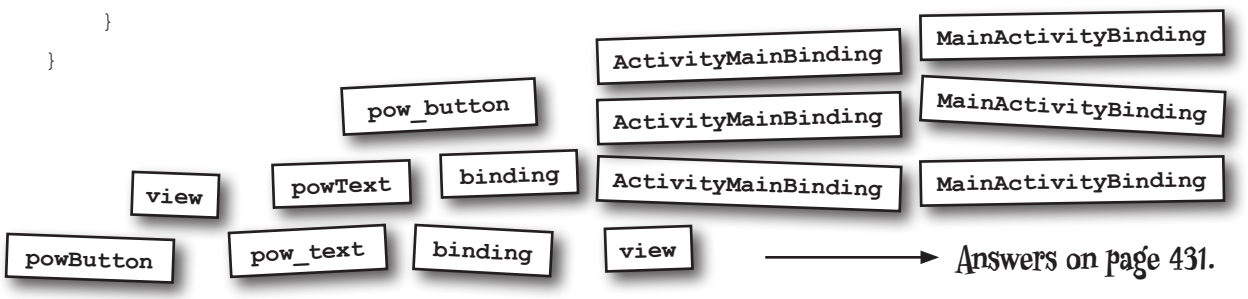
# Activity Magnets

An activity named `MainActivity` uses the file *activity_main.xml* for its layout. The layout includes a `Button` (with an ID of `pow_button`) and a `TextView` (with an ID of `pow_text`). When clicked, the `Button` needs to make the `TextView` display the text "Pow!"

Somebody used fridge magnets to write the `MainActivity` code, but a freak kitchen sandstorm dislodged some of the magnets. Can you piece the code back together again?

```
package com.hfad.myapp

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

import com.hfad.myapp.databinding._____

class MainActivity : AppCompatActivity() {

    private lateinit var binding:_____

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        binding = _____.inflate(layoutInflater)
        val view = binding.root
        setContentView(view)


        _____._____.setOnClickListener {

            _____._____.text = "Pow!"
        }
    }
}
```

You won't need to use all of these magnets.




Answers on page 431.

# Fragments can use view binding too (but the code's a little different)

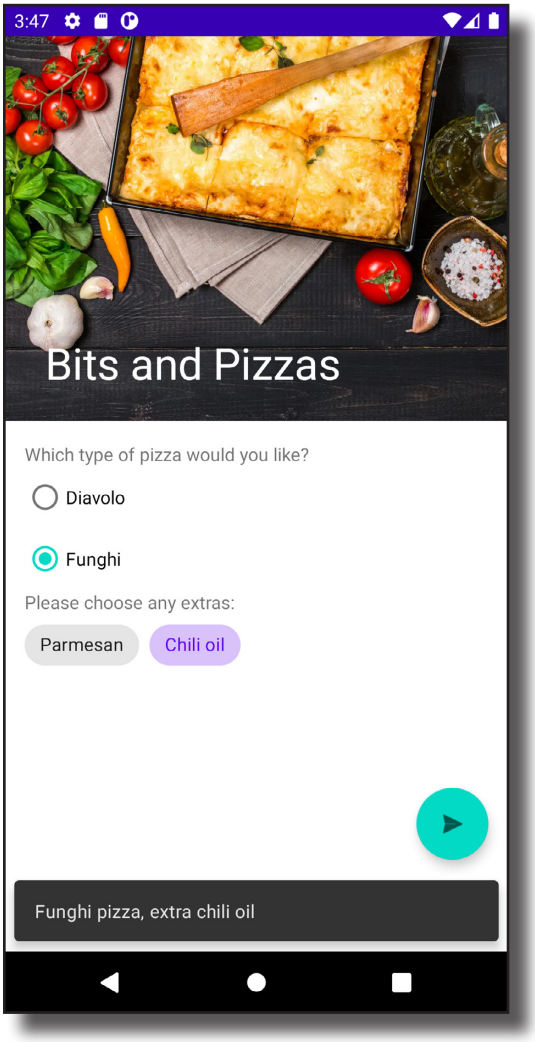
Now that you've learned how to implement view binding in your activity code, lets find out how to use it in fragments. As we said earlier, the code for fragments is slightly different to the code for activities.

## Let's implement view binding in the Bits and Pizzas app

We'll explore how view binding for fragments works by changing up the Bits and Pizzas app you created in Chapter 9. Open the project for this app now.

As we're sure you remember, the Bits and Pizzas app uses Material views such as collapsing toolbars and FABs to provide an interactive UI for a pizza app. The UI is defined by the app's fragment, which is named `FragmentOrder`.

Here's a reminder of what the Bits and Pizzas app looks like. It displays various views, which are controlled using fragment code. We can replace all of its calls to findViewByld() by implementing view binding.




The code for `FragmentOrder` includes many calls to `findViewById()`. Let's find out how to use view binding to replace these.

**Don't forget!**

Make sure you close the Stopwatch app and open the Bits and Pizzas app you created in Chapter 9 before continuing.

**Activity code**
**Fragment code**

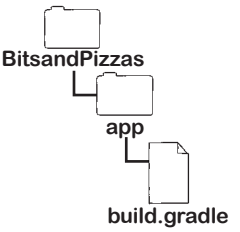
# Enable view binding for Bits and Pizzas

As before, to use view binding in the Bits and Pizzas project, you first need to enable it in the app's *build.gradle* file.

Open the file *BitsandPizzas/app/build.gradle*, and add the following lines to the `android` section:

```
android {
    ...
    buildFeatures {
        viewBinding true
    }
}
```

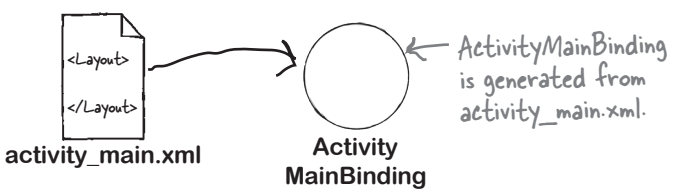
Add these lines to the android section of your app's build.gradle file.

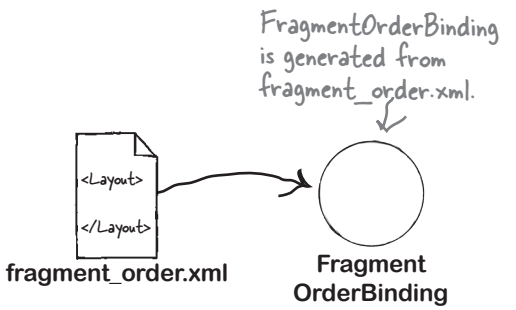



Then choose the Sync Now option to sync the change with the rest of your project.

## Binding classes are generated for every layout

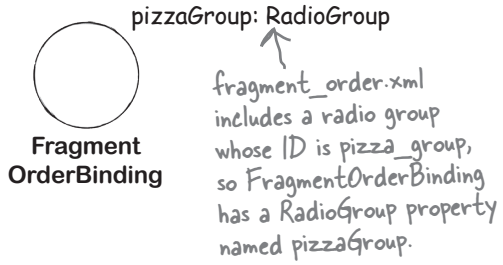
Just as with the Stopwatch app, enabling view binding in the Bits and Pizzas app automatically creates a binding class for each of the app's layout files. This app contains two layout files—*activity_main.xml* and *fragment_order.xml*—so two binding classes are generated: `ActivityMainBinding` and `FragmentOrderBinding`.




As before, each binding class includes a property for each view in its layout with an ID. The layout *fragment_order.xml*, for example, includes a radio group with an ID of `pizza_group`, so the binding class `FragmentOrderBinding` includes a property named `pizzaGroup` whose type is `RadioGroup`.

In the Bits and Pizzas app, the only code that interacts with views is in *OrderFragment.kt*, so we only need to update this file in order to use view binding.



# Fragment view binding code is a little different

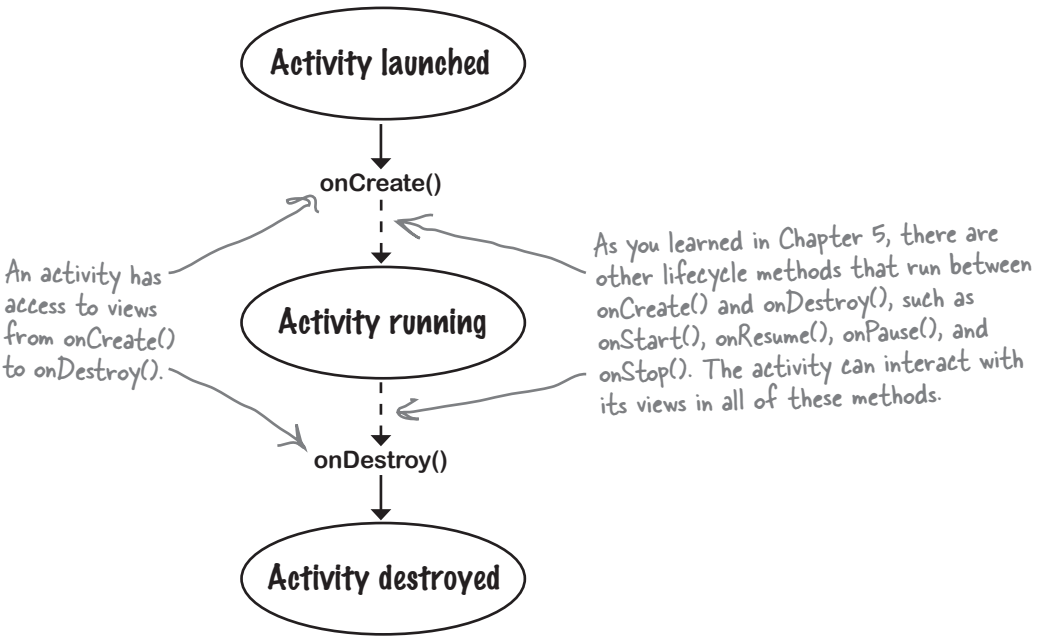
As we said earlier, the view binding code you use in a fragment is slightly different to the code you use in an activity. Before we show you what fragment view binding code looks like, we're going to dig a little deeper, and show you why the code is different.

## Activities can access views from onCreate()

As you already know, an activity first gets access to its layout when its `onCreate()` method runs. This is the first method in the activity's lifecycle, and it's used to inflate the layout (or bind to it with view binding) and perform any initial setup. If a button needs to respond to clicks, for example, the activity assigns an `OnClickListener` to it using code in its `onCreate()` method.

The activity continues to have access to its layout until its `onDestroy()` method has run. This is the final method in the activity's lifecycle, and the activity is destroyed after this method has finished running.

Because the activity has access to the views in its layout from `onCreate()` to `onDestroy()`, it can interact with them in any of its methods:




That's what happens with activities, but for fragments the situation's a little different. Let's see how.

**Activity code**
**Fragment code**

# Fragments can access views from onCreateView() to onDestroyView()

As you know, a fragment first gets access to its layout when its `onCreateView()` method runs. This gets called when an activity needs access to the fragment's layout, so it's used to inflate the layout and perform any initial setup such as setting `OnClickListeners`.

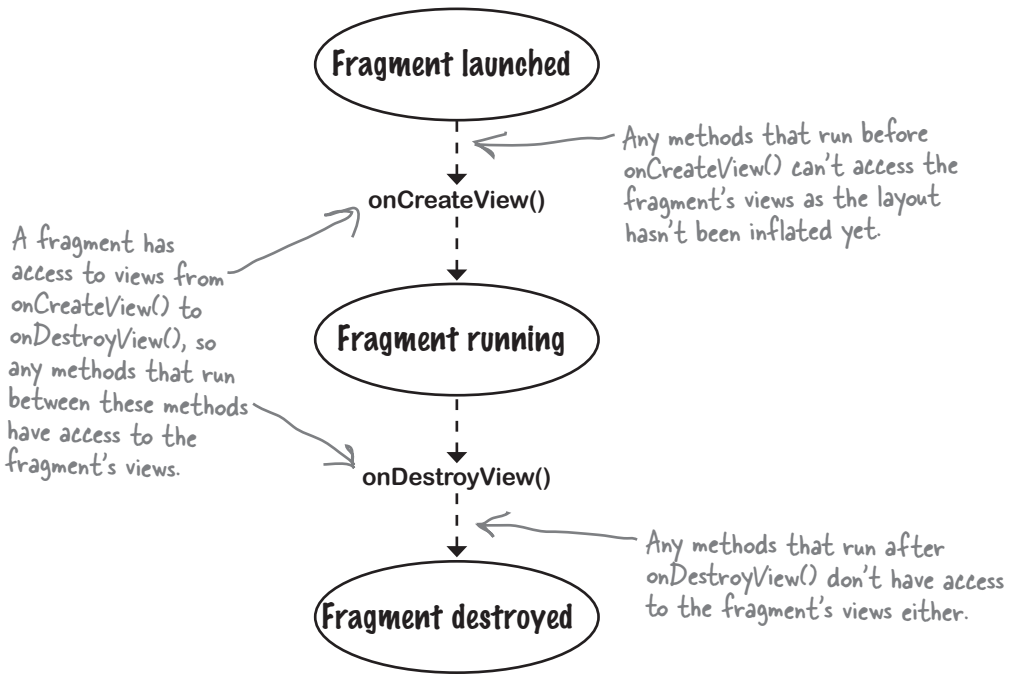
`onCreateView()`, however, isn't the first method in the fragment's lifecycle. There are other methods, such as `onCreate()`, that run before `onCreateView()`, so they can't interact with the fragment's views.

After its `onCreateView()` method has run, the fragment continues to have access to its views until its `onDestroyView()` method has finished running. This gets called when the activity no longer needs the fragment's layout, which may be because the activity needs to navigate to a different fragment, or it's being destroyed.

`onDestroyView()`, however, isn't the final method in the fragment's lifecycle. There are other methods, such as `onDestroy()`, that run after `onDestroyView()`, so these can't interact with the fragment's views either:

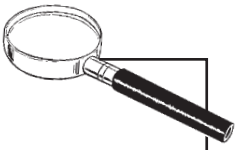
An activity can interact with its views from onCreate() to onDestroy().

A fragment can only interact with its views from onCreateView() to onDestroyView().




We'll show you an overview of the fragment lifecycle on the next page, and then see what fragment view binding code looks like.

# Fragment Lifecycle Methods Up Close



A fragment's lifecycle is very similar to an activity's, but it has a few extra steps. Here's when the main fragment lifecycle methods get called, what you use them for, and how they fit in with the state of the activity the fragment is displayed in:

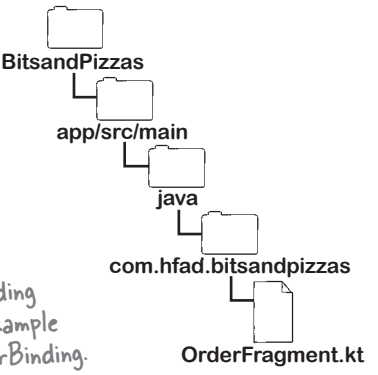
| Activity state | Fragment lifecycle method | |
|---|---|---|
| Created | onCreate() | **onCreate(Bundle)** This is called when the fragment is created. It can be used for any initial setup that doesn't involve views. |
| | onCreateView() | **onCreateView(LayoutInflater, ViewGroup, Bundle)** This is when the fragment first has access to its views. |
| Started | onStart() | **onStart()** This is called when the fragment is about to become visible. |
| Resumed | onResume() | **onResume()** This gets called when the fragment is actively running. |
| Paused | onPause() | **onPause()** This is called when the fragment is no longer actively running. |
| Stopped | onStop() | **onStop()** This gets called when the fragment is no longer visible. |
| Destroyed | onDestroyView() | **onDestroyView()** This is called when the fragment's view hierarchy is about to be destroyed so the fragment can clear resources. |
| | onDestroy() | **onDestroy()** This is when the fragment is about to be destroyed. Any additional resources can be cleared away at this point. |

**Activity code**
**Fragment code**

# What fragment view binding code looks like

You're now ready to see what fragment view binding code is like. Just like activities, you enable view binding in fragments by getting them to use a binding property, but the way in which you do this is slightly different.

Here's the code:




```
package com.hfad.bitsandpizzas

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.bitsandpizzas.databinding.FragmentOrderBinding

class OrderFragment : Fragment() {
    private var _binding: FragmentOrderBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
            inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        val view = inflater.inflate(R.layout.fragment_order, container, false)
        _binding = FragmentOrderBinding.inflate(inflater, container, false)
        val view = binding.root
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Import the binding class, in this example FragmentOrderBinding.

Add a backing property named _binding, which is set to null.

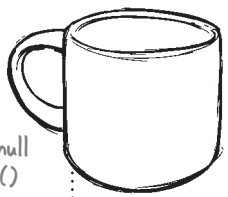
You access _binding using the binding property. If _binding is not null, it returns _binding. If _binding is null, trying to use it throws a null pointer exception.

Remove this line.

Set the _binding property.

This gets a FragmentOrderBinding object that's linked to the fragment's layout.

Set _binding to null in onDestroyView()



## Relax

**View binding code for fragments looks virtually identical for every fragment.**

Once you've included these changes, you can replace any `findViewById()` method calls as you did with activities.

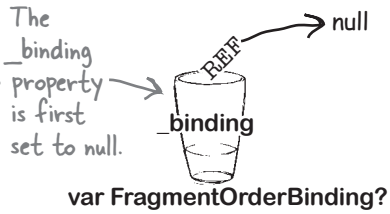
As you can see, the above code defines two extra properties: `binding` and `_binding`. Let's take a closer look at what these two properties do.

# _binding refers to the binding object...

As you've just seen, the view binding code for fragments defines a `_binding` property like this:

```
private var _binding: FragmentOrderBinding? = null
```

The _binding property is first set to null.

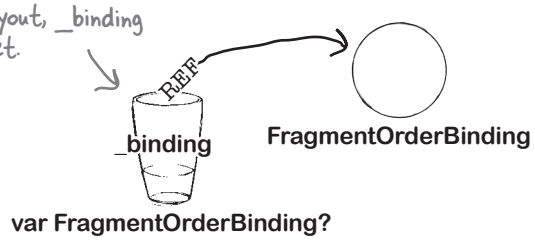



Its type is `FragmentOrderBinding?`, and it's initialized with `null`.

The `_binding` property is set to an instance of `FragmentOrderBinding` in the fragment's `onCreateView()` method using this code:

```
override fun onCreateView(...): View? {
    _binding = FragmentOrderBinding.inflate(inflater, container, false)
    ...
}
```

When the fragment has access to its layout, _binding is set to a FragmentOrderBinding object.




It's set in this method because it's when the fragment first has access to its views.

The `_binding` property is set back to `null` in the fragment's `onDestroyView()` method:

```
override fun onDestroyView() {
    super.onDestroyView()
    _binding = null
}
```

The fragment can't access its layout after onDestroyView() is called, so _binding is set to null again.

This is because the fragment can no longer access its views after its `onDestroyView()` method gets called.

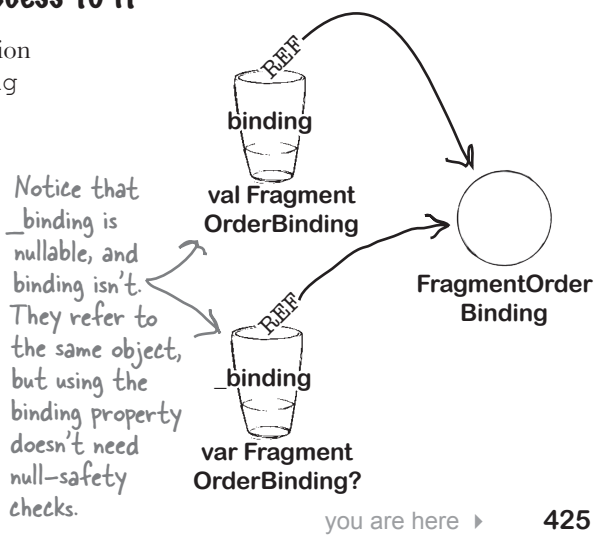
## ...and the binding property provides non-null access to it

The `binding` property uses a getter to return a non-`null` version of `_binding`, and throws a null pointer exception if `_binding` is `null`:

```
private val binding get() = _binding!!
```

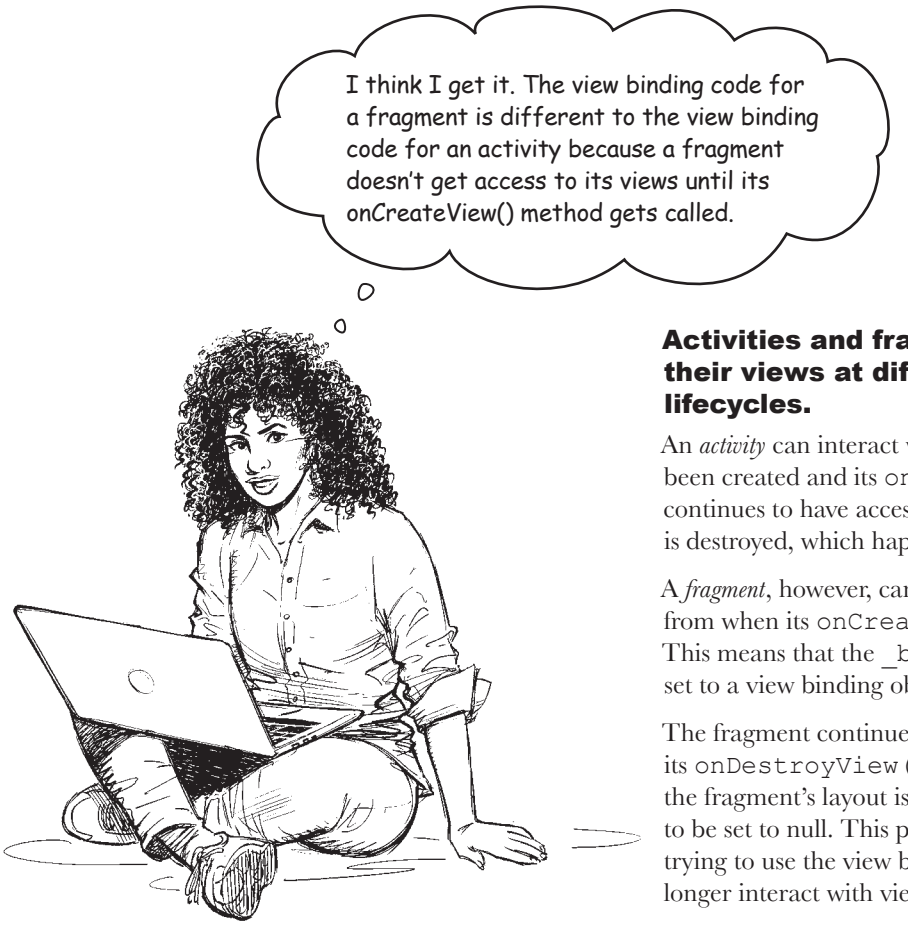
This means you can use the `binding` property to interact with the fragment's views without performing lots of messy null-safety checks. To make a fragment's FAB respond to clicks, for example, you simply use:

```
binding.fab.setOnClickListener {
    //Code that does something
}
```




Notice that _binding is nullable, and binding isn't. They refer to the same object, but using the binding property doesn't need null-safety checks.

**Activity code**
**Fragment code**




## Activities and fragments get access to their views at different points in their lifecycles.

An *activity* can interact with its views from when it's been created and its `onCreate()` method is called. It continues to have access to these views until the activity is destroyed, which happens at the end of its lifecycle.

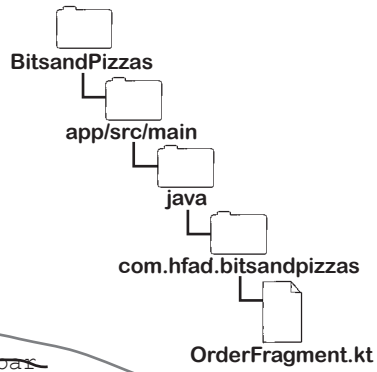
A *fragment*, however, can only interact with its views from when its `onCreateView()` method gets called. This means that the `_binding` property can *only* be set to a view binding object in this method.

The fragment continues to have access to its views until its `onDestroyView()` method runs. At this point, the fragment's layout is discarded, so `_binding` needs to be set to null. This prevents the fragment from trying to use the view binding object when it can no longer interact with views.

You now know everything that you need in order to use view binding with a fragment, so let's update the code for `OrderFragment` in the Bits and Pizzas app.

# The full code for OrderFragment.kt

Here's the code to make `OrderFragment` use view binding; update the code for *OrderFragment.kt* so it includes the changes shown here (in bold):




```
package com.hfad.bitsandpizzas

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.appcompat.app.AppCompatActivity
~~import com.google.android.material.appbar.MaterialToolbar~~
~~import android.widget.RadioGroup~~
~~import com.google.android.material.chip.Chip~~
~~import com.google.android.material.floatingactionbutton.FloatingActionButton~~
import android.widget.Toast
import com.google.android.material.snackbar.Snackbar
import com.hfad.bitsandpizzas.databinding.FragmentOrderBinding

class OrderFragment : Fragment() {
    private var _binding: FragmentOrderBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        ~~val view = inflater.inflate(R.layout.fragment_order, container, false)~~
        _binding = FragmentOrderBinding.inflate(inflater, container, false)
        val view = binding.root
        ~~val toolbar = view.findViewById<MaterialToolbar>(R.id.toolbar)~~
        (activity as AppCompatActivity).setSupportActionBar(binding.toolbar)
```

We no longer need these imports, so delete them.

Import the binding class.

Declare the _binding and binding properties.

Delete this line.

Set the _binding property.

Use view binding instead of findViewById().

The code continues on the next page.

**Activity code**
**Fragment code**

# OrderFragment.kt continued

```
        ~~val fab = view.findViewById<FloatingActionButton>(R.id.fab)~~
        binding.fab.setOnClickListener {
            ~~val pizzaGroup = view.findViewById<RadioGroup>(R.id.pizza_group)~~
            val pizzaType = binding.pizzaGroup.checkedRadioButtonId
            if (pizzaType == -1) {
                val text = "You need to choose a pizza type"
                Toast.makeText(activity, text, Toast.LENGTH_LONG).show()
            } else {
                var text = (when (pizzaType) {
                    R.id.radio_diavolo -> "Diavolo pizza"
                    else -> "Funghi pizza"
                })
                ~~val parmesan = view.findViewById<Chip>(R.id.parmesan)~~
                text += if (binding.parmesan.isChecked) ", extra parmesan" else ""
                ~~val chiliOil = view.findViewById<Chip>(R.id.chili_oil)~~
                text += if (binding.chiliOil.isChecked) ", extra chili oil" else ""
                Snackbar.make(binding.fab, text, Snackbar.LENGTH_LONG).show()
            }
        }
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Replace findViewById() here with view binding.

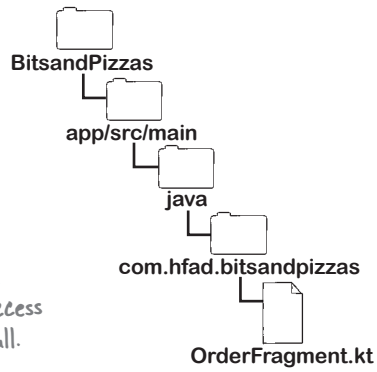
Delete this line.

Use view binding here too.

Replace these calls to findViewById() with view binding.

Access the FAB using binding.

Override the onDestroyView() lifecycle method so that when the fragment can no longer access its views, _binding is set to null.



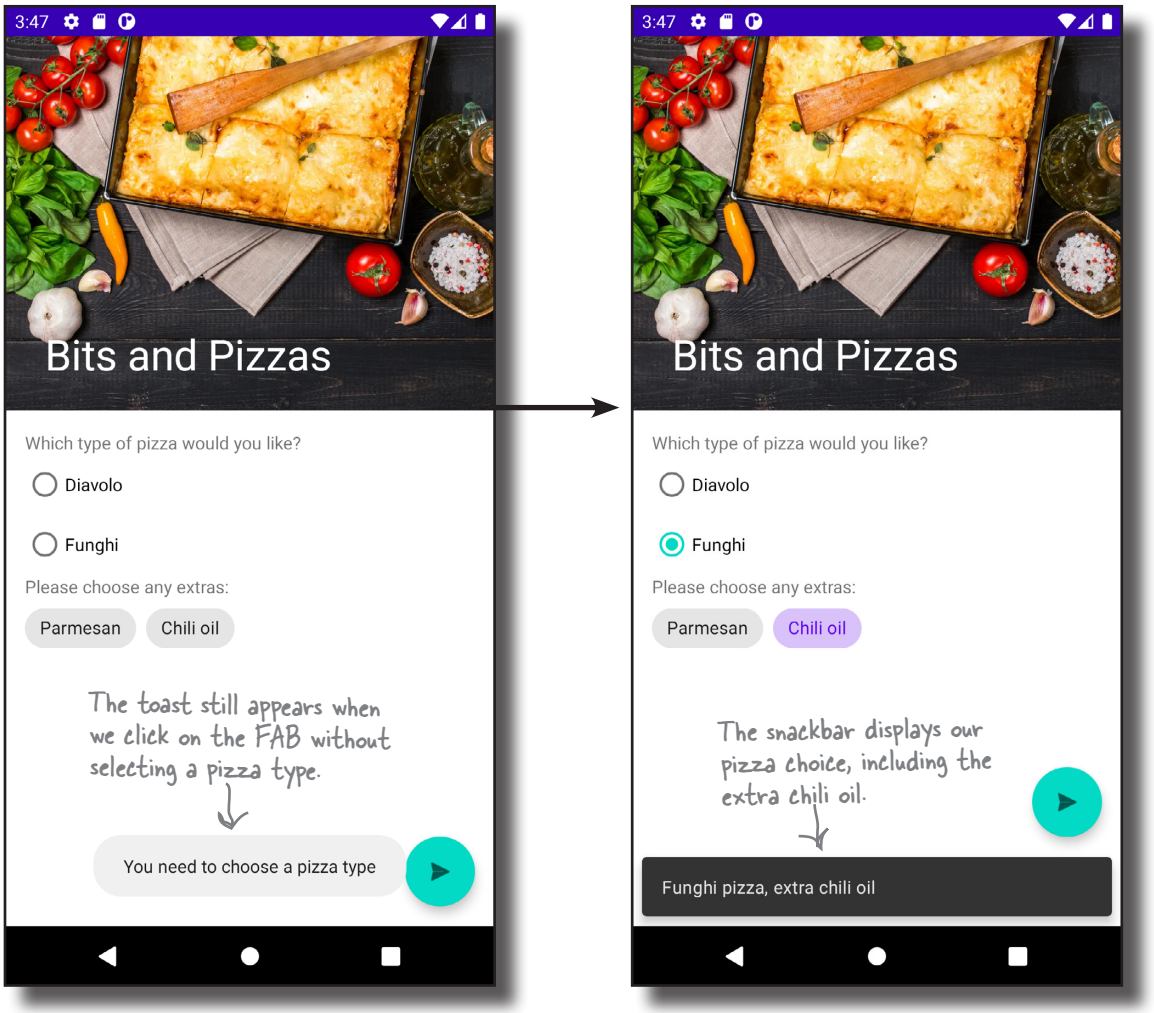

That's all the code we need to make `OrderFragment` use view binding. Let's take the app for a test drive, and make sure that it still works.

# Test Drive

When we run the app, it works in the same way as it did before.

When we click on the FAB without selecting a pizza type, a toast asks us to choose a pizza.

When we select a pizza type and click on the FAB again, a snackbar appears at the bottom of the screen showing us details of our order.




Congratulations! You now know how to implement view binding in your activity and fragment code. We'll continue to use view binding through the rest of this book.

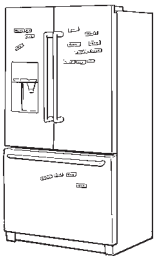
# Exercise

The fragment code below makes a FAB (with an ID of `fab`) respond to clicks. When clicked, a snackbar is displayed on the device screen. The snackbar includes an action that displays a toast.

Can you update the fragment so that it uses view binding?

```
... ← We've left out the package declaration and import statements..
class MyFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        val view = inflater.inflate(R.layout.fragment_my, container, false)
        val fab = view.findViewById<FloatingActionButton>(R.id.fab)
        fab.setOnClickListener {
            Snackbar.make(fab, "Your order has been updated", Snackbar.LENGTH_SHORT)
                    .setAction("Undo") {
                        Toast.makeText(activity, "Undone!", Toast.LENGTH_SHORT)
                        .show()
                    }
                    .show()
        }
        return view
    }
}
```

Answers on page 432.



# Activity Magnets Solution

An activity named `MainActivity` uses the file *activity_main.xml* for its layout. The layout includes a `Button` (with an ID of `pow_button`) and a `TextView` (with an ID of `pow_text`). When clicked, the `Button` needs to make the `TextView` display the text "Pow!"

Somebody used fridge magnets to write the `MainActivity` code, but a freak kitchen sandstorm dislodged some of the magnets. Can you piece the code back together again?

```
package com.hfad.myapp

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

import com.hfad.myapp.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        binding = ActivityMainBinding.inflate(layoutInflater)
        val view = binding.root
        setContentView(view)

        binding.powButton.setOnClickListener {
            binding.powText.text = "Pow!"
        }
    }
}
```

Import the ActivityMainBinding class.

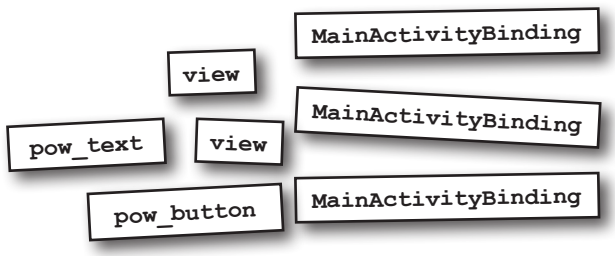
The binding property's type needs to be ActivityMainBinding.

Set the binding property.

Use view binding to set the button's OnClickListener.

Use view binding to set the text view's text.

You didn't need to use these magnets.



# Exercise Solution

The fragment code below makes a FAB (with an ID of `fab`) respond to clicks. When clicked, a snackbar is displayed on the device screen. The snackbar includes an action that displays a toast.

Can you update the fragment so that it uses view binding?

```
...
class MyFragment : Fragment() {
    private var _binding: FragmentMyBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        val view = inflater.inflate(R.layout.fragment_my, container, false)
        _binding = FragmentMyBinding.inflate(inflater, container, false)
        val view = binding.root

        val fab = view.findViewById<FloatingActionButton>(R.id.fab)
        binding.fab.setOnClickListener {
            Snackbar.make(binding.fab, "Your order has been updated",
                          Snackbar.LENGTH_SHORT)
                .setAction("Undo") {
                    Toast.makeText(activity, "Undone!", Toast.LENGTH_SHORT)
                    .show()
                }
                .show()
        }
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Declare _binding, set it to null, and add a binding getter.

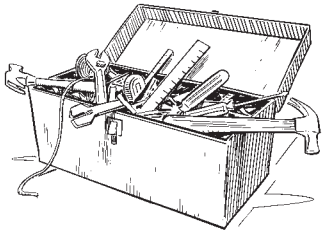
Get rid of this line as we want to use view binding

Set _binding.

Use view binding to access the FAB instead of using findViewById().

Override onDestroyView().

# Your Android Toolbox



**You've got Chapter 10 under your belt and now you've added view binding to your toolbox.**

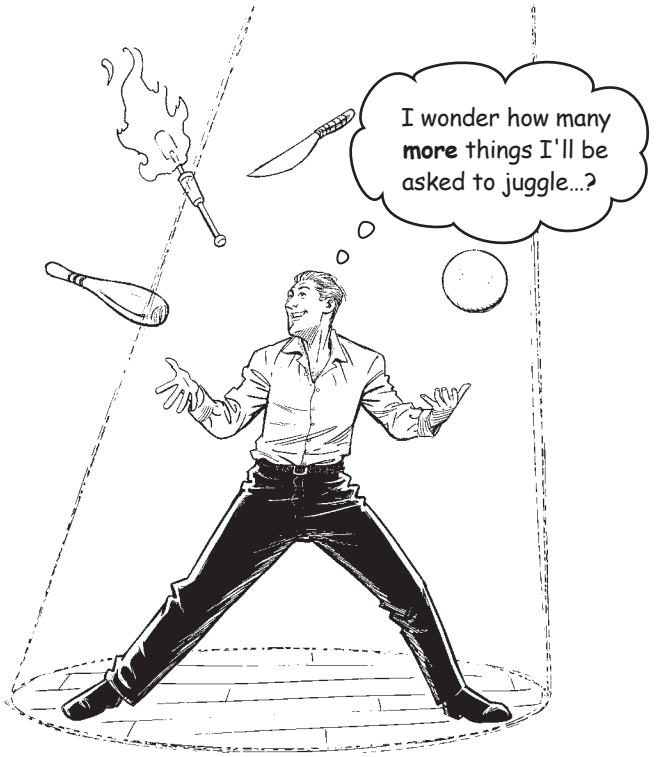
You can download the full code for the chapter from tinyurl.com/hfad3.

## Bullet Points

- View binding is a type-safe way of interacting with views. It's more efficient than using `findViewById()`, and less likely to produce errors at runtime.
- View binding can be used in activity and fragment code.
- Enable view binding in the app's *build.gradle* file.
- View binding generates a binding class for each of the app's layout files.
- Each binding class includes a property for each view in the layout with an ID.
- When you use view binding with an activity, you set its binding property in the activity's `onCreate()` method.
- When you use view binding with a fragment, you set its binding property in the fragment's `onCreateView()` method, and set it to null in `onDestroyView()`.
- Fragments have lifecycle methods that are linked to an activity's state.

# 11 view models

# Model Behavior

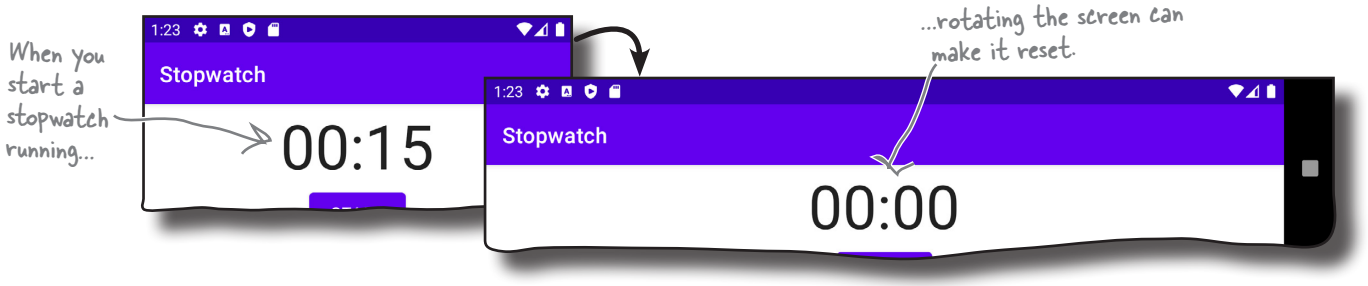



**As apps grow more complex, fragments have more to juggle.**

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.

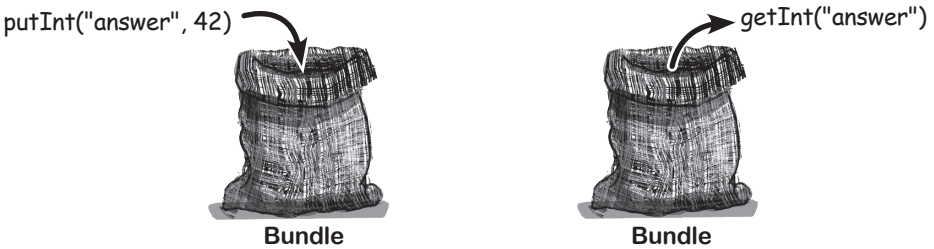
# Configuration changes revisited

As you learned in Chapter 5, Bad Things can happen when you rotate the screen while running an app. Changing the screen orientation is a configuration change, which causes Android to destroy and recreate the current activity. As a result, views and properties can lose their state and get reset:




In Chapter 5, you learned how to handle this using the activity's `onSaveInstanceState` method. This method gets triggered before the activity gets destroyed, and it's used to save any values that may get lost in a `Bundle`. When the activity gets recreated, you can use these saved values to restore the state of the activity's views and properties.

Fragments have an onSaveInstanceState method too. Even though we've only shown you how to use this approach with an activity, it works with fragments as well.




Using a `Bundle` to save state works well for relatively simple apps, but it's not an ideal solution for more complicated ones. This is because `Bundles` are only intended to hold a small amount of data, and only work with a limited number of types.

## There are other problems, too

Another problem apps can have is that the activity and fragment code can quickly become bloated. The code may need to control navigation, update the UI, save state, and also include more general business logic to control the app's behavior. Having all of this in one place makes the code longer, which can then be harder to read and maintain.

In this chapter we're going to learn how to solve all of these problems using a **view model**.

# Introducing the view model

A view model is a separate class that sits alongside the activity or fragment code. It's responsible for all of the data that needs to be displayed on the screen, along with any business logic. Whenever a fragment needs to update its layout, it asks the view model for the latest values it needs to display, and if it needs access to some business logic, it calls the methods held in the view model.

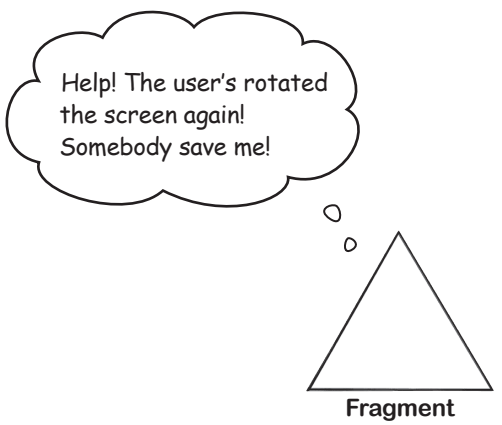
View models are another part of Android Jetpack.

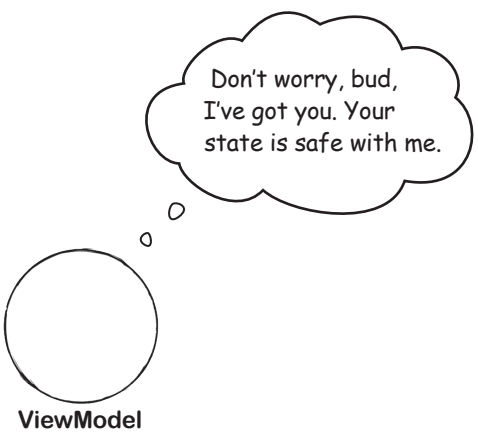
## Why use a view model?

There are a couple reasons why you might want to use a view model.

Using a view model simplifies your activity or fragment code. Your fragment no longer needs to include code relating to the app's business logic as this is all held in a separate class. Instead, it can focus on things like updating the screen or navigation.

Another reason is that **the view model can survive configuration changes**. It doesn't get destroyed when the user rotates the device screen, so the state of any variables doesn't get lost. This lets you restore the state of your app without having having to store values in a `Bundle`.




We're going to find out how to use view models by building a word guessing game. Before we start writing the code, let's go through how the game will work.

## there are no Dumb Questions

**Q: You said `Bundles` are only intended to hold a limited number of data types. Does that mean there's no way of adding extra types?**

A: Technically, you can add extra types to a `Bundle` using an interface named `Parcelable`. We're not teaching you this approach because we think it's better to use a view model.

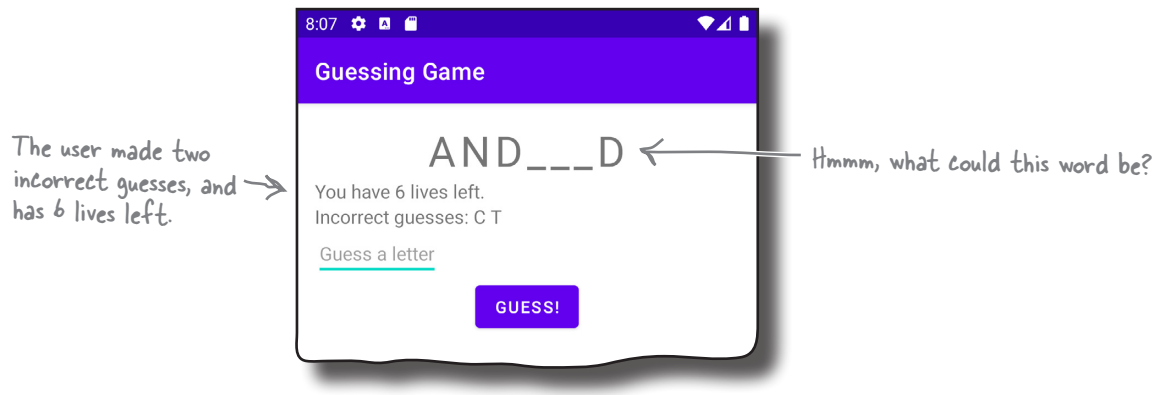
If you're feeling curious, however, you can find out more about using `Parcelable` in the Android documentation.

# What the guessing game will do

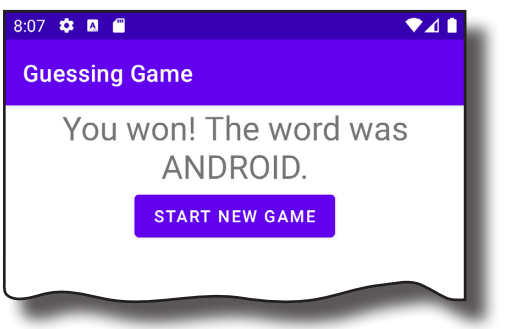
The aim of the Guessing Game app is for the user to try and guess a secret word.

When the game starts, it chooses a word at random from an array, and displays a blank for each of the word's letters:

The user suggests a letter that she thinks is in the secret word. If she guesses correctly, the game displays the letter where it appears in the secret word. If she guesses incorrectly, the game displays the incorrect guess, and she loses a life.




The user continues to make guesses until she finds all of the letters, or runs out of lives. When this happens, a new screen appears, telling her what the secret word was, and whether she won or lost the game. She's given the option to start another game.

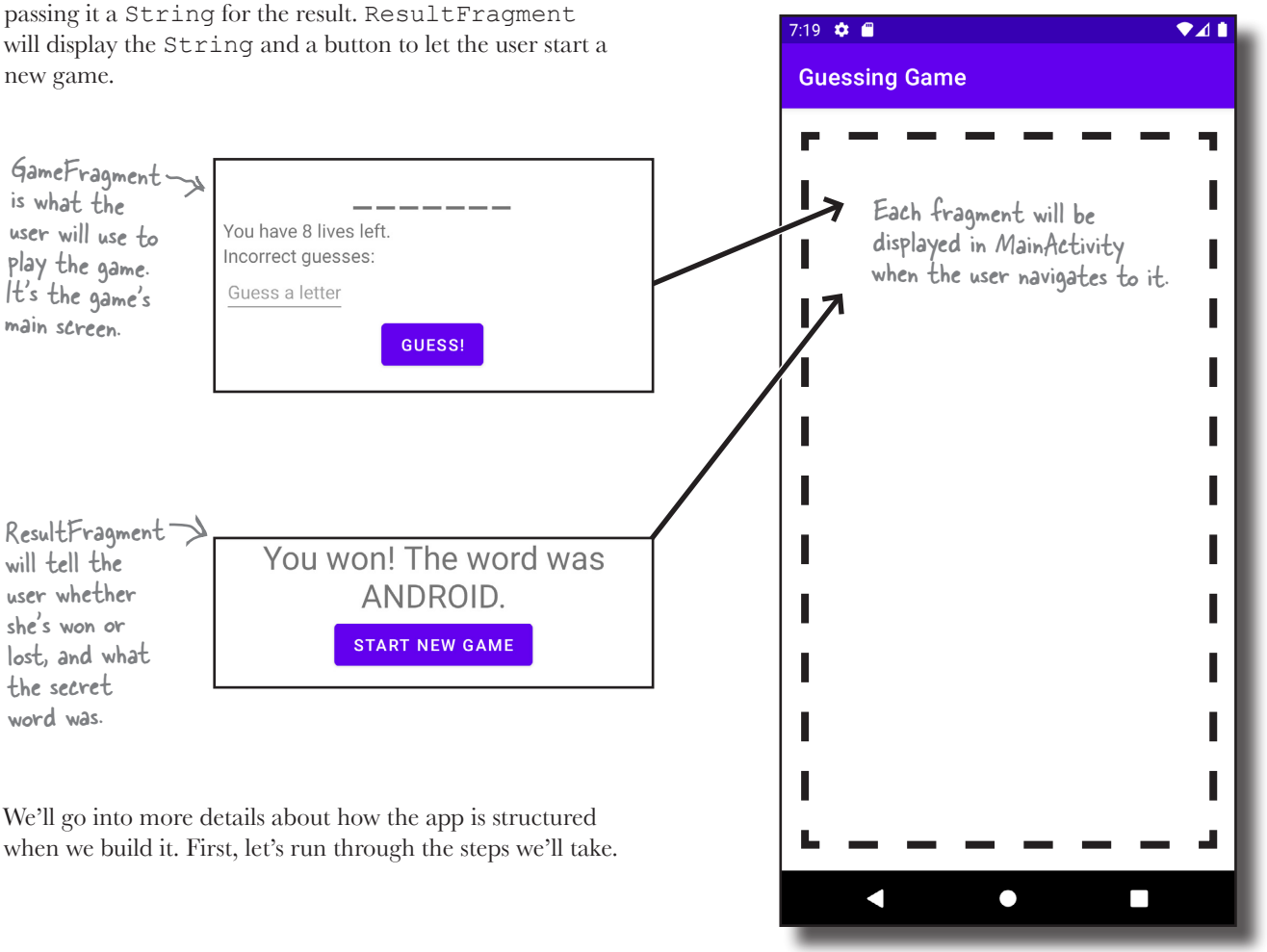


# How the app will be structured

The app will contain a single activity (named `MainActivity`), which will be used to display the game's fragments: `GameFragment` and `ResultFragment`.

`GameFragment` is the game's main screen. It will display blanks for the secret word's letters, let the user make guesses, and show any wrong choices and the number of lives left.

When the game is over, `GameFragment` will use the Navigation component to navigate to `ResultFragment`, passing it a `String` for the result. `ResultFragment` will display the `String` and a button to let the user start a new game.

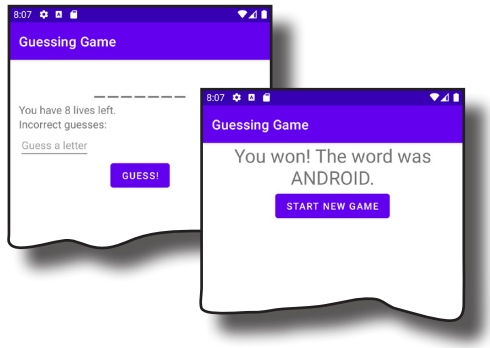



We'll go into more details about how the app is structured when we build it. First, let's run through the steps we'll take.

# Here's what we're going to do

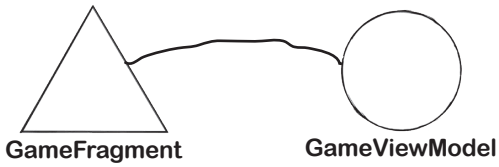
Here are the steps that we'll go through to write the app:

1. **Write the basic game.**
We'll create `GameFragment` and `ResultFragment`, write the game logic, and use the Navigation component to navigate between the two fragments.

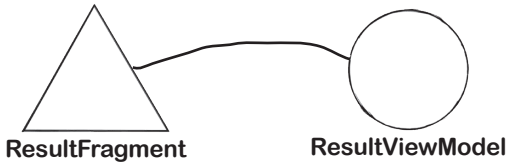



It will take a few pages to build the basic game. Think of it as revision for things you've already learned.

2. **Add a view model for GameFragment.**
We'll create a view model, `GameViewModel`, to hold `GameFragment`'s game logic and data. This will simplify the `GameFragment` code, and ensure that the game survives configuration changes.




3. **Add a view model for ResultFragment.**
We'll add a second view model, `ResultViewModel`, for `ResultFragment` to use. This view model will hold the result of the game that the user's just played.




## Create the Guessing Game project

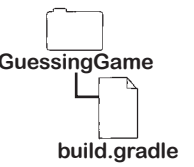
We're going to use a new project for the Guessing Game app, so create one now using the same steps you used in the previous chapters. Choose the Empty Activity option, enter a name of "Guessing Game" and a package name of "com.hfad.guessinggame", and accept the default save location. Make sure the language is set to Kotlin and the minimum SDK is API 21 so it will run on most Android devices.

The Guessing Game app will use view binding to reference its views, and the Navigation component to navigate between its fragments. Let's update the project and app *build.gradle* files to include these libraries.

# Update the project build.gradle file...

We're going to add a new variable to the project's *build.gradle* file to specify which version of the Navigation component we'll be using, along with a Safe Args classpath.

Open the file *GuessingGame/build.gradle*, and add the following lines (in bold) to the sections shown:




```
buildscript {
    ext.nav_version = "2.3.5"
    ...
    dependencies {
        classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version"
        ...
    }
}
```

We're holding the version number in nav_version to keep things nice and consistent.

Add the Safe Args classpath.

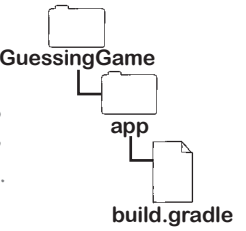
## ...and update the app build.gradle file too

In the app's *build.gradle* file, we need to enable view binding, add a dependency to the Navigation component library, and apply the Safe Args plug-in so that we can pass arguments to fragments.

Open the file *GuessingGame/app/build.gradle*, and add the following lines (in bold) to the appropriate sections:

```
plugins {
    ...
    id 'androidx.navigation.safeargs.kotlin'
}

android {
    ...
    buildFeatures {
        viewBinding true
    }
}

dependencies {
    ...
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
    ...
}
```

Apply this plug-in.

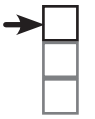



Enable view binding so we don't need to keep calling findViewById().

Add a dependency for the Navigation component.

Once you've made these changes, click on the Sync Now option to sync the changes you've made with the rest of your project.

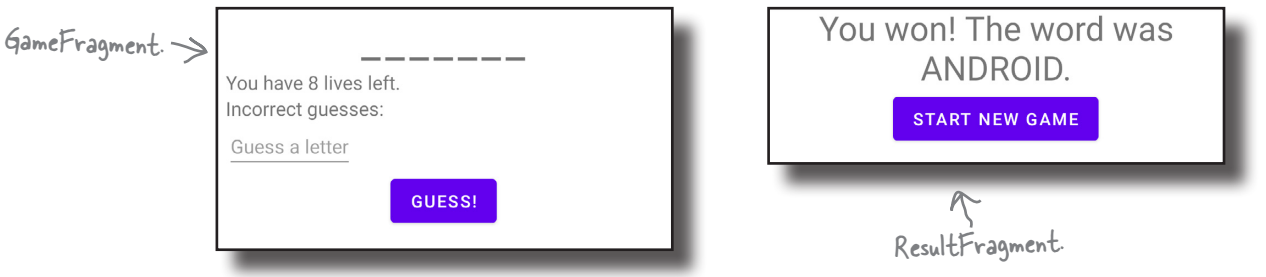
Next, we'll create the app's fragments.



**Basic game**
**GameFragment**
**ResultFragment**

# The Guessing Game app has two fragments

The Guessing Game app requires two fragments: `GameFragment` and `ResultFragment`. `GameFragment` is the app's main screen, and `ResultFragment` will be used to display the result:




Let's go ahead and add the two fragments to the project.

## Create GameFragment...

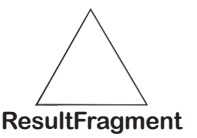
We'll add `GameFragment` to the project first.

Highlight the *com.hfad.guessinggame* package in the *app/src/main/java* folder, then go to File→New→Fragment→Fragment (Blank). Name the fragment "GameFragment", name its layout "fragment_game", and make sure the language is set to Kotlin.

Don't worry about the code for these fragments just yet. We'll update them after we've added them to a navigation graph.

## ...and then create ResultFragment

Next, add `ResultFragment` by highlighting the *com.hfad.guessinggame* package in the *app/src/main/java* folder again, and choosing File→New→Fragment→Fragment (Blank). This time, name the fragment "ResultFragment", name its layout "fragment_result", and make sure the language is set to Kotlin.




We'll update the code for these two fragments a few pages ahead. Before that, we'll add a navigation graph to the project, which will tell the app how to navigate between the two fragments.

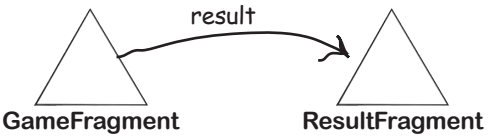
# How navigation should work

As you know, a navigation graph tells Android about an app's possible destinations, and how to navigate to them.

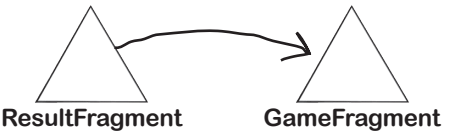
In the Guessing Game app, we want navigation to work as follows:

1. **When the app gets launched, display GameFragment.**

2. **When a game is won or lost, GameFragment navigates to ResultFragment, passing it the result.**




3. **When the user clicks on the New Game button, navigate to GameFragment.**




To implement this, we'll add `GameFragment` and `ResultFragment` to a new navigation graph (which we'll create), and specify that each fragment can navigate to the other.

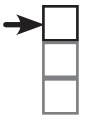
We'll also say that `GameFragment` will pass a `String` argument to `ResultFragment`, specifying a message that indicates whether the user has won or lost the game they've just played. `ResultFragment` will display this message when it's navigated to.

## Create the navigation graph

To create the navigation graph, select the *GuessingGame/app/src/main/res* folder in the project explorer, and then choose File→New→Android Resource File. When prompted, enter a file name of "nav_graph", choose a resource type of "Navigation", and click on OK. This creates a navigation graph named *nav_graph.xml*.

We need to update the navigation graph to include `GameFragment` and `ResultFragment`, along with navigation actions for each one. We'll do this next.

The navigation graph holds details of all the app's possible destinations, and how to get to them.



**Basic game**
GameFragment
ResultFragment

# Update the navigation graph

To update the navigation graph, open the file *nav_graph.xml* (if it's not already open), switch to the Code view, and update its code to include the changes shown here (in bold):

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/gameFragment">
    <fragment
        android:id="@+id/gameFragment"
        android:name="com.hfad.guessinggame.GameFragment"
        android:label="fragment_game"
        tools:layout="@layout/fragment_game" >
        <action
            android:id="@+id/action_gameFragment_to_resultFragment"
            app:destination="@id/resultFragment"
            app:popUpTo="@id/gameFragment"
            app:popUpToInclusive="true" />
    </fragment>
    <fragment
        android:id="@+id/resultFragment"
        android:name="com.hfad.guessinggame.ResultFragment"
        android:label="fragment_result"
        tools:layout="@layout/fragment_result" >
        <argument
            android:name="result"
            app:argType="string" />
        <action
            android:id="@+id/action_resultFragment_to_gameFragment"
            app:destination="@id/gameFragment" />
    </fragment>
</navigation>
```

Make sure the file includes this namespace.

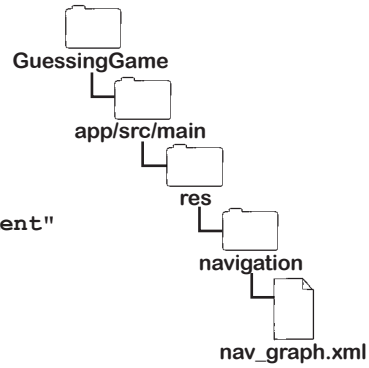
The game will start with GameFragment.

This action lets GameFragment navigate to ResultFragment.

This action pops fragments up to and including GameFragment off the back stack.

ResultFragment accepts a String argument.

This action lets ResultFragment navigate to GameFragment.

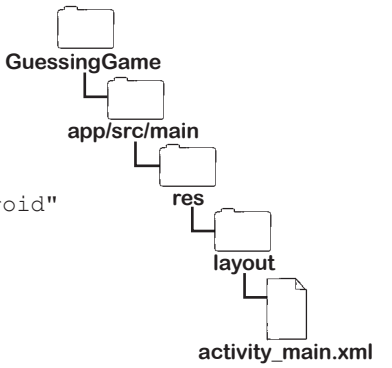



Now that you've updated the navigation graph, let's link it to `MainActivity` so that it displays each fragment when we navigate to it.

# Display the current fragment in MainActivity's layout

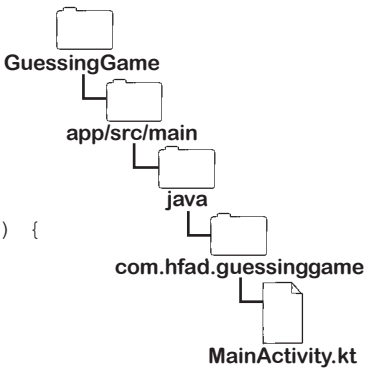
To display each fragment, we need to add a navigation host to `MainActivity`'s layout that's linked to the navigation graph we just created. We'll do this using a `FragmentContainerView`, just as we did in previous chapters.

You're already familiar with the code to do this, so update *activity_main.xml* so that it matches the code below:



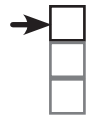

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/nav_host_fragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:name="androidx.navigation.fragment.NavHostFragment"
    app:navGraph="@navigation/nav_graph"
    app:defaultNavHost="true" />
```

When you've updated the layout, open *MainActivity.kt* and make sure it matches the code shown here:




```
package com.hfad.guessinggame

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)
    }
}
```

That's everything we need for `MainActivity`. Next, let's update the code for the two fragments, starting with `GameFragment`.



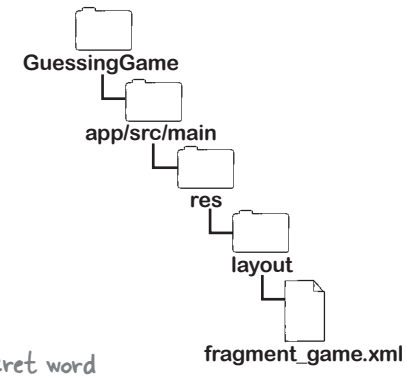
**Basic game**
GameFragment
ResultFragment

# Update GameFragment's layout

GameFragment is the app's main screen, and it's what the user will use to play the guessing game. We need to add several views to its layout to enable this: three text views for the word to guess, the number of lives left and the incorrect guesses that have been made, an edit text for typing guesses into, and a button to make the guess.

Add these views to the layout by updating the code in *fragment_game.xml* so it matches the code shown below:

```
<?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:orientation="vertical"
    android:padding="16dp"
    tools:context=".GameFragment">

    <TextView
        android:id="@+id/word"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:textSize="36sp"
        android:letterSpacing="0.1" />

    <TextView
        android:id="@+id/lives"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="16sp" />

    <TextView
        android:id="@+id/incorrect_guesses"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="16sp" />
```




The secret word

The lives left

Any incorrect guesses the user has made

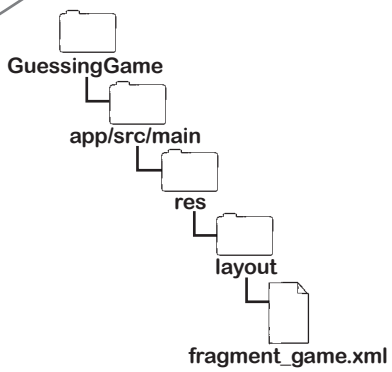
The code continues on the next page.

# fragment_game.xml continued

```
<EditText
    android:id="@+id/guess"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="16sp"
    android:hint="Guess a letter"
    android:inputType="text"
    android:maxLength="1" />

<Button
    android:id="@+id/guess_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:text="Guess!"/>
</LinearLayout>
```

An edit text that lets the user make a guess

The button



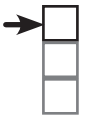

That's everything we need to include in `GameFragment`'s layout. Next, let's tell the game how to behave by updating its Kotlin code.

## What GameFragment needs to do

In the first version of the app, *GameFragment.kt* needs to include all of the code that's needed to play the game, as well as updating the screen and enabling navigation. It needs to:

- **Choose a word at random.**
  We'll provide it with a list of possible words to choose from.
- **Let the user guess a letter.**
- **Respond to the guess.**
  If the user guesses correctly, `GameFragment` needs to add the letter to a list of correct guesses, and show where the letter appears in the word. If the guess is incorrect, it needs to add the guess to a `String` of incorrect guesses, and decrement the number of lives left.
- **Navigate to ResultFragment when the game is over.**

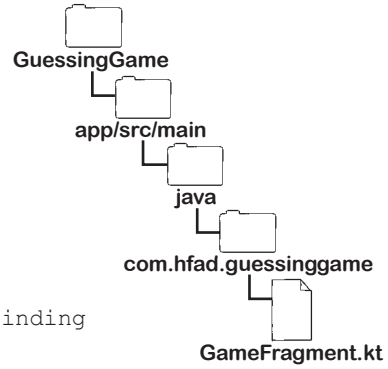
All of this code is pure Kotlin, or Android code that you've seen before. We'll show you the full code over the next few pages.



**Basic game**
GameFragment
ResultFragment

# The GameFragment.kt code

Here's the code for `GameFragment`; update the code in *GameFragment.kt* so that it matches the code shown below:




```
package com.hfad.guessinggame

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.guessinggame.databinding.FragmentGameBinding
import androidx.navigation.findNavController

class GameFragment : Fragment() {
    private var _binding: FragmentGameBinding? = null
    private val binding get() = _binding!!

    val words = listOf("Android", "Activity", "Fragment")
    val secretWord = words.random().uppercase()
    var secretWordDisplay = ""
    var correctGuesses = ""
    var incorrectGuesses = ""
    var livesLeft = 8

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentGameBinding.inflate(inflater, container, false)
        val view = binding.root

        secretWordDisplay = deriveSecretWordDisplay()
        updateScreen()
```

These properties are used for view binding.

These are the possible words. In real life, you'd have more to make the game harder.

The word the user has to guess

How the word is displayed

The correct and incorrect guesses made

The number of lives left

Derive how the secret word should be displayed, and update the screen.

The code continues on the next page.

# GameFragment.kt continued

```
        binding.guessButton.setOnClickListener() {
            makeGuess(binding.guess.text.toString().uppercase())
            binding.guess.text = null
            updateScreen()
            if (isWon() || isLost()) {
                val action = GameFragmentDirections
                    .actionGameFragmentToResultFragment(wonLostMessage())
                view.findNavController().navigate(action)
            }
        }
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }

    fun updateScreen() {
        binding.word.text = secretWordDisplay
        binding.lives.text = "You have $livesLeft lives left."
        binding.incorrectGuesses.text = "Incorrect guesses: $incorrectGuesses"
    }

    fun deriveSecretWordDisplay() : String {
        var display = ""
        secretWord.forEach {
            display += checkLetter(it.toString())
        }
        return display
    }
```

Call makeGuess to deal with the user's guess.

Reset the edit text and update the screen.

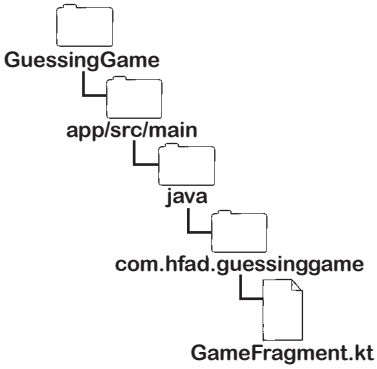
If the user has won or lost, navigate to ResultFragment, passing it the wonLostMessage() return value.

When the fragment no longer has access to its layout, set the _binding property to null.

Set the layout's text views.

This builds a String for how the secret word should be displayed on the screen.

Call checkLetter for each letter in secretWord, and add its return value to the end of the display variable.




The code continues on the next page.

# GameFragment.kt continued

**Basic game**
GameFragment
ResultFragment

```
    fun checkLetter(str: String) = when (correctGuesses.contains(str)) {
                                                true -> str
                                                false -> "_"
                                            }

    fun makeGuess(guess: String) {
        if (guess.length == 1) {
            if (secretWord.contains(guess)) {
                correctGuesses += guess
                secretWordDisplay = deriveSecretWordDisplay()
            } else {
                incorrectGuesses += "$guess "
                livesLeft--
            }
        }
    }

    fun isWon() = secretWord.equals(secretWordDisplay, true)

    fun isLost() = livesLeft <= 0

    fun wonLostMessage() : String {
        var message = ""
        if (isWon()) message = "You won!"
        else if (isLost()) message = "You lost!"
        message += " The word was $secretWord."
        return message
    }
}
```

This checks whether the secret word contains the letter the user has guessed. If so, it returns the letter. If not, it returns "_".

This gets called each time the user makes a guess.

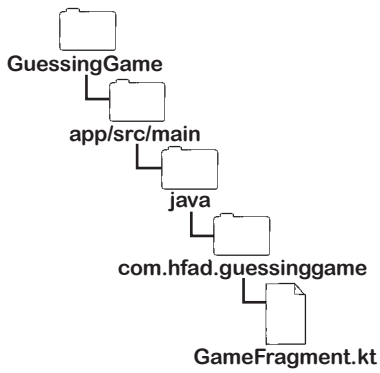
For each correct guess, update correctGuesses and secretWordDisplay.

For each wrong guess, update incorrectGuesses and livesLeft.

The game is won if the secret word matches secretWordDisplay.

The game is lost when the user runs out of lives.

wonLostMessage() returns a String indicating whether the user has won or lost, and what the secret word was.



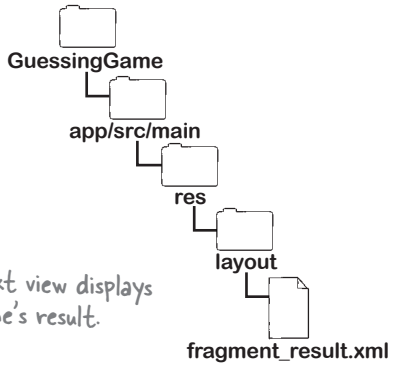

That's all the code that we need for `GameFragment`. Next, let's write the `ResultFragment` code.

# Update ResultFragment's layout

`ResultFragment` uses a text view to tell the user whether she's won or lost the game she's just played, and a button to let her start another game. We need to add both of these views to the fragment's layout.

You're already familiar with the code to do this, so open *fragment_result.xml*, and update it so that it matches the code shown below:

```
<?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:orientation="vertical"
    tools:context=".ResultFragment">

    <TextView
        android:id="@+id/won_lost"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:textSize="28sp" />

    <Button
        android:id="@+id/new_game_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:text="Start new game"/>
</LinearLayout>
```



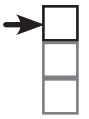

The text view displays the game's result.

The button lets the user start a new game.

## We also need to update ResultFragment.kt

Once we've added the text view and button to `ResultFragment`'s layout, we need to specify their behavior in the fragment's Kotlin code. We'll update the text view's text with the result, and we'll also make the button navigate back to `GameFragment` when it's clicked.

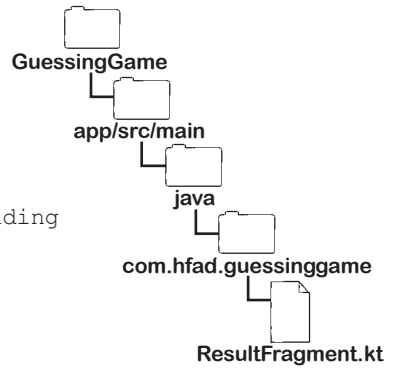
We'll show you the code for *ResultFragment.kt* on the next page.



**Basic game**
GameFragment
ResultFragment

# The ResultFragment.kt code

Here's the code for `ResultFragment`; update *ResultFragment.kt* so it matches the code below:




```
package com.hfad.guessinggame

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.guessinggame.databinding.FragmentResultBinding
import androidx.navigation.findNavController

class ResultFragment : Fragment() {
    private var _binding: FragmentResultBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentResultBinding.inflate(inflater, container, false)
        val view = binding.root

        binding.wonLost.text = ResultFragmentArgs.fromBundle(requireArguments()).result

        binding.newGameButton.setOnClickListener {
            view.findNavController()
                .navigate(R.id.action_resultFragment_to_gameFragment)
        }
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

We're using view binding in this fragment too.

Set the text view to the result String that's passed from GameFragment.

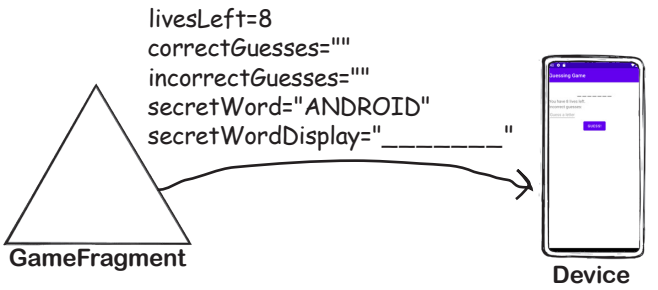
Navigate to GameFragment when the user clicks the button.

That's all the code that we need for this version of the Guessing Game app. Let's go through what happens when the code runs, and take the app for a test drive.

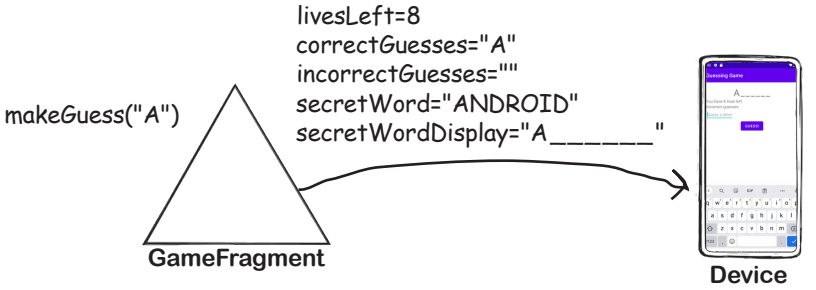
# What happens when the app runs

The following things happen when the app runs:

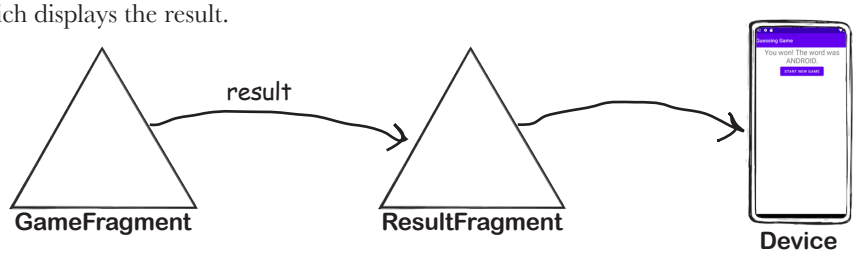
1. **The app is launched and GameFragment is displayed in MainActivity.**
`GameFragment` sets `livesLeft` to 8, `correctGuesses` and `incorrectGuesses` to `""`, `secretWord` to a word it chooses at random, and `secretWordDisplay` to the value of `deriveSecretWordDisplay()`. It then calls its `updateScreen()` method, which displays the values of `livesLeft`, `incorrectGuesses`, and `secretWordDisplay`.




2. **When the user makes a guess, GameFragment calls its makeGuess() method.**
The method checks if `secretWord` contains the letter the user has guessed. If it does, `makeGuess()` adds the letter to `correctGuesses` and updates `secretWordDisplay`. If it doesn't, the method adds the letter to `incorrectGuesses`, and subtracts 1 from `livesLeft`. It then calls `updateScreen()` again so that the new values are displayed.




3. **After each guess, GameFragment checks if isWon() or isLost() is true.**
These methods check if the user has guessed all the letters in the word, or if she has run out of lives. If either method returns *true*, `GameFragment` passes the result to `ResultFragment`, which displays the result.



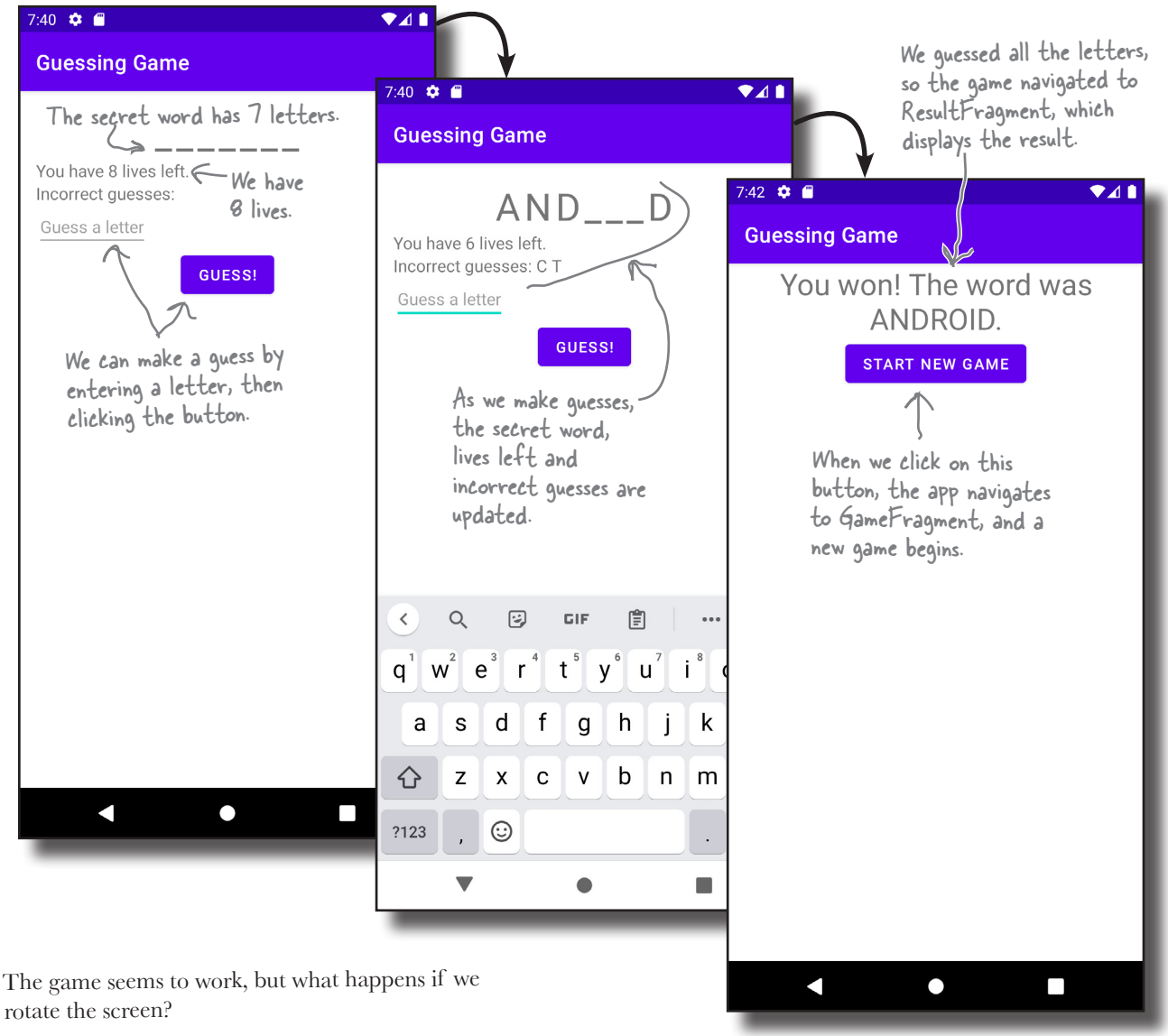
# Test Drive

**Basic game**
**GameFragment**
**ResultFragment**

When we run the app, `GameFragment` is displayed. It shows how many letters the secret word has, and how many lives we have.

We can make a guess by entering a letter in the edit text, and clicking the button. If we guess correctly, it places the letter in the secret word, but if we get it wrong, we lose a life.

If we guess all the letters or run out of lives, a message telling us whether we've won or lost is displayed in `ResultFragment`.

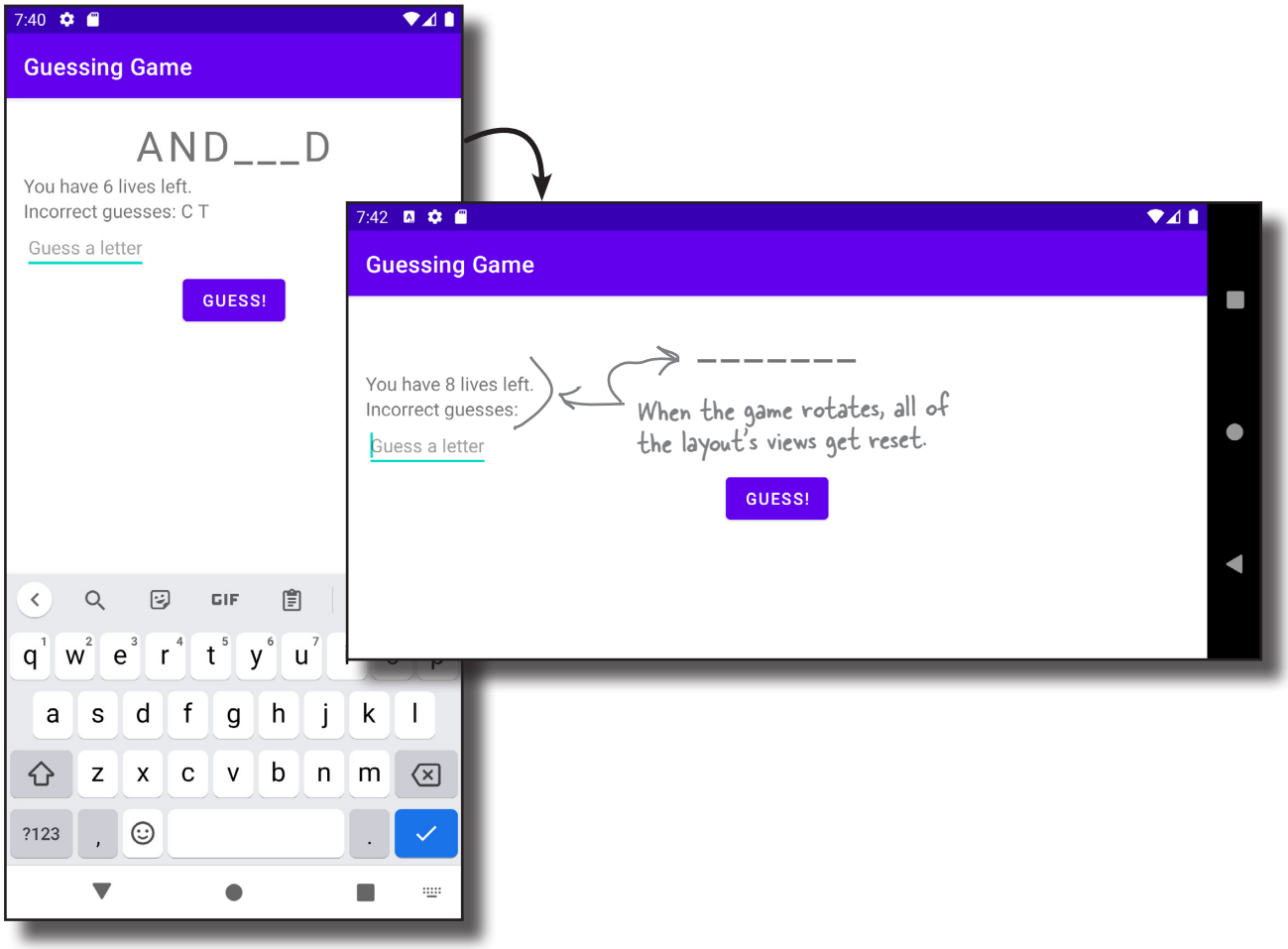



The game seems to work, but what happens if we rotate the screen?

**Basic game**
GameFragment
ResultFragment

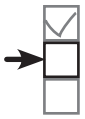
# The game loses state when the screen rotates

There's a problem with the game, however. If we rotate the screen when we're in the middle of a game, the app loses its state and the game starts from the beginning.




The game loses its state because rotating the screen changes the app's configuration, so Android destroys the activity (and the fragment it displays) and immediately recreates it. This resets the game's views and properties.

We *could* solve this problem using the fragment's `onSaveInstanceState` method to save the state of any properties, as we did earlier in the book. This time, however, we're going to fix it by **implementing a view model** instead.



Basic game
**GameFragment**
ResultFragment

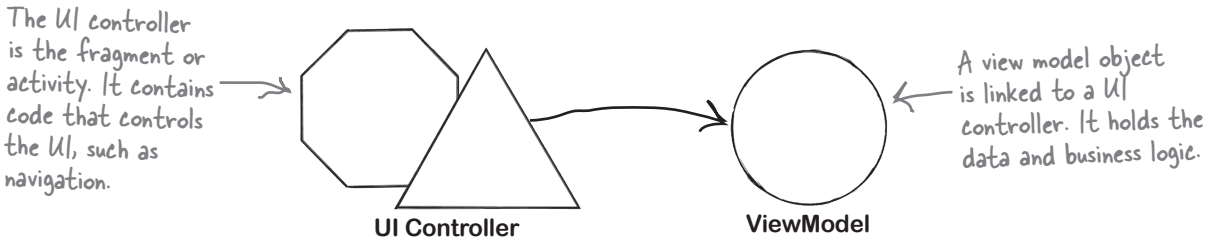
# A view model holds business logic

As we said earlier, a view model is a separate class that sits alongside your activity or fragment code. It's responsible for the data that needs to be displayed on the screen along with any business logic. In our Guessing Game app, for example, this means that the view model needs to hold the game's properties—like the secret word the user needs to guess, and the number of lives left—and any methods that control how the game is played.

When you implement a view model, all of the code relating to the app's data or business logic is moved out of the activity or fragment, and into the view model. Any code that controls the UI—such as displaying text, or getting user input—stays in the activity or fragment code.

This way of architecting Android apps follows a design principle known as *separation of concerns*. The app is split into different classes, where each class addresses a separate concern. A UI controller—the activity or fragment code—handles the UI, and the view model is responsible for the business logic and data:

**Use a view model to simplify your activity and fragment code, and hold the state of any properties so that they survive configuration changes.**

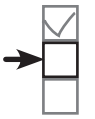



Using this sort of app architecture has two key advantages.

- **It simplifies your activity and fragment code.**
  Moving the app's data and business logic into a view model means that there's less activity and fragment code for you to maintain.

- **Your app survives configuration changes.**
  The view model is a separate class that sits alongside your activity or fragment code. It's not destroyed when you rotate the device screen, so the state of any properties held in the view model survives configuration changes without you having to add their values to a `Bundle`.

Now that you've learned what using a view model buys you, let's see how to add one to the Guessing Game app.





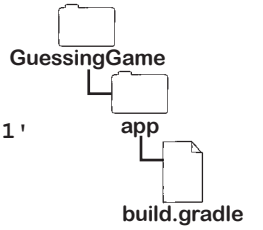
# Add a view model dependency to the app build.gradle file...

The view model library is part of Android Jetpack, so you need to update the app's *build.gradle* file to include it as a dependency.

Open the file *GuessingGame/app/build.gradle*, and add the following line (in bold) to the dependencies section:

```
dependencies {
    ...
    implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.3.1'
    ...
}
```

We're using this version of the library.




When prompted, sync your changes. You're now ready to go ahead and create a view model.

## ...and create a view model

We're going to create a view model named `GameViewModel`, which `GameFragment` will use for its game logic and data.

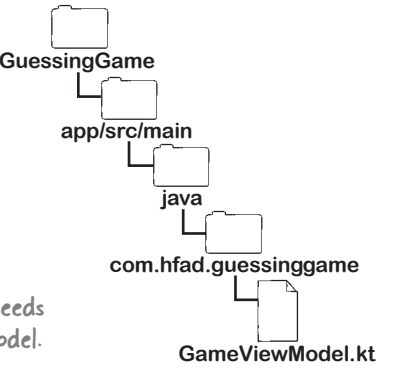
Select the *com.hfad.guessinggame* package in the *app/src/main/java* folder, then go to File→New→Kotlin Class/File. Name the file "GameViewModel" and choose the option to create a class.

When the *GameViewModel.kt* file has been created, update its code to match the code below (the changes are in bold):

```
package com.hfad.guessinggame

import androidx.lifecycle.ViewModel

class GameViewModel : ViewModel() {

}
```

Import this class.

GameViewModel needs to extend ViewModel.




As you can see, the `GameViewModel` class extends `androidx.lifecycle.ViewModel`. `ViewModel` is an abstract class that's used to turn a plain old class into a card-carrying view model.

Now that we've added `GameViewModel` to the Guessing Game project and turned it into a view model, let's write the rest of its code.

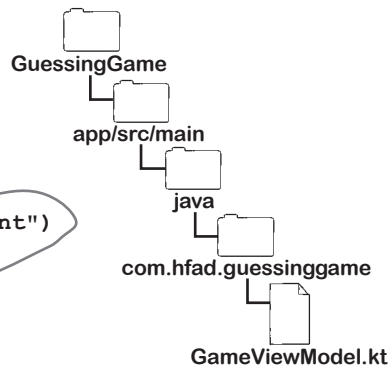
Basic game
**GameFragment**
ResultFragment

# The full code for GameViewModel.kt

As you've already learned, the view model is responsible for an activity or fragment's business logic and data. For the Guessing Game app, this means that we need to take all of `GameFragment`'s properties and methods that relate to how the game is played, and move them into `GameViewModel`. We'll leave any code relating to navigation or the UI in `GameFragment`.

Here's the full code for *GameViewModel.kt*; update the code to include the changes (in bold):

```
package com.hfad.guessinggame

import androidx.lifecycle.ViewModel

class GameViewModel : ViewModel() {
    val words = listOf("Android", "Activity", "Fragment")
    val secretWord = words.random().uppercase()
    var secretWordDisplay = ""
    var correctGuesses = ""
    var incorrectGuesses = ""
    var livesLeft = 8

    init {
        secretWordDisplay = deriveSecretWordDisplay()
    }

    fun deriveSecretWordDisplay() : String {
        var display = ""
        secretWord.forEach {
            display += checkLetter(it.toString())
        }
        return display
    }
```




These properties are needed to play the game.

Call this method in an init block. This runs when the class is initialized.

How the secret word should be displayed on the screen

The code continues on the next page.

# GameViewModel.kt continued

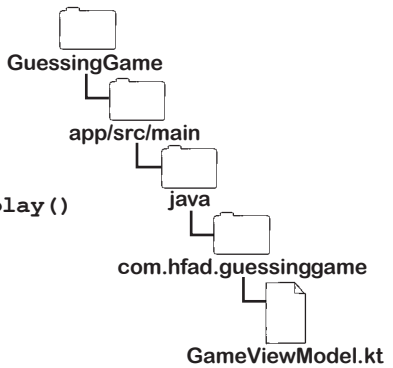
```
    fun checkLetter(str: String) = when (correctGuesses.contains(str)) {
        true -> str
        false -> "_"
    }

    fun makeGuess(guess: String) {
        if (guess.length == 1) {
            if (secretWord.contains(guess)) {
                correctGuesses += guess
                secretWordDisplay = deriveSecretWordDisplay()
            } else {
                incorrectGuesses += "$guess "
                livesLeft--
            }
        }
    }

    fun isWon() = secretWord.equals(secretWordDisplay, true)

    fun isLost() = livesLeft <= 0

    fun wonLostMessage() : String {
        var message = ""
        if (isWon()) message = "You won!"
        else if (isLost()) message = "You lost!"
        message += " The word was $secretWord."
        return message
    }
}
```

Returns str if it appears in the secret word, or "_" if it doesn't

This gets called when the user makes a guess.

Checks whether the game has been won or lost

This generates a String when the game is over.




That's everything that we need for `GameViewModel`. Next, let's link it to `GameFragment`, and update the code for that fragment.

Basic game
**GameFragment**
ResultFragment

# Create a GameViewModel object

To link a view model to an activity or fragment, you need to add a `ViewModel` property to the code, and initialize it with a `ViewModel` object, which you need to create. The code looks like this:

```
class GameFragment : Fragment() {
    ...
    lateinit var viewModel: GameViewModel

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        ...
        viewModel = ViewModelProvider(this).get(GameViewModel::class.java)
        ...
    }
}
```

Define the viewModel property, which will be initialized later in the code.

This line gets a GameViewModel object and assigns it to the viewModel property.

The :: in this code is used to get a reference to the GameViewModel class. We have to use this syntax because the ViewModelProvider's get() method requires a reference to the GameViewModel class instead of GameViewModel object.

As you can see, the above code uses a **view model provider** to create the `ViewModel` object. So what does using this class do?

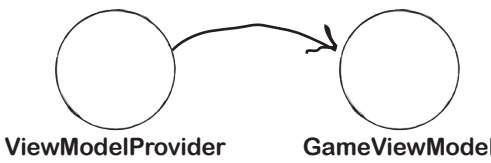
## Use a ViewModelProvider to create view models

As its name suggests, `ViewModelProvider` is a special class whose job is to provide activities and fragments with view models. It ensures that **a new view model object only gets created if one doesn't already exist**.

As you already know, when the screen is rotated, any fragments that are displayed on the screen get destroyed and recreated. When this happens, the view model provider makes sure that the same view model object continues to be used. The view model keeps its state, so any properties that are used by the fragment don't get reset.

The view model provider keeps hold of the view model so long as the activity or fragment stays alive. When a fragment is detached or removed from its activity, for example, the view model provider lets go of the fragment's view model. The next time the view model provider is asked to provide a view model object, it creates a new one.

Now that you've learned how to link a view model to a fragment, let's update the `GameFragment` code.

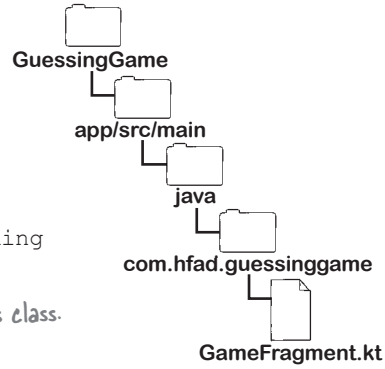



The ViewModelProvider provides GameFragment with an instance of GameViewModel. It only creates a new GameViewModel object if one doesn't already exist.

# The updated code for GameFragment.kt

We need to remove the properties and methods from `GameFragment` that we moved to `GameViewModel`, and make the fragment use the view model instead.

Here's the updated code for `GameFragment`; make sure that the file *GameFragment.kt* includes the changes shown below (in bold):




```
package com.hfad.guessinggame

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.guessinggame.databinding.FragmentGameBinding
import androidx.navigation.findNavController
import androidx.lifecycle.ViewModelProvider

class GameFragment : Fragment() {
    private var _binding: FragmentGameBinding? = null
    private val binding get() = _binding!!
    lateinit var viewModel: GameViewModel

    val words = listOf("Android", "Activity", "Fragment")
    val secretWord = words.random().uppercase()
    var secretWordDisplay = ""
    var correctGuesses = ""
    var incorrectGuesses = ""
    var livesLeft = 8

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentGameBinding.inflate(inflater, container, false)
        val view = binding.root
        viewModel = ViewModelProvider(this).get(GameViewModel::class.java)
```

Import this class.

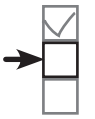
Add the viewModel property.

We moved these properties into GameViewModel, so remove them from GameFragment.

Set the viewModel property.

The code continues on the next page.

# GameFragment.kt continued

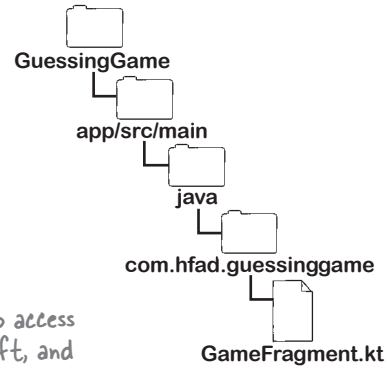


Basic game
**GameFragment**
ResultFragment

```
        ~~secretWordDisplay = deriveSecretWordDisplay()~~
        updateScreen()

        binding.guessButton.setOnClickListener() {
            viewModel.makeGuess(binding.guess.text.toString().uppercase())
            binding.guess.text = null
            updateScreen()
            if (viewModel.isWon() || viewModel.isLost()) {
                val action = GameFragmentDirections
                    .actionGameFragmentToResultFragment(viewModel.wonLostMessage())
                view.findNavController().navigate(action)
            }
        }
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }

    fun updateScreen() {
        binding.word.text = viewModel.secretWordDisplay
        binding.lives.text = "You have ${viewModel.livesLeft} lives left."
        binding.incorrectGuesses.text = "Incorrect guesses: ${viewModel.incorrectGuesses}"
    }

    ~~fun deriveSecretWordDisplay()...~~
    ~~fun checkLetter(str: String)...~~
    ~~fun makeGuess(guess: String)...~~
    ~~fun isWon()...~~
    ~~fun isLost()...~~
    ~~fun wonLostMessage()...~~
}
```

Delete this line.

The makeGuess(), isWon(), isLost(), and wonLostMessage() methods need to be accessed via the viewModel property.




Use the viewModel property to access the secretWordDisplay, livesLeft, and incorrectGuesses properties.

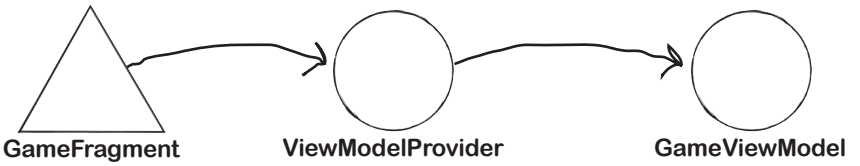
We moved these methods to GameViewModel, so delete them from GameFragment.

Those are all the changes that we need to make to `GameFragment`. Let's go through what the code does when it runs, and take it for a test drive.

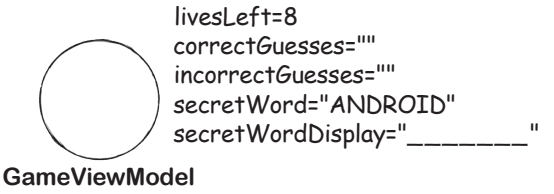
# What happens when the app runs

The following things happen when the app runs:

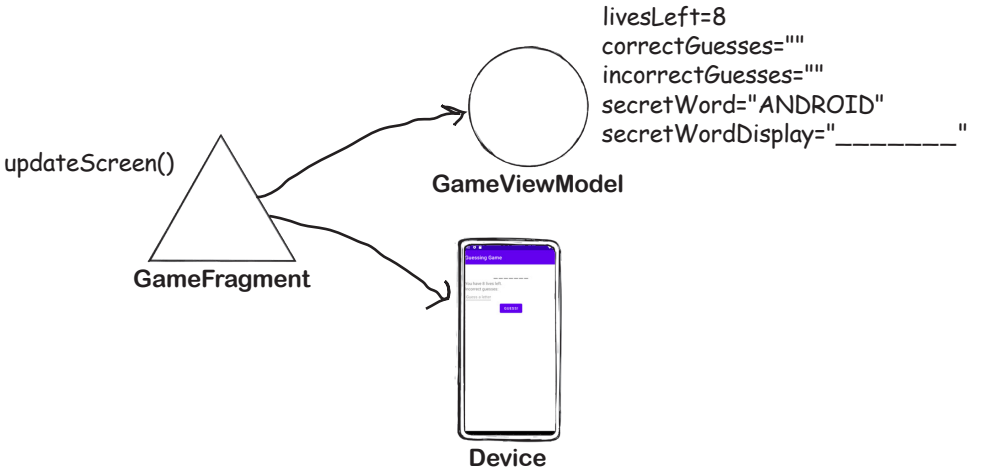
1. **GameFragment asks the ViewModelProvider class for an instance of GameViewModel.**
The view model provider sees that there's no existing `GameViewModel` object associated with the fragment, so it creates a new one.



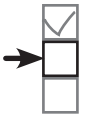

2. **The GameViewModel object is initialized.**
It sets `livesLeft` to 8, `correctGuesses` and `incorrectGuesses` to `""`, `secretWord` to a word it chooses at random, and `secretWordDisplay` to `deriveSecretWordDisplay()`.




3. **GameFragment calls its updateScreen() method.**
The method accesses the `GameViewModel` object's `secretWordDisplay`, `livesLeft`, and `incorrectGueses` properties, and shows them on the screen.



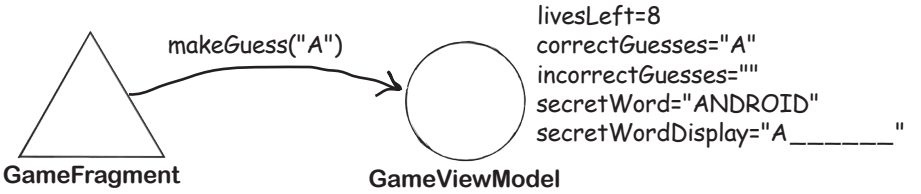
# The story continues





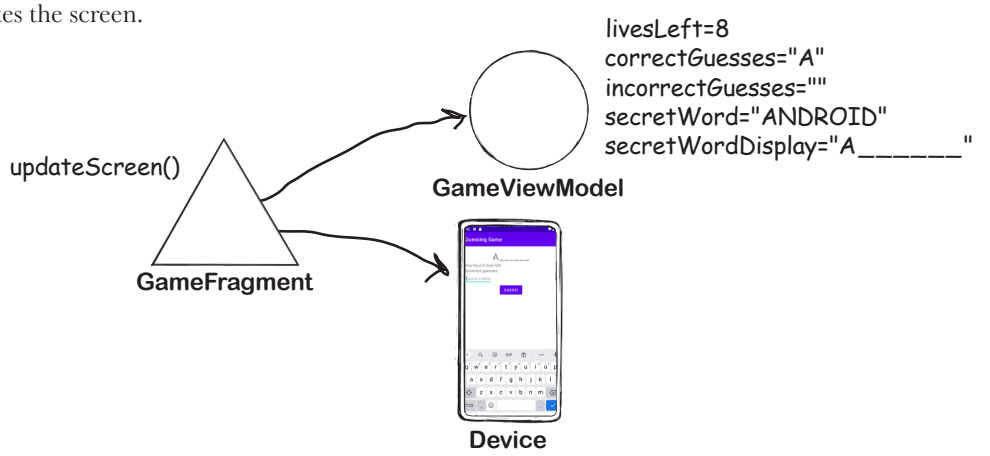
4 **When the user makes a guess, GameFragment calls the GameViewModel object's makeGuess() method.**

The method checks if `secretWord` contains the letter the user guesses. If it does, it adds the letter to `correctGuesses` and updates `secretWordDisplay`. If it doesn't, it adds it to `incorrectGuesses`, and subtracts 1 from `livesLeft`.



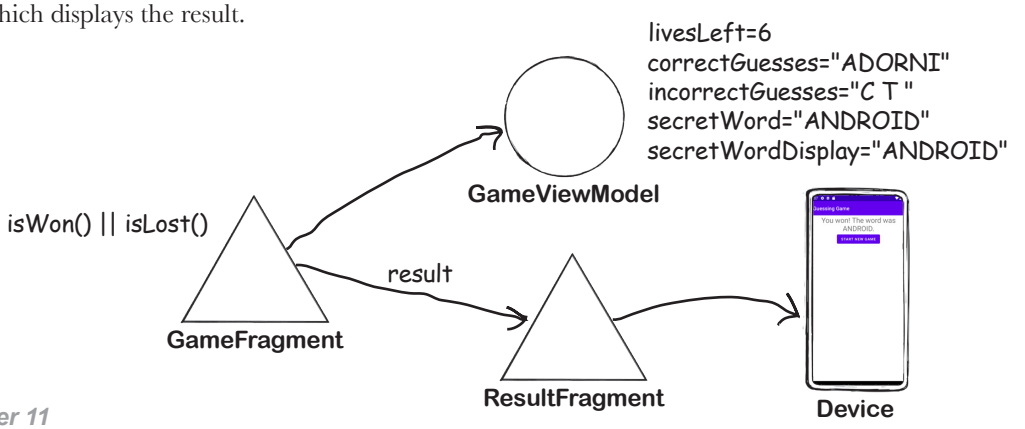

5 **GameFragment calls its updateScreen() method again.**

The method gets the updated property values from the `GameViewModel` object, and updates the screen.



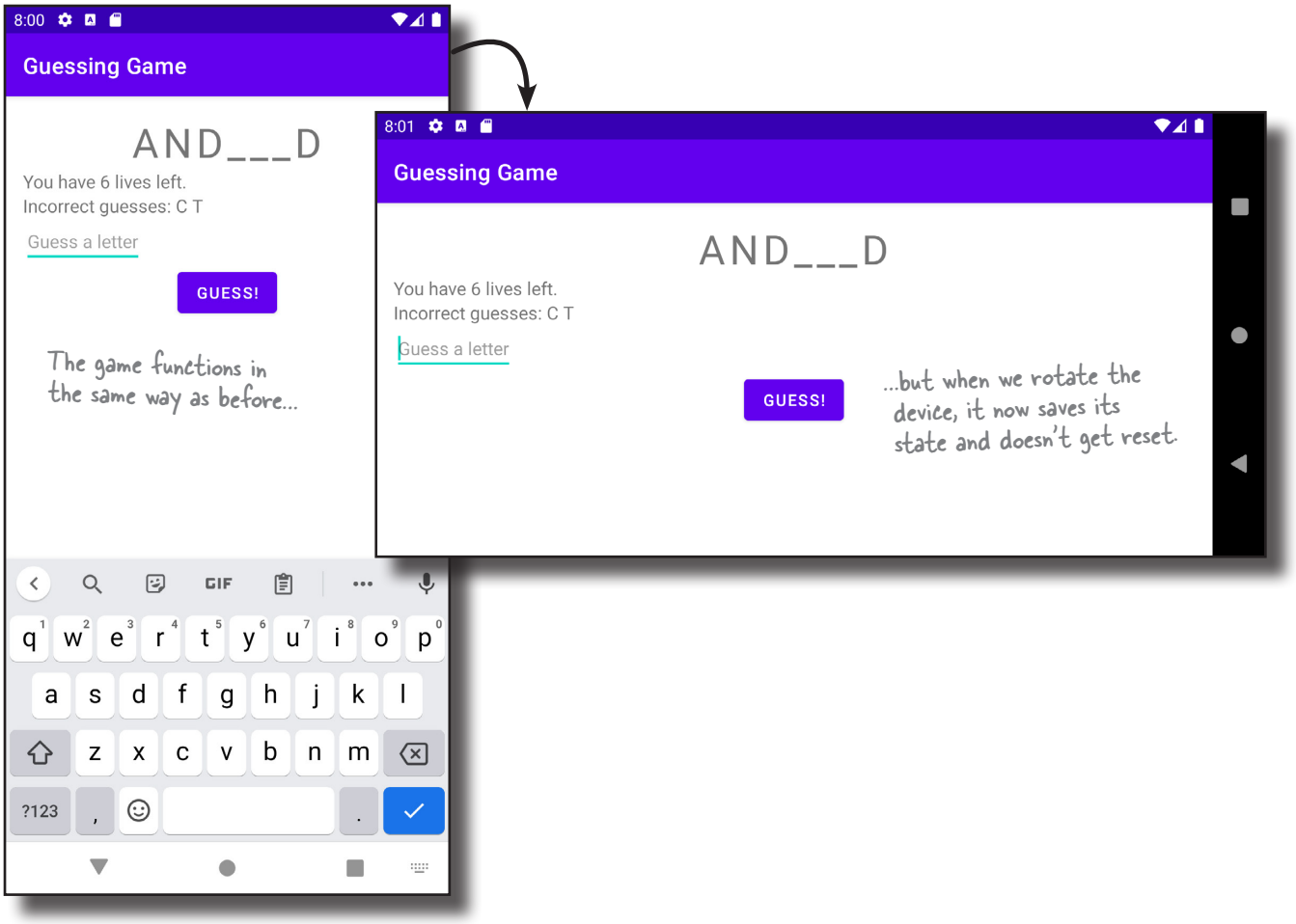

6 **After each guess, GameFragment checks if the view model's isWon() or isLost() methods return true.**

If either method returns *true*, `GameFragment` passes the result to `ResultFragment`, which displays the result.

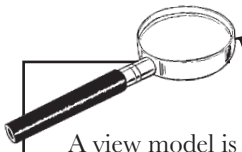


# Test Drive

When we run the app, `GameFragment` is displayed as before. If we start playing the game and rotate the screen, the game retains its state.




You've now learned how to add a view model to your apps, and used it to avoid the problems that can occur when the user rotates the device screen. Before we go any further, let's take a closer look at view models.



# View Models Up Close

A view model is a separate class you write that's responsible for a UI controller's business logic and data. It extends the `androidx.lifecycle.ViewModel` class, and may optionally override a method named **`onCleared()`**. This method gets called just before the view model gets cleared away, and it gives you a chance to dispose of any resources:

The UI controller can be an activity or fragment.

```
import androidx.lifecycle.ViewModel

class MyViewModel : ViewModel() {
    //Code to initialize the view model

    override fun onCleared() {
        //Code that runs when the view model is about to get cleared away
    }
}
```

You create a view model object using the `ViewModelProvider` class, for example:

```
viewModel = ViewModelProvider(this).get(MyViewModel::class.java)
```

This creates a view model in the same scope as the UI controller. It ensures that a fresh view model *only* gets created if there's not an existing one in the UI controller's scope. This means that the view model survives configuration changes, so you don't have to save the state of any instance variables in a separate `Bundle`.

The view model stays alive until its UI controller permanently goes away. If it's linked to an activity, this is when the activity has finished: when it's been destroyed and won't be recreated. If it's linked to a fragment, it's when the fragment has been removed or detached from its parent activity.

You can track when a view model has been created or cleared away using the **`android.util.Log`** class. This class lets you add messages to Android's log (or logcat) that you can examine in Android Studio. The following code, for example, logs a message when the view model is cleared:

```
override fun onCleared() {
    Log.i("MyViewModel", "ViewModel cleared")
}
```

This logs the message "ViewModel cleared", and tags it with "MyViewModel".

The `Log` class includes several methods you can use to log different types of message. The `Log.i()` method lets you log information, for example, while `Log.e()` is used for recording errors.

Here is a list of the different `Log` methods you can use, and what they're used for:

| | |
|---|---|
| Log.v(String tag, String message) | Logs a verbose message. |
| Log.d(String tag, String message) | Logs a debug message. |
| Log.i(String tag, String message) | Logs an information message. |
| Log.w(String tag, String message) | Logs a warning message. |
| Log.e(String tag, String message) | Logs an error message. |

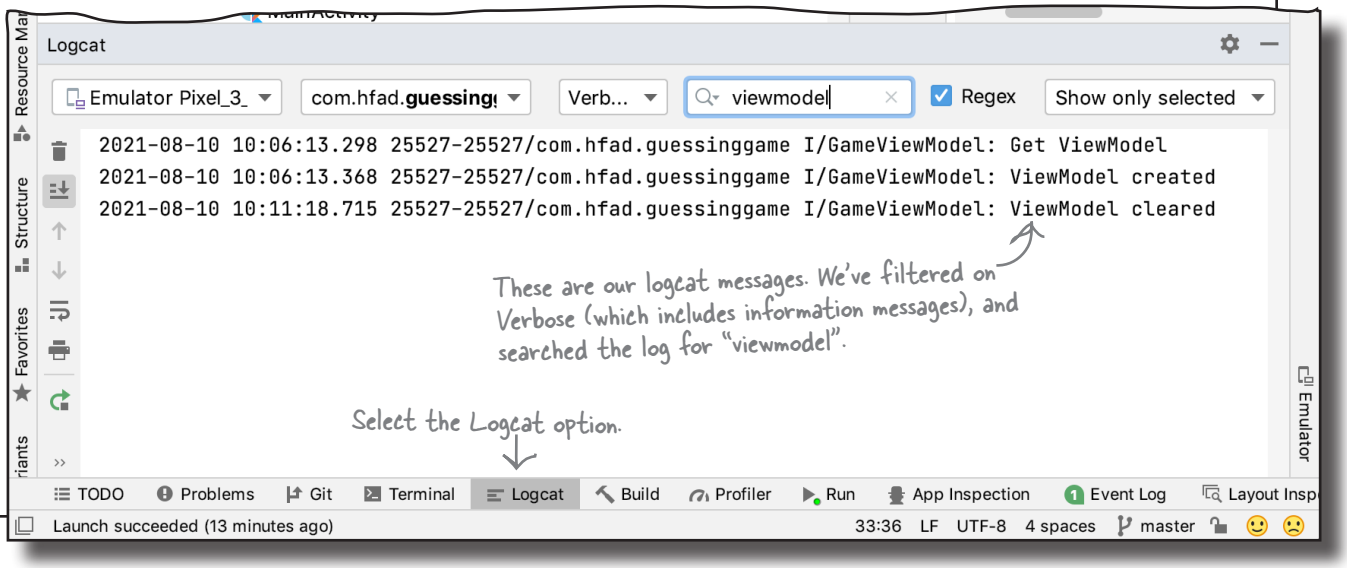
There's also a Log.wtf() method you can use to report conditions that should never happen. According to the Android documentation, wtf means "What a Terrible Failure." We know it really means "Welcome to Fiskidagurinn," which refers to the Great Fish Day festival held annually in Dalvik, Iceland. Android developers can often be heard to say, "My virtual device just took 8 minutes to boot up. WTF??" as a tribute to the small town that gave its name to the standard Android executable bytecode format.

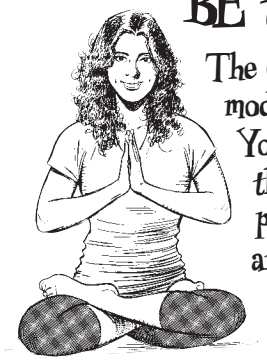
Each message is composed of a `String` tag you use to identify the source of the message, and the message itself. So the code:

```
Log.i("MyViewModel", "ViewModel cleared")
```

logs an information message that's come from `MyViewModel`.

You can view the logcat in Android Studio by choosing the Logcat option at the bottom of the screen. This lets you filter by the different types of message, and search for text:



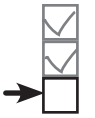


# BE the View Model

The code below describes a view model class named MyViewModel. Your job is to play like you're the view model, and say what problems there are with this code and how they could be fixed.

```
package com.hfad.myapp

import androidx.lifecycle.ViewModel
import android.util.Log
import android.widget.TextView

class MyViewModel {
    val num = 2

    init {
        Log.i("MyViewModel", "ViewModel created")
    }

    override fun onCleared() {
        Log.i("MyViewModel", "ViewModel cleared")
    }

    fun calculation(val1: Int, val2: Int): Int {
        Log.i("MyViewModel", "Called Calculation")
        return (val1 + val2) * num
    }

    fun joinTogether(text1: TextView, text2: TextView): String {
        Log.i("MyViewModel", "Called JoinTogether")
        return ("${text1.text} ${text2.text}")
    }
}
```

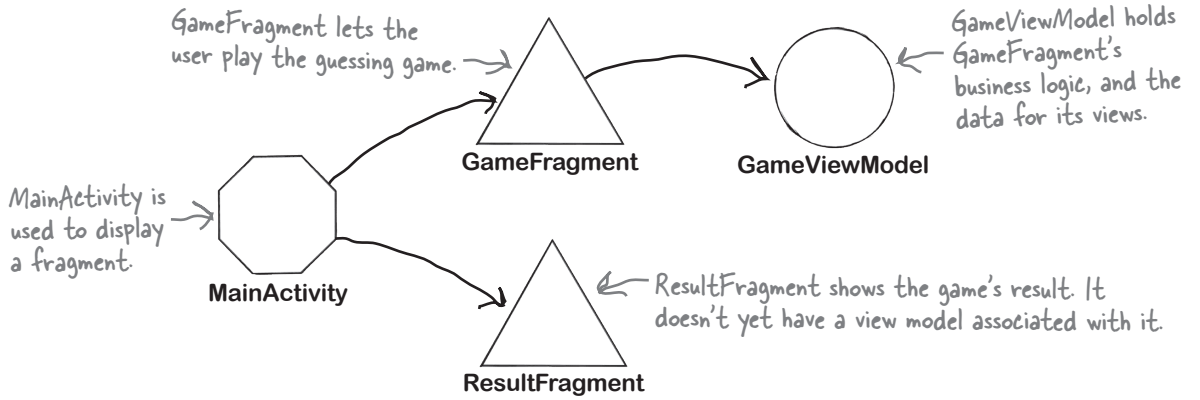
Answers on page 481.



**Basic game**
**GameFragment**
**ResultFragment**

# We've added a view model for GameFragment

So far, we've updated the Guessing Game app so that `GameFragment` uses a view model named `GameViewModel`, which is responsible for all of the fragment's business logic and data. Using a view model in this way simplifies the code in *GameFragment.kt*, and means that the app doesn't lose its state when the screen is rotated.




## ResultFragment needs a view model too

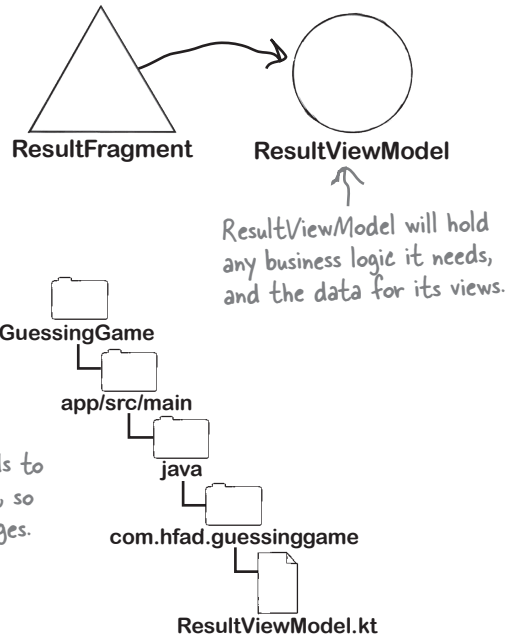
Each view model you create is associated with a single UI controller—an activity or fragment. This means that if we want `ResultFragment` to use a view model too, we need to create a new one for this fragment.

Let's do this now. Highlight the *com.hfad.guessinggame* package in the *app/src/main/java* folder, then go to File→New→Kotlin Class/File. Name the file "ResultViewModel" and choose the option to create a class.

When the *ResultViewModel.kt* file has been created, update its code to match ours below (our changes are in bold):

```
package com.hfad.guessinggame

import androidx.lifecycle.ViewModel

class ResultViewModel : ViewModel(){
}
```

The new class needs to extend ViewModel, so include these changes.




That's the basic code you need to define a view model. So what other code do we need to add?

Basic game
GameFragment
**ResultFragment**

# ResultViewModel needs to hold the result

As you may recall, `ResultFragment` displays a message in its layout that tells the user whether they've won or lost the game they've just played. This message is passed to `ResultFragment` by `GameFragment` when the game is over.

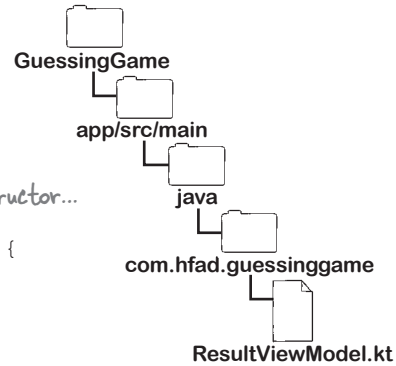
In the new version of the app, `ResultViewModel` is responsible for `ResultFragment`'s game logic and data, so we need to add a property to `ResultViewModel` to store the result. We'll also use a `String` constructor to make sure this property gets set as soon as `ResultViewModel` gets created.

Here's the full code for *ResultViewModel.kt*; update its code to include the changes below (in bold):

```
package com.hfad.guessinggame

import androidx.lifecycle.ViewModel

class ResultViewModel(finalResult: String) : ViewModel(){
    val result = finalResult
}
```

We'll add a String constructor...

...which we'll use to set the result property.




Next, we'll update `ResultFragment` so that it uses the new view model.

## We need to link ResultViewModel to ResultFragment

Earlier, we were able to add a `GameViewModel` reference to `GameFragment` using this code:

```
viewModel = ViewModelProvider(this).get(GameViewModel::class.java)
```

This code uses the ViewModelProvider to get a reference to a GameViewModel object.

This tells the view model provider to get the `GameViewModel` object that's linked to the fragment, or create a new one if it doesn't already exist.

The above approach, however, can't be used to add a `ResultViewModel` reference to `ResultFragment`. This is because it **only works for view models with a no argument constructor**.

The code worked for `GameViewModel` because we could construct it without passing any arguments. The constructor for the `ResultViewModel` class, however, requires a `String`, so the above code won't work.

# A view model factory creates view models

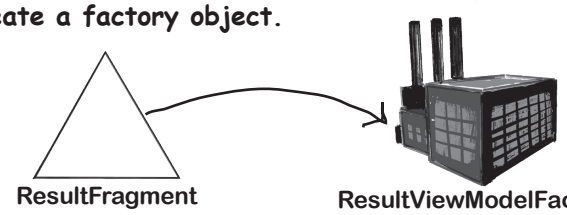
An alternative way of creating a view model is to pass the view model provider a **view model factory**: a separate class whose sole purpose is to create and initialize view models. This approach means that the view model provider doesn't have to worry about creating a view model by itself. Instead, it uses the view model factory.

While view model factories can be used for any sort of view model, they're mostly used for ones whose constructors require arguments. This is because the view model provider can't pass arguments to the constructor by itself: it needs a view model factory to do so.

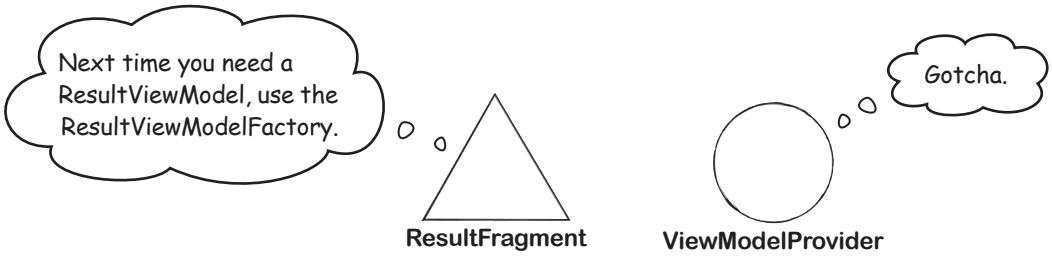
Here's how the view model factory will be used in the Guessing Game app:

**Use a view model factory to create a view model that doesn't have a no-argument constructor.**

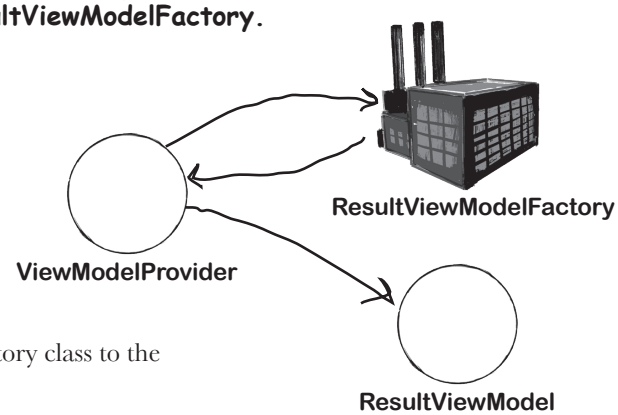
1. **We'll define a class named ResultViewModelFactory, which ResultFragment will use to create a factory object.**




2. **ResultFragment will tell the view model provider to use the factory object.**




3. **When the view model provider needs a new ResultViewModel object, it will use the ResultViewModelFactory.**



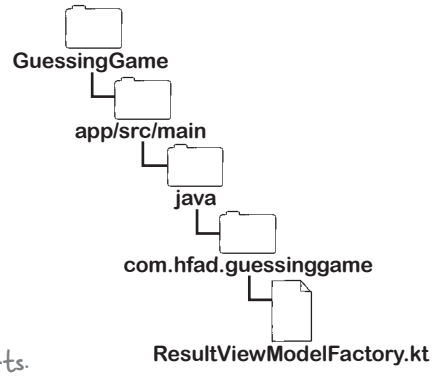

Let's go ahead and add a factory class to the Guessing Game app.

# Create the ResultViewModelFactory class

**Basic game**
**GameFragment**
**ResultFragment**

We're going to add a view model factory class named `ResultViewModelFactory` to the Guessing Game app. This class will be used by the view model provider to create a `ResultViewModel` object.

To create the class, highlight the *com.hfad.guessinggame* package in the *app/src/main/java* folder, then go to File→New→Kotlin Class/File. Name the file "ResultViewModelFactory" and choose the option to create a class.




When the *ResultViewModelFactory.kt* file has been created, update its code to match ours below (our changes are in bold):

```
package com.hfad.guessinggame

import androidx.lifecycle.ViewModel
import androidx.lifecycle.ViewModelProvider
import java.lang.IllegalArgumentException

class ResultViewModelFactory(private val finalResult: String)
        : ViewModelProvider.Factory {
    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(ResultViewModel::class.java))
            return ResultViewModel(finalResult) as T
        throw IllegalArgumentException("Unknown ViewModel")
    }
}
```

Add these imports.

The String in this constructor is needed to create the ResultViewModel object.

Override this method, which the view model provider uses to create view model objects.

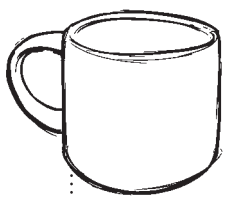
The class needs to implement this interface.

This checks if the view model the view model provider wants to create is the correct type. If so, it returns one.

If the view model type is incorrect, throw an exception.

As you can see, the `ResultViewModelFactory` class implements an interface named `ViewModelProvider.Factory`, and overrides its `create()` method. This turns the class into a view model factory, which the view model provider can use to create a `ResultViewModel` object.

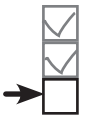
The above code is everything we need for `ResultViewModelFactory`. Let's find out how to use it in our `ResultFragment` code.



## Relax

**All view model factory code is nearly identical.**

Don't worry too much about the code details. All you really need to know is it creates a view model of a particular type, and passes an argument to its constructor.

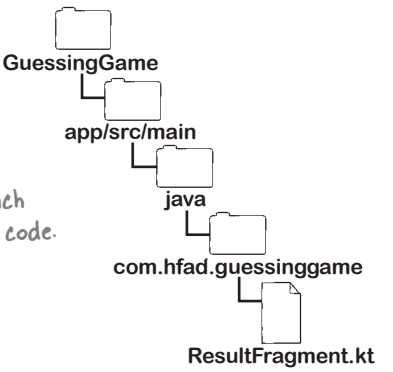




# Use the factory to create the view model

As we said earlier, you use a view model factory to create a view model by passing the factory to the view model provider. The view model provider decides when a new view model object is required, and when necessary, uses the factory to create one.

The code to make the view model provider use a view model factory is virtually identical for every view model you want to create. It looks like this:




```
class ResultFragment : Fragment() {
    ...
    lateinit var viewModel: ResultViewModel
    lateinit var viewModelFactory: ResultViewModelFactory

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        ...
        val result = ResultFragmentArgs.fromBundle(requireArguments()).result
        viewModelFactory = ResultViewModelFactory(result)
        viewModel = ViewModelProvider(this, viewModelFactory)
                                .get(ResultViewModel::class.java)
        ...
    }
}
```

Add these two properties, which will be initialized later in the code.

Create a view model factory object.

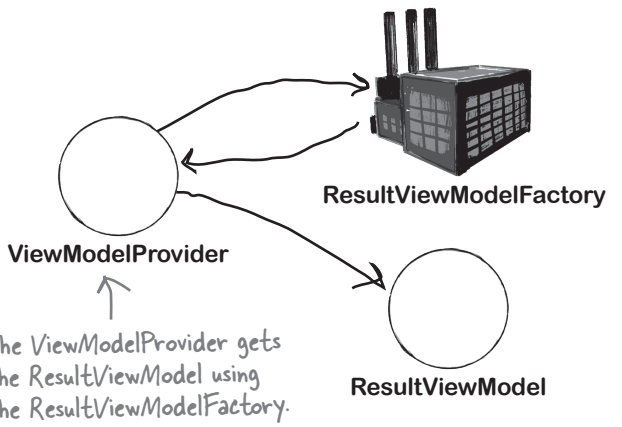
Pass the view model factory to the view model provider.

The view model provider uses the view model factory to create a view model object if one doesn't already exist.

As you can see, the above code defines two properties: `viewModel` and `viewModelFactory`. These are set in the fragment's `onCreateView()` method.

In `onCreateView()`, the code uses the result `String` that's passed to it from `GameFragment` to create a new `ResultViewModelFactory` object. It passes the factory to the view model provider, which uses it to get a `ResultViewModel` object.

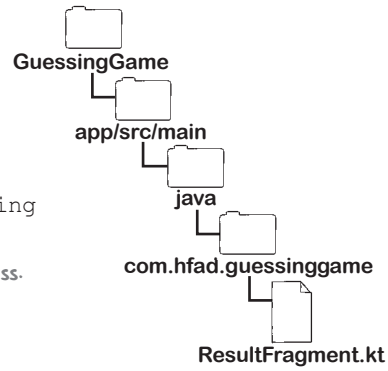
Now that you know how to use a factory to link a view model to a fragment, let's update the `ResultFragment` code.



# The updated code for ResultFragment.kt

**Basic game**
**GameFragment**
**ResultFragment**

Here's the full code for `ResultFragment`, so update the code in file *ResultFragment.kt* so it includes the changes shown below (in bold):




```
package com.hfad.guessinggame

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.guessinggame.databinding.FragmentResultBinding
import androidx.navigation.findNavController
import androidx.lifecycle.ViewModelProvider

class ResultFragment : Fragment() {
    private var _binding: FragmentResultBinding? = null
    private val binding get() = _binding!!
    lateinit var viewModel: ResultViewModel
    lateinit var viewModelFactory: ResultViewModelFactory

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentResultBinding.inflate(inflater, container, false)
        val view = binding.root

        val result = ResultFragmentArgs.fromBundle(requireArguments()).result
        viewModelFactory = ResultViewModelFactory(result)
        viewModel = ViewModelProvider(this, viewModelFactory)
                                .get(ResultViewModel::class.java)
        binding.wonLost.text = ResultFragmentArgs.fromBundle(requireArguments()).result
        binding.wonLost.text = viewModel.result

        binding.newGameButton.setOnClickListener {
            view.findNavController()
                .navigate(R.id.action_resultFragment_to_gameFragment)
        }
        return view
    }
```

Import this class.

Add these properties.

Get the result String that's been passed to ResultFragment.

Create the factory.

Get the view model.

Delete this line.

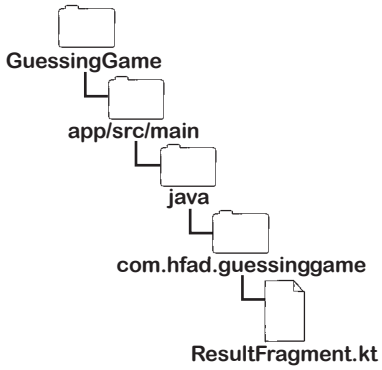
Display the result from the view model.

The code continues on the next page.

# ResultFragment.kt continued

```
    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

You don't need to change this method.




Those are all the changes that we need to make to `ResultFragment`. After some questions, we'll go through what happens when the app runs.

## there are no Dumb Questions

**Q: Does `ResultFragment` really need its own view model?**

A: We decided to use one so that we could separate out any game logic and data required by `ResultFragment`. This is only a result `String` in the current app, but it's good practice in case the app gets more complicated. And let's face it, it was a good excuse for us to teach you about using view model factories.

**Q: Why didn't we use a factory when we created `GameViewModel`?**

A: Factories are only necessary when you need to create a view model that doesn't have a no-arguments constructor. As `GameViewModel` uses the default constructor which has no arguments, we didn't need to use a factory.

**Q: Could we still have used a factory for `GameViewModel` if we'd wanted to?**

A: Yes. You can use factories to create any sort of view model, but if the view model has a no-argument constructor, you don't need to use one.

**Q: Why did we need to add a constructor to `ResultViewModel`?**

A: So that we could set the result `String` as soon as the view model was created. This guarantees that the result `String` has a value whenever `ResultFragment` uses the view model.

**Q: Do you have to create view models using the `ViewModelProvider` class? What happens if you don't?**

A: Yes, you should always use the view model provider to create a view model object. The view model provider *only* creates a new view model object if one doesn't already exist for the UI controller's scope, which means that the view model survives configuration changes.

If you instantiate a view model without using the view model provider, it will get recreated each time the UI controller is destroyed and recreated. The view model would lose its state each time the device screen is rotated.

**Q: Can you unit test view models?**

A: Yes! View models don't refer to any of the screen's views, so they can be unit tested independently of activities and fragments. You don't have to run the app to test them.

**Q: Why didn't we create a view model for `MainActivity`?**

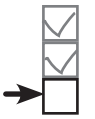
A: In the Guessing Game app, `MainActivity` is only used to display the game's fragments. As `MainActivity` doesn't include any game logic or data, we didn't need to create a view model for it.

**Q: Can two fragments share the same view model?**

A: If you have two fragments in an activity that need to communicate with each other, you might want to add a single view model to the activity that the two fragments then have access to. You can find out more about this here:

*https://developer.android.com/topic/libraries/architecture/viewmodel*

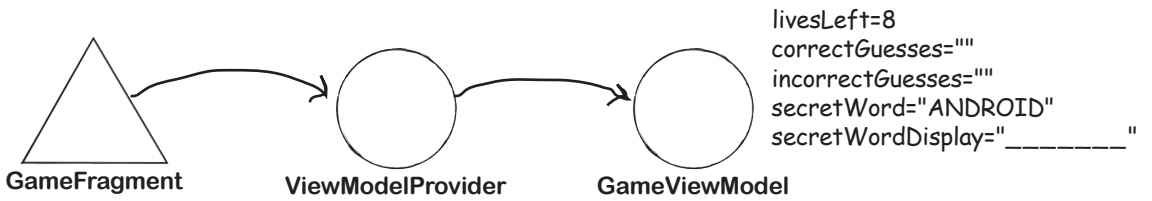
# What happens when the app runs



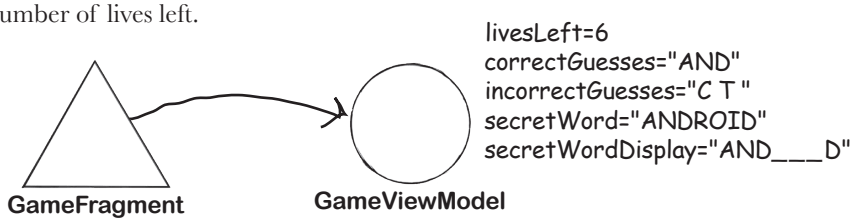
**Basic game**
**GameFragment**
**ResultFragment**

The following things happen when the app runs:

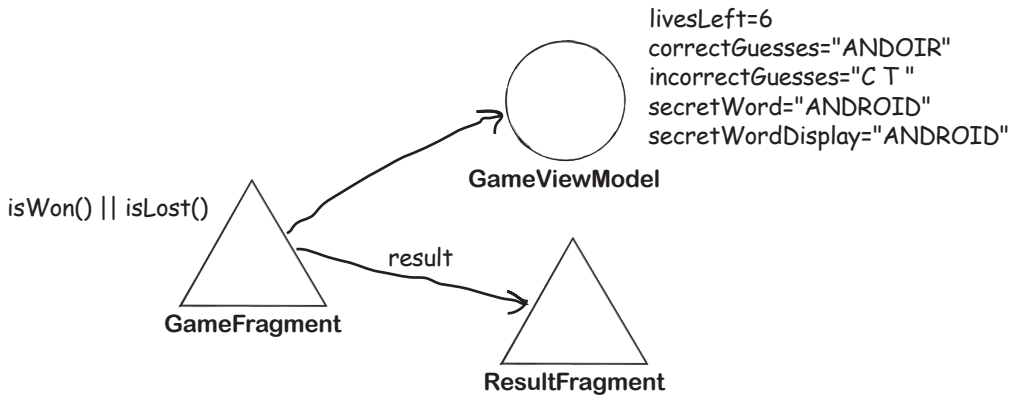
1. **GameFragment asks the ViewModelProvider class for an instance of GameViewModel.**
The `GameViewModel` object is initialized, and chooses a word at random.



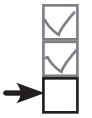

2. **GameFragment interacts with the GameViewModel object.**
The `GameViewModel` object records any guesses the user has made, and keeps track of the number of lives left.



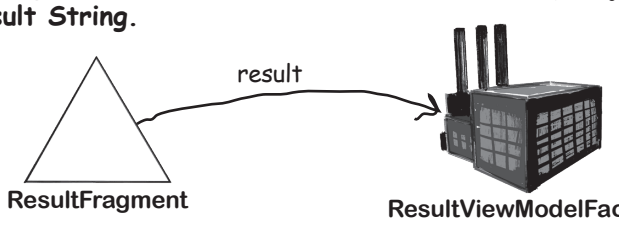

3. **After each guess, GameFragment checks if the view model's isWon() or isLost() methods return true.**
If either method is *true*, `GameFragment` navigates to `ResultFragment`, passing it the result.



# The story continues

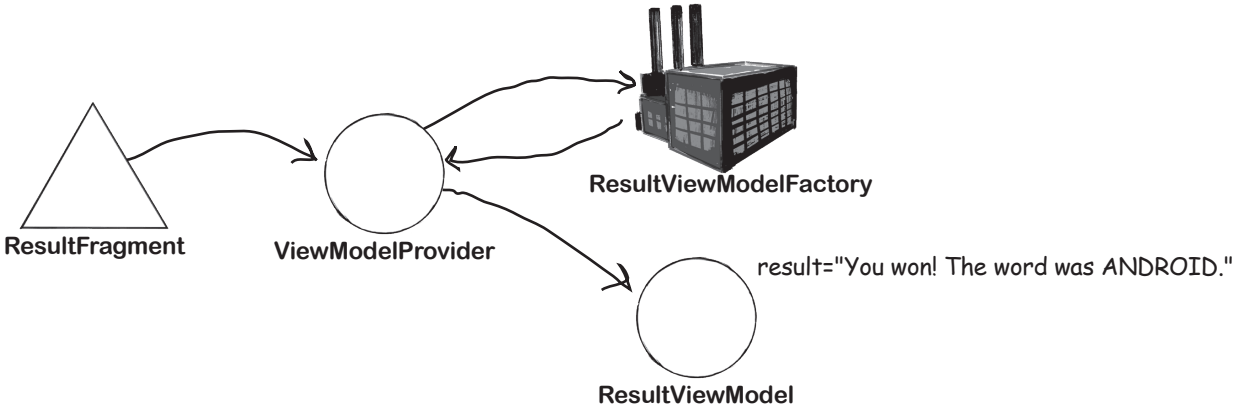


4 **ResultFragment creates a ResultViewModelFactory object, passing it the result String.**

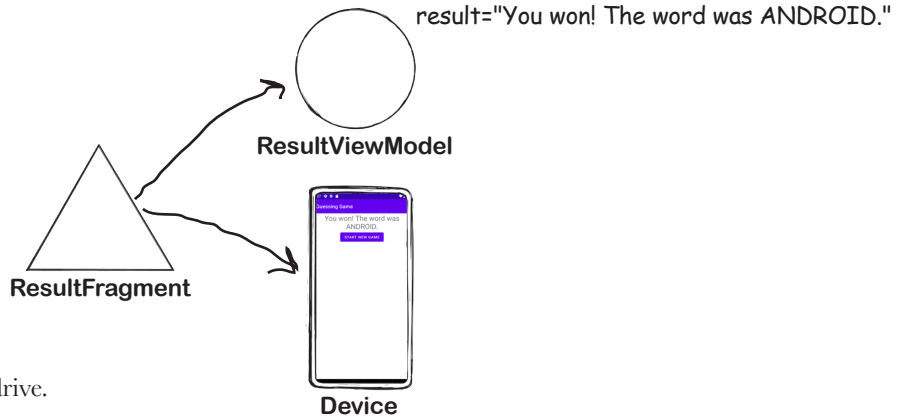



5 **ResultFragment asks the ViewModelProvider class for an instance of ResultViewModel.**

The `ViewModelProvider` class sees that there's no existing `ResultViewModel` object, so it uses the `ResultViewModelFactory` to create one. The `ResultViewModel` object's `result` property is initialized with the result `String`.




6 **ResultFragment gets the result String from the ResultViewModel object, and displays it on the screen.**



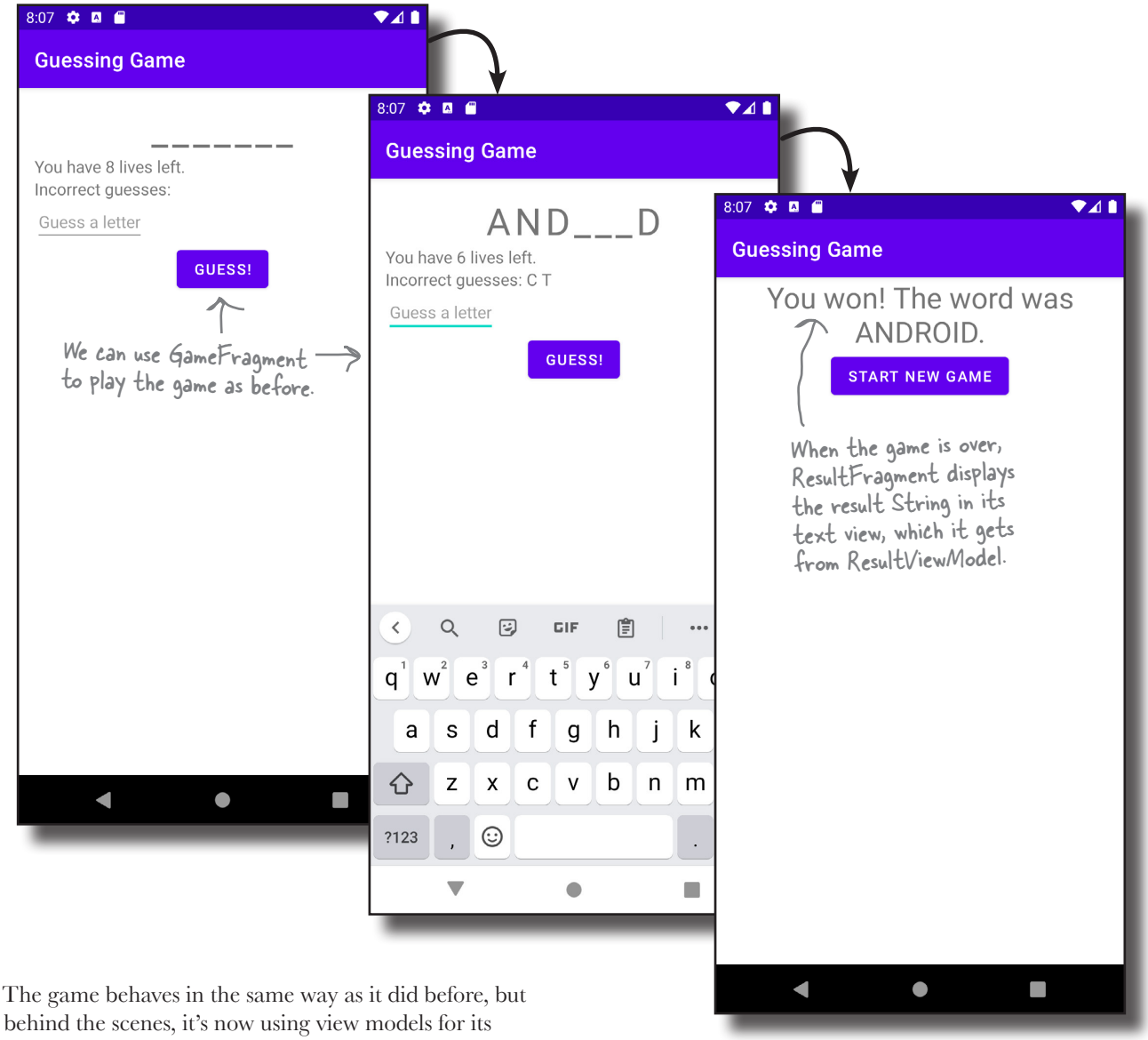

Let's take the app for a test drive.

# Test Drive

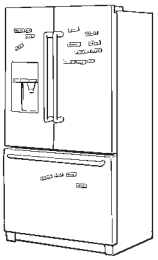
**Basic game**
**GameFragment**
**ResultFragment**

When we run the app, `GameFragment` is displayed as before.

If we guess all the letters or lose all our lives, the app navigates to `ResultFragment`. A message is displayed telling us whether we won or lost, and what the secret word was.




The game behaves in the same way as it did before, but behind the scenes, it's now using view models for its game logic and data.



# View Model Magnets

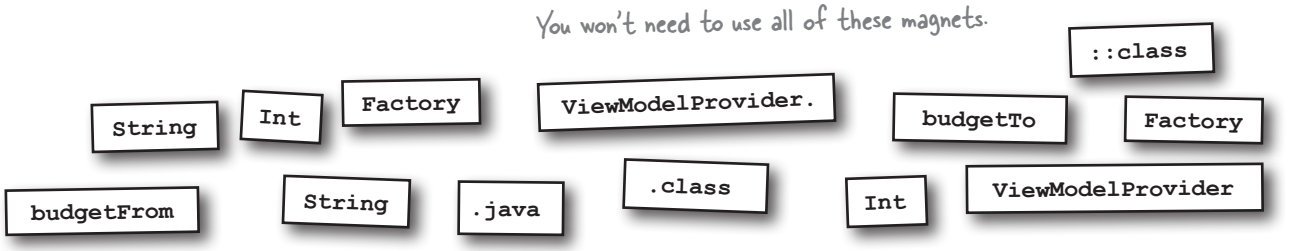
Here's the code to define a view model named `GiftViewModel`:

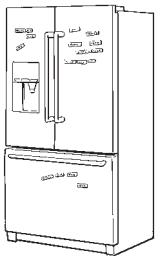
```
import androidx.lifecycle.ViewModel

class GiftViewModel(budgetFrom: Int, budgetTo: Int) : ViewModel(){
    val from = budgetFrom
    val to = budgetTo
}
```

See if you can piece together the code for the view model factory class that will be used to create `GiftViewModel` objects.

```
import androidx.lifecycle.ViewModel

import androidx.lifecycle.....
import java.lang.IllegalArgumentException

class GiftViewModelFactory(private val budgetFrom: ....., private val budgetTo: .....)

                           : ..... {

    override fun <T : ViewModel?> create(modelClass: Class<T>): T {

        if (modelClass.isAssignableFrom(GiftViewModel.....))

            return GiftViewModel(....., .....) as T

        throw IllegalArgumentException("Unknown ViewModel")
    }
}
```

You won't need to use all of these magnets.





# View Model Magnets Solution

Here's the code to define a view model named `GiftViewModel`:

```
import androidx.lifecycle.ViewModel

class GiftViewModel(budgetFrom: Int, budgetTo: Int) : ViewModel(){
    val from = budgetFrom
    val to = budgetTo
}
```

See if you can piece together the code for the view model factory class that will be used to create `GiftViewModel` objects.

```
import androidx.lifecycle.ViewModel
import androidx.lifecycle.ViewModelProvider
import java.lang.IllegalArgumentException

class GiftViewModelFactory(private val budgetFrom: Int, private val budgetTo: Int)
    : ViewModelProvider.Factory {

    override fun <T : ViewModel?> create(modelClass: Class<T>): T {

        if (modelClass.isAssignableFrom(GiftViewModel::class.java)) {

            return GiftViewModel(budgetFrom, budgetTo) as T

        throw IllegalArgumentException("Unknown ViewModel")
    }
}
```

Import ViewModelProvider.

The factory needs two Int values in order to create GiftViewModel.

The class needs to extend ViewModelProvider.Factory.

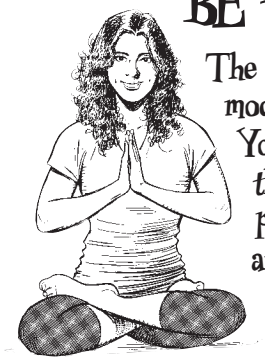
isAssignableFrom() needs a reference to the GiftViewModel class.

Construct the GiftViewModel object.

You didn't need to use these magnets.

`String` `Factory` `String` `.class`

# BE the View Model Solution



The code below describes a view model class named MyViewModel. Your job is to play like you're the view model, and say what problems there are with this code and how they could be fixed.

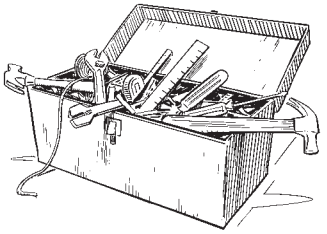
```
package com.hfad.myapp

import androidx.lifecycle.ViewModel
import android.util.Log
import android.widget.TextView

class MyViewModel : ViewModel() {
    val num = 2

    init {
        Log.i("MyViewModel", "ViewModel created")
    }

    override fun onCleared() {
        Log.i("MyViewModel", "ViewModel cleared")
    }

    fun calculation(val1: Int, val2: Int): Int {
        Log.i("MyViewModel", "Called Calculation")
        return (val1 + val2) * num
    }

    fun joinTogether(text1: TextView String, text2: TextView String): String {
        Log.i("MyViewModel", "Called JoinTogether")
        return ("${text1.text} ${text2.text}")
    }
}
```

View models shouldn't reference any views, so this import isn't needed.

The class needs to extend ViewModel.

View models shouldn't refer to views, so change this method to use String values instead.

# Your Android Toolbox



**You've got Chapter 11 under your belt and now you've added view models to your toolbox.**

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

## Bullet Points

- A view model simplifies your activity or fragment code by separating out any code relating to its business logic and data.
- View models can survive configuration changes.
- A view model is usually associated with a single UI controller.
- The view model extends the class `androidx.lifecycle.ViewModel`.
- Create a view model by calling the `ViewModelProvider` class `get()` method. This ensures that a view model is only created where one doesn't already exist.
- A view model stays alive until its UI controller permanently goes away. This is when the activity is finished, or the fragment is removed from its activity.
- If a view model's constructor requires arguments, you need to write an additional factory class for it. The `ViewModelProvider` class uses the factory each time it needs to create an instance of the view model.
- The view model factory class must implement an interface named `ViewModelProvider.Factory`.

# 12 live data

# Leaping into Action

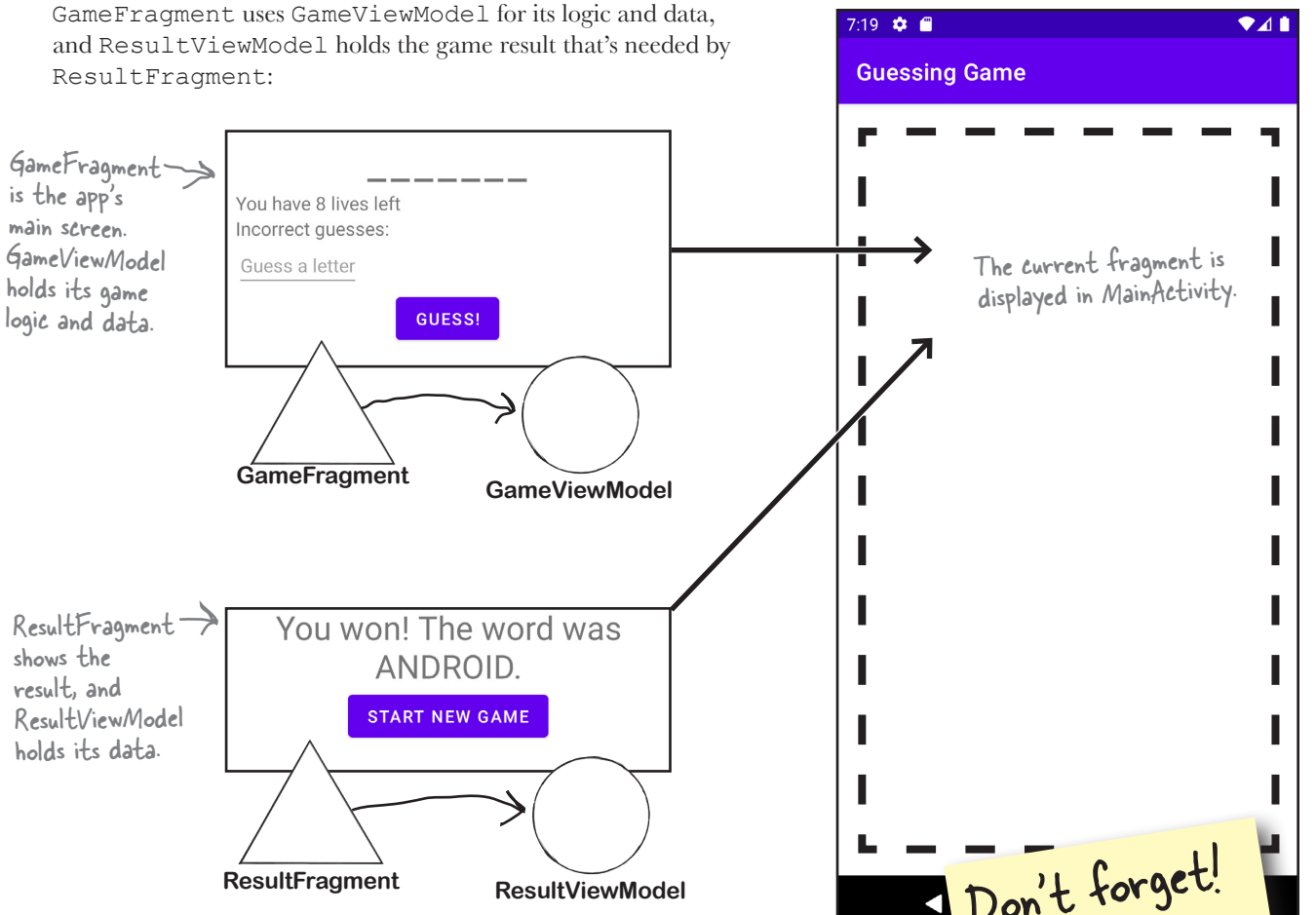
## Your code often needs to react to property value changes.

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**. Soon, you'll be writing apps that are **more responsive than ever before**...

# The Guessing Game app revisited

In the previous chapter, we built a Guessing Game app that lets the user guess which letters are included in a secret word. When the user guesses all the letters or runs out of lives, the game ends.

To stop the fragment code from becoming too bloated and maintain the app's state when the user rotated the device screen, we used view models for the app's game logic and data. `GameFragment` uses `GameViewModel` for its logic and data, and `ResultViewModel` holds the game result that's needed by `ResultFragment`:

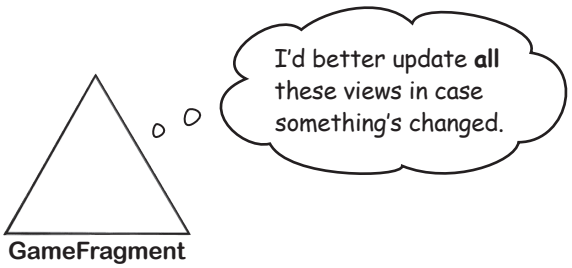



When each fragment is displayed, or the user makes a guess, the fragments get the latest values from their view models and display them on the screen.

While this approach works, however, there are some disadvantages.

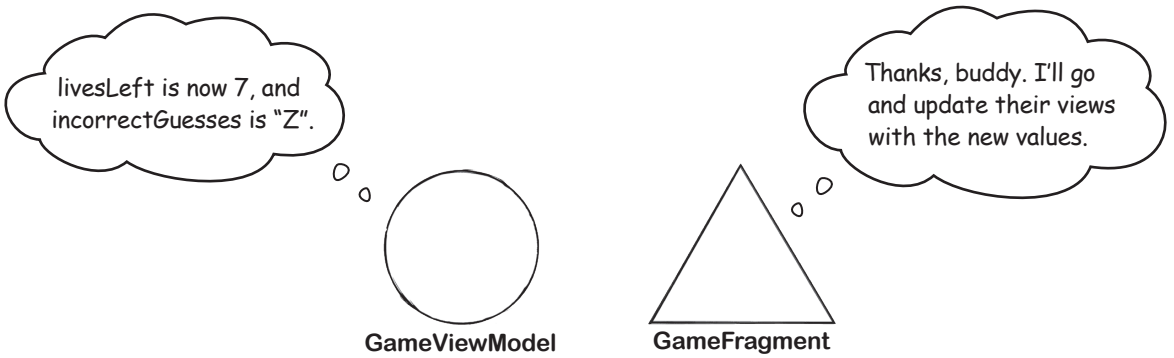
# The fragments decide when to update views

The downside to this approach is that each fragment decides when to fetch the latest property values from the view model and update its views. And some of the time, these values won't have changed. If the user makes a correct guess, for example, `GameFragment` updates the text that's displayed for the number of lives left and the incorrect guesses, even though these values won't have changed.




## Let the view model say when a value's changed

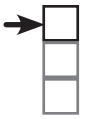
An alternative approach is for `GameViewModel` to let `GameFragment` know when each of its properties has been updated. If the fragment is notified of these changes, it will no longer need to decide for itself when to fetch the latest property values from the view model and update its views. Instead, it will only need to update its views once it's told that the underlying property has been updated.




We're going to implement this change in the Guessing Game app using Android's **live data** library: part of Android Jetpack. Live data lets the view model tell interested parties—such as fragments and activities—when its property values have been updated. They can then react to these changes by updating views, or calling other methods.

You'll find out how to use live data through the rest of this chapter. First, let's go through the steps we'll take to update the app.

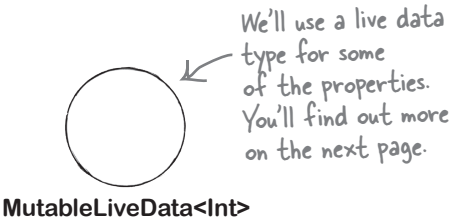
# Here's what we're going to do



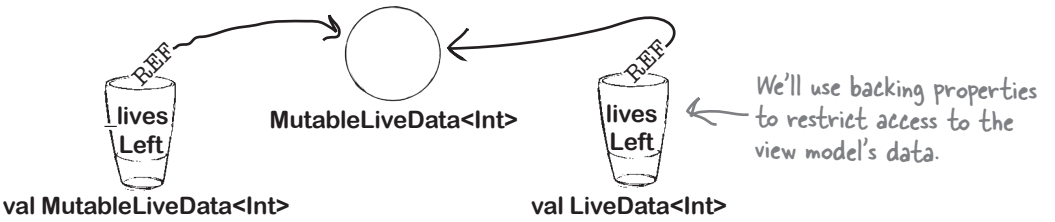
**Use live data**
**Protect properties**
**Add gameOver**

Here are the steps that we'll go through to write the app:

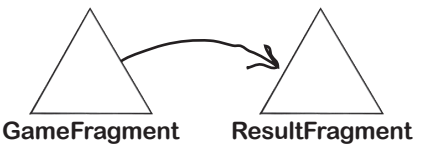
1. **Make the Guessing Game app use live data.**
We'll update `GameViewModel` so that the `livesLeft`, `incorrectGuesses`, and `secretWordDisplay` properties use live data. We'll then make `GameFragment` update its views when the values of these properties change.




2. **Protect GameViewModel's properties and methods.**
We'll restrict access to `GameViewModel`'s properties so that only `GameViewModel` can update them. We'll also make sure that `GameFragment` only has access to the methods it requires to do its job.




3. **Add a gameOver property.**
We'll make `GameViewModel` decide when each game is over using a new `gameOver` property. `GameFragment` will navigate to `ResultFragment` when the value of this property changes.



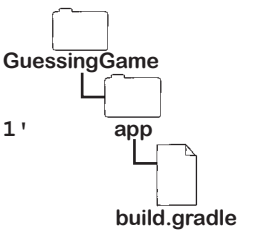

## Add a live data dependency to the app build.gradle file

As we're going to be using live data, we'll begin by adding a live data dependency to the app's *build.gradle* file.

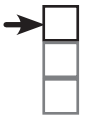
Open the project for the Guessing Game app (if you haven't already done so), open the file *GuessingGame/app/build.gradle*, and add the following line (in bold) to the `dependencies` section:

```
dependencies {
    ...
    implementation 'androidx.lifecycle:lifecycle-livedata-ktx:2.3.1'
    ...
}
```

This adds the live data library to the app.




When prompted, sync your changes.



**Use live data**
Protect properties
Add gameOver

# GameViewModel and GameFragment need to use live data

We want to use live data in the Guessing Game app so that `GameViewModel` notifies `GameFragment` when its property values have changed. `GameFragment` will then react to these changes.

We'll tackle this in two stages:

1. **Specify which GameViewModel property changes GameFragment needs to know about.**
2. **Tell GameFragment how to respond to each change.**

For now, we'll focus on the `GameViewModel` code changes.

## Which view model properties should use live data?

`GameViewModel` includes three properties—`secretWordDisplay`, `incorrectGuesses`, and `livesLeft`—that `GameFragment` uses to update its views. We'll specify that these three properties use live data so that `GameFragment` is notified when their values change.

You specify that a property uses live data by changing its type to **`MutableLiveData<Type>`**, where `Type` is the type of data the property should hold. The `livesLeft` property, for example, is currently defined as an `Int` using the code:

```
var livesLeft = 8
```

To make the property use live data, you change its type to `MutableLiveData<Int>` so that it looks like this instead:

```
var livesLeft = MutableLiveData<Int>(8)
```

This initializes the MutableLiveData<Int> with an Int value of 8.

This specifies that `livesLeft` is now a `MutableLiveData<Int>` with an initial value of 8.

Similarly, we can define the `incorrectGuesses` and `secretWordDisplay` properties using code like this

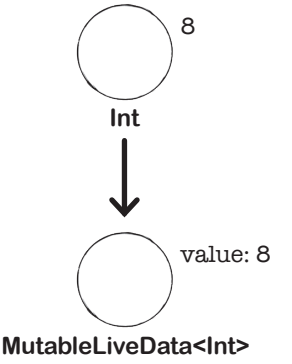
```
var incorrectGuesses = MutableLiveData<String>("")
var secretWordDisplay = MutableLiveData<String>()
```

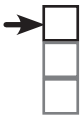
We're not setting secretWordDisplay's value here because we're going to do it in the view model's init block instead.

Here, each property's type is set to `MutableLiveData<String>`. `incorrectGuesses`'s value is set to `""`, while `secretWordDisplay`'s value will get set in `GameViewModel`'s `init` block.

So that's how you define a live data property. Next up, how to update its value.

The livesLeft property goes from being an Int to a MutableLiveData<Int>.





**Use live data**
**Protect properties**
**Add gameOver**

# Live data objects use a value property

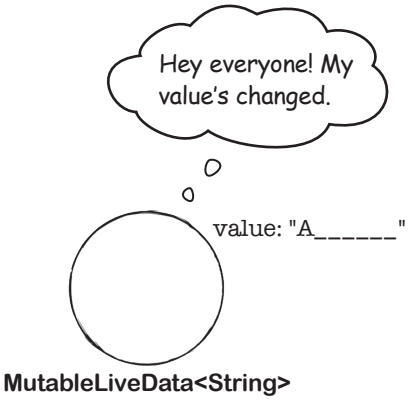
When you use `MutableLiveData` properties, you update their values using a property named **`value`**. To update the `secretWordDisplay` property with the `deriveSecretWordDisplay()` method's return value, for example, you don't use the code:

```
secretWordDisplay = deriveSecretWordDisplay()
```

as you did before. You use the following code instead:

```
secretWordDisplay.value = deriveSecretWordDisplay()
```

Changing the `value` property in this way is important, because it's how any interested parties—in this case `GameFragment`—get notified of any changes. Each time `secretWordDisplay`'s `value` property gets updated, `GameFragment` is informed, so it can respond by updating its views.




## The value property can be null

There's one extra thing to be aware of when you use live data: the `value`'s type is *nullable*. This means that when you use live data values in your code, you need to perform null-safety checks, or your code won't compile.

As an example, the `livesLeft` property is defined using the code:

```
var livesLeft = MutableLiveData<Int>(8)
```

This means that the livesLeft property's value can be set to an Int or be null.

This property's type is `MutableLiveData<Int>`, so its `value` property can accept an `Int`, or be `null`.

As the `value` property may be `null`, we can't, say subtract one from its value using the code:

```
livesLeft.value--
```

Instead, we need to use:

```
livesLeft.value = livesLeft.value?.minus(1)
```

This subtracts one from the value, so long as it isn't null.

which subtracts 1 from its value so long as it isn't `null`.

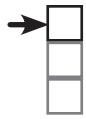
Similarly, the following `isLost()` method won't compile because `liveLeft.value` may be `null`:

```
fun isLost() = livesLeft.value <= 0
```

We can, however, change it to use Kotlin's Elvis operator like this:

```
fun isLost() = livesLeft.value ?: 0 <= 0
```

livesLeft.value ?: 0 returns 0 if value is null. Here, isLost() returns true if the value is null or <= 0.



**Use live data**
**Protect properties**
**Add gameOver**

That's got me thinking... You said that whenever you need to update a live data property, you assign a new value to the existing object's value. You don't replace the live data object itself. Does that mean we can define the live data properties using val instead of var?

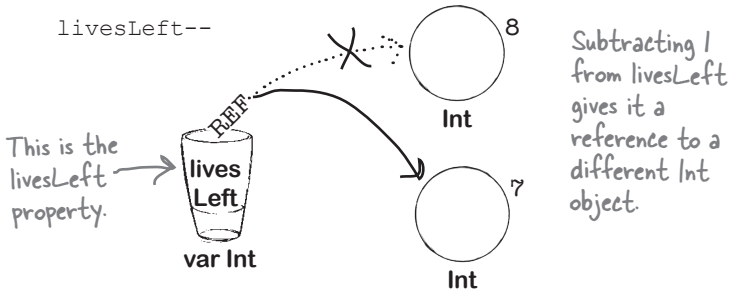
## Live Data properties can be defined using val.

As you already know, you use `val` and `var` in Kotlin to specify whether or not a property can be given a reference to a new object.

When we first defined each property, we used `var` so that we could update it. We defined the `livesLeft` property, for example, using:

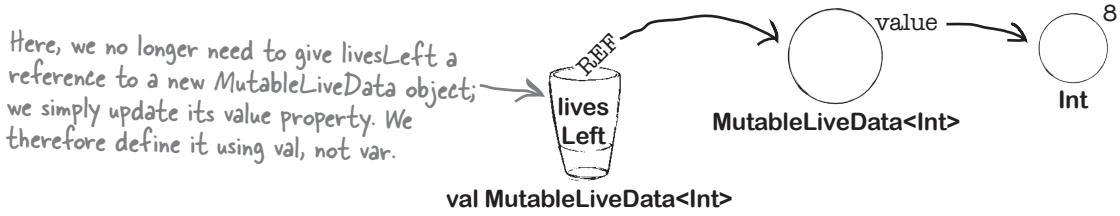
```
var livesLeft = 8
```

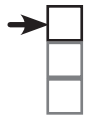
which initialized `livesLeft` with an `Int` object whose value is 8. Each time the user made an incorrect guess, we subtracted 1 from `livesLeft`, which gave it a reference to a new `Int` object:




With live data, you update the `value` property of the *existing* object, instead of replacing it with another object, and interested parties are notified of this change. As the object is no longer being replaced, you can define the property using `val` instead of `var` like this:

```
val livesLeft = MutableLiveData<Int>(8)
```



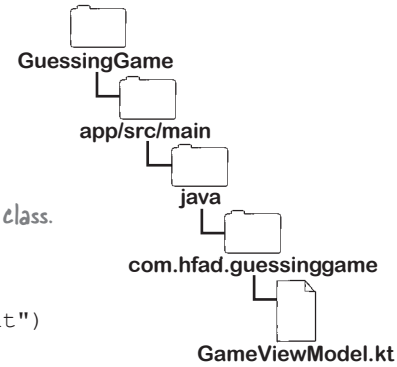


**Use live data**
**Protect properties**
**Add gameOver**

# The full code for GameViewModel.kt

Now that you know how live data works, let's update `GameViewModel` so that its `livesLeft`, `incorrectGuesses`, and `secretWordDisplay` properties all use live data.

Here's the full code for *GameViewModel.kt*; update the code to include the changes (in bold):




```
package com.hfad.guessinggame

import androidx.lifecycle.ViewModel
import androidx.lifecycle.MutableLiveData

class GameViewModel : ViewModel() {
    val words = listOf("Android", "Activity", "Fragment")
    val secretWord = words.random().uppercase()
    var secretWordDisplay = MutableLiveData<String>()
    var correctGuesses = ""
    var incorrectGuesses = MutableLiveData<String>("")
    var livesLeft = MutableLiveData<Int>(8)

    init {
        secretWordDisplay.value = deriveSecretWordDisplay()
    }

    fun deriveSecretWordDisplay() : String {
        var display = ""
        secretWord.forEach {
            display += checkLetter(it.toString())
        }
        return display
    }

    fun checkLetter(str: String) = when (correctGuesses.contains(str)) {
        true -> str
        false -> "_"
    }
```

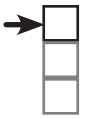
Import this class.

Define the live data properties using val, not var.

Change the types of these properties so they use live data.

We need to use secretWordDisplay's value property.

The code continues on the next page.



**Use live data**
Protect properties
Add gameOver

# GameViewModel.kt continued

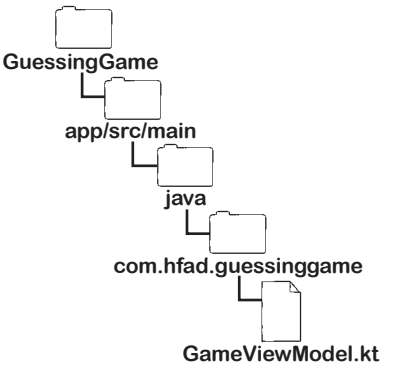
```
    fun makeGuess(guess: String) {
        if (guess.length == 1) {
            if (secretWord.contains(guess)) {
                correctGuesses += guess
                secretWordDisplay.value = deriveSecretWordDisplay()
            } else {
                incorrectGuesses.value += "$guess "
                livesLeft--
                livesLeft.value = livesLeft.value?.minus(1)
            }
        }
    }

    fun isWon() = secretWord.equals(secretWordDisplay.value, true)

    fun isLost() = livesLeft.value ?: 0 <= 0

    fun wonLostMessage() : String {
        var message = ""
        if (isWon()) message = "You won!"
        else if (isLost()) message = "You lost!"
        message += " The word was $secretWord."
        return message
    }
}
```

We need to use the value property for secretWordDisplay and incorrectGuesses.

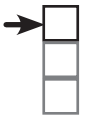
This subtracts 1 from livesLeft's value in a way that's null-safe.

We need to use .value here too.

If the value is null, the Elvis operator returns 0. This means that if the livesLeft value is null, or less than or equal to 0, isLost() returns true.




That's everything that we need for `GameViewModel`. Next up, let's make `GameFragment` respond when the `livesLeft`, `incorrectGuesses`, and `secretWordDisplay` properties get updated.



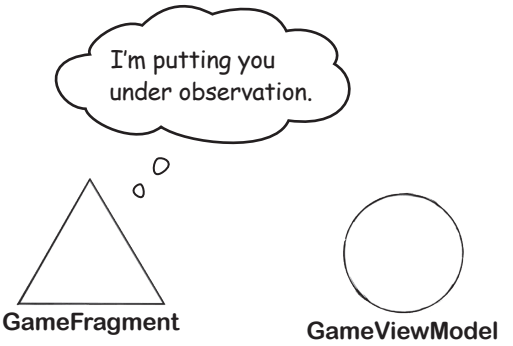
**Use live data**
Protect properties
Add gameOver

# The fragment observes the view model properties and reacts to changes.

You make a fragment respond to `value` changes in a view model's `MutableLiveData` property by calling the property's **`observe()`** method. If the following code is added to `GameFragment`, for example, it will observe the view model's `livesLeft` property, and take action when it changes:

Adding this code to your fragment lets it respond when the property's value changes.

```
viewModel.livesLeft.observe(viewLifecycleOwner, Observer { newValue ->
    //Code that uses the new value
})
```




As you can see, the above code passes `viewLifecycleOwner` and `Observer` arguments to the `observe()` method.

`viewLifecycleOwner` refers to the lifecycle of the fragment's views. It's tied to when the fragment has access to its UI: from when it's created in the fragment's `onCreateView()` method, to when its destroyed, and `onDestroyView()` gets called.

`Observer` is a class that can receive live data. It's tied to the `viewLifecycleOwner`, so it's only active—and able to receive live data notifications—when the fragment has access to its views. If the live data property's value changes while the fragment doesn't have access to its UI, the observer isn't notified, so the fragment doesn't respond. This stops the fragment from trying to update views when they're not available, which might cause the app to crash.

It also makes your coding life easier because you don't need to check whether the view's available for yourself. Using live data handles it all for you.

The `Observer` class accepts a lambda parameter, which specifies how the property's new value should be used. In the Guessing Game app, for example, we want `GameFragment` to update its `lives` text each time the view model's `livesLeft` property gets updated, and we can achieve this using the following code:

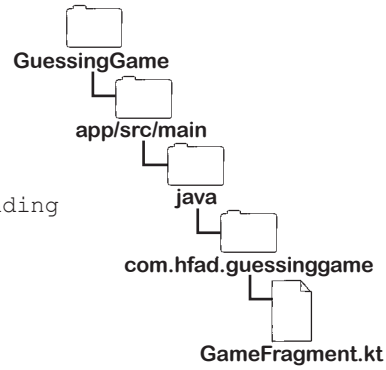
```
viewModel.livesLeft.observe(viewLifecycleOwner, Observer { newValue ->
    binding.lives.text = "You have $newValue lives left"
})
```

This shows the user how many lives are now left.

That's everything that we need to know in order to make `GameFragment` update its views when the view model property values get changed. Let's update its code.

# The full code for GameFragment.kt

Here's the updated code for `GameFragment`; make sure that the file *GameFragment.kt* includes the changes shown below (in bold):




```
package com.hfad.guessinggame

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.guessinggame.databinding.FragmentGameBinding
import androidx.navigation.findNavController
import androidx.lifecycle.ViewModelProvider
import androidx.lifecycle.Observer

class GameFragment : Fragment() {
    private var _binding: FragmentGameBinding? = null
    private val binding get() = _binding!!
    lateinit var viewModel: GameViewModel

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentGameBinding.inflate(inflater, container, false)
        val view = binding.root
        viewModel = ViewModelProvider(this).get(GameViewModel::class.java)
        updateScreen()
        viewModel.incorrectGuesses.observe(viewLifecycleOwner, Observer { newValue ->
            binding.incorrectGuesses.text = "Incorrect guesses: $newValue"
        })
        viewModel.livesLeft.observe(viewLifecycleOwner, Observer { newValue ->
            binding.lives.text = "You have $newValue lives left"
        })
```

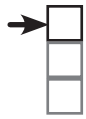
Import this class.

We no longer need this method, as we're using live data to update the values displayed on the screen.

Each time the incorrectGuesses and livesLeft properties in the view model get changed, update the text in the layout.

The code continues on the next page.

# GameFragment.kt continued



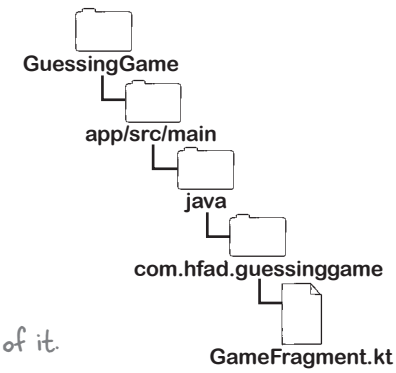
**Use live data**
**Protect properties**
**Add gameOver**

```
        viewModel.secretWordDisplay.observe(viewLifecycleOwner, Observer { newValue ->
            binding.word.text = newValue
        })

        binding.guessButton.setOnClickListener() {
            viewModel.makeGuess(binding.guess.text.toString().uppercase())
            binding.guess.text = null
            updateScreen()
            if (viewModel.isWon() || viewModel.isLost()) {
                val action = GameFragmentDirections
                    .actionGameFragmentToResultFragment(viewModel.wonLostMessage())
                view.findNavController().navigate(action)
            }
        }
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }

    fun updateScreen() {
        binding.word.text = viewModel.secretWordDisplay
        binding.lives.text = "You have ${viewModel.livesLeft} lives left"
        binding.incorrectGuesses.text = "Incorrect guesses:${viewModel.incorrectGuesses}"
    }
}
```

This updates the word view's text when the secretWordDisplay property in the view model gets updated.

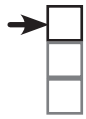
Remove this line, as we're using live data to update the screen.

This method is no longer needed, so get rid of it.




Those are all the changes we need to make to `GameFragment` so that it updates its views each time it observes changes to the `incorrectGuesses`, `livesLeft`, and `secretWordDisplay` properties.

We've now implemented live data in the Guessing Game app Let's go through what happens when the app runs.



**Use live data**
Protect properties
Add gameOver

Wait a second! Aren't you **forgetting something?** The Guessing Game app has **two** view models, not one. Don't we need to update ResultViewModel and ResultFragment too?

## ResultViewModel doesn't need to use live data, so we don't need to update it.

As you may recall, `ResultViewModel` has one property (named `result`), which gets set when the view model gets created. The code looks like this:

```
class ResultViewModel(finalResult: String) : ViewModel(){
    val result = finalResult
}
```

As you can see, result is defined using `val`, so once it's been initialized, it can't be updated to another value. `ResultFragment` doesn't need to be notified when `result` changes because **once set, `result` can never change**. There's no need to make `ResultFragment` respond to any changes, because there can't be any.

So let's walk through what happens when the code runs, and then take the app for a test drive.

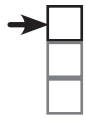
## there are no Dumb Questions

**Q: What happens if `GameFragment` is inactive when one of `GameViewModel`'s live data properties gets updated? Won't this make the app crash?**

A: No. Live data is lifecycle-aware, so it only notifies active lifecycle observers when its property values have changed. If `GameFragment` is inactive when `GameViewModel`'s properties get updated, `GameFragment` won't get notified, so you don't have to worry about the app crashing.

**Q: Is there a `LiveData` type too?**

A: Yes! The `LiveData` type is like `MutableLiveData` except that its `value` property can't be updated. You'll find out how the `LiveData` type is used a few pages ahead.

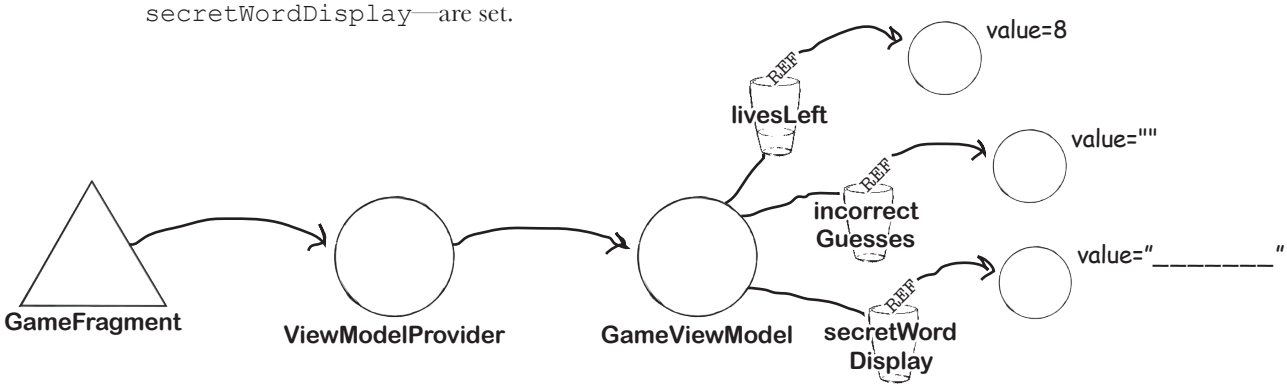


**Use live data**
**Protect properties**
**Add gameOver**

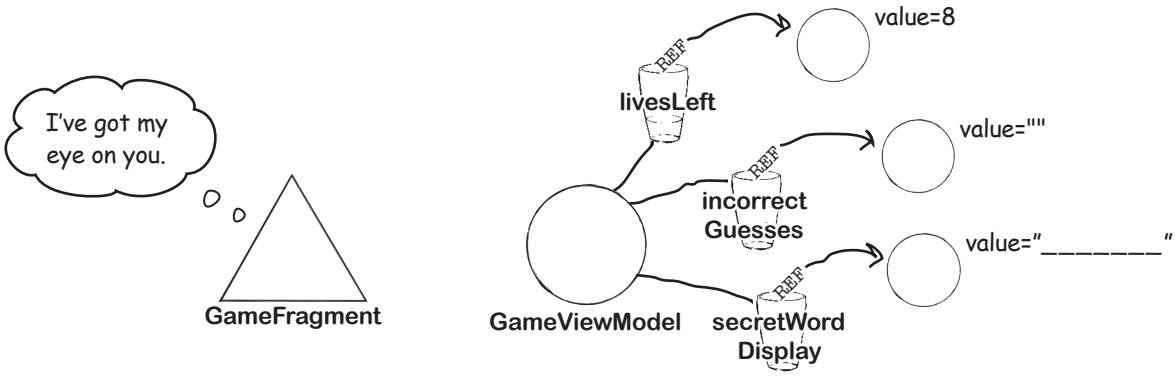
# What happens when the app runs

The following things happen when the app runs:

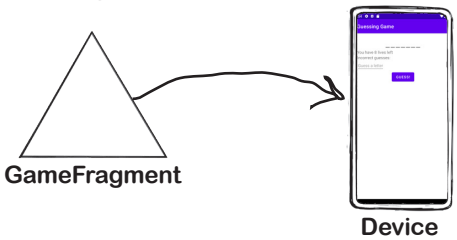
1. **GameFragment asks the ViewModelProvider class for an instance of GameViewModel.**
   The `GameViewModel` object is initialized, and the values of its three `MutableLiveData` properties—`livesLeft`, `incorrectGuesses`, and `secretWordDisplay`—are set.




2. **GameFragment observes the livesLeft, incorrectGuesses, and secretWordDisplay properties in the GameViewModel.**

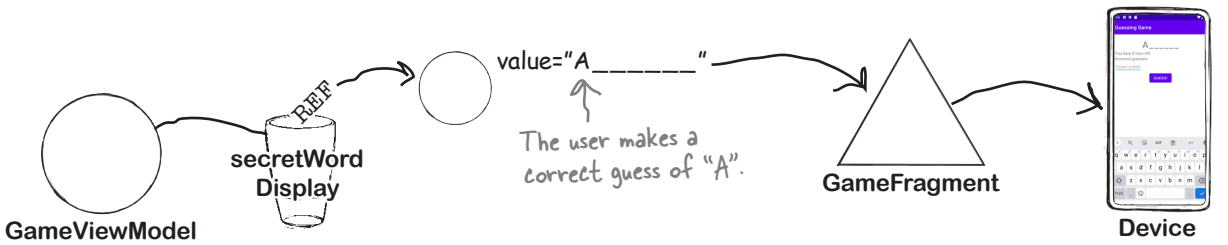



3. **GameFragment updates its views with the values of the properties it's observing.**

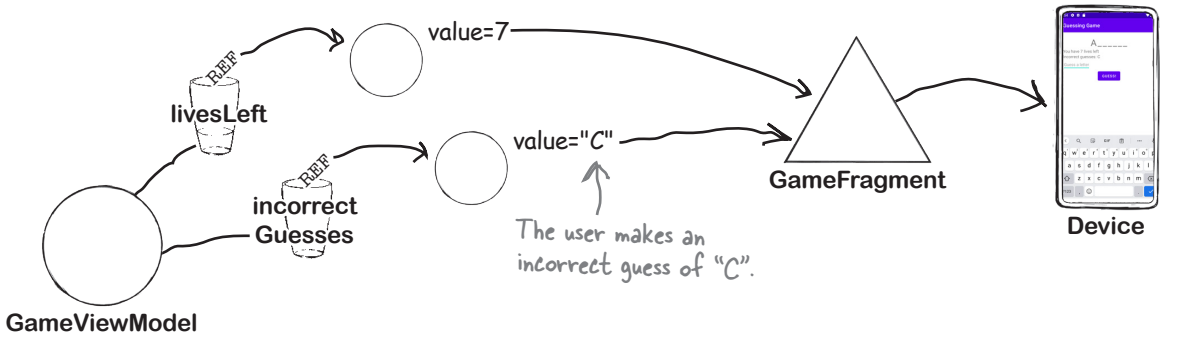


# The story continues

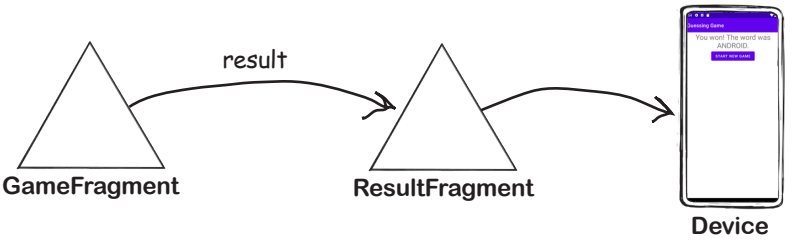
**4** **When the user makes a correct guess, secretWordDisplay's value is updated, and the new value is passed to GameFragment.**
`GameFragment` responds by updating its `word` view, which is displayed on the screen.




**5** **When the user makes an incorrect guess, the values for incorrectGuesses and livesLeft are updated, and passed to GameFragment.**
`GameFragment` responds by updating its views.




**6** **When isWon() or isLost() return true, GameFragment navigates to ResultFragment, passing it the result.**
`ResultFragment` displays the result.




Let's take the app for a test drive.

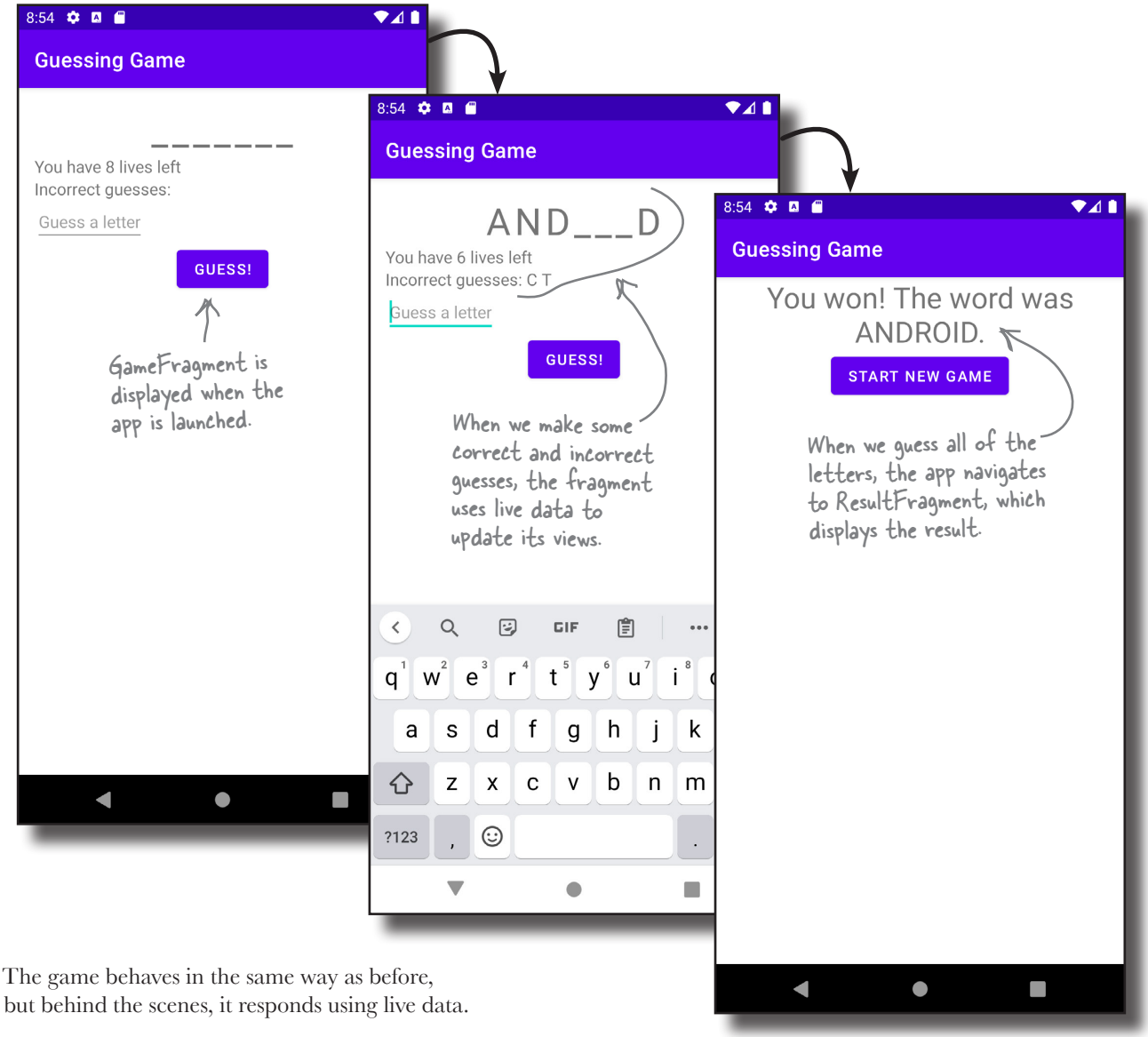
# Test Drive

**Use live data**
**Protect properties**
**Add gameOver**

When we run the app, `GameFragment` is displayed as before.

When we make a correct guess, the secret word display gets updated. When we make an incorrect guess, the lives left is updated, and our guess is added to the incorrect guesses display.

If we guess all the letters or lose all our lives, the app navigates to `ResultFragment`, which displays the result.




The game behaves in the same way as before, but behind the scenes, it responds using live data.

## Sharpen your pencil

The code below describes a view model named `MyViewModel`. See if you can change the code so that it uses live data to let fragments respond to the `rate` property changing.

```
package com.hfad.myapp

import androidx.lifecycle.ViewModel
import android.util.Log

class MyViewModel : ViewModel() {
    var rate = 2

    init {
        Log.i("MyViewModel", "ViewModel created")
    }

    override fun onCleared() {
        Log.i("MyViewModel", "ViewModel cleared")
    }

    fun updateRate(newRate: Int) {
        rate = newRate
    }

    fun Calculation(val1: Int, val2: Int): Int {
        Log.i("MyViewModel", "Called Calculation")
        return (val1 + val2) * rate
    }

    fun JoinTogether(text1: String, text2: String): String {
        Log.i("MyViewModel", "Called JoinTogether")
        return ("${text1} ${text2}")
    }
}
```

## Sharpen your pencil Solution

The code below describes a view model named `MyViewModel`. See if you can change the code so that it uses live data to let fragments respond to the `rate` property changing.

```
package com.hfad.myapp

import androidx.lifecycle.ViewModel
import android.util.Log
import androidx.lifecycle.MutableLiveData   <- We need to import this class.

class MyViewModel : ViewModel() {
    val ~~var~~ rate = MutableLiveData<Int>(2)   <- The rate property needs to use val instead of var, and its type needs to be MutableLiveData<Int>.

    init {
        Log.i("MyViewModel", "ViewModel created")
    }

    override fun onCleared() {
        Log.i("MyViewModel", "ViewModel cleared")
    }

    fun updateRate(newRate: Int) {
        rate.value = newRate   <- We need to set rate's value property.
    }

    fun Calculation(val1: Int, val2: Int): Int {
        Log.i("MyViewModel", "Called Calculation")
        return (val1 + val2) * rate.value ?: 0   <- We need to use rate's value property. We're using the Elvis operator in case the value is null.
    }

    fun JoinTogether(text1: String, text2: String): String {
        Log.i("MyViewModel", "Called JoinTogether")
        return ("${text1} ${text2}")
    }
}
```

# Fragments can update GameViewModel's properties

So far, we've updated the code in `GameViewModel` and `GameFragment` so that it uses live data. Each time the value of a `MutableLiveData` property in `GameViewModel` gets updated, `GameFragment` responds by updating its views.

There's a slight problem with the code, however. `GameFragment` has full access to `GameViewModel`'s properties and methods, so if it wanted to, it could use them inappropriately. There's nothing to stop the fragment from, say, updating the `livesLeft` property to 100 so that the user can make many more guesses and win the game each time she plays.

To get around this problem, we're going to restrict direct access to the `GameViewModel`'s properties so that **only methods in the view model can update them**.

**You can restrict direct access to a view model's property by marking it private, and providing read-only access via another property's getter.**

## Keep your privates private

To protect `GameViewModel`'s properties, we'll mark each one as `private` so that only code in `GameViewModel` can update their values. We'll then expose a *read-only version* of each `MutableLiveData` property that `GameFragment` needs to observe. Instead of defining the `livesLeft` property using code like this:

```
val livesLeft = MutableLiveData<Int>(8)
```

we'll use the following:

```
private val _livesLeft = MutableLiveData<Int>(8)
val livesLeft: LiveData<Int>
    get() = _livesLeft
```

The MutableLiveData property is marked as private so that no other class has direct access to it.

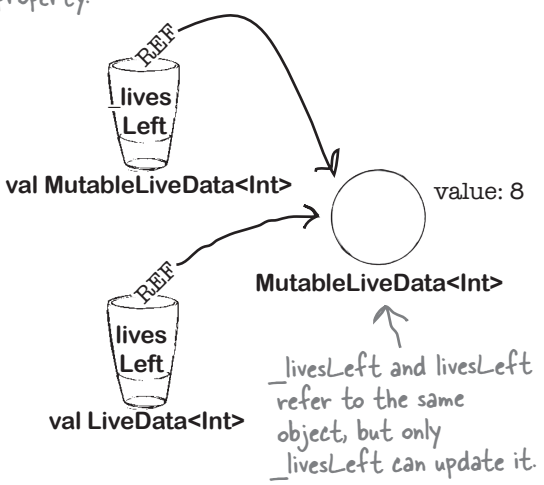
This returns a read-only version of the MutableLiveData property.

Here, the `_livesLeft` property holds a reference to a `MutableLiveData` object. `GameFragment` doesn't have access to this property because it's marked as `private`.

`GameFragment` can, however, access the value of this property using `livesLeft`'s getter. `livesLeft`'s type is **`LiveData`**, which is like `MutableLiveData` except that it can't be used to update the underlying object's `value` property: `GamesFragment` can read the value, but it can't update it.

When you structure your code in this way, the private property is sometimes called a **backing property**. It holds a reference to an object that other classes only have access to via another property.

Let's update the `GameViewModel` code.

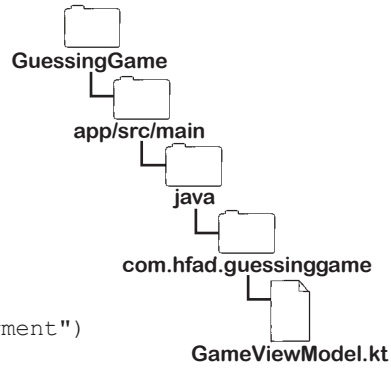


Use live data
**Protect properties**
Add gameOver

# The full code for GameViewModel.kt

We'll update the code for `GameViewModel` so that it uses backing properties to restrict direct access to its live data properties. We'll also mark as private any properties and methods that `GameFragment` doesn't need to use.

Here's the full code for *GameViewModel.kt*; update the code to include our changes (in bold):




```
package com.hfad.guessinggame

import androidx.lifecycle.ViewModel
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.LiveData

class GameViewModel : ViewModel() {
    private val words = listOf("Android", "Activity", "Fragment")
    private val secretWord = words.random().uppercase()
    private val _secretWordDisplay = MutableLiveData<String>()
    val secretWordDisplay: LiveData<String>
        get() = _secretWordDisplay
    private var correctGuesses = ""
    private val _incorrectGuesses = MutableLiveData<String>("")
    val incorrectGuesses: LiveData<String>
        get() = _incorrectGuesses
    private val _livesLeft = MutableLiveData<Int>(8)
    val livesLeft: LiveData<Int>
        get() = _livesLeft

    init {
        _secretWordDisplay.value = deriveSecretWordDisplay()
    }
```

Import this class.

Mark these as private.

Use secretWordDisplay to provide read-only access to the _secretWordDisplay backing property.

These need to be private.

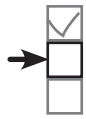
incorrectGuesses provides read-only access to its _incorrectGuesses backing property.

This is private too.

livesLeft returns a read-only version of the _livesLeft backing property.

We're updating the property value, so we need to use _secretWordDisplay instead of secretWordDisplay.

The code continues on the next page.



**Use live data**
**Protect properties**
**Add gameOver**

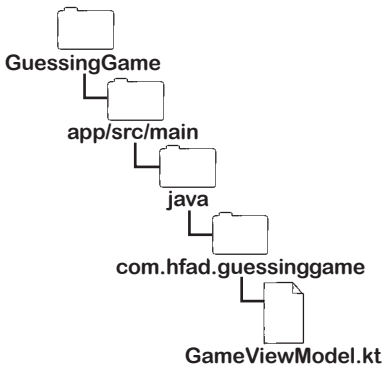
# GameViewModel.kt continued

Only GameViewModel needs to call these methods, so mark them as private.

```
    private fun deriveSecretWordDisplay() : String {
        var display = ""
        secretWord.forEach {
            display += checkLetter(it.toString())
        }
        return display
    }

    private fun checkLetter(str: String) = when (correctGuesses.contains(str)) {
        true -> str
        false -> "_"
    }

    fun makeGuess(guess: String) {
        if (guess.length == 1) {
            if (secretWord.contains(guess)) {
                correctGuesses += guess
                _secretWordDisplay.value = deriveSecretWordDisplay()
            } else {
                _incorrectGuesses.value += "$guess "
                _livesLeft.value = _livesLeft.value?.minus(1)
            }
        }
    }

    fun isWon() = secretWord.equals(secretWordDisplay.value, true)

    fun isLost() = livesLeft.value ?: 0 <= 0

    fun wonLostMessage() : String {
        var message = ""
        if (isWon()) message = "You won!"
        else if (isLost()) message = "You lost!"
        message += " The word was $secretWord."
        return message
    }
}
```

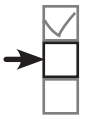
We need to use the mutable versions of these properties, so prefix them with "_".

Add this "_" as well.




Let's go through what happens when the code runs.

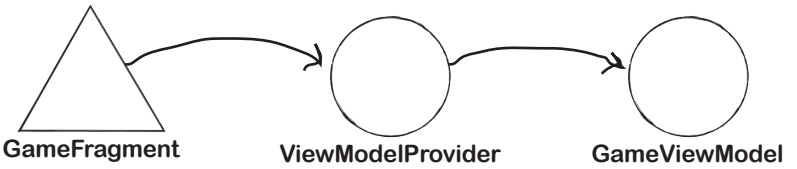
# What happens when the app runs



**Use live data**
**Protect properties**
**Add gameOver**

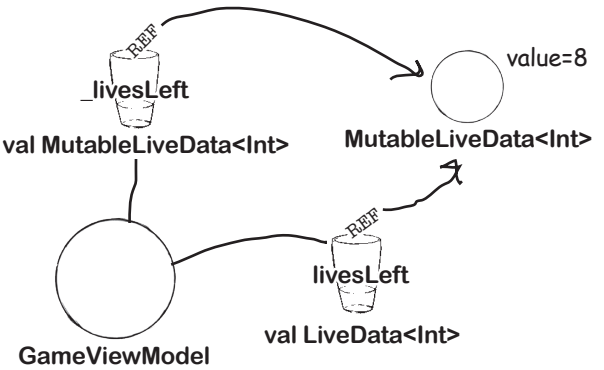
The following things happen when the app runs:

1. **GameFragment asks the ViewModelProvider class for an instance of GameViewModel.**

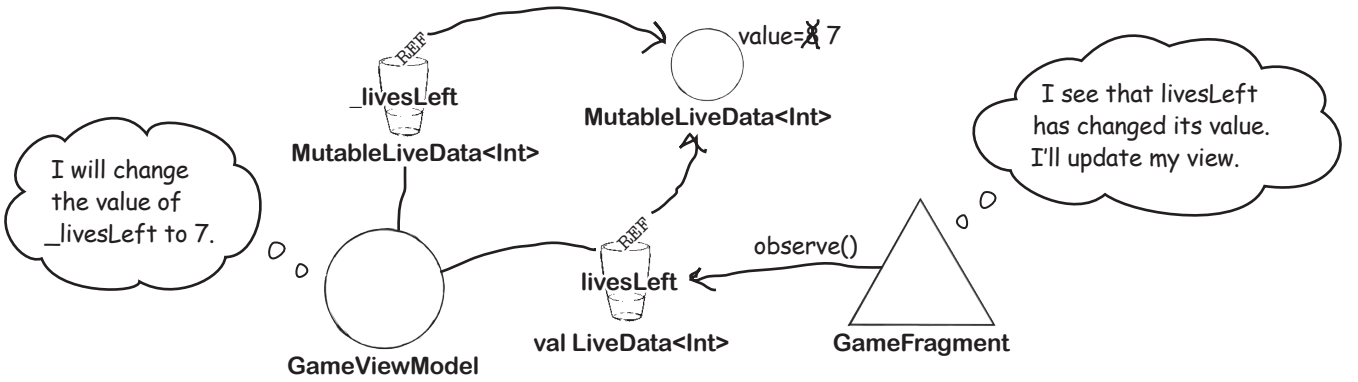



2. **GameViewModel's properties are initialized.**
`livesLeft`, `incorrectGuesses`, and `secretWordDisplay` are `LiveData` properties that refer to the same underlying objects as their `MutableLiveData` backing properties.

We're just showing you the _livesLeft and livesLeft properties to make the diagrams a bit simpler.



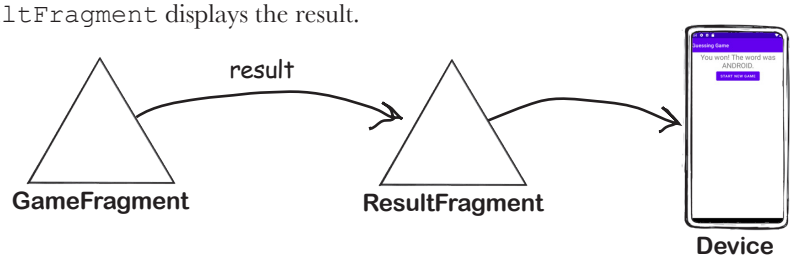

3. **GameFragment observes the livesLeft, incorrectGuesses, and secretWordDisplay properties.**
`GameFragment` can't update any of these properties, but it responds when `GameViewModel` updates any of the backing properties because they refer to the same underlying object.



# The story continues

4 **GameFragment continues to respond to value changes until isWon() or isLost() returns true.**

`GameFragment` navigates to `ResultFragment`, passing it the result. `ResultFragment` displays the result.

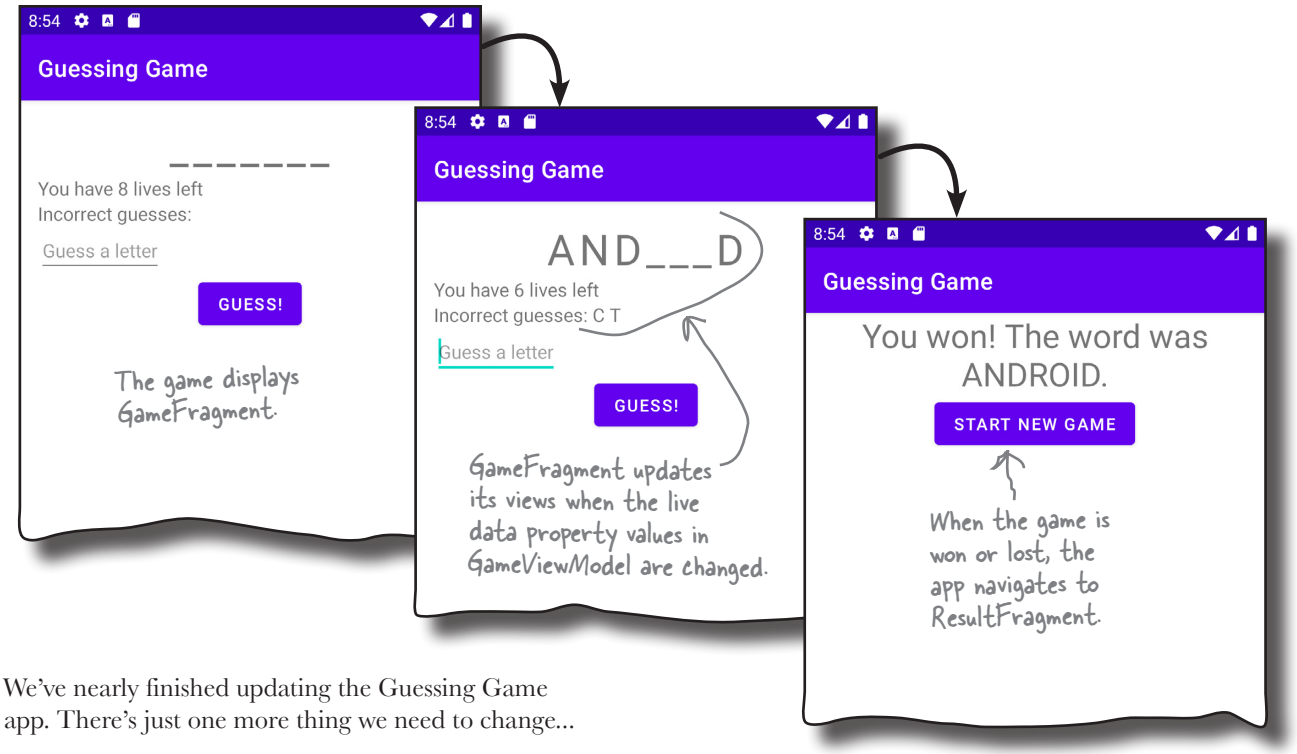



Let's take the app for a test drive.

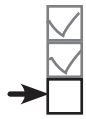
## Test Drive

When we run the app, it works in the same way as before. This time, however, we've protected `GameViewModel`'s `MutableLiveData` properties by restricting `GameFragment`'s access to them.

**Use live data**
**Protect properties**
**Add gameOver**




We've nearly finished updating the Guessing Game app. There's just one more thing we need to change...



Use live data
Protect properties
**Add gameOver**

# GameFragment still includes game logic

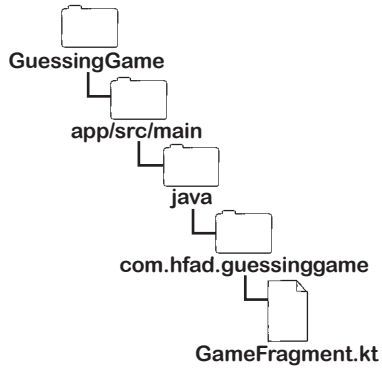
In the current version of the app, `GameFragment` decides when the game is finished by calling `GameViewModel`'s `isWon()` and `isLost()` methods after the user has made each guess. If either of these returns *true*, `GameFragment` navigates to `ResultFragment`, passing it the result.

Here's the current code:

```
...
    binding.guessButton.setOnClickListener() {
        viewModel.makeGuess(binding.guess.text.toString().uppercase())
        binding.guess.text = null
        if (viewModel.isWon() || viewModel.isLost()) {
            val action = GameFragmentDirections
                .actionGameFragmentToResultFragment(viewModel.wonLostMessage())
            view.findNavController().navigate(action)
        }
    }
...
```

Each time the user makes a guess, GameFragment checks if the game has been won or lost.

If the game has been won or lost, GameFragment navigates to ResultFragment.

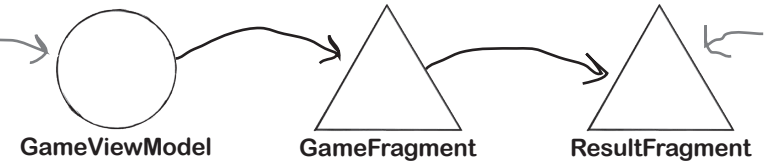



The problem with this approach is that `GameFragment` decides when the game is over instead of `GameViewModel`. As determining when the game has finished is a game decision, it's something that `GameViewModel` should be responsible for, not `GameFragment`.

## Make GameViewModel decide when the game is over

To deal with this, we'll add a `MutableLiveData<Boolean>` property named `_gameOver` to `GameViewModel`, whose value we'll expose using a `LiveData` property named `gameOver`. We'll set this property to *true* when the user wins or loses the game. When this happens, `GameFragment` will respond by navigating to `ResultFragment`.

GameViewModel will use the gameOver property to tell GameFragment when the game is over.




When the game is over, GameFragment will navigate to ResultFragment.

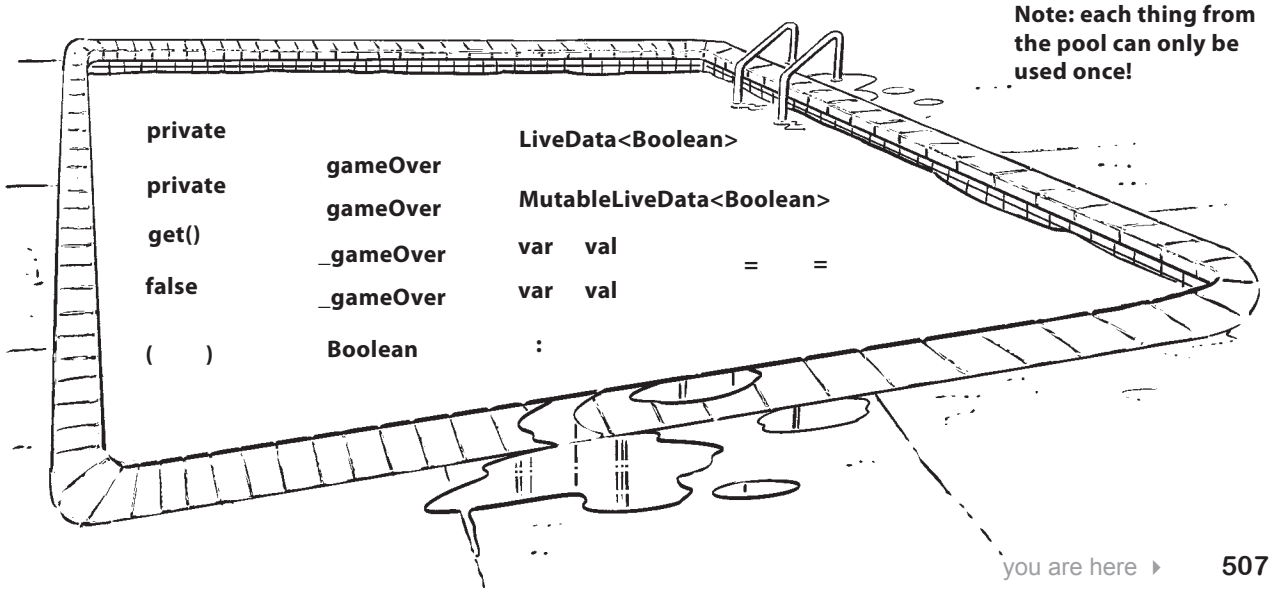
You're already familiar with the changes that are required to do this, so before we show you the code, try the following exercise.

# Pool Puzzle

Your **goal** is to add a `gameOver` property (with a `_gameOver` backing property) to `GameViewModel` so that `GameFragment` can respond when its value (a `Boolean`) gets updated. The property should be initialized with a value of *false*. Take code snippets from the pool and place them into the blank lines in the code. You may **not** use the same snippet more than once, and you won't need to use all the snippets.

```
package com.hfad.guessinggame

import androidx.lifecycle.ViewModel
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.LiveData

class GameViewModel : ViewModel() {
    ...
    ..........
    ..........
    ..........
    ...
}
```

**Note: each thing from the pool can only be used once!**



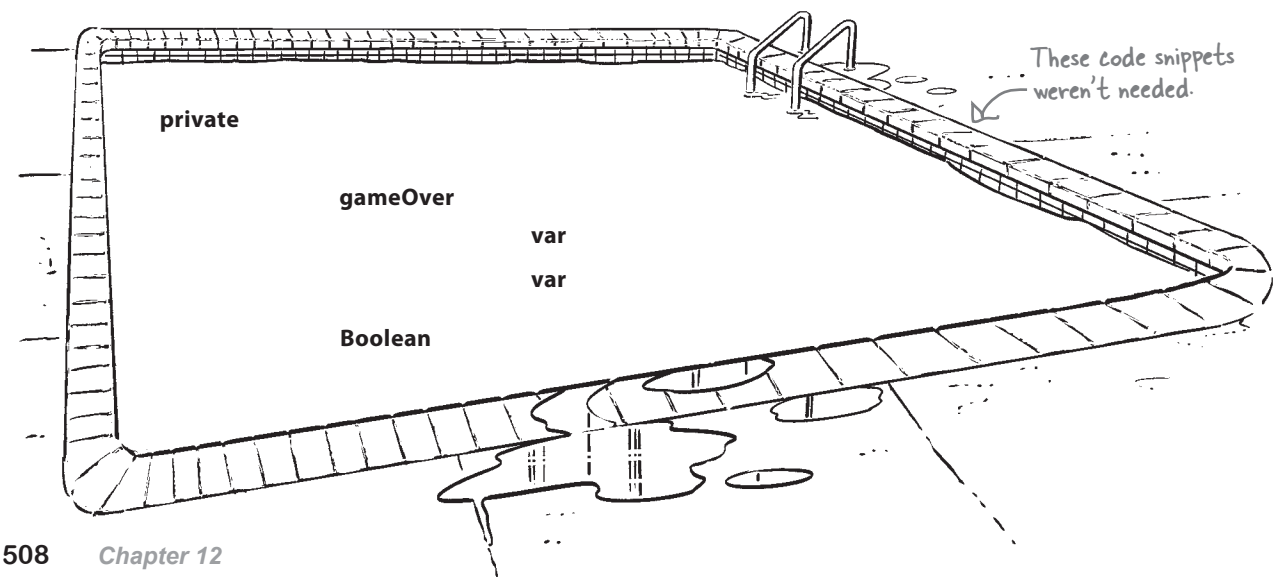
# Pool Puzzle Solution

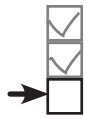
Your **goal** is to add a `gameOver` property (with a `_gameOver` backing property) to `GameViewModel` so that `GameFragment` can respond when its value (a `Boolean`) gets updated. The property should be initialized with a value of *false*. Take code snippets from the pool and place them into the blank lines in the code. You may **not** use the same snippet more than once, and you won't need to use all the snippets.

```
package com.hfad.guessinggame

import androidx.lifecycle.ViewModel
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.LiveData

class GameViewModel : ViewModel() {
    ...
    private val _gameOver = MutableLiveData<Boolean>(false)
    val gameOver : LiveData<Boolean>
        get() = _gameOver
    ...
}
```

_gameOver is a private MutableLiveData<Boolean> property that's set to false.

gameOver gives GameFragment read-only access to _gameOver's value.





**Use live data**
**Protect properties**
**Add gameOver**

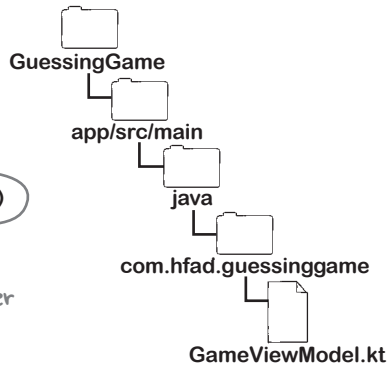
# The full code for GameViewModel.kt

We need to add a `gameOver` property to `GameViewModel`, along with a `_gameOver` backing property. We'll get the `makeGuess()` method to set it to *true* when the user has guessed all of the letters in the secret word, or run out of lives.

Here's the full code for `GameViewModel`; update the code in *GameViewModel.kt* to include the changes shown below (in bold):

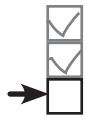
```
package com.hfad.guessinggame

import androidx.lifecycle.ViewModel
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.LiveData

class GameViewModel : ViewModel() {
    private val words = listOf("Android", "Activity", "Fragment")
    private val secretWord = words.random().uppercase()
    private val _secretWordDisplay = MutableLiveData<String>()
    val secretWordDisplay: LiveData<String>
        get() = _secretWordDisplay
    private var correctGuesses = ""
    private val _incorrectGuesses = MutableLiveData<String>("")
    val incorrectGuesses: LiveData<String>
        get() = _incorrectGuesses
    private val _livesLeft = MutableLiveData<Int>(8)
    val livesLeft: LiveData<Int>
        get() = _livesLeft
    private val _gameOver = MutableLiveData<Boolean>(false)
    val gameOver: LiveData<Boolean>
        get() = _gameOver

    init {
        _secretWordDisplay.value = deriveSecretWordDisplay()
    }
```

This adds the gameOver property, and its _gameOver backing property.




The code continues on the next page.

# GameViewModel.kt continued

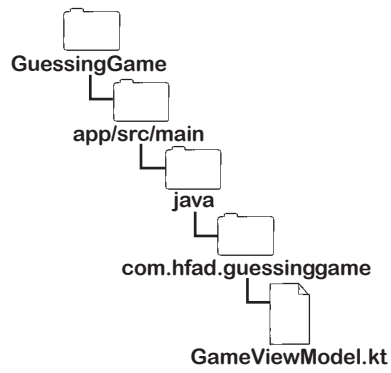


Use live data
Protect properties
**Add gameOver**

```
    private fun deriveSecretWordDisplay() : String {
        var display = ""
        secretWord.forEach {
            display += checkLetter(it.toString())
        }
        return display
    }

    private fun checkLetter(str: String) = when (correctGuesses.contains(str)) {
        true -> str
        false -> "_"
    }

    fun makeGuess(guess: String) {
        if (guess.length == 1) {
            if (secretWord.contains(guess)) {
                correctGuesses += guess
                _secretWordDisplay.value = deriveSecretWordDisplay()
            } else {
                _incorrectGuesses.value += "$guess "
                _livesLeft.value = _livesLeft.value?.minus(1)
            }
            if (isWon() || isLost()) _gameOver.value = true
        }
    }

    private fun isWon() = secretWord.equals(secretWordDisplay.value, true)

    private fun isLost() = livesLeft.value ?: 0 <= 0

    fun wonLostMessage() : String {
        var message = ""
        if (isWon()) message = "You won!"
        else if (isLost()) message = "You lost!"
        message += " The word was $secretWord."
        return message
    }
}
```

This sets the _gameOver property's value to true if the game is won or lost.

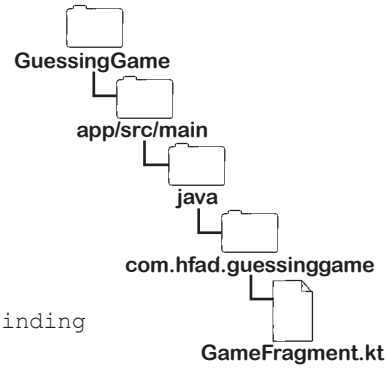
Since GameFragment no longer needs to call these methods, we can make them private.



# Make GameFragment observe the new property

Now that we've added the `gameOver` property to `GameViewModel`, we need to make `GameFragment` respond to when it's been updated. We'll make the fragment observe the property, so that when its value is changed to *true*, the fragment will navigate to `ResultFragment`.

Here's the code for `GameFragment`; update the code in *GameFragment.kt* so that it includes the changes below (in bold):

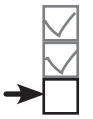



```
package com.hfad.guessinggame

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.guessinggame.databinding.FragmentGameBinding
import androidx.navigation.findNavController
import androidx.lifecycle.ViewModelProvider
import androidx.lifecycle.Observer

class GameFragment : Fragment() {
    private var _binding: FragmentGameBinding? = null
    private val binding get() = _binding!!
    lateinit var viewModel: GameViewModel

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentGameBinding.inflate(inflater, container, false)
        val view = binding.root
        viewModel = ViewModelProvider(this).get(GameViewModel::class.java)

        viewModel.incorrectGuesses.observe(viewLifecycleOwner, Observer { newValue ->
            binding.incorrectGuesses.text = "Incorrect guesses: $newValue"
        })

        viewModel.livesLeft.observe(viewLifecycleOwner, Observer { newValue ->
            binding.lives.text = "You have $newValue lives left"
        })
```

You don't need to change any of the code on this page.

The code continues on the next page.

# GameFragment.kt continued



**Use live data**
**Protect properties**
**Add gameOver**

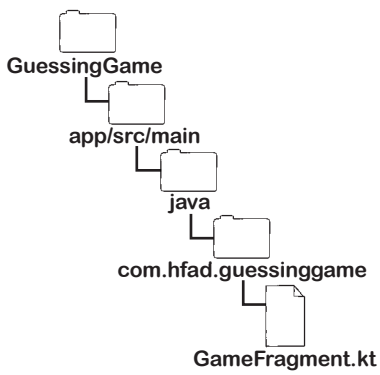
```
        viewModel.secretWordDisplay.observe(viewLifecycleOwner, Observer { newValue ->
            binding.word.text = newValue
        })
```

This makes the fragment observe the view model's gameOver property.

```
        viewModel.gameOver.observe(viewLifecycleOwner, Observer { newValue ->
            if (newValue) {
                val action = GameFragmentDirections
                    .actionGameFragmentToResultFragment(viewModel.wonLostMessage())
                view.findNavController().navigate(action)
            }
        })

        binding.guessButton.setOnClickListener() {
            viewModel.makeGuess(binding.guess.text.toString().uppercase())
            binding.guess.text = null
            if (viewModel.isWon() || viewModel.isLost()) {
                val action = GameFragmentDirections
                    .actionGameFragmentToResultFragment(viewModel.wonLostMessage())
                view.findNavController().navigate(action)
            }
        }
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

If gameOver's value has been set to true, navigate to ResultFragment.

Delete these lines.

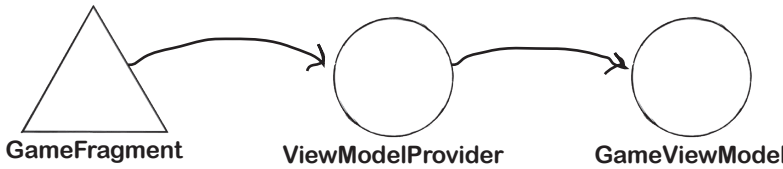



That's it! Let's go through what happens when it runs.

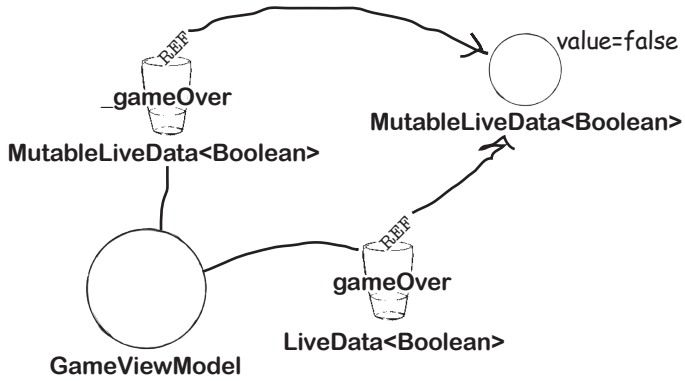
# What happens when the app runs

The following things happen when the app runs:

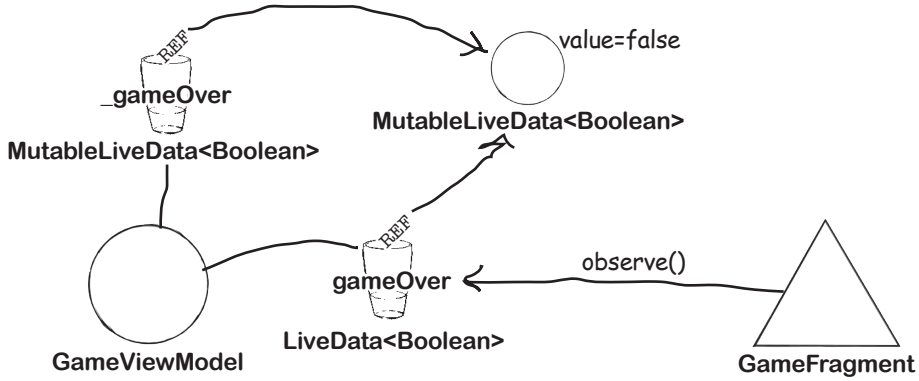
1. **GameFragment asks the ViewModelProvider class for an instance of GameViewModel.**



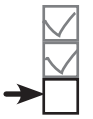

2. **GameViewModel's properties are initialized.**
The `_gameOver` and `gameOver` properties refer to a `MutableLiveData<Boolean>` object whose value is set to *false*.




3. **GameFragment observes the GameViewModel's gameOver property.**
`GameFragment` can't update the `MutableLiveData` object that the `gameOver` property refers to, but it can respond to when its value changes.

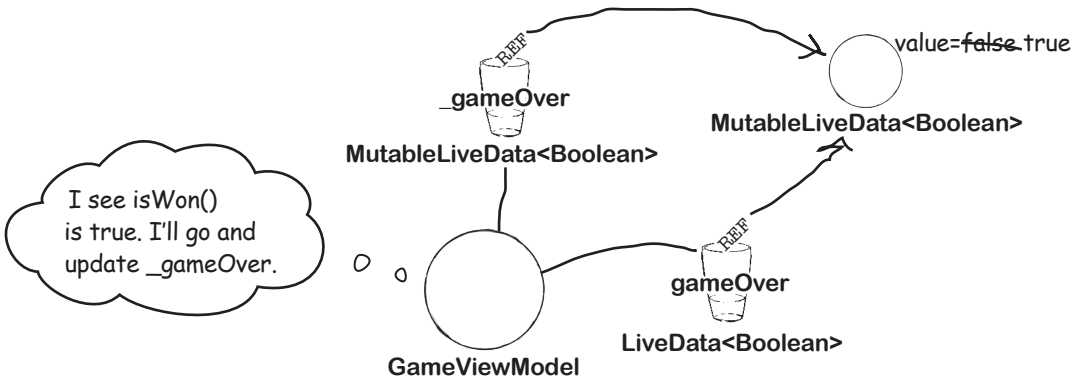


# The story continues

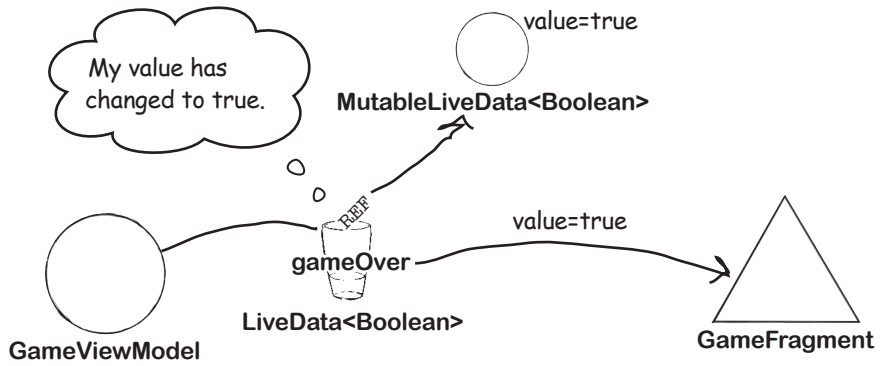


**Use live data**
**Protect properties**
**Add gameOver**

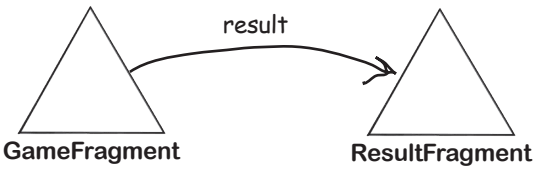
**4** **Each time the GameViewModel's makeGuess() method is called, it checks whether isWon() or isLost() returns true.**
If either is *true*, it sets the value of its `_gameOver` property to *true*.



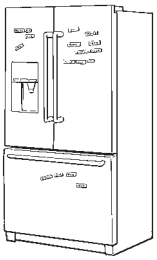

**5** **GameFragment observes the value has been updated to true via the GameViewModel's gameOver property.**
The new value is passed to `GameFragment`.




**6** **GameFragment responds by navigating to ResultFragment, passing it the result.**




We'll take the app for a test drive after the following exercise.



# Fragment Magnets

Somebody wrote the code for a fragment named `LotteryFragment` on the fridge door, but a freak kitchen blizzard blew some of the code away. Can you piece it back together again?

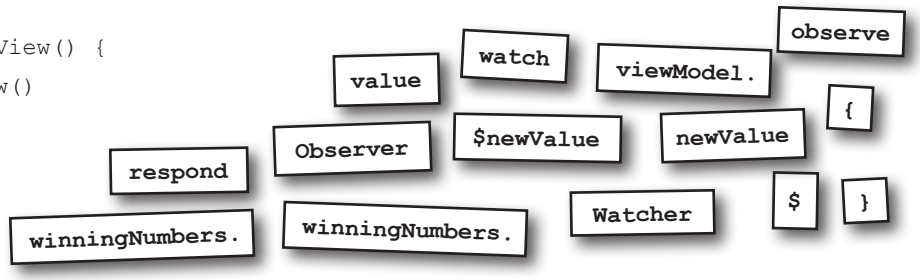
The fragment needs to observe `LotteryViewModel`'s `winningNumbers` property, which is defined like this:

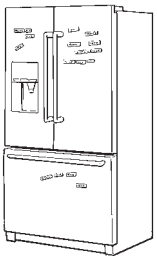
```
private val _winningNumbers = MutableLiveData<String>()
val winningNumbers: LiveData<String>
    get() = _winningNumbers
```

When `winningNumbers` changes, `LotteryFragment` needs to update its `numbers` view with the new value.

```
class LotteryFragment : Fragment() {
    private var _binding: FragmentLotteryBinding? = null
    private val binding get() = _binding!!
    lateinit var viewModel: LotteryViewModel

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentLotteryBinding.inflate(inflater, container, false)
        val view = binding.root
        viewModel = ViewModelProvider(this).get(LotteryViewModel::class.java)


        viewModel.                                  (viewLifecycleOwner,                { newValue ->

            binding.numbers.text = "Winning numbers:                                                                  "
        })
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

You won't need to use all of these magnets.





# Fragment Magnets Solution

Somebody wrote the code for a fragment named `LotteryFragment` on the fridge door, but a freak kitchen blizzard blew some of the code away. Can you piece it back together again?

The fragment needs to observe `LotteryViewModel`'s `winningNumbers` property, which is defined like this:

```
private val _winningNumbers = MutableLiveData<String>()
val winningNumbers: LiveData<String>
    get() = _winningNumbers
```

When `winningNumbers` changes, `LotteryFragment` needs to update its `numbers` view with the new value.

```
class LotteryFragment : Fragment() {
    private var _binding: FragmentLotteryBinding? = null
    private val binding get() = _binding!!
    lateinit var viewModel: LotteryViewModel

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentLotteryBinding.inflate(inflater, container, false)
        val view = binding.root
        viewModel = ViewModelProvider(this).get(LotteryViewModel::class.java)

        viewModel.winningNumbers.observe(viewLifecycleOwner, Observer { newValue ->
            binding.numbers.text = "Winning numbers: $newValue"
        })
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

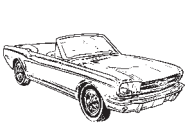
Observe the winningNumbers property.

Display the property's new value in the numbers view.

You didn't need to use these magnets.

watch  value  $newValue  respond  {

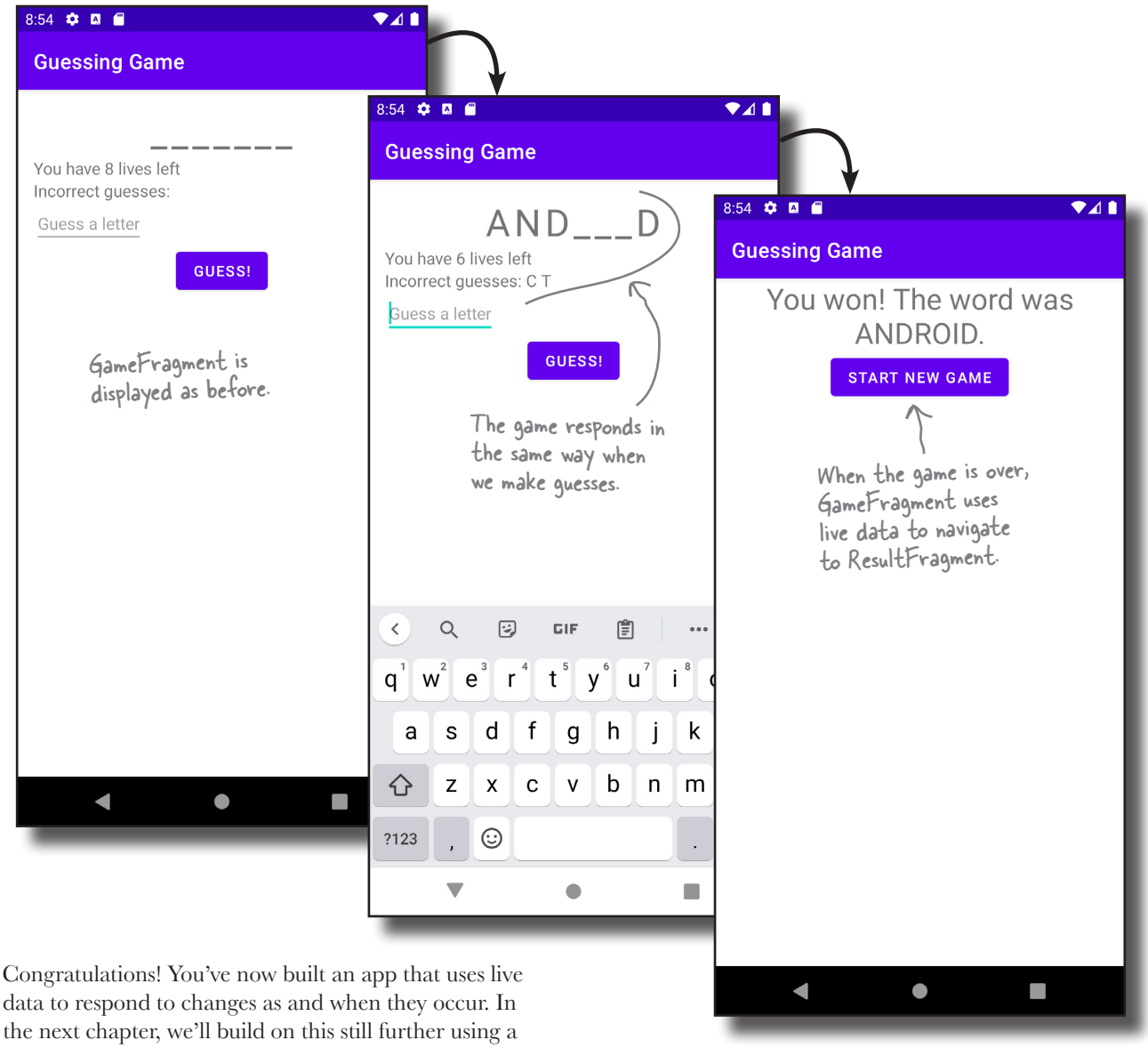
viewModel.  winningNumbers.  Watcher  }



# Test Drive

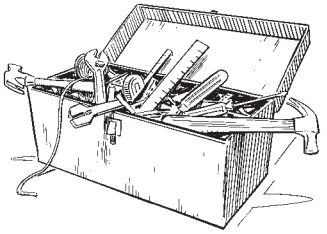
**Use live data**
**Protect properties**
**Add gameOver**

When we run the app, it works in the same way as before. This time, however, `GameViewModel` decides when the game is over instead of `GameFragment`. The fragment simply observes the view model's `gameOver` property, and navigates to `ResultFragment` when it's changed to *true*.




Congratulations! You've now built an app that uses live data to respond to changes as and when they occur. In the next chapter, we'll build on this still further using a new technique called data binding.

# Your Android Toolbox



**You've got Chapter 12 under your belt and now you've added live data to your toolbox.**

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

## Bullet Points

- Live data lets the view model's properties tell interested parties (such as fragments and activities) when its values have changed.
- To make a property use live data, define the property using `val`, and change its type to `MutableLiveData<Type>` where `Type` is the type of data the property needs to hold.
- The `MutableLiveData` object's `value` property is used to hold its value. This property is nullable.
- You make a fragment or activity respond to changes in the view model by calling the property's `observe` method and specifying how the new value should be used.
- You can restrict direct access to a view model's properties using backing properties.
- The `LiveData` type is like `MutableLiveData` except that it's immutable. It's used to expose a `MutableLiveData` object's value while preventing it from being updated.

# Building Smart Layouts

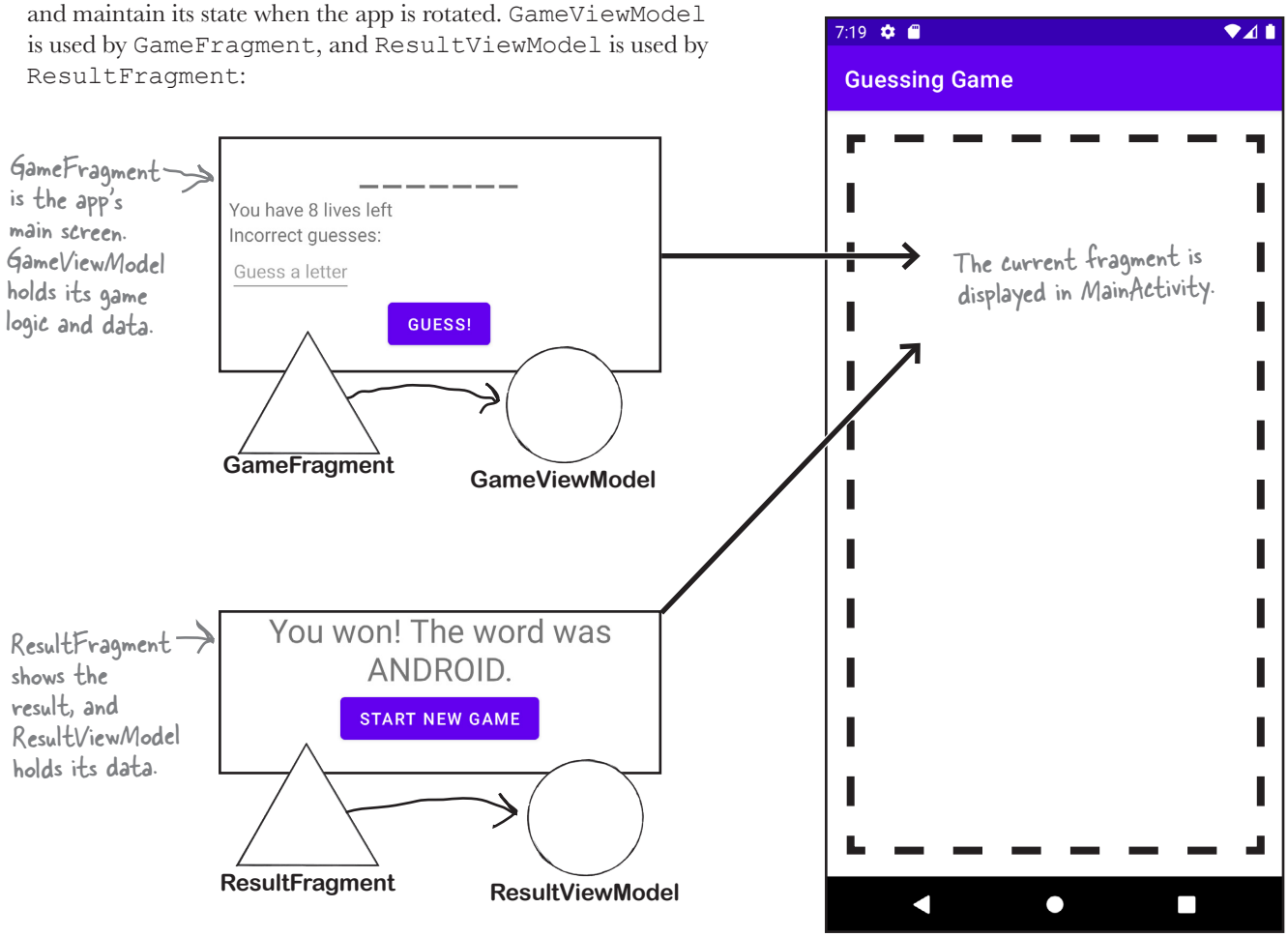
## Layouts can do more than control your app's appearance.

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**. Soon, your layouts will be more powerful than ever before.

# Back to the Guessing Game app

In the previous two chapters, you built a Guessing Game app that lets the user guess which letters are included in a secret word. When the user guesses all the letters or runs out of lives, the game ends.

The app uses two view models—`GameViewModel` and `ResultViewModel`—which hold the app's game logic and data, and maintain its state when the app is rotated. `GameViewModel` is used by `GameFragment`, and `ResultViewModel` is used by `ResultFragment`:

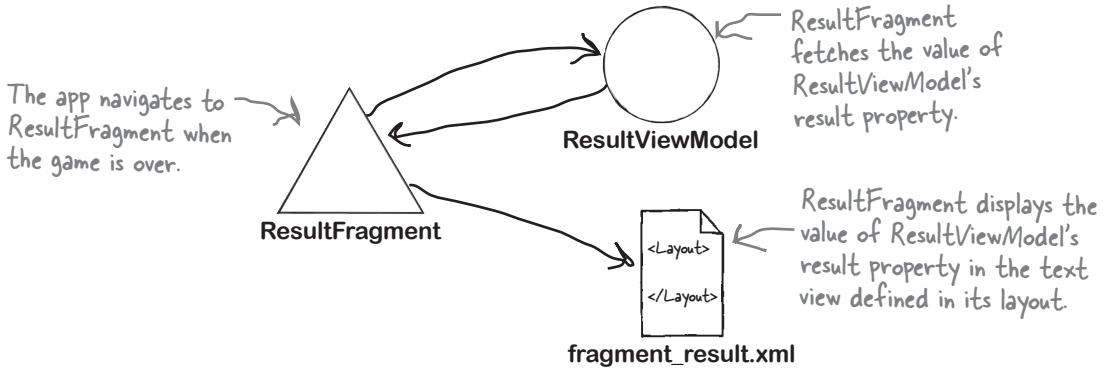



The first screen the user sees is `GameFragment`, and she interacts with it to play the game. It displays information—such as the number of lives left and any incorrect guesses the user has made—and lets the user make guesses.

When the game is over, the app navigates to `ResultFragment`. This screen tells the user whether she has won or lost the game, and what the secret word was.

# The fragments update the views in their layouts

Each of the app's fragments are responsible for updating the views in their layouts. `ResultFragment`, for example, sets the text in its `won_lost` text view to the value of `ResultViewModel`'s `result` property:

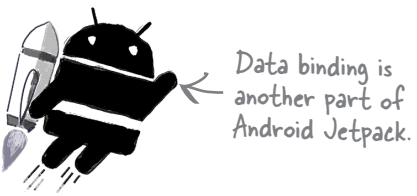


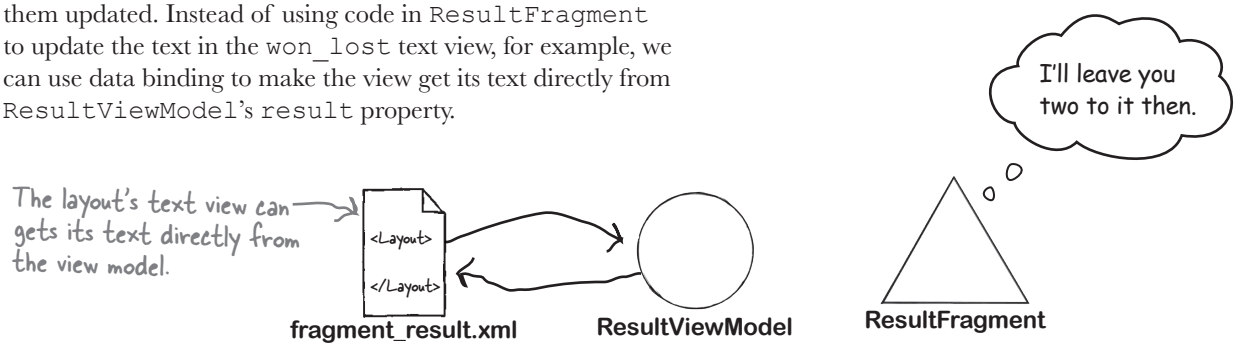

But while this approach works, it has a downside. When you make fragments responsible for keeping views up-to-date in this way, it can make your Kotlin code more complicated, and harder to read and maintain.

So what's the alternative?

## We can make the views update themselves instead

The alternative is to use a technique called **data binding**. With data binding, **views can get their values directly from the view model**, so the fragment code no longer has to keep them updated. Instead of using code in `ResultFragment` to update the text in the `won_lost` text view, for example, we can use data binding to make the view get its text directly from `ResultViewModel`'s `result` property.



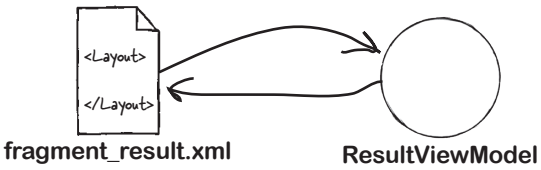




In this chapter, you'll learn how to use data binding by implementing it in the Guessing Game app. Let's go through the steps we'll take to do this.

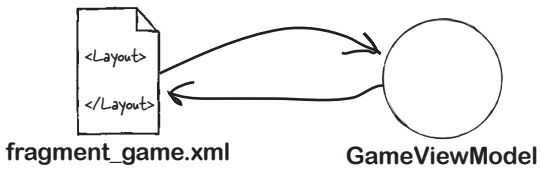
# Here's what we're going to do

Here are the steps that we'll go through to make the Guessing Game app use data binding:

1. **Implement data binding for ResultFragment.**
We'll update `ResultFragment` so that its layout uses data binding to display the value of the `result` property directly from `ResultViewModel`.




2. **Implement data binding for GameFragment.**
We'll then change up `GameFragment` so that its layout uses data binding to display property values from `GameViewModel`. Along the way, we'll also make the layout dynamically respond to any live data updates.



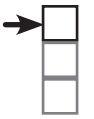

3. **Add a Finish Game button to GameFragment.**
Finally, we'll add a new button to `GameFragment`'s layout that, when clicked, immediately finishes the game.

When the user clicks on the Finish Game button, the app will immediately navigate to ResultFragment and display the secret word.

Let's get started.

**Don't forget!**

In this chapter, you're going to modify the Guessing Game app again, so make sure you open the project for this app.

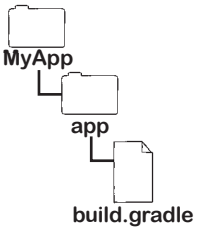


# Enable data binding in the app build.gradle file

Just like with view binding, you need to enable data binding in the `android` section of the app's *build.gradle* file before you can use it in your app. The code to enable data binding looks like this:

```
android {
    ...
    buildFeatures {
        dataBinding true
    }
}
```

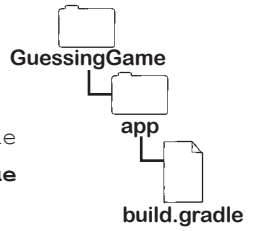
You need to add these lines to the android section of the app's build.gradle.




We've already enabled view binding in the Guessing Game app, so let's enable data binding too. Open the file *GuessingGame/app/build.gradle*, and update the `buildFeatures` section so that it looks like this:

```
android {
    ...
    buildFeatures {
        viewBinding true
        dataBinding true
    }
}
```

You need to add this line.




Then choose the Sync Now option to sync this change with the rest of your project.

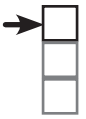
Now that we've enabled data binding, let's implement it in `ResultFragment`.

## there are no Dumb Questions

**Q: I see that in the Guessing Game app, both view binding *and* data binding are enabled. Do you always need to enable view binding in order to use data binding?**

A: No, you don't. It's quite common to enable data binding without enabling view binding, and you'll find out more about this later in the chapter.

For now, all you really need to know is that we're keeping the view binding option enabled so that all the app's existing code continues to work.

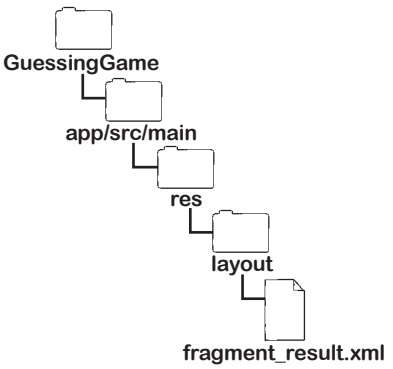


**ResultFragment**
GameFragment
Add button

# ResultFragment updates the text in its layout

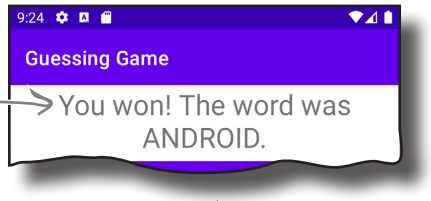
As you may recall, `ResultFragment`'s layout contains a text view with an ID of `won_lost`, which is defined like this:

```
<LinearLayout ...>
    <TextView
        android:id="@+id/won_lost"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:textSize="28sp" />
    ...
</LinearLayout>
```




When the app navigates to `ResultFragment`, the fragment gets the value of the `result` property from its view model, and displays it in the text view using this code:

```
binding.wonLost.text = viewModel.result
```




The won_lost text is currently set by the code in ResultFragment. We're going to change the fragment to use data binding so that the view fetches its text value directly from the view model's result property.

We're going to change the fragment and its layout so that it uses data binding. Instead of getting `ResultFragment` to update the text view, we'll make the text view get its value directly from the view model.

## How we'll implement data binding

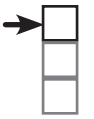
To make the text view get its text from `ResultViewModel`, we need to update the code for `ResultFragment` and its layout. We'll do this by going through the following steps:

1. **Add <layout> and <data> elements to the layout.**
The `<layout>` element is required for data binding, and the `<data>` element is used to set up a data binding variable that links the layout to the view model.

2. **Set the layout's data binding variable to an instance of the view model.**
We'll do this using fragment code.

3. **Use the data binding variable to access the view model's properties.**
We'll update the `won_lost` text view so that it gets its text value directly from the view model.

Let's begin by updating the layout code.



**ResultFragment**
GameFragment
Add button

# 1. Add <layout> and <data> elements

Every layout that uses data binding takes the following form:

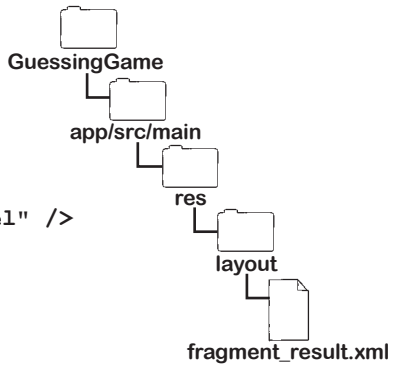
```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".ResultFragment">

    <data>
        <variable
            name="resultViewModel"
            type="com.hfad.guessinggame.ResultViewModel" />
    </data>

    ...

</layout>
```

A <layout> element goes at the layout's root.

You don't need to update your code just yet: we'll show you the full code a few pages ahead.

A <data> section defines any variables that the layout will use for data binding.

A view or view group element for the layout's view hierarchy goes here. This is like the layout code for a "normal" layout that doesn't use data binding.

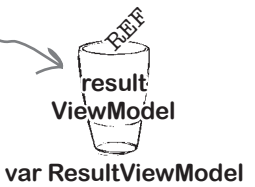



The above layout has a **<layout>** element as its root element. This enables data binding in the layout, so it's required by *all* layouts that need to use data binding.

Within the <layout> element sits a **<data>** element. This is used to specify any variables—defined using the **<variable>** element—that are needed for data binding, such as the view model you want the layout's views to gets their data from. The above code, for example, uses:

```
<data>
    <variable
        name="resultViewModel"
        type="com.hfad.guessinggame.ResultViewModel" />
</data>
```

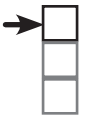
This defines a ResultViewModel data binding variable named resultViewModel.




to specify a data binding variable named `resultViewModel` whose type is `ResultViewModel`.

The <layout> element also includes a view or view group for the layout's view hierarchy. If you want to display views in a linear layout, for example, you'd put the <LinearLayout> element and its views inside the <layout> element along with all of its views. You'll see how this is done a few pages ahead.

Once you've defined a data binding variable, you need to set it. We'll see how to do this next.



**ResultFragment**
GameFragment
Add button

# 2. Set the layout's data binding variable

You set the layout's data binding variable using Kotlin code.

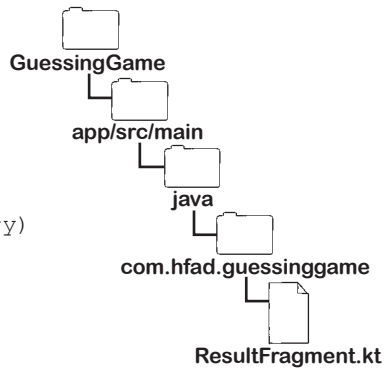
On the previous page, we showed you the code to add a data binding variable (named `resultViewModel`) to `ResultFragment`'s layout file *fragment_result.xml*:

```
<data>
    <variable
        name="resultViewModel"
        type="com.hfad.guessinggame.ResultViewModel" />
</data>
```

The variable's type is `ResultViewModel`, so we need to set it to an object with a matching type.

As you may recall, `ResultFragment` includes code in its `onCreateView()` method that gets a `ResultViewModel` object like so:

```
...
viewModelFactory = ResultViewModelFactory(result)
viewModel = ViewModelProvider(this, viewModelFactory)
    .get(ResultViewModel::class.java)
...
```

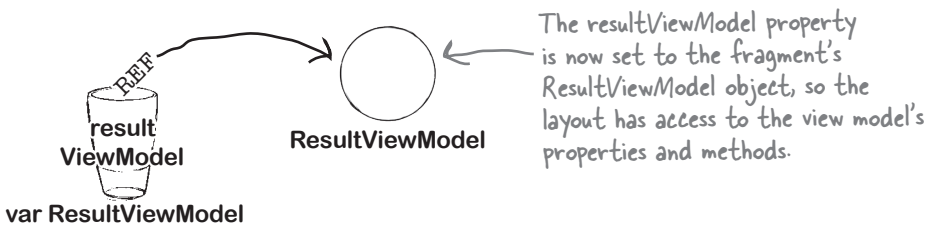



We can assign this view model to the layout's `resultViewModel` data binding variable by adding the following line to the `onCreateView()` code:

```
binding.resultViewModel = viewModel
```

This sets the layout's data binding variable.

This uses the fragment's `binding` property to set the layout's `resultViewModel` property to the fragment's view model object:




Once you've set the data binding variable, the layout can use it to access the view model's properties and methods. Let's see how.

# 3. Use the layout's data binding variable to access the view model

`ResultFragment`'s layout defines a `won_lost` text view that needs to display the value of `ResultViewModel`'s `result` property. Its text is set in the fragment code like this:

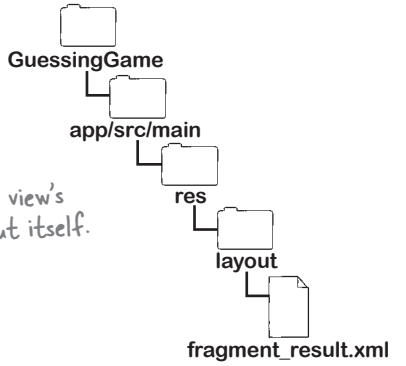
```
binding.wonLost.text = viewModel.result
```

The fragment sets the won_lost view's text to the value of the view model's result property.

Now that we've set a data binding variable for this view model in the layout code, we can use data binding to set its text instead. The code to do so looks like this:

```
<TextView
    android:id="@+id/won_lost"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:gravity="center"
    android:textSize="28sp"
    android:text="@{resultViewModel.result}" />
```

With data binding, the view's text is set in the layout itself.




The above code uses the `resultViewModel` variable to get the value of the view model's `result` property. It then sets the text view's text to the value of this property.

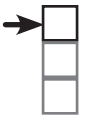
That's everything that you need to know in order to implement data binding for `ResultFragment`. We'll show you the full code for *result_fragment.xml* and *ResultFragment.kt* over the next couple pages.

## there are no Dumb Questions

**Q: All of the other elements we've added to layouts, like `<LinearLayout>` and `<Button>`, have all had an initial capital letter. Why are `<layout>`, `<data>`, and `<variable>` all lowercase?**

A: All of the other elements you've learned about so far have had initial capital letters because they are class names for types of `View`. The `<LinearLayout>` element, for example, refers to the `LinearLayout` class found in the `android.view` package.

`<layout>`, `<data>`, and `<variable>` are all lowercase because they're instructions, not `View`s. When Android sees the `<variable>` element, for example, it treats it as an instruction to create a variable.



**ResultFragment**
GameFragment
Add button

# The full code for fragment_result.xml

Here's the full code for `ResultFragment`'s layout; update the code for *fragment_result.xml* to include the changes (in bold):

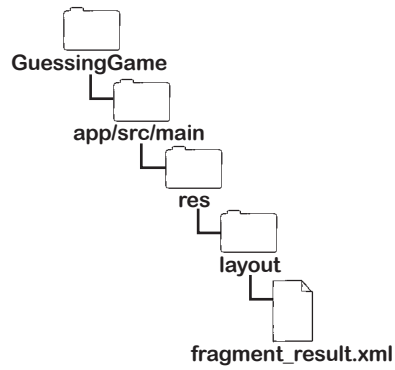
```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".ResultFragment">

    <data>
        <variable
            name="resultViewModel"
            type="com.hfad.guessinggame.ResultViewModel" />
    </data>

    <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:orientation="vertical"
        ~~tools:context=".ResultFragment"~~>

        <TextView
            android:id="@+id/won_lost"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:gravity="center"
            android:textSize="28sp"
            android:text="@{resultViewModel.result}" />

        <Button
            android:id="@+id/new_game_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:text="Start new game"/>
    </LinearLayout>
</layout>
```

Add the <layout> element.

Move these lines from the <LinearLayout> to the <layout> element so they're still in the file's root element.

Define the data binding variable.

Delete these lines from here, and move them to the <layout>.

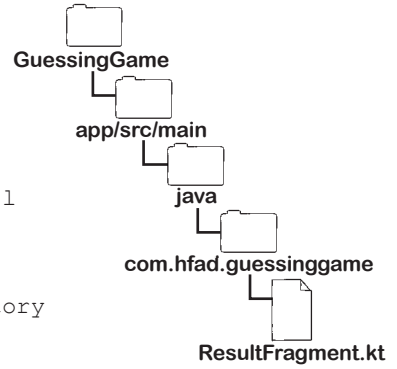



Add this line, which uses data binding to set the view's text.

The <layout> element's closing tag

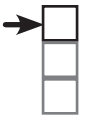
# The full code for ResultFragment.kt

You need to update the fragment code too. Update the code for *ResultFragment.kt* so that it includes the changes shown here (in bold):




```
package com.hfad.guessinggame

...   <-- You don't need to change any of the imports, so we've left them out.

class ResultFragment : Fragment() {
    private var _binding: FragmentResultBinding? = null
    private val binding get() = _binding!!
    lateinit var viewModel: ResultViewModel
    lateinit var viewModelFactory: ResultViewModelFactory

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentResultBinding.inflate(inflater, container, false)
        val view = binding.root

        val result = ResultFragmentArgs.fromBundle(requireArguments()).result
        viewModelFactory = ResultViewModelFactory(result)
        viewModel = ViewModelProvider(this, viewModelFactory)
            .get(ResultViewModel::class.java)
        binding.resultViewModel = viewModel   <-- Set the layout's data binding variable.
        ~~binding.wonLost.text = viewModel.result~~   <-- This line is no longer needed as we're setting the view's text using data binding.

        binding.newGameButton.setOnClickListener {
            view.findNavController()
                .navigate(R.id.action_resultFragment_to_gameFragment)
        }
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Let's go through what happens when the app runs.

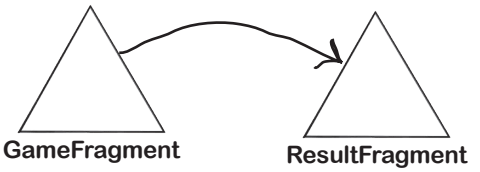


**ResultFragment**
GameFragment
Add button

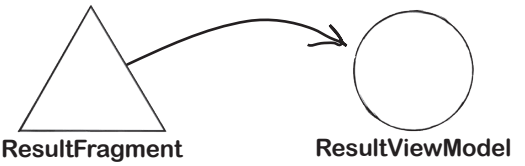
# What happens when the app runs

The following things happen when the app runs:

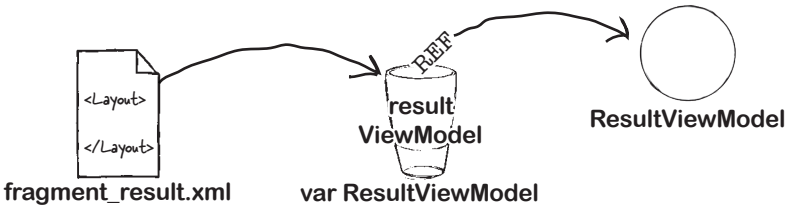
1. **When the user completes a game, the app navigates to ResultFragment.**



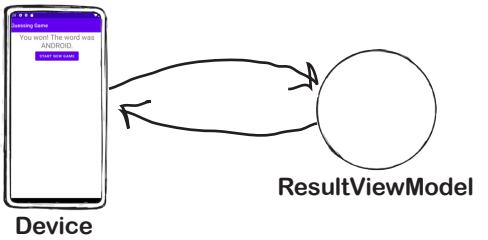

2. **ResultFragment gets a reference to its ResultViewModel object.**



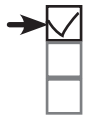

3. **ResultFragment assigns the ResultViewModel object to the layout's resultViewModel data binding variable.**
The layout's views can now use the variable to access the view model's properties and methods.




4. **The won_lost view's text is set to the value of the ResultViewModel object's result property.**
The text is displayed on the device screen.



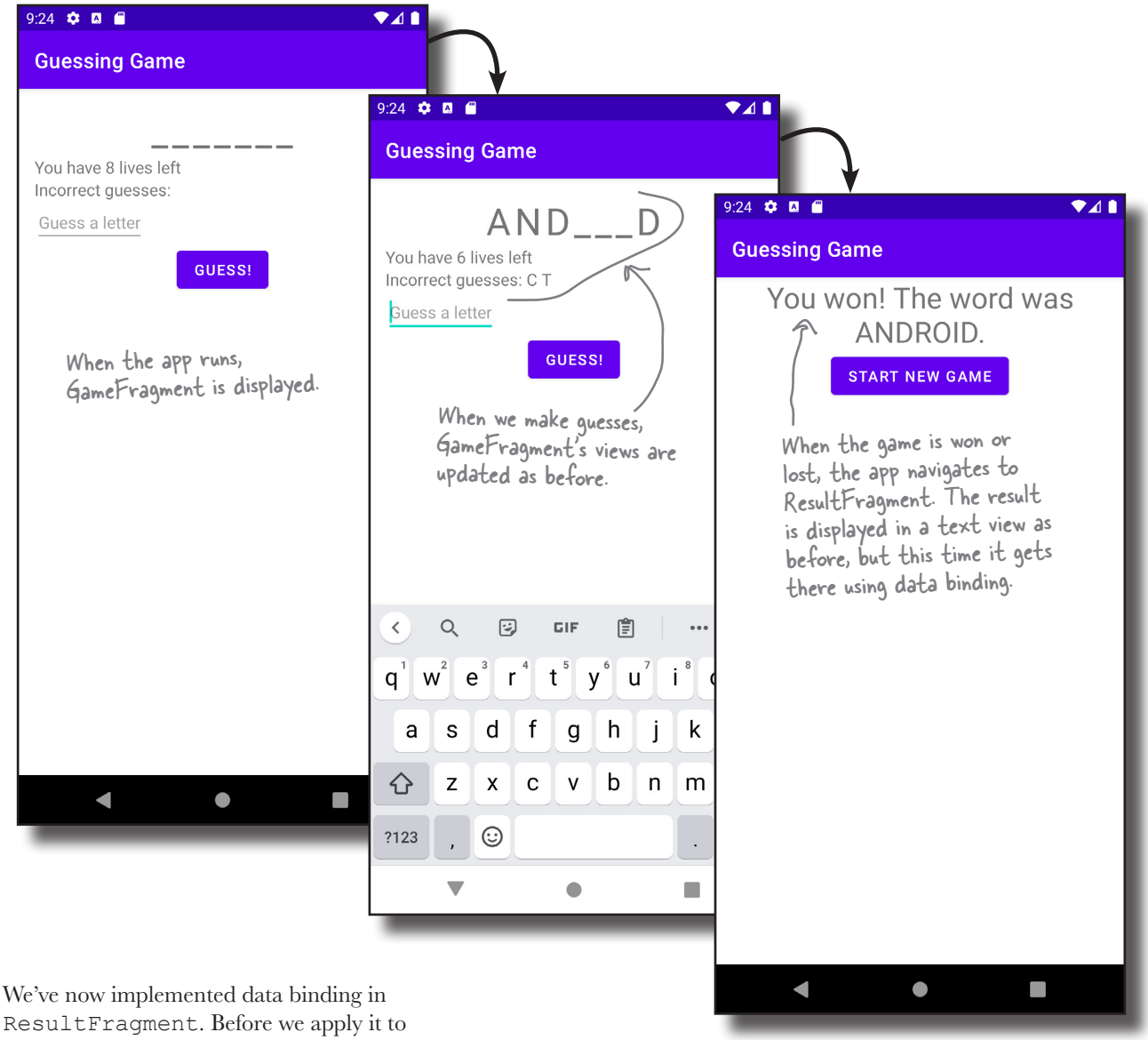

Let's take the app for a test drive.



**ResultFragment**
GameFragment
Add button

# Test Drive

When we run the app, it works in the same way as before. This time, however, `ResultFragment`'s layout is getting the game's result directly from `ResultViewModel`.




We've now implemented data binding in `ResultFragment`. Before we apply it to `GameFragment`, let's take a closer look at the `<layout>` element, and why it's necessary for data binding.

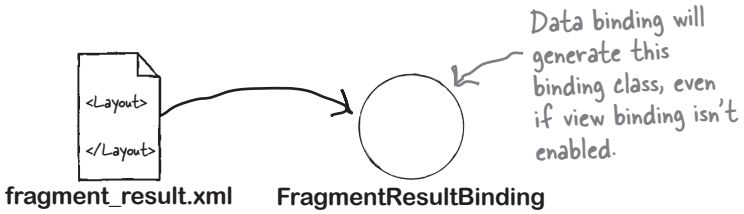
## <layout> Up Close

As we said earlier, every layout that uses data binding *must* have a `<layout>` element as its root element like this:

```
<layout ... >
    ...
</layout>
```

When you enable data binding in an app's *build.gradle* file, it generates a binding class for each layout with a `<layout>` element at its root. **This is the same binding class that gets generated when you enable view binding**.

In the Guessing Game app, the layout file *fragment_result.xml* has a `<layout>` element as its root element. This means that its binding class `FragmentResultBinding` will still get generated by data binding even if view binding is switched off:




You add one or more data binding variables to the layout using the `<data>` and `<variable>` elements:

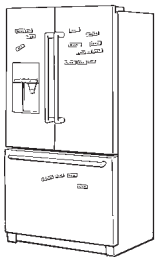
```
<layout ... >
    <data>
        <variable
            name="variableName"
            type="com.hfad.myapp.ClassName" />
    </data>

    ...

</layout>
```

Each data binding variable goes inside its own <variable> element. Here we're just using one, but your layout can have multiple such variables.

You set each variable to an object with a matching type in Kotlin code. Once set, the layout's views can use the data binding variable to access its object's properties and methods.

**View binding and data binding generate the same binding classes, but in different situations.**

**View binding generates a binding class for every layout, while data binding generates one for every layout with a <layout> element.**



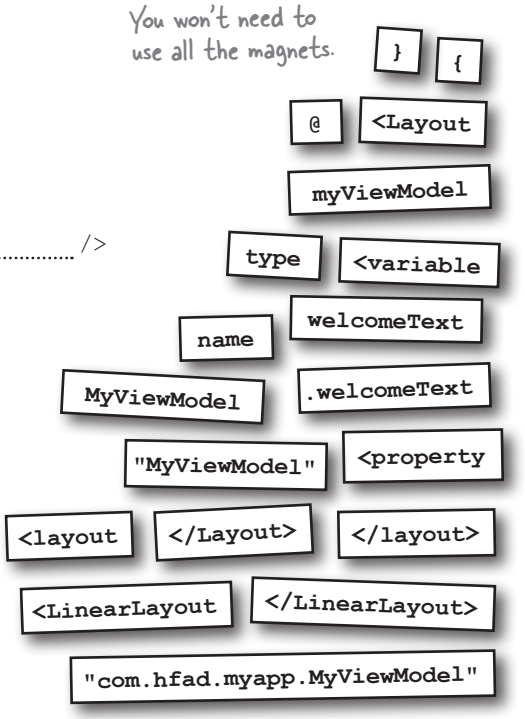
# Layout Magnets

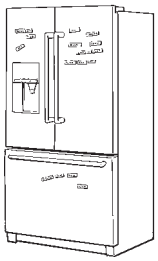
Somebody wrote the code for a layout using fridge magnets, but some of them fell off when we shut the fridge door too firmly. Can you piece the code back together again?

The layout needs to use data binding to display the `welcomeText` property from a `MyViewModel` object in its `welcome` text view. The view model resides in a package named `com.hfad.myapp.MyViewModel`.

```
.....
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".MyFragment">

    <data>

        .....

            ..... ="myViewModel"

            ..... = ..... />
    </data>


      .....
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <TextView
            android:id="@+id/welcome"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"

            android:text="..... " />

      .....

.....
```

You won't need to use all the magnets.





# Layout Magnets Solution

Somebody wrote the code for a layout using fridge magnets, but some of them fell off when we shut the fridge door too firmly. Can you piece the code back together again?

The layout needs to use data binding to display the `welcomeText` property from a `MyViewModel` object in its `welcome` text view. The view model resides in a package named `com.hfad.myapp.MyViewModel`.

Put a <layout> element at the layout's root.

```
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".MyFragment">

    <data>
        <variable
            name
            ="myViewModel"
            type
            =
            "com.hfad.myapp.MyViewModel"
            />
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <TextView
            android:id="@+id/welcome"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="
            @
            {
            myViewModel
            .welcomeText
            }
            " />
    </LinearLayout>
</layout>
```

Define the data binding variable in the <data> element.

The variable needs a name and type.

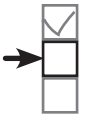
Display any views in a <LinearLayout>.

Set the view's text to the view model's welcomeText property.

Closing tags for the <LinearLayout> and <layout>

You didn't need to use these magnets.

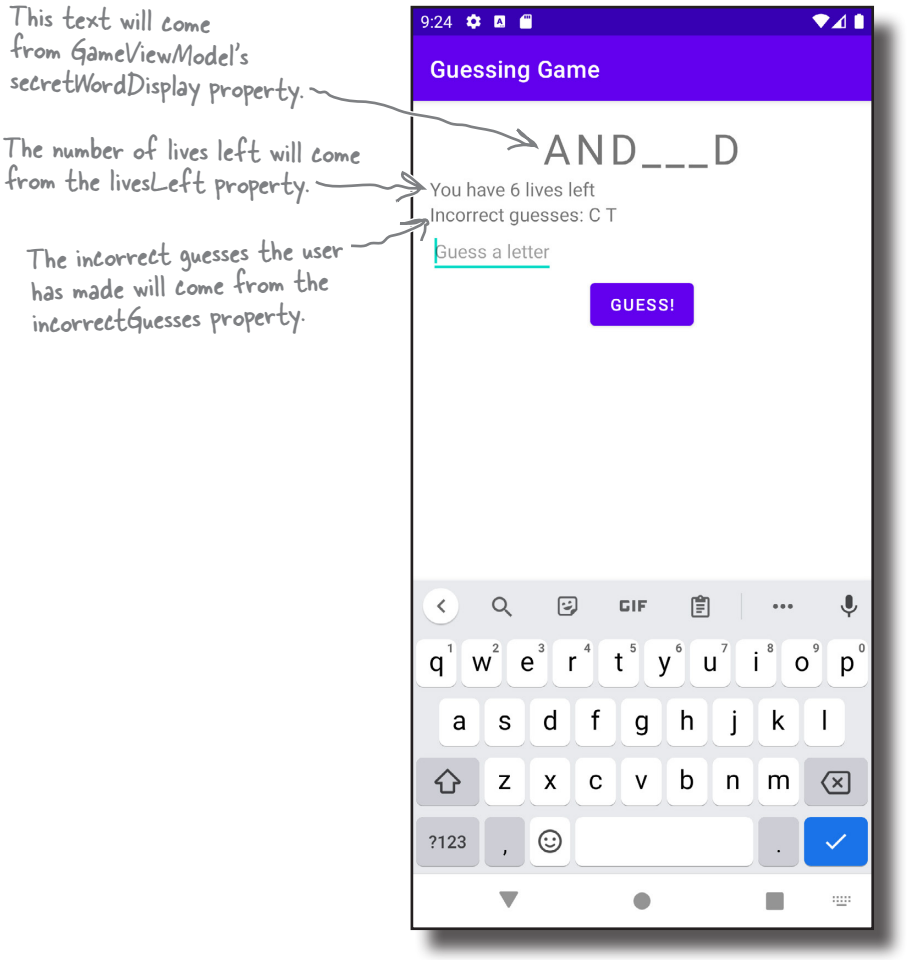
<property>
<Layout
"MyViewModel"
welcomeText
MyViewModel
</Layout>



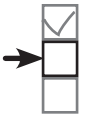
# GameFragment can use data binding too

You've now implemented data binding for `ResultFragment` so that the text view in its layout gets its text directly from `ResultViewModel`'s result property.

Next, we'll do a similar thing with `GameFragment`: we'll implement data binding so that the text views in its layout get their values directly from `GameViewModel`:




To do this, we need to update the code for `GameFragment` and its layout. We'll begin with the layout code.



ResultFragment
**GameFragment**
Add button

# Add <layout> and <data> elements to fragment_game.xml

Just as we did with `ResultFragment`, we'll begin by adding `<layout>` and `<data>` elements to `GameFragment`'s layout. These elements specify that the layout uses data binding, and define a data binding variable. The code looks like this; update the code in *fragment_game.xml* to include these changes (in bold):

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

    <data>
        <variable
            name="gameViewModel"
            type="com.hfad.guessinggame.GameViewModel" />
    </data>

    <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:orientation="vertical"
        android:padding="16dp"
        ~~tools:context=".GameFragment~~">

        ...

    </LinearLayout>
</layout>
```

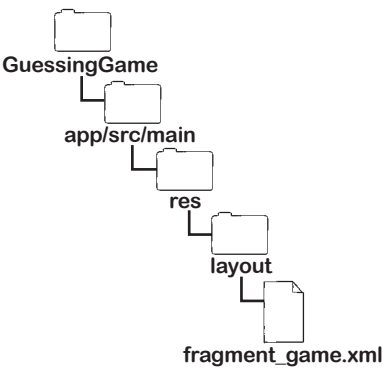
Put a <layout> element at the layout's root.

Move these lines from the <LinearLayout> element.

Add a data binding variable.

Delete these lines from here, and move them to the <layout> element.

The <layout> element's closing tag



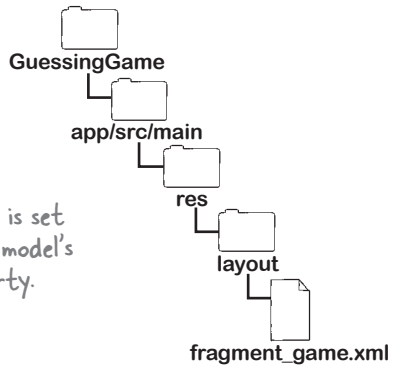

As you can see, the code defines a data binding variable named `gameViewModel` whose type is `GameViewModel`. Before we set this variable in the fragment's Kotlin code, let's update the layout's views so that they get their values from the view model.

# Use the data binding variable to set the layout's text

As you learned earlier, views can get data directly from the view model using the layout's data binding variable. *fragment_game.xml*'s `word` text view, for example, can display the value of `GameViewModel`'s `secretWordDisplay` property using this code:

```
<TextView
    android:id="@+id/word"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:textSize="36sp"
    android:letterSpacing="0.1"
    android:text="@{gameViewModel.secretWordDisplay}" />
```

The view's text property is set to the value of the view model's secretWordDisplay property.




## Two of the views need to show extra text

The `lives` and `incorrect_guesses` text views, however, don't just display the property values: **they include extra text**. The `GameFragment` code, for example, currently sets the `incorrectGuesses` view's text using this code:

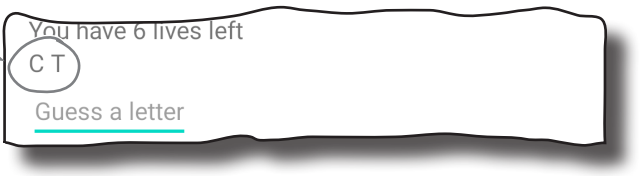
When GameFragment sets the incorrect_guesses text, it includes some extra text.

```
binding.incorrectGuesses.text = "Incorrect guesses: $newValue"
```

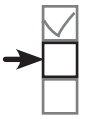
so we can't simply use the view model's `incorrectGuesses` property using the code below, or the extra text won't be displayed:

```
<TextView
    android:id="@+id/incorrect_guesses"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="16sp"
    android:text="@{gameViewModel.incorrectGuesses}" />
```

This code sets the view's text to the value of the incorrectGuesses property, and doesn't include any extra text.




To get around this problem, we'll use `String` formatting to pass the property value to a `String` resource. Let's find out how.

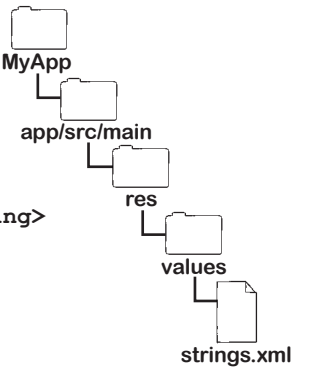


ResultFragment
GameFragment
Add button

# String resources revisited

As you learned earlier in the book, you can add `String` resources to your app's `String` resource file to avoid using hardcoded text. The following code, for example, defines a `String` resource named `my_string` that displays the text "This is a String resource":

```
<resources>
    ...
    <string name="my_string">This is a String resource</string>
</resources>
```




## String resources can have arguments

You can also define `String` resources that accept one or more arguments, which is useful if you want to display more complex text. The following resource, for example, uses a **`%s`** placeholder to specify where to place a `String` argument that's passed to it:

```
<string name="hello">Hello, %s.</string>
```

Use this to display a personalized message to the user, like "Hello, Ingo."

and this one uses **`%d`** to say where a number argument should go:

```
<string name="messages">You have %d messages.</string>
```

For example: "You have 10 messages."

You can pass multiple arguments to a `String` resource by numbering each one. The following resource, for example, takes a `String` as its first argument, and a number as its second:

```
<string name="welcome">Hello, %1$s. You have %2$d messages.</string>
```

"Hello, Ingo. You have 10 messages."

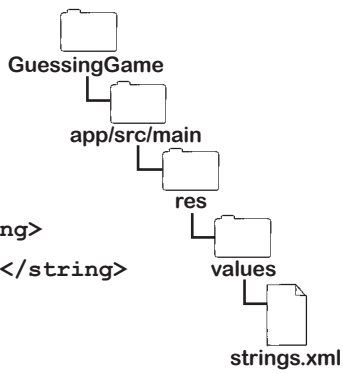
**Use %s for String arguments and %d for numbers.**

## Add two new String resources to strings.xml

In the Guessing Game app, we'll define two new `String` resources: one for the number of lives left, and another for the incorrect guesses the user has made. Open the *strings.xml* file in the *app/src/main/res/values* folder, and add the two resources shown here:

```
<resources>
    ...
    <string name="lives_left">You have %d lives left</string>
    <string name="incorrect_guesses">Incorrect guesses: %s</string>
</resources>
```

Make sure you add both String resources.




Now that we've defined the two `String` resources, let's use them in `GameFragment`'s layout.

# The layout can pass parameters to String resources

In order to use the two `String` resources we just created, we need to pass each one a parameter.

You pass parameters to `String` resources in your layout code like this:

```
android:text="@{@string/string_name(arg1, arg2)}"
```

This passes two parameters—arg1 and arg2—to a String resource named string_name.

where `string_name` is the name of the `String` resource, and `arg1` and `arg2` are two parameter values. `arg1` is passed to the `String`'s first argument, and `arg2` is passed to its second.

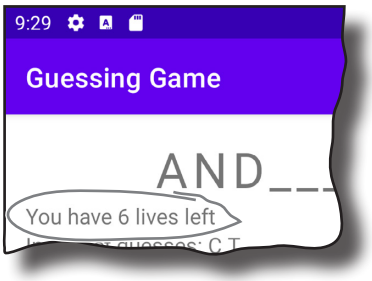
In the Guessing Game app, the `lives_left String` resource is defined using:

```
<string name="lives_left">You have %d lives left</string>
```

so we can use it to display the `livesLeft` property from `GameViewModel` using the following code:

```
<TextView
    android:id="@+id/lives"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="16sp"
    android:text="@{@string/lives_left(gameViewModel.livesLeft)}" />
```

This displays the lives left as a formatted String.




Similarly, the `incorrect_guesses String` resource is defined using:

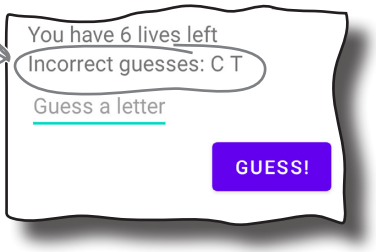
```
<string name="incorrect_guesses">Incorrect guesses: %s</string>
```

so we display the value of the `incorrectGuesses` property like this:

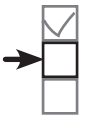
```
<TextView
    android:id="@+id/incorrect_guesses"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="16sp"
    android:text="@{@string/incorrect_guesses(gameViewModel.incorrectGuesses)}" />
```

That's everything that we need to know in order to change `GameFragment`'s layout so that it uses data binding. Let's see what the full code looks like.

This displays a formatted String of the user's incorrect guesses.



# The full code for fragment_game.xml



**ResultFragment**
**GameFragment**
**Add button**

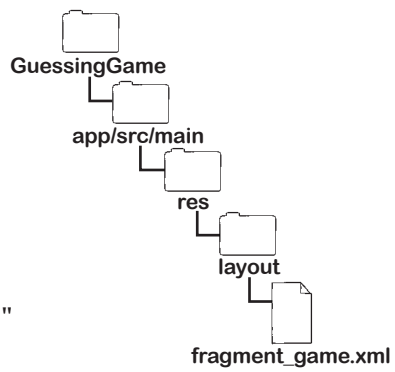
Here's the full code for `GameFragment`'s layout; update the code for *fragment_game.xml* to include the changes (in bold):

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

    <data>
        <variable
            name="gameViewModel"
            type="com.hfad.guessinggame.GameViewModel" />
    </data>

    <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:orientation="vertical"
        android:padding="16dp"
        ~~tools:context=".GameFragment"~~>

        <TextView
            android:id="@+id/word"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:textSize="36sp"
            android:letterSpacing="0.1"
            android:text="@{gameViewModel.secretWordDisplay}" />
```

We want to use data binding, so the layout needs to include a <layout> element at its root.

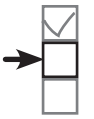
Define a data binding variable.

Delete these lines from here, and move them to the <layout> element.




Display the value of the view model's secretWordDisplay property.

The code continues on the next page.

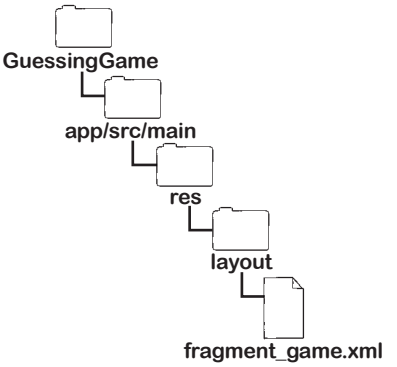


# fragment_game.xml continued

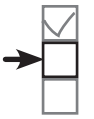
```
        <TextView
            android:id="@+id/lives"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="16sp"
            android:text="@{@string/lives_left(gameViewModel.livesLeft)}" />

        <TextView
            android:id="@+id/incorrect_guesses"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="16sp"
            android:text="@{@string/incorrect_guesses(gameViewModel.incorrectGuesses)}" />

        <EditText
            android:id="@+id/guess"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="16sp"
            android:hint="Guess a letter"
            android:inputType="text"
            android:maxLength="1" />

        <Button
            android:id="@+id/guess_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:text="Guess!"/>
    </LinearLayout>
</layout>
```

Display the lives left and incorrect guesses.

The <layout> element's closing tag




That's all the code we need to change in the layout file. Before we take the app for a test drive, however, we first need to update the corresponding fragment code in *GameFragment.kt*.



ResultFragment
**GameFragment**
Add button

# We need to set the gameViewModel variable

The first change we need to make to the `GameFragment` code is to set the layout's data binding variable `gameViewModel` to an instance of the fragment's view model.

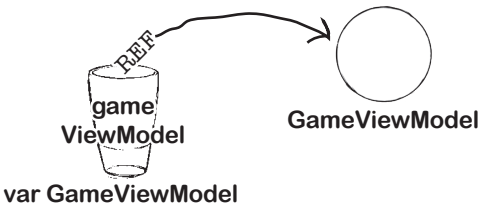
`GameFragment` currently includes the following line in its `onCreateView()` method, which uses the view model provider to get a `GameViewModel` object:

```
viewModel = ViewModelProvider(this).get(GameViewModel::class.java)
```

We can therefore assign this view model to the `gameViewModel` data binding variable by adding the following line:

```
binding.gameViewModel = viewModel
```

Once the `gameViewModel` variable is linked to the view model in this way, the layout's views can use it to get their values.




## We also need to make the data binding use live data

As well as setting the `gameViewModel` variable, we need to make the layout's data binding work with live data.

Unlike `ResultViewModel`, the `GameViewModel` code uses live data so that when its property values are updated, the `GameFragment` code responds. The following code in *GameFragment.kt*, for example, observes the `incorrectGuesses` property, and updates the `incorrect_guesses` view when it's updated:

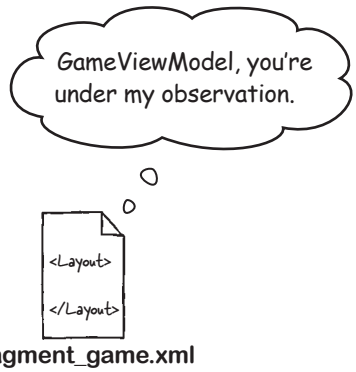
```
viewModel.incorrectGuesses.observe(viewLifecycleOwner, Observer { newValue ->
    binding.incorrectGuesses.text = "Incorrect guesses: $newValue"
})
```

As we're now using data binding, we can make each view respond to live data changes by adding the following line to the fragment code:

```
binding.lifecycleOwner = viewLifecycleOwner
```

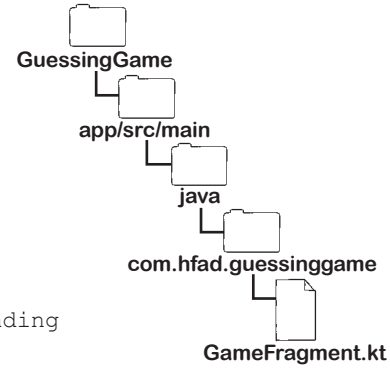
Each time a live data property value in the view model gets updated, the layout no longer has to rely on the fragment code updating its views. The layout can respond to any updates without any further intervention from the fragment.

That's everything that we need to know in order to update the `GameFragment` code. Let's see what it looks like.



# The full code for GameFragment.kt

Here's the full code for `GameFragment`; update the code for *GameFragment.kt* so that it includes the changes shown here (in bold):




```
package com.hfad.guessinggame

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.guessinggame.databinding.FragmentGameBinding
import androidx.navigation.findNavController
import androidx.lifecycle.ViewModelProvider
import androidx.lifecycle.Observer

class GameFragment : Fragment() {
    private var _binding: FragmentGameBinding? = null
    private val binding get() = _binding!!
    lateinit var viewModel: GameViewModel

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentGameBinding.inflate(inflater, container, false)
        val view = binding.root
        viewModel = ViewModelProvider(this).get(GameViewModel::class.java)

        binding.gameViewModel = viewModel
        binding.lifecycleOwner = viewLifecycleOwner

        viewModel.incorrectGuesses.observe(viewLifecycleOwner, Observer { newValue ->
            binding.incorrectGuesses.text = "Incorrect guesses: $newValue"
        })
```

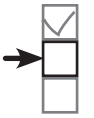
Set the data binding variable.

This lets the layout respond to live data updates.

Delete these lines.

The code continues on the next page.

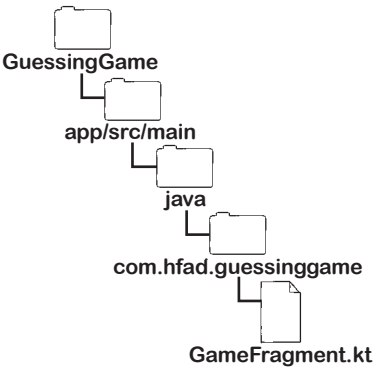
# GameFragment.kt continued



ResultFragment
**GameFragment**
Add button

Delete these lines.

```
        viewModel.livesLeft.observe(viewLifecycleOwner, Observer { newValue ->
            binding.lives.text = "You have $newValue lives left"
        })

        viewModel.secretWordDisplay.observe(viewLifecycleOwner, Observer { newValue ->
            binding.word.text = newValue
        })

        viewModel.gameOver.observe(viewLifecycleOwner, Observer { newValue ->
            if (newValue) {
                val action = GameFragmentDirections
                    .actionGameFragmentToResultFragment(viewModel.wonLostMessage())
                view.findNavController().navigate(action)
            }
        })

        binding.guessButton.setOnClickListener() {
            viewModel.makeGuess(binding.guess.text.toString().uppercase())
            binding.guess.text = null
        }
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

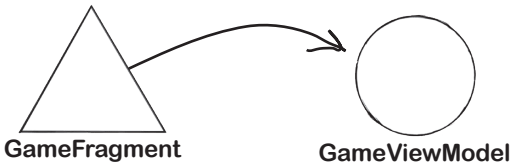



Those are all the changes we need to make to `GameFragment`. Let's run through what happens when the code runs, and take the app for a test drive.

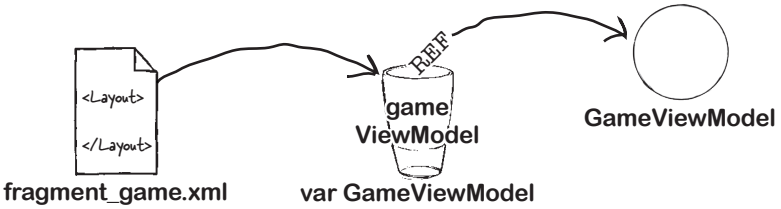
# What happens when the app runs

The following things happen when the app runs:

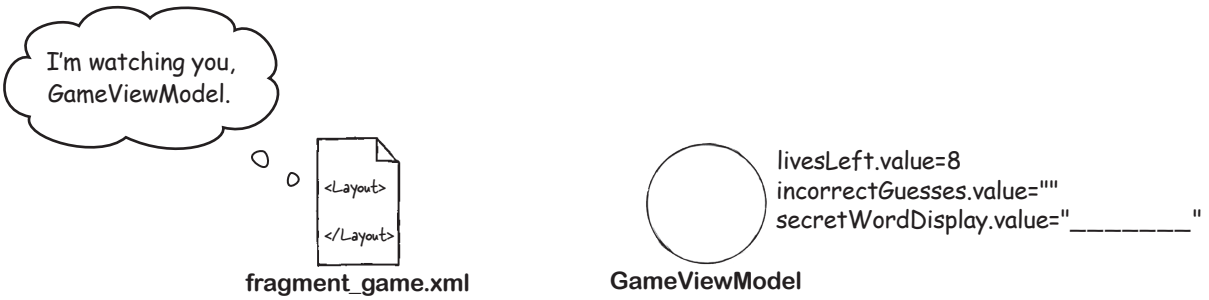
1. **GameFragment gets a reference to its GameViewModel object.**



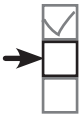

2. **GameFragment assigns the GameViewModel object to the layout's gameViewModel data binding variable.**
The layout's views can now use this variable to access the view model's properties and methods.




3. **GameFragment sets the layout's lifecycle owner.**
The layout can now observe the view model's live data properties and respond to any changes.

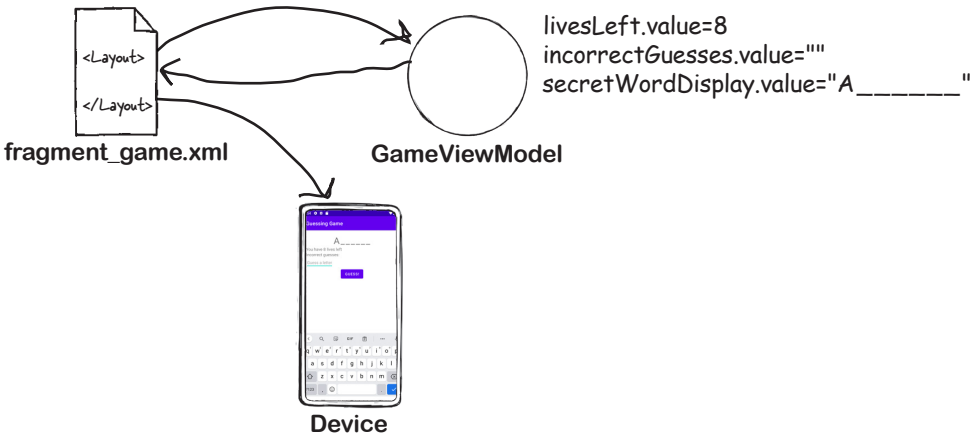


# The story continues

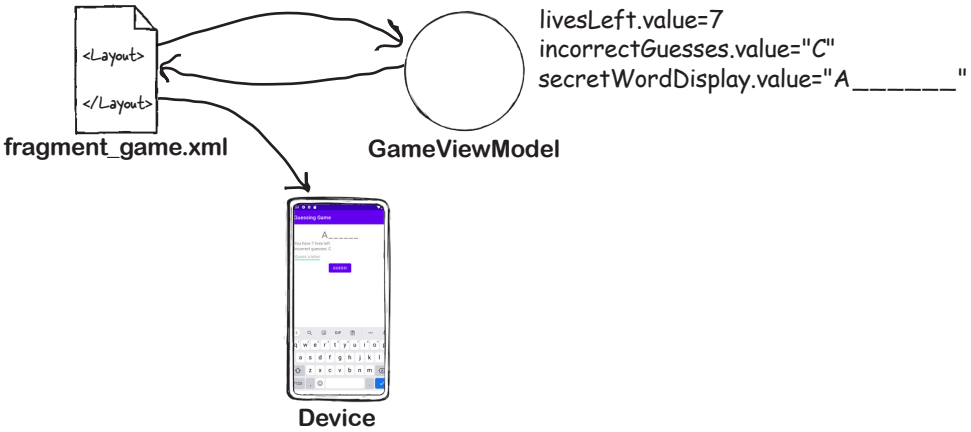


ResultFragment
**GameFragment**
Add button

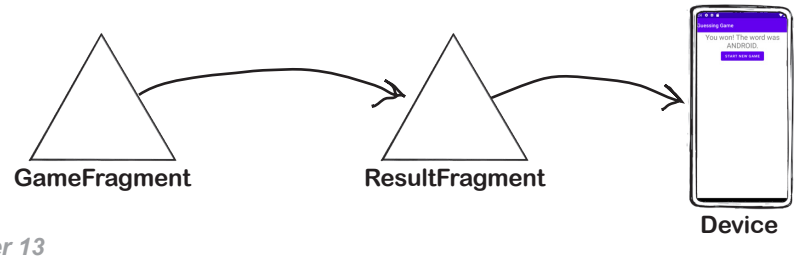
4 **When the user makes a correct guess, the GameViewModel object's secretWordDisplay property gets updated.**
The layout observes this change and its `word` view displays the new text.




5 **When the user makes an incorrect guess, the GameViewModel object's livesLeft and incorrectGuesses properties are updated.**
The layout observes the changes and its corresponding views display the new text.

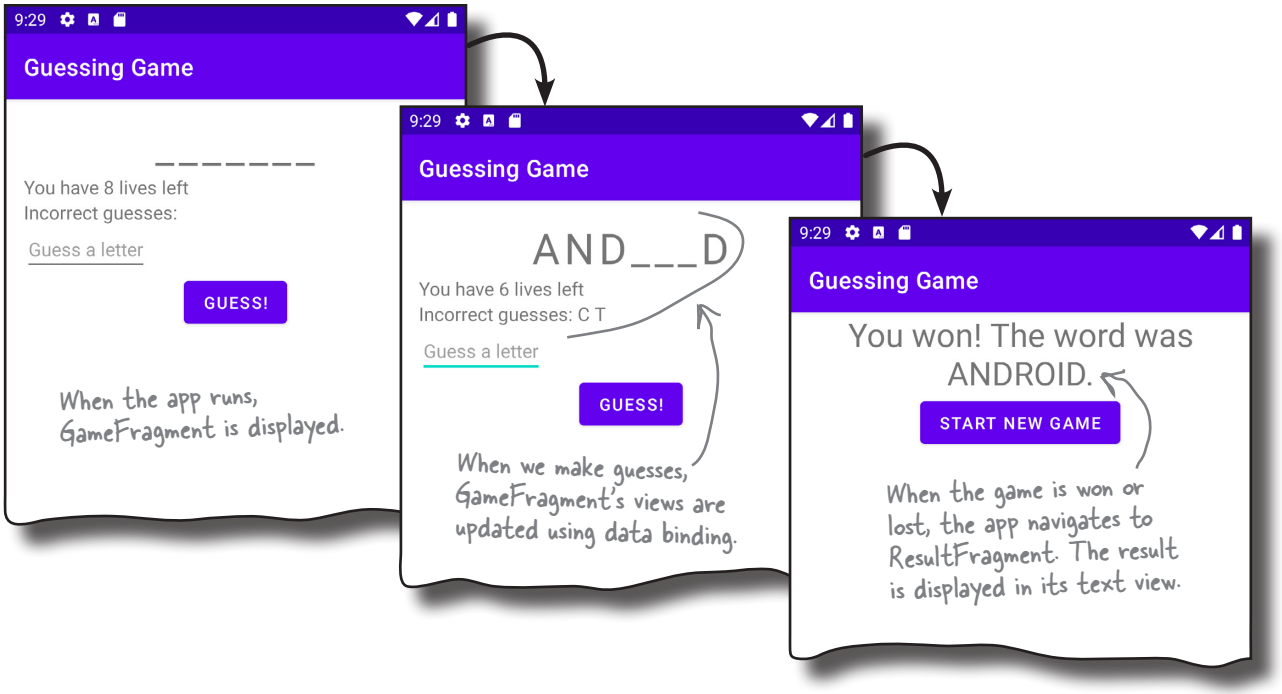



6 **When the game is over, GameFragment navigates to ResultFragment.**



# Test Drive

When we run the app, it works in the same way as before. This time, however, `GameFragment` uses data binding to get property values directly from `GameViewModel`.




## there are no Dumb Questions

**Q: Can you use data binding to make views update properties in the view model?**

A: Yes! Views can update mutable properties using `@=` instead of `@`. We'll show you an example of this in the next chapter.

**Q: You're setting the `word` view's text to `"@{gameViewModel.secretWordDisplay}"`. As `secretWordDisplay`'s type is `LiveData`, shouldn't you use `secretWordDisplay.value` instead?**

A: No. The layout will automatically use the `value` property without you telling it to.

**Q: I tried to set the text of the `lives` view to `"@{gameViewModel.livesLeft}"` but the app crashed. Any idea why?**

A: The `livesLeft.value` property is an `Int?`, and the view's `text` property needs to be a `String`. Try converting the property to a `String` using: `"@{gameViewModel.livesLeft.toString()}"` That should stop the app from crashing.

## Sharpen your pencil

The layout below needs to use a `String` resource (named `msg`) to display this text in a text view: "Hello, `user`. The numbers are `numbersText`. You guessed `numberGuessed` right."

`user`, `numbersText`, and `numberGuessed` are properties in `LotteryViewModel` where `user` is a `String`, `numbersText` is a `String`, and `numberGuessed` is an `Int`.

Can you write the code for the `String` resource?

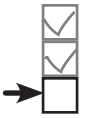
The layout is:

```
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".LotteryFragment">

    <data>
        <variable
            name="vm"
            type="com.hfad.lottery.LotteryViewModel" />
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <TextView
            android:id="@+id/message"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="16sp"
            android:text="@{@string/msg(vm.user, vm.numbersText, vm.numberGuessed)}" />
    </LinearLayout>
</layout>
```

The `String` resource is:

```
<string name="msg">                                                                    </string>
```

Answers on page 565.

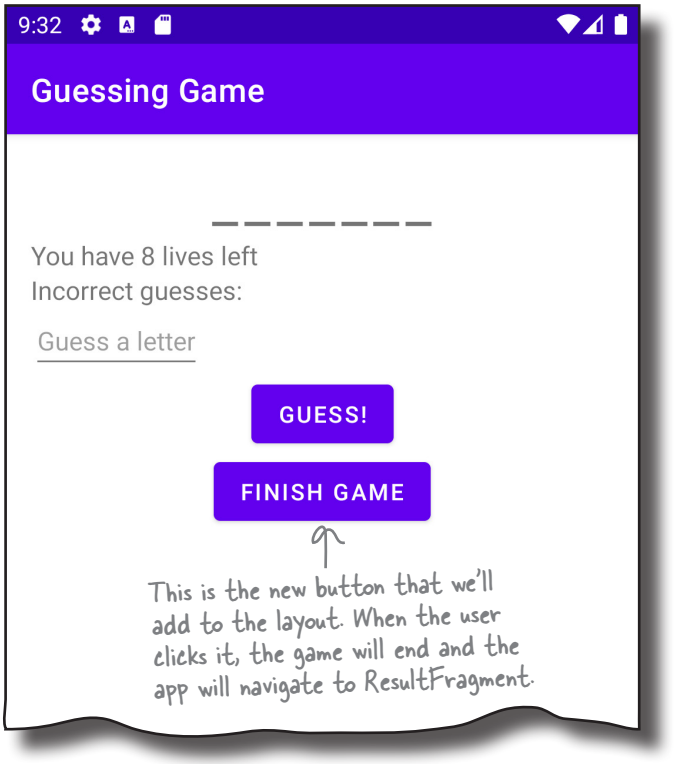


# You can use data binding to call methods

So far, you've implemented data binding so that views get their values directly from the view model instead of relying on fragment code.

Another way in which you can use data binding is to make a button call a method in the view model *without writing any extra fragment code*. To see how this works, we're going to add a new Finish Game button to `GameFragment`'s layout that, when clicked, will finish the game.

Here's what the new version of the screen will look like:



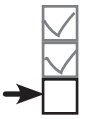

To make the new button finish the game, we'll do the following:

1. **Add a new finishGame() method to GameViewModel.**
   This method will set the view model's `_gameOver` property to *true*. When this happens, the existing code in `GameFragment` will respond by navigating to `ResultFragment`.

2. **Add the new button to the layout and make it call finishGame().**
   The button will call the method using data binding.

We'll begin by updating the `GameViewModel` code.

# Add finishGame() to GameViewModel.kt



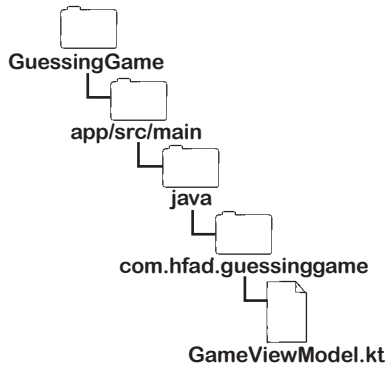
**ResultFragment**
**GameFragment**
**Add button**

We need to add a new `finishGame()` method to the `GameViewModel` code that sets the value of `_gameOver` to *true*.

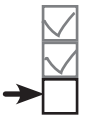
You already know how to do this, so here's the new code for *GameViewModel.kt*; make sure that you update the code to include the new method (in bold):

```
package com.hfad.guessinggame

...

class GameViewModel : ViewModel() {
    private val words = listOf("Android", "Activity", "Fragment")
    private val secretWord = words.random().uppercase()
    private val _secretWordDisplay = MutableLiveData<String>()
    val secretWordDisplay: LiveData<String>
        get() = _secretWordDisplay
    private var correctGuesses = ""
    private val _incorrectGuesses = MutableLiveData<String>("")
    val incorrectGuesses: LiveData<String>
        get() = _incorrectGuesses
    private val _livesLeft = MutableLiveData<Int>(8)
    val livesLeft: LiveData<Int>
        get() = _livesLeft
    private val _gameOver = MutableLiveData<Boolean>(false)
    val gameOver: LiveData<Boolean>
        get() = _gameOver

    init {
        _secretWordDisplay.value = deriveSecretWordDisplay()
    }

    private fun deriveSecretWordDisplay() : String {
        var display = ""
        secretWord.forEach {
            display += checkLetter(it.toString())
        }
        return display
    }
```

You don't need to change any of the code on this page.



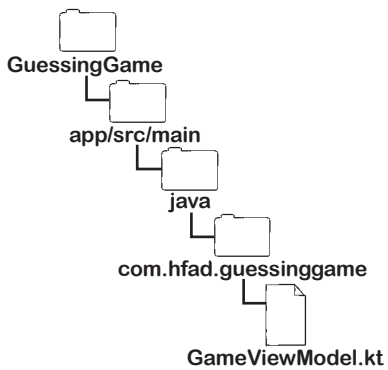

The code continues on the next page.



# GameViewModel.kt continued

```
    private fun checkLetter(str: String) = when (correctGuesses.contains(str)) {
        true -> str
        false -> "_"
    }

    fun makeGuess(guess: String) {
        if (guess.length == 1) {
            if (secretWord.contains(guess)) {
                correctGuesses += guess
                _secretWordDisplay.value = deriveSecretWordDisplay()
            } else {
                _incorrectGuesses.value += "$guess "
                _livesLeft.value = _livesLeft.value?.minus(1)
            }
            if (isWon() || isLost()) _gameOver.value = true
        }
    }

    private fun isWon() = secretWord.equals(secretWordDisplay.value, true)

    private fun isLost() = livesLeft.value ?: 0 <= 0

    fun wonLostMessage() : String {
        var message = ""
        if (isWon()) message = "You won!"
        else if (isLost()) message = "You lost!"
        message += " The word was $secretWord."
        return message
    }

    fun finishGame() {
        _gameOver.value = true
    }
}
```

Add this method.




That's all the `GameViewModel` code we need to add. Next, we'll add the new Finish Game button to `GameFragment`'s layout, and make it call the `finishGame()` method.

ResultFragment
GameFragment
**Add button**

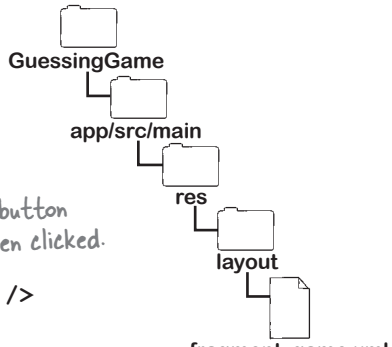
# Use data binding to make a button call a method when clicked

As we said earlier, you can use data binding to make a button call a method in its view model without writing any extra fragment code. To do this, you add an **`android:onClick`** attribute to the button, and set its value to an expression that calls the method.

In the Guessing Game app, we want to add a new button to `GameFragment`'s layout that calls `GameViewModel`'s `finishGame()` method when clicked. The code for this button looks like this:

```
<Button
    android:id="@+id/finish_game_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:text="Finish Game"
    android:onClick="@{() -> gameViewModel.finishGame()}" />
```

This makes the new button call finishGame() when clicked.




The line:

```
android:onClick="@{() -> gameViewModel.finishGame()}"
```

is just like adding the following code to your fragment:

```
binding.finishGameButton.setOnClickListener() {
    viewModel.finishGame()
}
```

except that this time, no fragment code is required. Behind the scenes, data binding uses the line:

```
android:onClick="@{() -> gameViewModel.finishGame()}"
```

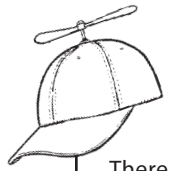
to create an `onClickListener` for the button. When the user clicks the button, the code is executed, which calls the view model's `finishGame()` method.

When you use a binding expression to call a method such as:

```
"@{() -> gameViewModel.finishGame()}"
```

the binding expression is sometimes known as a **listener binding**.

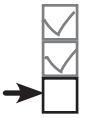
That's everything that we need to know in order to add the button to `GameFragment`'s layout and make it respond to clicks. Let's see what the full code looks like.



## Serious Coding

There are lots of other things you can do with binding expressions. Find out more here:

*https://developer.android.com/topic/libraries/data-binding/expressions*

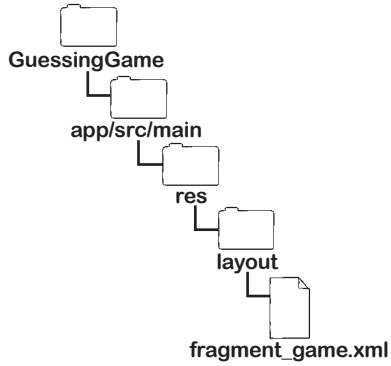


# The full code for fragment_game.xml

Here's the full code for `GameFragment`'s layout; update the code for *fragment_game.xml* to include the new button (in bold):

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

    <data>
        <variable
            name="gameViewModel"
            type="com.hfad.guessinggame.GameViewModel" />
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:padding="16dp">

        <TextView
            android:id="@+id/word"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:textSize="36sp"
            android:letterSpacing="0.1"
            android:text="@{gameViewModel.secretWordDisplay}" />

        <TextView
            android:id="@+id/lives"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="16sp"
            android:text="@{@string/lives_left(gameViewModel.livesLeft)}" />
```

You don't need to change any of the code on this page.

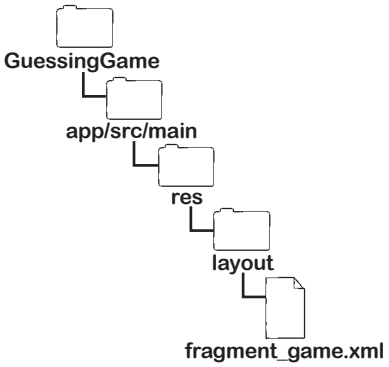



The code continues on the next page.

# fragment_game.xml continued

ResultFragment
GameFragment
**Add button**

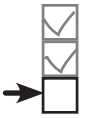
```
        <TextView
            android:id="@+id/incorrect_guesses"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="16sp"
            android:text="@{@string/incorrect_guesses(gameViewModel.incorrectGuesses)}" />

        <EditText
            android:id="@+id/guess"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="16sp"
            android:hint="Guess a letter"
            android:inputType="text"
            android:maxLength="1" />

        <Button
            android:id="@+id/guess_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:text="Guess!"/>
```




Add the new button.

```
        <Button
            android:id="@+id/finish_game_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:text="Finish Game"
            android:onClick="@{() -> gameViewModel.finishGame()}" />
    </LinearLayout>
</layout>
```

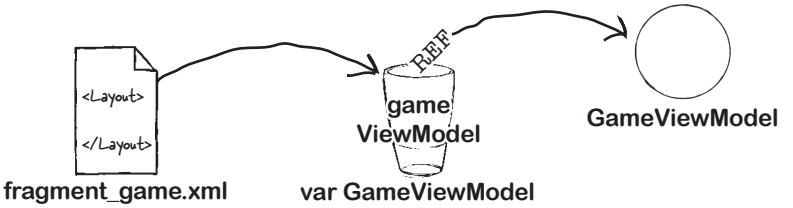
Let's see what happens when the code runs.



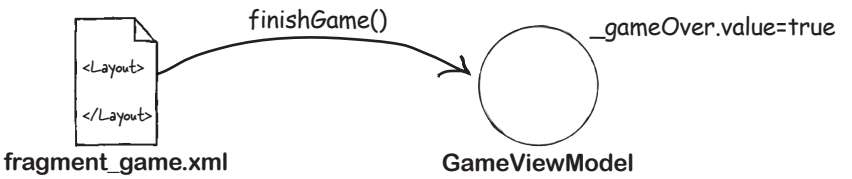
# What happens when the app runs

The following things happen when the app runs:

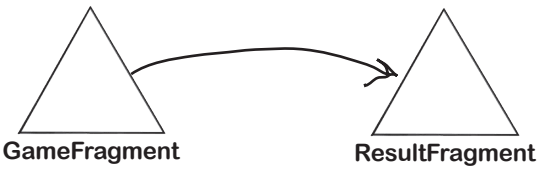
1. **GameFragment assigns the GameViewModel object to the layout's gameViewModel data binding variable.**
The layout's views can now use the variable to access the view model's properties and methods.




2. **The user clicks on the Finish Game button.**
The button calls the `finishGame()` method in the `GameViewModel` object, which sets its `_gameOver` property to *true*.

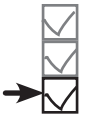



3. **GameFragment observes that _gameOver has a new value.**
It responds by navigating to `ResultFragment`.



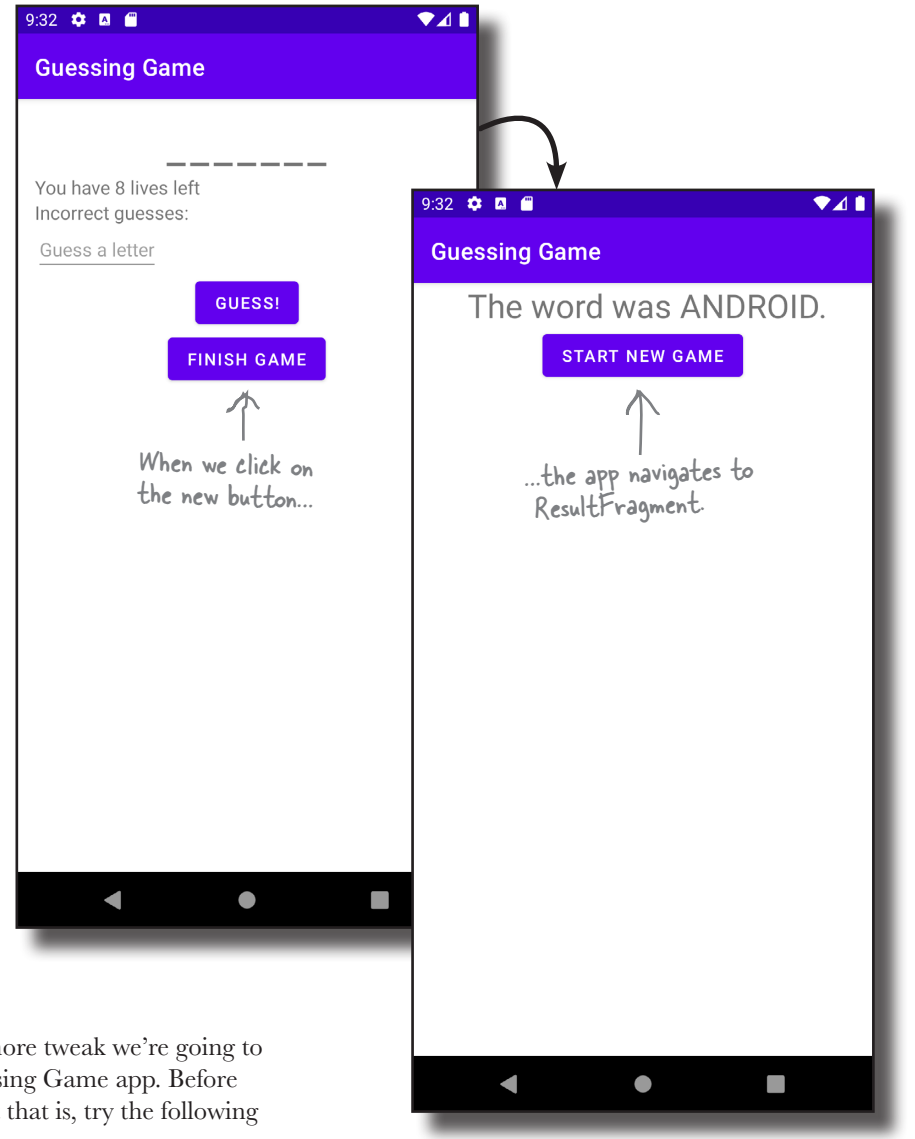

Let's take the app for a test drive.

# Test Drive



**ResultFragment**
**GameFragment**
**Add button**

When we run the app, `GameFragment`'s layout includes a new Finish Game button. When we click on the button, the app navigates to `ResultFragment`, which displays the secret word.



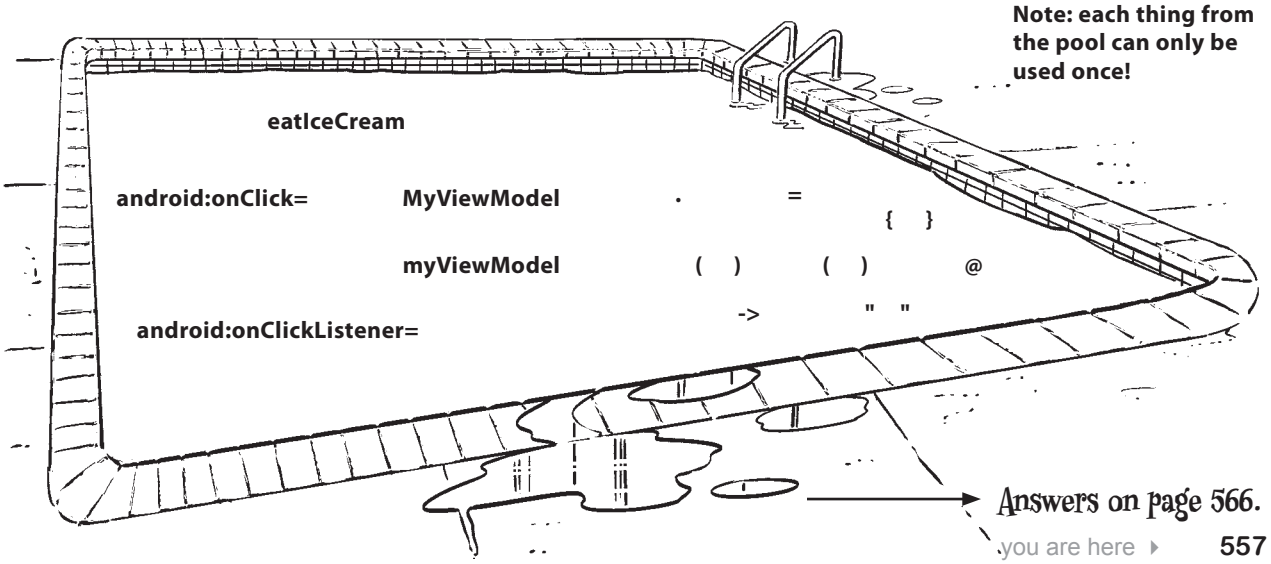

There's just one more tweak we're going to make to the Guessing Game app. Before we show you what that is, try the following exercise.

# Pool Puzzle

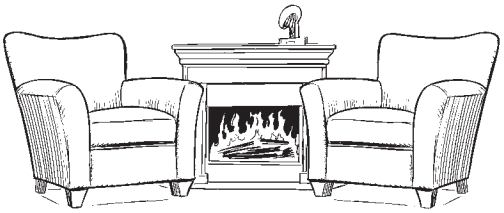
Your **goal** is to update the layout code below so that its button, when clicked, calls the `eatIceCream()` method in `MyViewModel`. Take code snippets from the pool and place them into the blank lines in the code. You may **not** use the same snippet more than once, and you won't need to use all the snippets.

```
<layout ...>
    <data>
        <variable
            name="myViewModel"
            type="com.hfad.myapp.MyViewModel" />
    </data>

    <LinearLayout ...>

       <Button
            android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Eat Ice Cream"
            ............................................................................................/>

    </LinearLayout>
</layout>
```

**Note: each thing from the pool can only be used once!**




Answers on page 566.



# Fireside Chats

Tonight's talk: **View Binding and Data Binding air their differences**

**Data Binding:**

View Binding, we need to talk.

Surely you've noticed how people keep getting the two of us confused. I mean, just because we both have Binding in our names, it doesn't mean we're the same thing.

You think so?

Maybe we do, but under completely different circumstances.

You can't claim sole responsibility for getting rid of `findViewById`. That's down to the binding class, not you. The binding class that *I* generate, too.

Besides, generating a binding class for every single layout is such a waste of space.

**View Binding:**

We do? Why?

You've got to admit our names sound similar. Besides, we have a lot in common.

Yes! We both generate binding classes. The *same* binding classes. Don't pretend that's not true.

It's true that I'm much more convenient and helpful than you. I generate a binding class for every layout, and most of the time, that's exactly what's needed. Developers thank me for it. I even get fan mail. "Oh, View Binding, I'm your biggest fan! Thank you for eliminating `findViewById`. I thought my pain would never end!"

Whatever.

Strictly speaking, I ignore any layouts with a `viewBindingIgnore` attribute that's set to *true*. That's way less complicated than your approach.

**Data Binding:**

Complicated? I'm just more discerning than you are. I only generate binding classes for layouts with a `<layout>` element. That way, the developer gets to decide exactly where I'm needed.

So it means a bit of extra code, but I can do things that you can only dream of. I let views get their values directly from the view model, for instance.

But that's the point. You rely on fragment code to tell the views what they should do, but when I'm involved, that's not necessary. I let the views get the values directly, which means simpler fragment code. I even let them use live data properties so that they can keep themselves up to date.

No, I don't. I have two-way data binding where views can update values *and* respond to changes. Look it up. It'll change your life.

But of course. And why not?

I hardly call simple method calls irresponsible, and it saves the developer from writing yet more fragment code. But I accept that for more complex things, the code really belongs in the fragment, not the layout.

You're just jealous.

**View Binding:**

Yes, complicated! Who really wants to add even more XML to their layout code? That's so much extra typing!

I can do that too. Using fragment code.

Huh. But what about updating property values? I bet you still need fragment code for that.

Huh. You'll be telling me next you let views call methods.

It makes the layout code overly complicated. I just can't believe you'd be so irresponsible.

So even you admit you have your limitations.

Riiiiiight. You keep telling yourself that…

ResultFragment
GameFragment
**Add button**

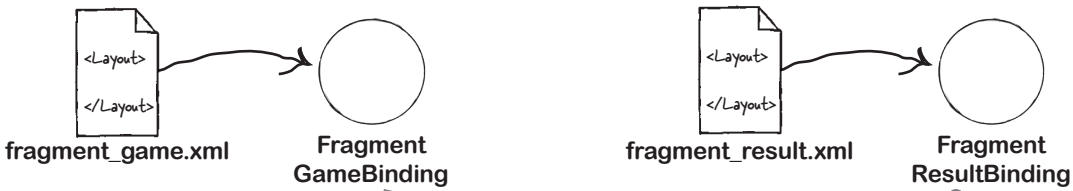
# We can switch off view binding

We've nearly finished updating the Guessing Game app, but there's one more tweak we can make so that only the binding classes we're using for data binding will get generated.

As you know, data binding and view binding generate the same binding classes, but under different circumstances: view binding generates a binding class for *every* layout, while data binding generates the class for every layout with a `<layout>` element at its root.

In the Guessing Game app, we're only using the binding classes for *fragment_game.xml* and *fragment_result.xml*. Since both of these files include a `<layout>` element, we can safely switch off the view binding option, and their binding classes will continue to get generated via data binding:

The only layout file that data binding won't generate a binding class for is activity_main.xml because it doesn't include a <layout> element. We don't need to use this binding class, so view binding can be safely switched off.



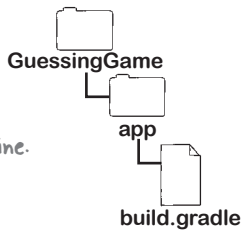

Data binding generates these two files for us as the layout files have root <layout> elements.

## Switch off view binding in the app's build.gradle file

To switch off view binding, open the *GuessingGame/app/build.gradle* file, and remove the view binding line from the `buildFeatures` section like this:

```
android {
    ...
    buildFeatures {
        viewBinding true
        dataBinding true
    }
}
```

Delete this line.

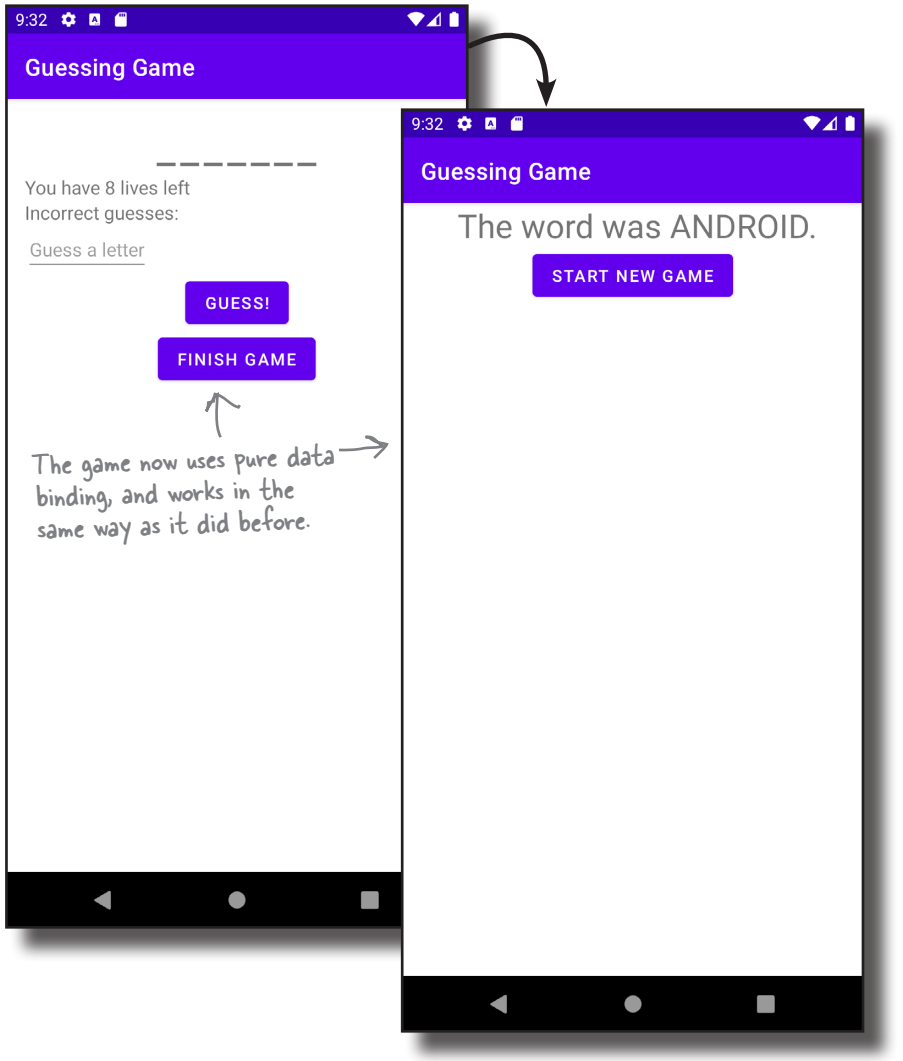



Then choose the Sync Now option to sync this change with the rest of the project.

Let's take the app for a quick test drive and make sure that it still works.

# Test Drive

When we run the app, the game works in the same way as before. Switching off view binding has made no difference because data binding has generated all the binding classes that the app needs.

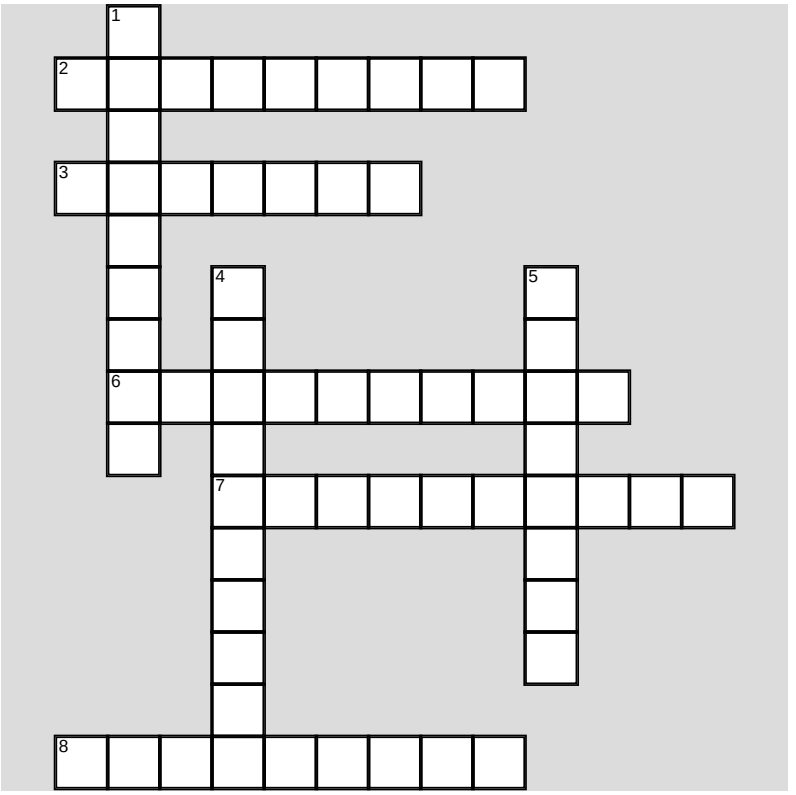



Congratulations! You've now learned how to use data binding, and used it in conjunction with view models and live data. This approach can be used to simplify your fragment code and build reactive, responsive apps.

# Binding-Cross

Each of the clues for the crossword below is a binding expression that's used for a `TextView`'s `android:text` property. Can you solve the crossword using the output for each clue?

Hint: Each binding expression uses a `String` resource and a view model. You'll find the code for these on the next page, along with a layout that includes the `TextView`.




## Across

**2** "@{@string/string2(vm.d[1])}"

**3** "@{@string/string3(vm.d[2], vm.g[0])}"

**6** "@{@string/string1(vm.a)}"

**7** "@{@string/string3(vm.f[1], vm.h[0])}"

**8** "@{@string/string1(vm.d[vm.i[1]])}"

## Down

**1** "@{@string/string3(vm.e[2], vm.g[0])}"

**4** "@{@string/string4(vm.h[2], vm.a)}"

**5** "@{@string/string4(vm.h[1], vm.g[2])}"

**Layout:**

```
<?xml version="1.0" encoding="utf-8"?>
<layout...>

    <data>
        <variable
            name="vm"
            type="com.hfad.crossword.CrosswordViewModel" />
    </data>

    <LinearLayout...>
        <TextView
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text=...../>
    </LinearLayout>
</layout>
```

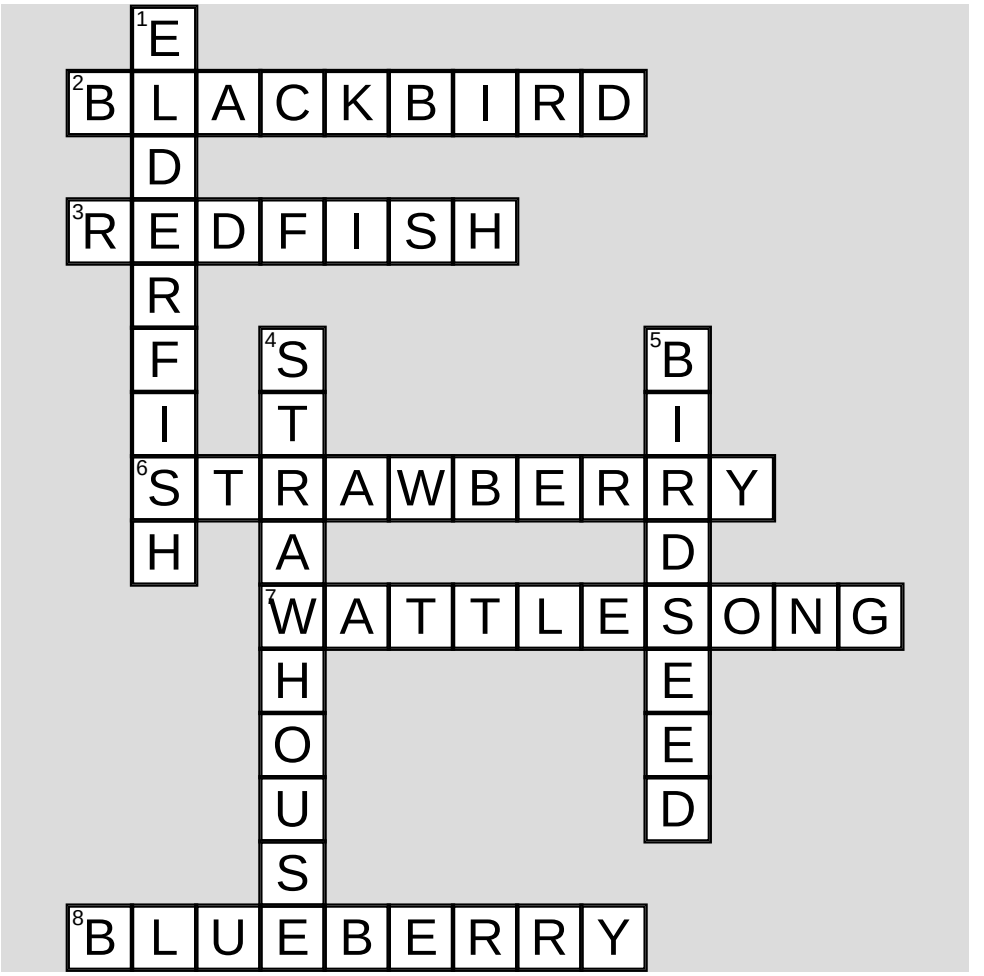
**View model:**

```
...
class CrosswordViewModel : ViewModel() {
    val a = "STRAW"
    val b = "RASP"
    val c = "CRAN"
    val d = listOf("BLUE", "BLACK", "RED")
    val e = listOf("GOOSE", "MOOSE", "ELDER")
    val f = listOf("LINGON", "WATTLE", "WEAVER")
    val g = listOf("FISH", "FLY", "BIRD", "HORSE")
    val h = listOf("SONG", "SEED", "HOUSE")
    val i = listOf(2,0,1,2,0)
}
```

**String resource file:**

```
<resources>
    <string name="app_name">Crossword</string>
    <string name="string1">%sBERRY</string>
    <string name="string2">%sBIRD</string>
    <string name="string3">%1$s%2$s</string>
    <string name="string4">%2$s%1$s</string>
</resources>
```

# Binding-Cross Solution



## Sharpen your pencil Solution

The layout below needs to use a `String` resource (named `msg`) to display this text in a text view: "Hello, `user`. The numbers are `numbersText`. You guessed `numberGuessed` right."

`user`, `numbersText`, and `numberGuessed` are properties in `LotteryViewModel` where `user` is a `String`, `numbersText` is a `String`, and `numberGuessed` is an `Int`.

Can you write the code for the `String` resource?

The layout is:

```
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".LotteryFragment">

    <data>
        <variable
            name="vm"
            type="com.hfad.lottery.LotteryViewModel" />
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <TextView
            android:id="@+id/message"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="16sp"
            android:text="@{@string/msg(vm.user, vm.numbersText, vm.numberGuessed)}" />
    </LinearLayout>
</layout>
```

The String resource needs to accept these three arguments.

The `String` resource is:

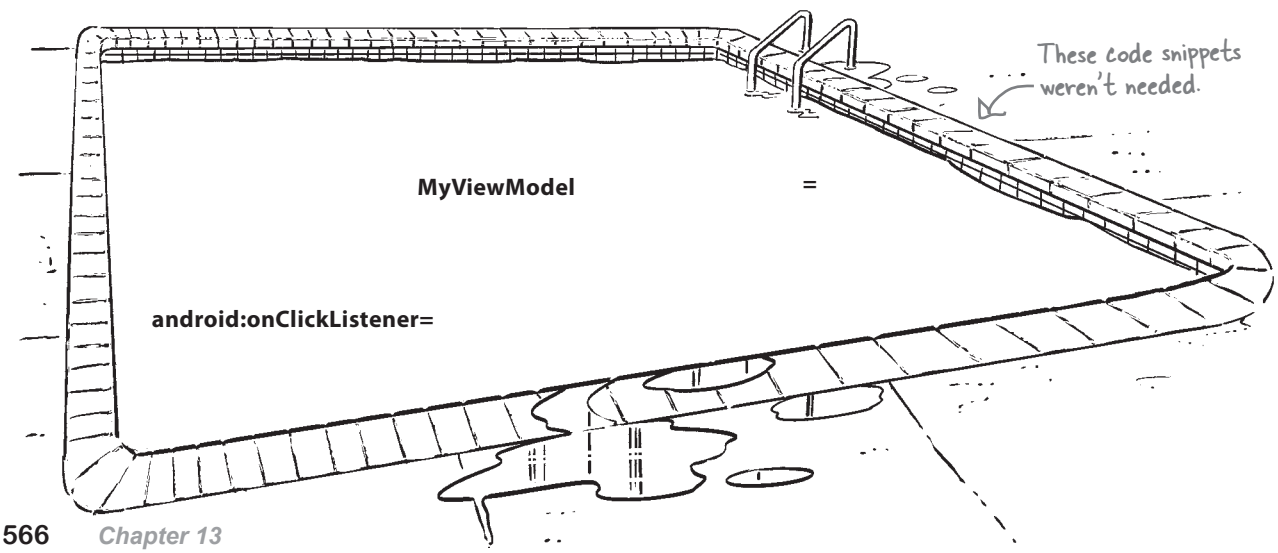
```
<string name="msg">Hello, %1$s. The numbers are %2$s. You guessed %3$d right.</string>
```

# Pool Puzzle Solution

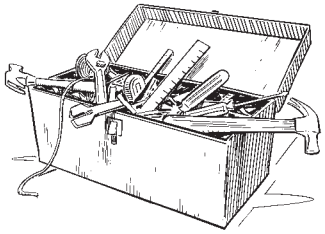
Your **goal** is to update the layout code below so that its button, when clicked, calls the `eatIceCream()` method in `MyViewModel`. Take code snippets from the pool and place them into the blank lines in the code. You may **not** use the same snippet more than once, and you won't need to use all the snippets.

```
<layout ...>
    <data>
        <variable
            name="myViewModel"
            type="com.hfad.myapp.MyViewModel" />
    </data>

    <LinearLayout ...>

       <Button
            android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Eat Ice Cream"
            android:onClick="@{() -> myViewModel.eatIceCream()}" />

    </LinearLayout>
</layout>
```

This expression uses data binding to call the view model's eatIceCream() method when the button is clicked.



# Your Android Toolbox



**You've got Chapter 13 under your belt and now you've added data binding to your toolbox.**

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

## Bullet Points

- Data binding gives layouts direct access to properties and methods.
- Enable data binding in the app's *build.gradle* file.
- Add a `<layout>` element to the root of each layout that needs to use data binding.
- Data binding generates a binding class for each layout with a `<layout>` element. This is the same binding class that gets generated with view binding.
- Use `<data>` and `<variable>` elements to define variables.
- `String` resources can accept arguments.
- Data binding lets views respond to when a live data property's value gets updated.

# 14 room databases

# 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()***.

# Most apps need to store data

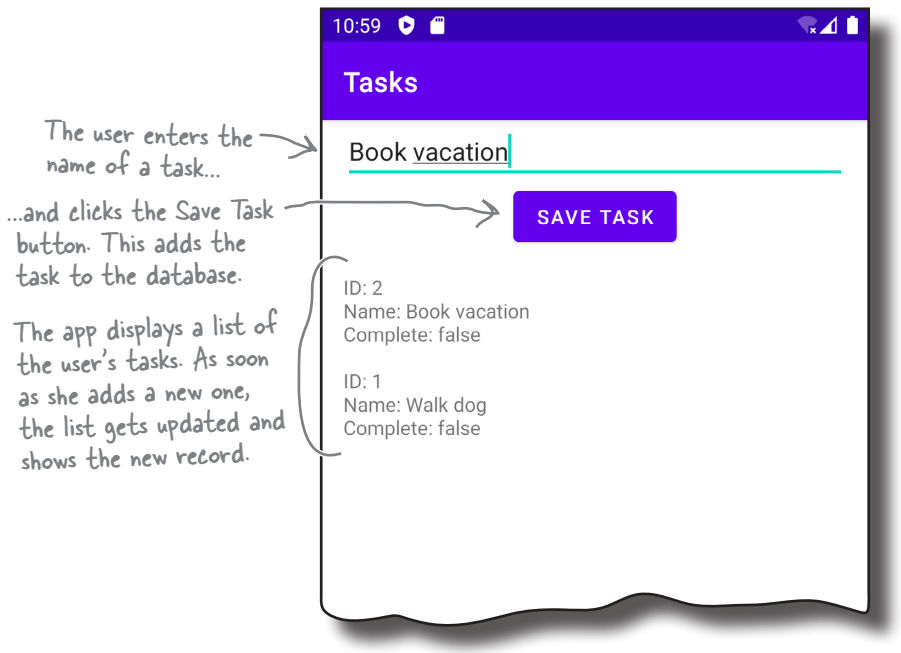
Nearly all of the apps that you've written so far have used a small amount of static data in order to function. The Guessing Game app we built in Chapters 11 to 13, for example, holds an array of `Strings` in its view model so that the game can select one at random for you to guess.

In the real world, however, most apps require more than just static data; they need to be able to save data that can change so that it doesn't get lost when the user shuts down the app. A music app might need to store playlists, for example, and a game might need to record the user's progress so that they can go back to where they left off.

## Apps can persist data using a database

In most cases, the best way of persisting the user's data is to use a database, so in this chapter, you're going to learn how to use one by building a Tasks app. This app will let the user add tasks to a database, and display a list of all the tasks they have already entered.

Here's what the app will look like:

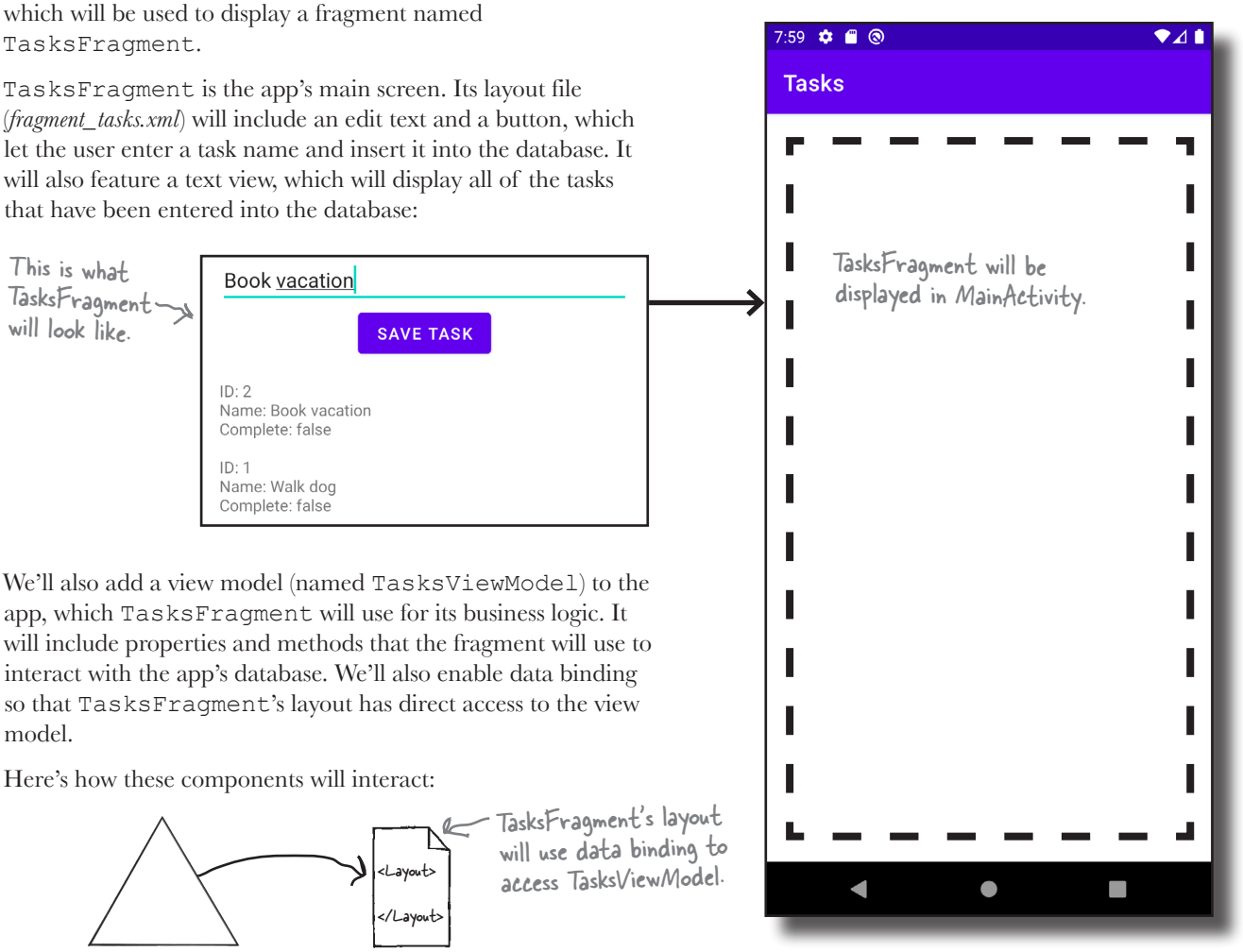



Before we start building the app, let's run through how it will be structured.

# How the app will be structured

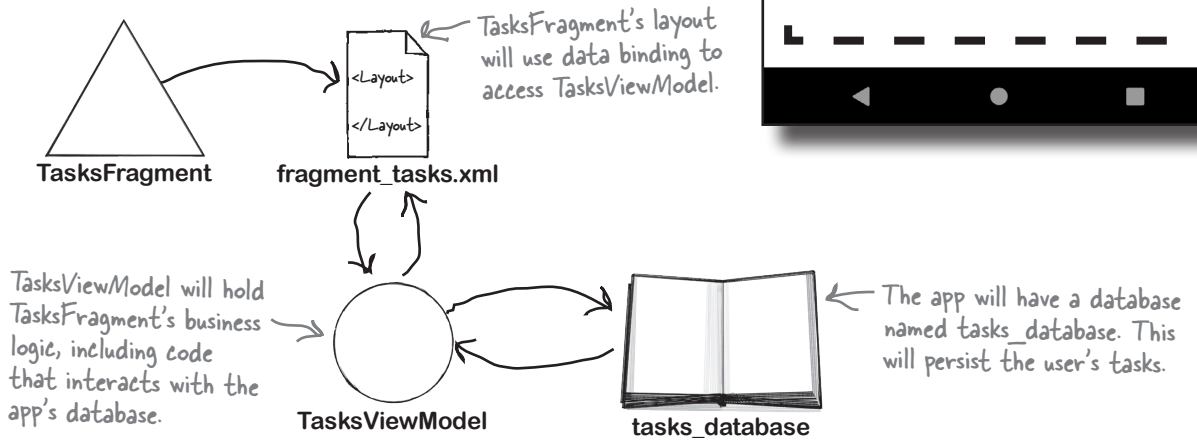
The app will contain a single activity (named `MainActivity`), which will be used to display a fragment named `TasksFragment`.

`TasksFragment` is the app's main screen. Its layout file (*fragment_tasks.xml*) will include an edit text and a button, which let the user enter a task name and insert it into the database. It will also feature a text view, which will display all of the tasks that have been entered into the database:




We'll also add a view model (named `TasksViewModel`) to the app, which `TasksFragment` will use for its business logic. It will include properties and methods that the fragment will use to interact with the app's database. We'll also enable data binding so that `TasksFragment`'s layout has direct access to the view model.

Here's how these components will interact:




To create the database, we're going to use an Android library called ***Room***. So what's Room?

# Room is a database library that sits on top of SQLite

Room is another part of Android Jetpack.

Behind the scenes, most Android databases use SQLite. SQLite is lightweight, stable, fast, and optimized for a single user, and these features make it a good choice for Android apps. Writing the code to create, manage, and interact with SQLite databases, however, can be tricky.

To make things easier, Android Jetpack includes a persistence library named ***Room*** that sits on top of SQLite. With Room, you get all the benefits of using SQLite, but using simpler code. It provides you with convenient annotations, for example, that let you quickly write database code in a way that's less repetitive and less error-prone.

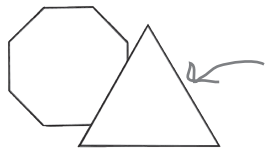
**MVVM is an architectural design pattern that's used to structure apps. It stands for Model-View-ViewModel.**

## Room apps are usually structured using MVVM

Apps that use Room—including the Tasks app—are usually structured using an architectural design pattern called ***MVVM***, which stands for Model-View-ViewModel. The structure looks like this:

### View

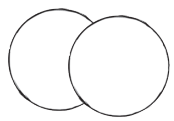
This includes all the code that's responsible for the UI, so the app's activity, fragment, and layout code.




In the Tasks app, the View layer is MainActivity, TasksFragment, and their layouts.

### ViewModel

View model objects are responsible for business logic and each view's data.




For example, TasksViewModel

### Model

This is the data that underpins the app. Here, it's a database, but in other apps, it could be a remote data source instead.

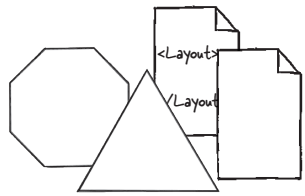
You'll find out more about the database layer as we go through this chapter.

This structure is similar to the one we used for the Guessing Game app we built, except that there's an extra Model layer for the database. It means that the activity and fragment's UI code is cleanly separated from the business logic that's held in the view model, and the view model is separated from any code that underpins the database.

You'll learn more about how to use the MVVM structure as we build the Tasks app.

# Here's what we're going to do

Here are the steps that we'll go through to build the Tasks app:




1. **Set up the basic app.**
   We'll create the app, update its *build.gradle* files so that it uses the libraries we need, and create the basic activity, fragment, and layout code.

2. **Write the database code.**
   In this step, we'll add the code to create a database with a table, along with the data access methods needed to interact with the table's data.

We'll define a database named tasks_database.

tasks_database

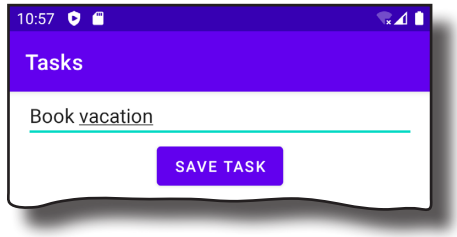
| taskId | task_name | task_done |
|---|---|---|
| | | |

task_table

The database will include a table named task_table.

3. **Insert task records.**
   We'll create a view model and update the app's fragment so that we can use the app to insert records.

We'll add views to the layout, which we'll use to insert task records.



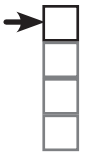

In the final step, we'll display a list of tasks so that we can see the records we've entered.

4. **Display a list of task records.**
   Finally, we'll update the view model and fragment code so that the app displays a list of all the task records held in the database.

## Create the Tasks project

We're going to use a new project for the Tasks app, so create one now using the same steps you used in the previous chapters. Choose the Empty Activity option, enter a name of "Tasks" and a package name of "com.hfad.tasks", and accept the default save location. Make sure the language is set to Kotlin and the minimum SDK is API 21 so it will run on most Android devices.

Next, we'll update the project's *build.gradle* files so that they include all the features and dependencies that are needed for the app.

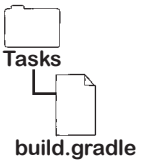



# Add a variable to the project build.gradle file...

We're going to use two Room libraries in this chapter, so we'll add a new variable to the project's *build.gradle* file to specify which version we'll be using, and to keep things consistent. To do this, open the file *Tasks/build.gradle*, and add the following lines (in bold) to the `buildscript` section:

```
buildscript {
    ext.lifecycle_version = "2.3.1"
    ext.room_version = "2.4.1"
    ...
}
```

We're using these to say which versions of the Room and lifecycle libraries we'll be using. They help keep things nice and consistent.




## ...and update the app build.gradle file, too

In the app's *build.gradle* file, we need to enable data binding, and add dependencies for the view model, live data, and Room libraries.

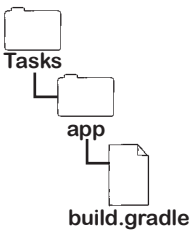
Open the file *Tasks/app/build.gradle*, and add the following lines (in bold) to the appropriate sections:

```
plugins {
    ...
    id 'kotlin-kapt'
}
```

This is a Kotlin annotation processing tool that Room needs.

```
android {
    ...
    buildFeatures {
        dataBinding true
    }
}
...
```

We're going to use data binding, so enable it.




```
dependencies {
    ...
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"
    implementation "androidx.lifecycle:lifecycle-livedata-ktx:$lifecycle_version"
    implementation "androidx.room:room-runtime:$room_version"
    implementation "androidx.room:room-ktx:$room_version"
    kapt "androidx.room:room-compiler:$room_version"
    ...
}
```

These are the view model and live data libraries.

We'll use these Room libraries.

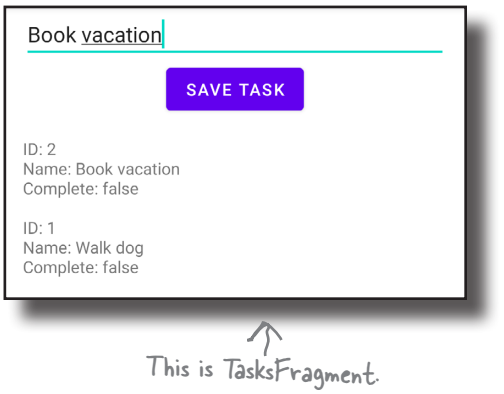
We need to use a Room compiler too.

Then click on the Sync Now option to sync the changes you've made with the rest of the project.

# Create TasksFragment

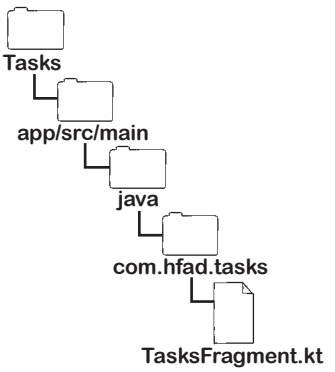
The app will include a single fragment named `TasksFragment`, which we'll use to display a list of all the tasks in the database, and to insert new ones.

To create `TasksFragment`, highlight the *com.hfad.tasks* package in the *app/src/main/java* folder, then go to File→New→Fragment→Fragment (Blank). Name the fragment "TasksFragment", name its layout "fragment_tasks", and make sure the language is set to Kotlin.

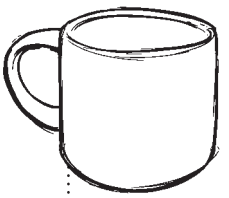



## Update TasksFragment.kt

Once you've added `TasksFragment` to the project, make sure that *TasksFragment.kt* matches the code shown here:




```
package com.hfad.tasks

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.tasks.databinding.FragmentTasksBinding

class TasksFragment : Fragment() {
    private var _binding: FragmentTasksBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentTasksBinding.inflate(inflater, container, false)
        val view = binding.root
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```



## Relax

**Don't worry if the compiler doesn't recognize the FragmentTasksBinding class.**

It's because we haven't yet added a `<layout>` element to the fragment's layout, which it needs to generate the binding class.

**Setup**
Database
Insert
Get tasks

# Update fragment_tasks.xml

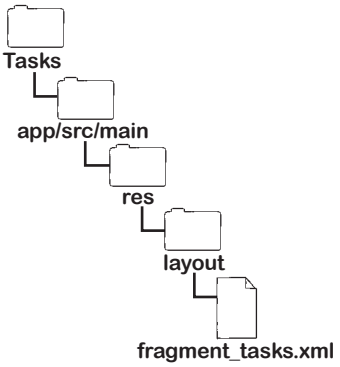
We also need to update `TasksFragment`'s layout so that it uses data binding, and includes views to let us enter new tasks and display a list of existing ones.

Open the file *fragment_tasks.xml*, and update it so that it matches the code shown here:

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

    <data>
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <EditText
            android:id="@+id/task_name"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:inputType="text"
            android:hint="Enter a task name" />

        <Button
            android:id="@+id/save_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:text="Save Task" />

        <TextView
            android:id="@+id/tasks"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
    </LinearLayout>
</layout>
```

The layout needs a <layout> element at its root in order to use data binding.

Later, we'll include a data binding variable here.




The edit text will be used to enter a task name.

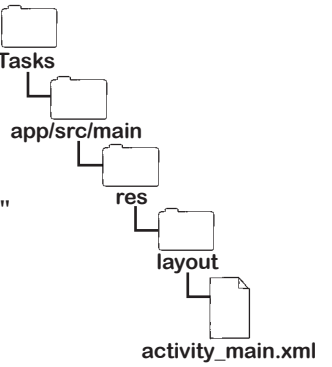
The button will save the task.

We'll use the text view to display a list of task records held in the database.

# Display TasksFragment in MainActivity's layout...

In order to use `TasksFragment`, we need to add it to `MainActivity`'s layout in a `FragmentContainerView`.

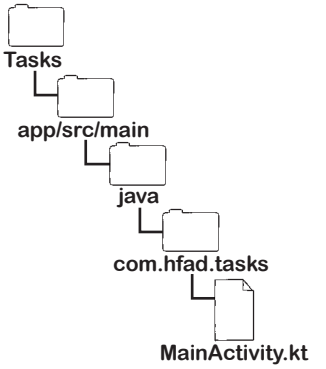
Update the file *activity_main.xml* so that it matches the code shown here:




```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/fragment_container_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:name="com.hfad.tasks.TasksFragment"
    tools:context=".MainActivity" />
```

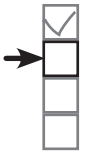
## ...and check the MainActivity.kt code

Once you've updated the layout, open *MainActivity.kt* and make sure its code looks like this:




```
package com.hfad.tasks

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)
    }
}
```

Now that we've updated the fragment and activity code, let's get to work on the Room database.



Setup
**Database**
Insert
Get tasks

# How Room databases are created

Room uses a set of annotated classes and interfaces to create and configure a SQLite database for your app. It requires three main things:

## 1. A database class

This defines the database, including its name and version number. It's used to get an instance of the database.

Database

## 2. Data classes for the tables

All of the data in a database is stored in tables. You define each table using a data class, which includes annotations to specify the table's name and columns.

A Kotlin data class lets you create objects whose main purpose is to hold data.

| taskId | task_name | task_done |
|---|---|---|
| 1 | Walk dog | false |
| 2 | Book vacation | false |
| 3 | Arrange picnic | false |

Table

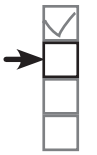
## 3. Interfaces for data access

You interact with each table using an interface, which specifies the data access methods your app needs. If you need to insert records, for example, you can add an `insert()` method to the interface, and to get all records, you can add a `getAll()` method.

Data Access Interface

Room uses these three things to generate all the code your app needs to create the SQLite database, its tables, and any data access methods.

Over the next few pages, we're going to show you how to write the code for these three components by defining a database for the Tasks app. We'll begin by defining its table.




# We'll store tasks data in a table

As we said before, all of a database's data is stored in one or more tables. Each table is made up of rows and columns, where each row is a record, and each column holds a single piece of data, like a number or piece of text.

You create a separate table for each type of data you want to hold. A calendar app, for example, might have a table to record events, and a weather app might include a table for its locations.

We want to store task records in the Tasks app, so we're going to create a table named "task_table". The table will look something like this:

These are records, which the user will enter using the app.

| taskId | task_name | task_done |
|---|---|---|
| 1 | Walk dog | false |
| 2 | Book vacation | false |
| 3 | Arrange picnic | false |

The table has three columns: taskId, task_name, and task_done.

## You define tables with an annotated data class

You specify any tables you want the database to include by defining a data class for each table. The data class needs to include a property for each of the table's columns, and you use annotations to tell Room how the table should be configured.

To see how this works, we're going to define a data class named `Task`, which we'll use to create a table in the Tasks app's database.

| (Data) Task |
|---|
| taskId: Long |
| taskName: String |
| taskDone: Boolean |

The Task data class includes a property for each of the table's columns.

## Create the Task data class

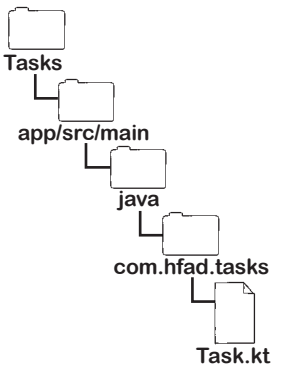
We'll begin by creating the data class. Highlight the *com.hfad.tasks* package in the *app/src/main/java* folder, then go to File→New→Kotlin Class/File. Name the file "Task" and choose the Class option.

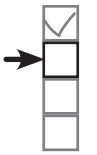
When you've created the file, update its code so that it looks like this:

```
package com.hfad.tasks

data class Task(
    var taskId: Long = 0L,
    var taskName: String = "",
    var taskDone: Boolean = false
)
```

Make sure that you make Task a data class.

Add these three properties.






# Specify a table name with @Entity

Now that we've created the `Task` data class, we need to add annotations to tell Room how to configure the table. We'll begin with the table name.

You name a table by adding an **`@Entity`** annotation to the data class, specifying the table name. We want to give the table in the Tasks app a name of "task_table", and the code to do so looks like this:

```
...
@Entity(tableName = "task_table")
data class Task(
...
```

This tells Room that the class describes the task_table entity.

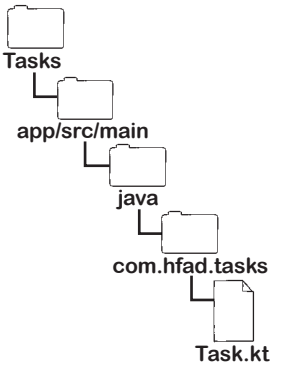
## Specify the primary key with @PrimaryKey

Next, we'll specify the table's primary key. This is used to uniquely identify a single record, and can't contain any duplicate values.

In the Tasks app, we'll use the `taskId` property for `task_table`'s primary key, and we'll get the table to autogenerate its values so that they're unique. This is done using a **`@PrimaryKey`** annotation like this:

```
...
    @PrimaryKey(autoGenerate = true)
    var taskId: Long = 0L,
...
```

taskId is the primary key, and Room will autogenerate its values.




## Specify column names with @ColumnInfo

The final thing we'll do is specify column names for the `taskName` and `taskDone` properties. This is done using the **`@ColumnInfo`** annotation like this:

```
...
    @ColumnInfo(name = "task_name")
    var taskName: String = "",

    @ColumnInfo(name = "task_done")
    var taskDone: Boolean = false
}
```

These annotations override the column names that will be used for these two properties.

Note that the `@ColumnInfo` annotation is *only needed if you want the column name to be different to the property name*. If you omit the annotation, Room will give the column the same name as the property.

That's everything that we need to know to complete the `Task` data class. We'll show you the full code on the next page.

# The full code for Task.kt

Here's the full code for the `Task` data class; update the code for *Task.kt* so that it includes the changes shown (in bold):

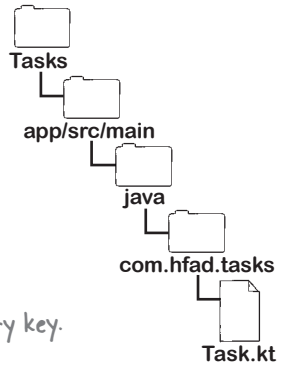
```
package com.hfad.tasks

import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "task_table")
data class Task(

    @PrimaryKey(autoGenerate = true)
    var taskId: Long = 0L,

    @ColumnInfo(name = "task_name")
    var taskName: String = "",

    @ColumnInfo(name = "task_done")
    var taskDone: Boolean = false
)
```

Import these classes.

Name the table.

Specify the primary key.

Name these two columns.




Room uses this file to create a table named `task_table` with an autogenerated primary key named `taskId`, and two extra columns named `task_name` and `task_done`. Here's what the table looks like:

| taskId | task_name | task_done |
|---|---|---|
| | | |

task_table

You'll find out how Room adds this table to the database when we write the database class a few pages ahead. First, we'll define some database access methods so that the app can insert, read, update, and delete the table's data.

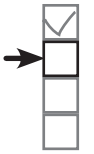
## there are no Dumb Questions

**Q: If I leave out the `@ColumnInfo` annotation, will Room still create a column for that data class property?**

A: Yes. It will create a column with the same name as the property. The `@ColumnInfo` annotation simply lets you rename the column.

**Q: Can I stop Room from creating a column for one of the data class properties?**

A: Yes. If you don't want to persist the data for one of the properties in the data class, you can annotate it with `@Ignore`. When you do so, Room won't create a column in the table for this property.



Setup
**Database**
Insert
Get tasks

# Use an interface to specify data operations

You specify how the app will access the table's data by creating an annotated interface. This interface defines a DAO—or *Data Access Object*—which includes all the methods the app needs to insert, read, update, and delete the data.

To see how this works, we'll create a new interface named `TaskDao`, which the Tasks app will use to interact with `task_table`'s data.

## Create the TaskDao interface

We'll begin by creating the interface. Highlight the *com.hfad.tasks* package in the *app/src/main/java* folder, then go to File→New→Kotlin Class/File. Name the file "TaskDao" and choose the Interface option.

When you've created the file, make sure that its code looks like this:

```
package com.hfad.tasks

interface TaskDao {

}
```

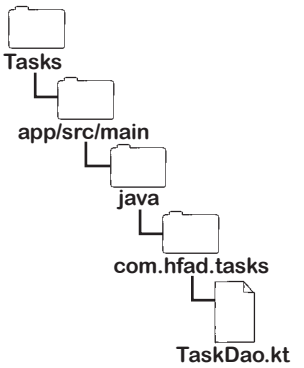
Make sure that TaskDao is defined as an interface.

## Use @Dao to mark the interface for data access

Next, we need to tell Room that the `TaskDao` interface defines data access methods. This is done by marking the interface with an **`@Dao`** annotation like this:

```
...
@Dao
interface TaskDao {
}
```

This tells Room that the interface is used for data access.

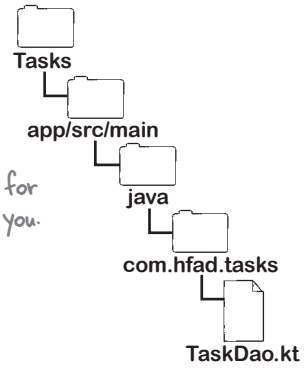



Once you've annotated the interface with `@Dao`, you add annotated methods that the app will use to interact with the data. If you want the app to insert records, for example, you need to add a method for this to the interface, and if you want it to get records, you need to add this method too.

The great news is that Room provides four annotations—`@Insert`, `@Update`, `@Delete`, and `@Query`—that make adding these methods a breeze. Let's find out how these are used by adding some data access methods to `TaskDao`.

# Use @Insert to insert a record

The first data access method we'll define is an `insert()` method, which the app will use to insert a task into `task_table`. The method will have one parameter—a `Task`—for the task we want to insert. We'll also mark the method with an **`@Insert`** annotation, which tells Room that the method is used to insert records.

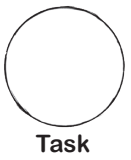



Here's the full code for the `insert()` method:

```
@Insert
fun insert(task: Task)
```

Yes! That's all the code you need to add to TaskDao for this method. Room takes care of everything else for you.

When Room sees an `@Insert` annotation, it automatically generates all the code the app needs to insert a record into the table so that you don't have to write it yourself. For the `insert()` method above, for example, it generates all the code needed to insert a `Task` object's data into `task_table` like this:




insert()

| taskId | task_name | task_done |
|---|---|---|
| 1 | Walk dog | false |

The insert() method takes the values of the Task's properties, and inserts them into the table. The table replaces the taskId with one that's autogenerated.

Any methods marked with `@Insert` can accept one or more entity objects for their arguments—objects whose type is marked with `@Entity`. `@Insert` methods can also accept collections of entity objects. The following method, for example, will insert all the tasks that are included in the `List<Task>` argument:

```
@Insert
fun insertAll(tasks: List<Task>)
```

If you wanted to add many task records to the database, you could use this method. It will insert every Task in the List<Task>.

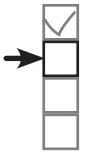
## Use @Update to update a record

Room can also generate all the code that's needed to update one or more existing records in the table. This is done by adding a method to the DAO interface that's marked with **`@Update`**. The following `update()` method, for example, generates all the code that's needed to update an existing task record:

```
@Update
fun update(task: Task)
```

This will update the task record with a matching taskId.

When called, this method updates the record with a matching `taskId` so that its data matches the `Task` object's property values.



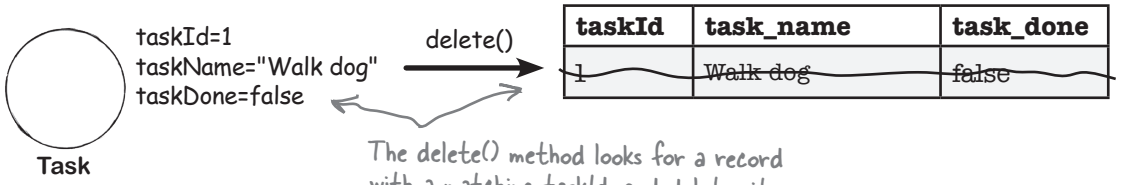

# Use @Delete to delete a record

There's also a **`@Delete`** annotation that's used to mark any methods that need to delete specific records from the table. To delete a single task record, for example, you could use a method defined like so:

```
@Delete
fun delete(task: Task)
```

This deletes the task record with a matching taskId.

The code that Room generates for this method will delete the record with a matching `taskId`:




## Use @Query for everything else

Any other data access methods are marked with `@Query`. This annotation lets you define a SQL statement (using `SELECT`, `INSERT`, `UPDATE`, or `DELETE`) that will be run when the method is called.

We're not going to teach you how to use SQL in this book, but if you want to learn more, we suggest reading Head First SQL by Lynn Beighley.

In the Tasks app, for example, we can use the following code to define a method named `get()` to return a live data `Task` for the record with a matching `taskId`:

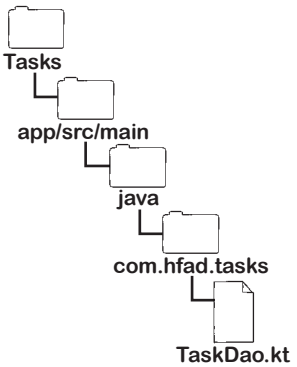
```
@Query("SELECT * FROM task_table WHERE taskId = :taskId")
fun get(taskId: Long): LiveData<Task>
```

This returns the task record with a certain taskId.

We can also define a `getAll()` method, which will return a live data `List` of all the records held in the table:

This returns all the task records in descending order of taskId.

```
@Query("SELECT * FROM task_table ORDER BY taskId DESC")
fun getAll(): LiveData<List<Task>>
```




As these methods return live data objects, the app can use them to be notified when the data changes. We'll use this feature later in the chapter to keep the list of task records displayed in `TasksFragment` up-to-date.

You now know everything that's needed to complete the code for `TaskDao`. We'll show you the full code on the next page.

# The full code for TaskDao.kt

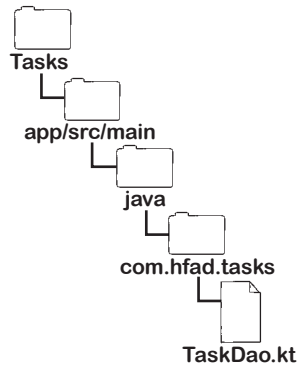
Here's the full code for the `TaskDao` interface; update the code for *TaskDao.kt* so that it includes the changes shown (in bold):

```
package com.hfad.tasks

import androidx.lifecycle.LiveData
import androidx.room.Dao
import androidx.room.Delete
import androidx.room.Insert
import androidx.room.Query
import androidx.room.Update

@Dao
interface TaskDao {
    @Insert
    fun insert(task: Task)

    @Update
    fun update(task: Task)

    @Delete
    fun delete(task: Task)

    @Query("SELECT * FROM task_table WHERE taskId = :taskId")
    fun get(taskId: Long): LiveData<Task>

    @Query("SELECT * FROM task_table ORDER BY taskId DESC")
    fun getAll(): LiveData<List<Task>>
}
```

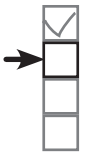
Import these classes.

Mark the interface with @Dao.

Include these data access methods.



We've now written the code for the `Task` data class (which defines a table) and the `TaskDao` interface (which specifies the data access methods). Next, we'll learn how to define the actual database.



Setup
**Database**
Insert
Get tasks

# Create a TaskDatabase abstract class

You define the app's database by creating an abstract class. The abstract class specifies the database name and version number, and any classes or interfaces that define the tables and data access methods.

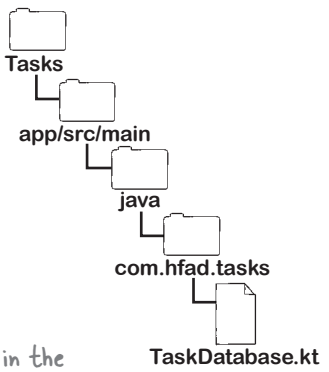
In the Tasks app, we're going to define the database using an abstract class named `TaskDatabase`. Highlight the *com.hfad.tasks* package in the *app/src/main/java* folder, then go to File→New→Kotlin Class/File. Name the file "TaskDatabase" and choose the Class option.

The `TaskDatabase` class needs to extend `RoomDatabase`, so update the code for *TaskDatabase.kt* so that it looks like this:

```
package com.hfad.tasks

import androidx.room.RoomDatabase

abstract class TaskDatabase : RoomDatabase() {
}
```

Import this class.

Make sure the class extends RoomDatabase.




## Annotate the class with @Database

Next, we need to mark the class with **`@Database`**, which tells Room that it defines a database. Here's the code to do this:

```
package com.hfad.tasks

import androidx.room.Database
import androidx.room.RoomDatabase

@Database(entities = [Task::class], version = 1, exportSchema = false)
abstract class TaskDatabase : RoomDatabase() {
}
```

Import the Database class.

This adds the table defined in the Task data class to the database.

As you can see, the **`@Database`** annotation includes three attributes: **`entities`**, **`version`**, and **`exportSchema`**.

`entities` specifies any classes—marked with `@Entity`—that define the tables you want Room to add to the database. For the Tasks app, this is the `Task` data class.

`version` is an `Int` that specifies the database version. In this case, it's 1, as this is the first version of the database.

Finally, `exportSchema` tells Room whether to export the database schema into a folder so that you can record its version history. Here, we're setting it to *false*.

**If you modify the database entity files, you need to update the version number.**

If you change the schema without updating the version number, the app will crash when you try to run it.

# Add properties for any DAO interfaces

Next, we need to specify any interfaces (marked with `@Dao`) that will be used for data access. This is done by adding a property for each interface.

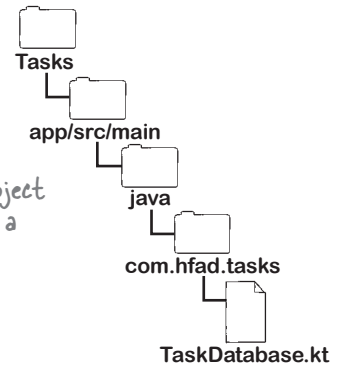
In the Tasks app, for example, we've defined a single DAO interface named `TaskDao`, so we need to add a new `taskDao` property to the `TaskDatabase` code like this:

```
...
@Database(entities = [Task::class], version = 1, exportSchema = false)
abstract class TaskDatabase : RoomDatabase() {
    abstract val taskDao: TaskDao
}
```

This tells Room that you want to use the data access methods specified in TaskDao.

## Create and return an instance of the database

The final thing we need is a `getInstance()` method that will create the database and return an instance of it. The code looks like this:




```
...
abstract class TaskDatabase : RoomDatabase() {
    ...
    companion object {
        @Volatile
        private var INSTANCE: TaskDatabase? = null

        fun getInstance(context: Context): TaskDatabase {
            synchronized(this) {
                var instance = INSTANCE
                if (instance == null) {
                    instance = Room.databaseBuilder(
                        context.applicationContext,
                        TaskDatabase::class.java,
                        "tasks_database"
                    ).build()
                    INSTANCE = instance
                }
                return instance
            }
        }
    }
}
```

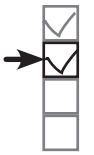
We're putting getInstance() in a companion object so that we can call it without first creating a TaskDatabase. You'll see why later on.

This method is used to get an instance of the database. It will look nearly the same for every database you create.

If the database doesn't already exist, the getInstance() method goes and builds it.

Return an instance of TaskDatabase.

That's everything that we need for the `TaskDatabase` class. We'll show you the full code on the next page.



Setup
**Database**
Insert
Get tasks

# The full code for TaskDatabase.kt

Here's the full code for the `TaskDatabase` abstract class; update the code for *TaskDatabase.kt* so that it includes the changes shown (in bold):

```
package com.hfad.tasks

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase

@Database(entities = [Task::class], version = 1, exportSchema = false)
abstract class TaskDatabase : RoomDatabase() {
    abstract val taskDao: TaskDao

    companion object {
        @Volatile
        private var INSTANCE: TaskDatabase? = null

        fun getInstance(context: Context): TaskDatabase {
            synchronized(this) {
                var instance = INSTANCE
                if (instance == null) {
                    instance = Room.databaseBuilder(
                        context.applicationContext,
                        TaskDatabase::class.java,
                        "tasks_database"
                    ).build()
                    INSTANCE = instance
                }
                return instance
            }
        }
    }
}
```

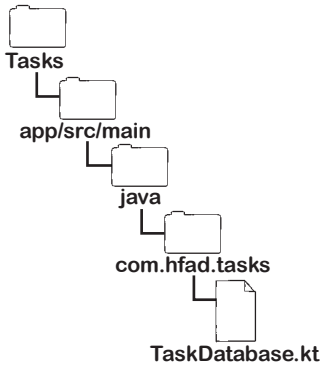
Import these classes.

Add this annotation.

Define a property for TaskDao.

Define a companion object so that we can call the getInstance() method using TaskDatabase.getInstance().

This returns an instance of TaskDatabase. It builds the database if it doesn't yet exist.




We've now written all the database code needed by the Tasks app. Before we build the rest of the app, try the following exercise.

# Sharpen your pencil

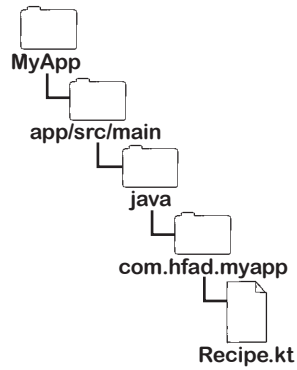
A table named `recipe_table` needs to have the following columns:

| recipeId | name | ingredients | method |
|---|---|---|---|
| | | | |

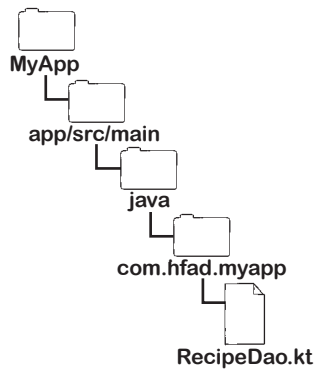
recipeId is the primary key, and should be autogenerated.

See if you can annotate the `Recipe` data class and `RecipeDao` interface below so that Room creates the specified table, along with methods to insert, update, and delete its data.

```
data class Recipe(
    var recipeId: Long = 0L,
    var recipeName: String = "",
    var recipeIngredients: String = "",
    var recipeMethod: String = ""
)
```




```
interface RecipeDao {
    fun insert(recipe: Recipe)
    fun update(recipe: Recipe)
    fun delete(recipe: Recipe)
}
```



# Sharpen your pencil Solution

A table named `recipe_table` needs to have the following columns:

| recipeId | name | ingredients | method |
|---|---|---|---|
| | | | |

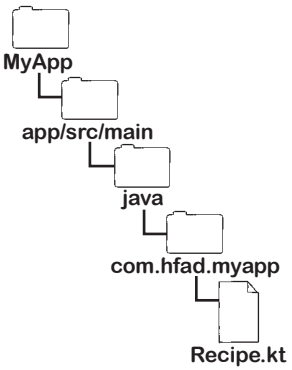
See if you can annotate the `Recipe` data class and `RecipeDao` interface below so that Room creates the specified table, along with methods to insert, update, and delete its data.

Name the table.

```
@Entity(tableName = "recipe_table")
data class Recipe(
    @PrimaryKey(autoGenerate = true)
    var recipeId: Long = 0L,

    @ColumnInfo(name = "name")
    var recipeName: String = "",

    @ColumnInfo(name = "ingredients")
    var recipeIngredients: String = "",

    @ColumnInfo(name = "method")
    var recipeMethod: String = ""
)
```

Make recipeId the primary key, and autogenerate its values.

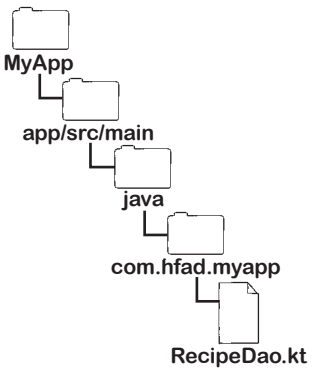
Column names for these properties




```
@Dao
interface RecipeDao {
    @Insert
    fun insert(recipe: Recipe)

    @Update
    fun update(recipe: Recipe)

    @Delete
    fun delete(recipe: Recipe)
}
```

Mark RecipeDao as a DAO.

Annotate these methods so they can be used to insert, update, and delete records.

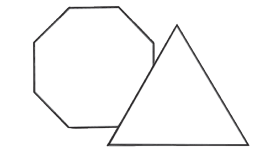


# MVVM revisited

Earlier in the chapter, we said we were going to structure the Tasks app using the MVVM (or Model-View-ViewModel) architecture pattern. Here's a reminder of what this kind of structure looks like:

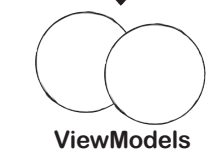
## View

This is the code that's responsible for the UI, so the app's activity, fragment, and layout code.




## ViewModel

View model objects are responsible for business logic, and each view's data.




## Model

This is the app's database code, including its entities and DAOs.

## We've completed all the Model code...

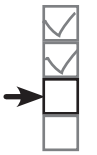
So far, we've written all the database code that the app needs by creating its entity, DAO, and database definition files (`Task`, `TaskDao`, and `TaskDatabase`). Writing all the database code means that we've now completed the Model part of the app's architecture.

## ...so let's move on to the ViewModel

The next thing we'll work on is the ViewModel part. To do this, we'll create a view model—named `TasksViewModel`—which will hold `TasksFragment`'s business logic. The view model will include methods that will use `TaskDao` to insert records into the database.

We want to display a list of the tasks held in the database as well, but for now, we're going to focus on inserting records.

Let's go ahead and create `TasksViewModel`.



Setup
Database
**Insert**
Get tasks

# Create TasksViewModel

To create `TasksViewModel`, highlight the *com.hfad.tasks* package in the *app/src/main/java* folder, then go to File→New→Kotlin Class/File. Name the file "TasksViewModel" and choose the Class option.

We're going to update the view model code so that `TasksFragment` can use it to insert new task records. To do this, the code needs three things:

1. **A reference to a TaskDao object**
   `TasksViewModel` will use this object to interact with the database, so we'll pass it to the view model in its constructor.

2. **A String property that holds the name of the new task**
   When the user enters a new task name, `TasksFragment` will update this property with its value.

3. **An addTask() method, which TasksFragment will call**
   This method will create a new `Task` object, set its name, and insert it into the database by calling `TaskDao`'s `insert()` method.

The basic view model code for these three things looks like this; update the code for *TasksViewModel.kt* so that it matches the code below:

```
package com.hfad.tasks

import androidx.lifecycle.ViewModel

class TasksViewModel(val dao: TaskDao) : ViewModel() {

    var newTaskName = ""

    fun addTask() {
        val task = Task()
        task.taskName = newTaskName
        dao.insert(task)
    }
}
```

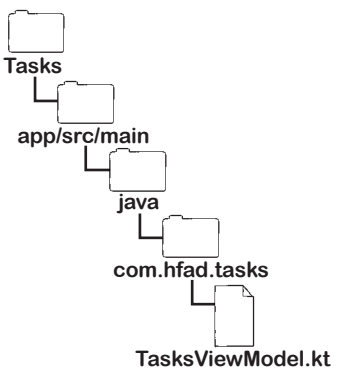
Import this class.

Make TasksViewModel extend ViewModel.

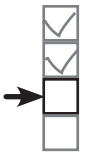
Pass a TaskDao object to TasksViewModel.

This is for the task name.

This method creates a Task object, and uses the TaskDao's insert() method to add it to the database.




Before `TasksFragment` can call the `addTask()` method, however, there's one more change we need to make.



- Setup
- Database
- **Insert**
- Get tasks

# Database operations can run in slooooow-moooo

Some tasks in Androidville, such as inserting records into a database, are potentially quite time-consuming. For this reason, the Room persistence library insists that ***any data access operations must be performed on a background thread*** so they don't block Android's main thread and hold up the UI.

Technically, there's a setting you can use to override this, but it's way better to run tasks such as this on a background thread so they don't hold up the rest of the app.

As we're developing Android apps using Kotlin, we're going to use ***coroutines*** to make sure all of `TaskDao`'s data access methods run in the background.

## We'll use coroutines to run data access code in the background

As you may already know, a coroutine is like a lightweight thread that lets you run multiple pieces of code asynchronously. Using coroutines means that you can launch a background job—such as inserting records into a database—without the rest of the code having to wait around for it to complete. This gives your user a more fluid experience, so it's not like trying to watch YouTube on Thanksgiving.

**A coroutine is a piece of suspendable code that can be run in the background.**

Changing your data access code to use coroutines is pretty straightforward. You simply have to make these two changes:

1. **Mark each of the DAO's data access methods with suspend.** This turns each method into a coroutine that's run in the background and can be suspended, for example:

This turns the insert() method into a coroutine.

```
@Insert
suspend fun insert(task: Task)
```

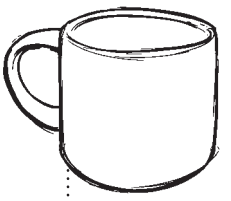
2. **Launch the DAO's coroutines in the background.** To call `TaskDao`'s `insert()` method from `TasksViewModel`, for example, you'd use the code:

This launches the coroutine in the same scope as the view model.

```
viewModelScope.launch {
    ...
    dao.insert(task)
}
```

***These changes are needed for all data access methods except for ones that return live data.*** Room already uses a background thread for methods that return live data objects, which means you don't need to make any extra changes to your code.

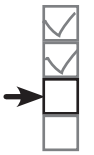
Let's update the code for `TaskDao` and `TasksViewModel` so they use coroutines.



### Relax

**Don't worry if you don't feel confident using coroutines.**

The information we've given here is everything you need to know to run data access methods in the background.

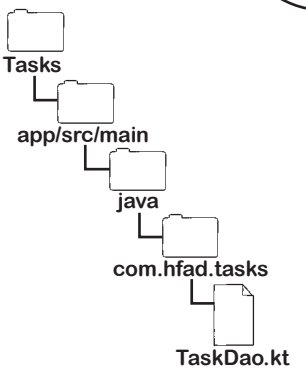


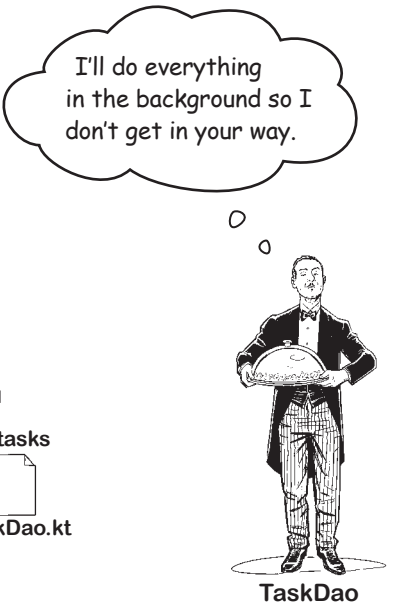
Setup
Database
**Insert**
Get tasks

# 1. Mark TaskDao's methods with suspend

The first thing we need to do is mark all of `TaskDao`'s data access methods that don't use live data with `suspend`. This means that we need to apply this change to all of its methods except for `get()` and `getAll()`.

Here's the full code for *TaskDao.kt*; update the code so that it includes the changes shown here (in bold):




```
package com.hfad.tasks

import androidx.lifecycle.LiveData
import androidx.room.Dao
import androidx.room.Delete
import androidx.room.Insert
import androidx.room.Query
import androidx.room.Update

@Dao
interface TaskDao {
    @Insert
    suspend fun insert(task: Task)

    @Update
    suspend fun update(task: Task)

    @Delete
    suspend fun delete(task: Task)

    @Query("SELECT * FROM task_table WHERE taskId = :key")
    fun get(key: Long): LiveData<Task>

    @Query("SELECT * FROM task_table ORDER BY taskId DESC")
    fun getAll(): LiveData<List<Task>>
}
```

Mark the insert(), update(), and delete() methods with suspend.

The get() and getAll() methods use live data, which runs in the background. This means we can keep these methods exactly as they are.

**Marking methods with suspend turns each one into a suspendable coroutine.**

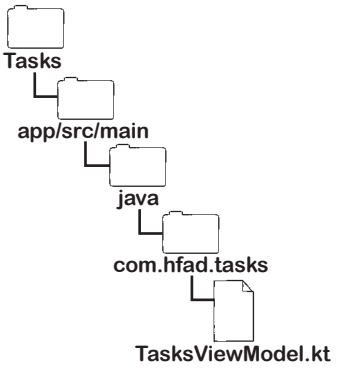
That's all the code that we need to turn `TaskDao`'s methods into coroutines that can run in the background.

# 2. Launch the insert() method in the background

We next need to update `TasksViewModel`'s `addTask()` method so that it launches `TaskDao`'s `insert()` method as a coroutine. The code to do this is shown below, so update *TasksViewModel.kt* to include the changes (in bold):

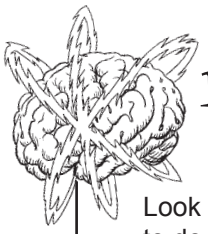
```
package com.hfad.tasks

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch

class TasksViewModel(val dao: TaskDao) : ViewModel() {
    var newTaskName = ""

    fun addTask() {
        viewModelScope.launch {
            val task = Task()
            task.taskName = newTaskName
            dao.insert(task)
        }
    }
}
```

Launch the coroutine in the same scope as the view model.




This change means that each time the `addTask()` method gets called, it will use `TaskDao`'s `insert()` method (a coroutine) to insert records in the background.

We've now written all the code we need for the Task app's view model. Next, we'll add a `TasksViewModel` object to `TasksFragment` so that it can access the view model's properties and methods, and let the user insert task records.



## Brain Power

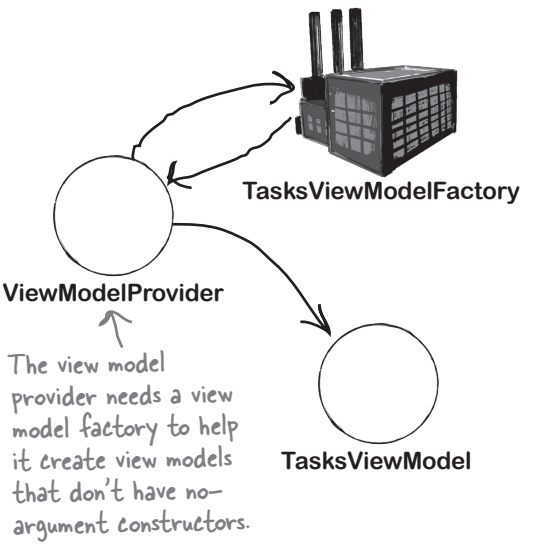
Look at the code for `TasksViewModel`. What do you think we need to do next in order to add an instance of it to `TasksFragment`?

# TasksViewModel needs a view model factory

As you learned in Chapter 11, you add a view model to fragment code by asking the view model provider to provide one. The view model provider will return the fragment's current view model object if one exists, or create a new one if it doesn't.

If the view model includes a no-argument constructor, the view model provider can create an instance of it with no extra help. But if the constructor has arguments, it needs the assistance of a view model factory.

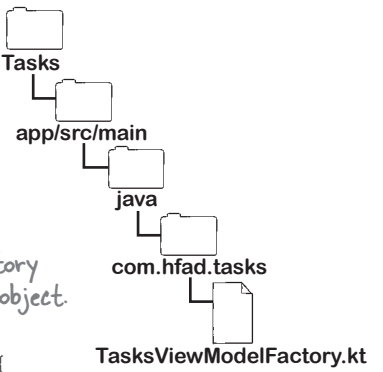
In the Tasks app, we need the view model provider to get a `TasksViewModel` object. Since `TasksViewModel`'s constructor requires a `TaskDao` argument, we must first define a `TasksViewModelFactory` class.




## Create TasksViewModelFactory

To create the factory, highlight the *com.hfad.tasks* package in the *app/src/main/java* folder, then go to File→New→Kotlin Class/File. Name the file "TasksViewModelFactory" and choose the Class option.

The `TasksViewModelFactory` code is almost identical to the view model factory code you wrote in Chapter 11, so update *TasksViewModelFactory.kt* so that it looks like this:

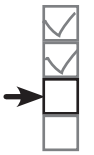



```
package com.hfad.tasks

import androidx.lifecycle.ViewModel
import androidx.lifecycle.ViewModelProvider

class TasksViewModelFactory(private val dao: TaskDao)
                            : ViewModelProvider.Factory {

    override fun <T: ViewModel?> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(TasksViewModel::class.java)) {
            return TasksViewModel(dao) as T
        }
        throw IllegalArgumentException("Unknown ViewModel")
    }
}
```

TasksViewModelFactory accepts a TaskDao object.

Use the TaskDao object to create a TasksViewModel object.

Now that we've written the view model factory, let's use it to add a `TasksViewModel` object to `TasksFragment`.




# TasksViewModelFactory needs a TaskDao

To add a `TasksViewModel` to `TasksFragment`, we need to create a `TasksViewModelFactory` object and pass it to the view model provider. The provider will then use the factory to create the view model.

But there's a problem: `TasksViewModelFactory`'s constructor requires a `TaskDao` argument, so we need to get one of these before we can create the `TasksViewModelFactory` object. `TaskDao`, however, is an interface, not a concrete class, so how can we get a `TaskDao` object?

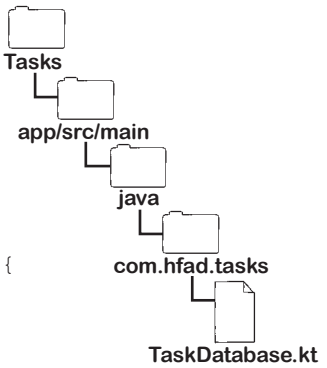
## The TaskDatabase code has a TaskDao property

When we wrote the code for `TaskDatabase`, we included two key things: a `TaskDao` property named `taskDao`, and a `getInstance()` method to return an instance of the database. Here's a reminder of what that code looks like:

```
abstract class TaskDatabase : RoomDatabase() {
    abstract val taskDao: TaskDao

    companion object {
        ...
        fun getInstance(context: Context): TaskDatabase {
            ...
        }
    }
}
```

Here's the taskDao property.

This returns an instance of the database.




To get a reference to a `TaskDao` object in the `TasksFragment` code, we can therefore call `TaskDatabase`'s `getInstance()` method, and access its `taskDao` property. The code to do so looks like this:

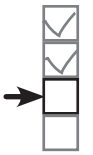
requireNotNull() is a Kotlin function that throws an IllegalArgumentException if its argument is null. We're using it here to be extra careful, and make sure that application isn't null.

```
val application = requireNotNull(this.activity).application
val dao = TaskDatabase.getInstance(application).taskDao
```

We'll add this code to TasksFragment on the next page.

The above code gets a reference to the current application, builds the database if it doesn't already exist, and returns an instance of it. It then assigns its `TaskDao` object to a local variable named `dao`.

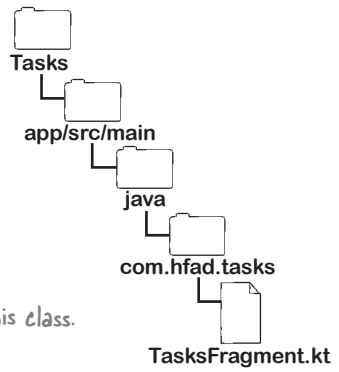
Now that we know how to get a reference to a `TaskDao` object, we can update the `TasksFragment` code to create a `TasksViewModelFactory` object, which we'll use to get a `TasksViewModel`. Let's see what the code looks like.




# The updated code for TasksFragment.kt

The updated `TasksViewModel` code is shown below. As you can see, it now includes code to get a `TaskDao` object and create a `TasksViewModelFactory`. The code then passes the factory to the view model provider, which uses it to get an instance of `TasksViewModel`.

Update the code in *TasksFragment.kt* so that it includes these changes (in bold):




```
package com.hfad.tasks

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.tasks.databinding.FragmentTasksBinding
import androidx.lifecycle.ViewModelProvider

class TasksFragment : Fragment() {
    private var _binding: FragmentTasksBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentTasksBinding.inflate(inflater, container, false)
        val view = binding.root

        val application = requireNotNull(this.activity).application
        val dao = TaskDatabase.getInstance(application).taskDao
        val viewModelFactory = TasksViewModelFactory(dao)
        val viewModel = ViewModelProvider(
                this, viewModelFactory).get(TasksViewModel::class.java)

        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Import this class.

Build the database (if it doesn't already exist) and get a reference to the taskDao property.

Get the view model.

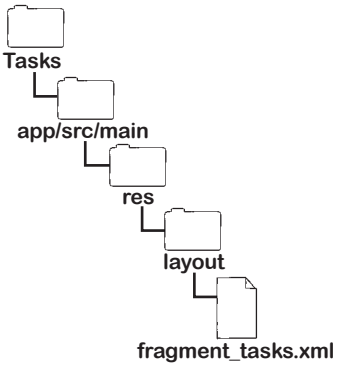
# TasksFragment can use data binding

Now that we've added a `TasksViewModel` object to `TasksFragment`, we can make the fragment use its properties and methods to insert a record into the database. We'll do this using data binding, which will give the layout direct access to the view model's properties and methods.

To set up data binding, we first need to add a data binding variable to the fragment's layout, so we'll add the code shown here (in bold) to the `<data>` section of *fragment_tasks.xml*:

```
<layout
    ...>
    <data>
        <variable
            name="viewModel"
            type="com.hfad.tasks.TasksViewModel" />
    </data>
    ...
</layout>
```

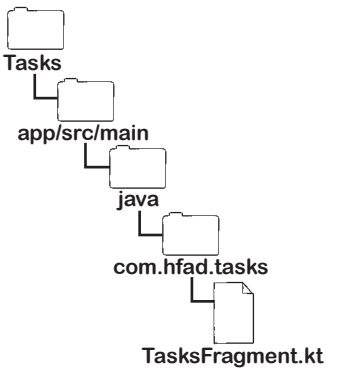
This defines a data binding variable named viewModel whose type is TasksViewModel.



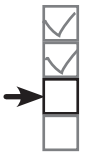

We'll then assign the fragment's `viewModel` property to the data binding variable by adding the line shown below (in bold) to `TasksFragment`'s `onCreateView()` method:

```
...
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        ...
        val viewModel = ViewModelProvider(
                this, viewModelFactory).get(TasksViewModel::class.java)
        binding.viewModel = viewModel

        return view
    }
...
```

This sets the data binding so that the layout can use it to access the view model's properties and methods.




We'll show you the full code for both files a few pages ahead. First, let's update *fragment_tasks.xml* so that it uses its `viewModel` variable to insert records into the database.



Setup
Database
**Insert**
Get tasks

# We'll use data binding to insert a record

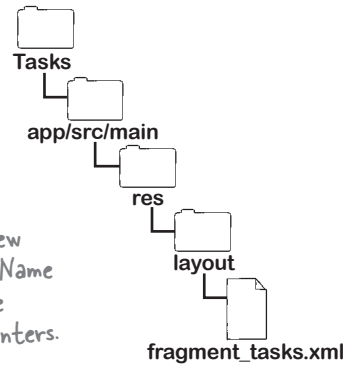
To insert a new task record into the database, we need to do two things: set `TasksViewModel`'s `newTaskName` property to the name of the new task, and call its `addTask()` method. We can do both of these in `TasksFragment`'s layout with data binding.

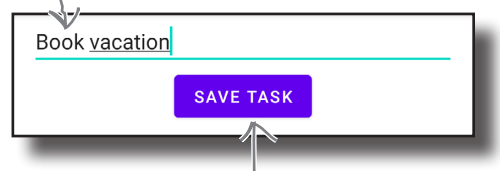
## Set TasksViewModel's newTaskName property

To set the view model's `newTaskName` property, we'll bind it to the `task_name` edit text in the fragment's layout. The code to do so looks like this:

```
<EditText
    android:id="@+id/task_name"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:inputType="text"
    android:hint="Enter a task name"
    android:text="@={viewModel.newTaskName}" />
```

This sets the view model's newTaskName property to the text the user enters.




The new task will be inserted when the user clicks this button.

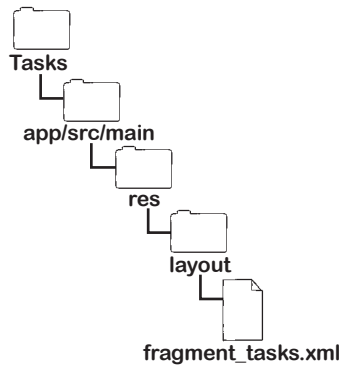
Notice that we're using **`@=`** to bind the property to the edit text instead of `@`. The `@=` means that the edit text can update the property it's bound to: in this case `newTaskName`.

## Call TasksViewModel's addTask() method

To insert the task, we'll use data binding to make the layout's Save Task button call the view model's `addTask()` method when clicked. You're already familiar with how to do this, so here's the code:

```
<Button
    android:id="@+id/save_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:text="Save Task"
    android:onClick="@{() -> viewModel.addTask()}" />
```

This makes the button call the view model's addTask() method when it's clicked.



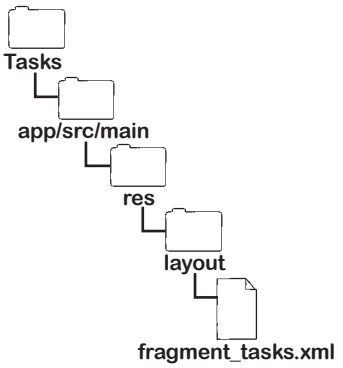

Those are all the code changes we need to make to *fragment_tasks.xml* for it to insert a record into the database. Let's see what the full code looks like.

# The full fragment_tasks.xml code

Here's the full code for `TasksFragment`'s layout; make sure that the code in *fragment_tasks.xml* includes the changes shown (in bold):

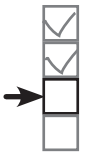
```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".TasksFragment">

    <data>
        <variable
            name="viewModel"
            type="com.hfad.tasks.TasksViewModel" />
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <EditText
            android:id="@+id/task_name"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:inputType="text"
            android:hint="Enter a task name"
            android:text="@={viewModel.newTaskName}" />

        <Button
            android:id="@+id/save_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:text="Save Task"
            android:onClick="@{() -> viewModel.addTask()}" />

        <TextView
            android:id="@+id/tasks"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
    </LinearLayout>
</layout>
```

Add the data binding variable.



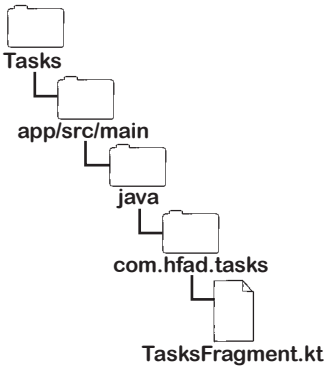

Set the view model's newTaskName property to the name of the task in the EditText.

Call the view model's addTask() method when the user clicks on the button to save the task.




# The full TasksFragment.kt code

Before running the app, we also need to make sure that `TasksFragment` includes all the code that's needed to set the layout's data binding variable. Here's the full code; update *TasksFragment.kt* (if you haven't already done so) to include the changes below (in bold):




```
package com.hfad.tasks

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.lifecycle.ViewModelProvider
import com.hfad.tasks.databinding.FragmentTasksBinding

class TasksFragment : Fragment() {
    private var _binding: FragmentTasksBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentTasksBinding.inflate(inflater, container, false)
        val view = binding.root

        val application = requireNotNull(this.activity).application
        val dao = TaskDatabase.getInstance(application).taskDao
        val viewModelFactory = TasksViewModelFactory(dao)
        val viewModel = ViewModelProvider(
                this, viewModelFactory).get(TasksViewModel::class.java)
        binding.viewModel = viewModel

        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Import this class.

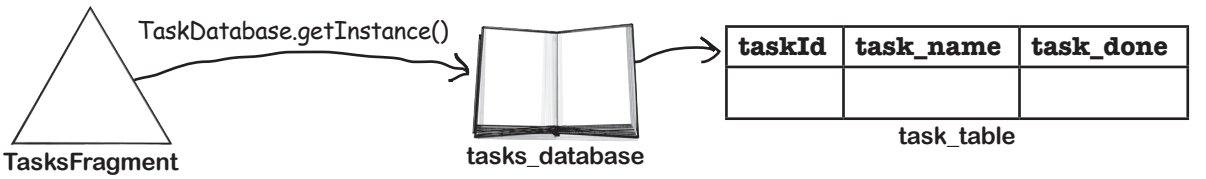
Add these lines.

Let's go through what happens when the code runs.

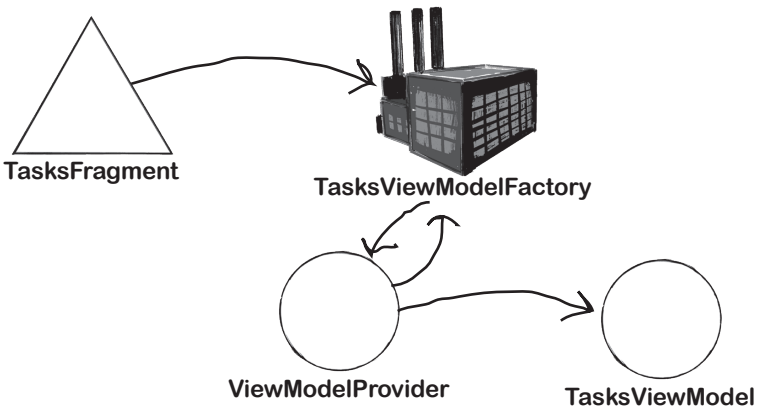
# What happens when the code runs

The following things happen when the app runs:

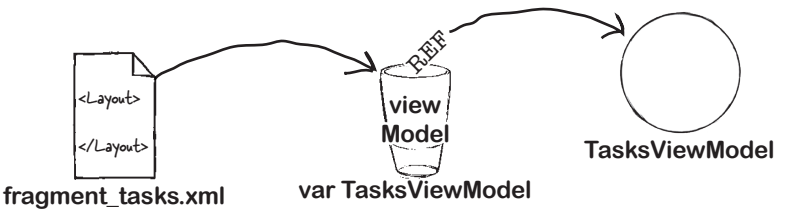
1. **TasksFragment calls TaskDatabase.getInstance(), which builds the database if it doesn't already exist.**
It names the database `tasks_database`, and uses the `Task` data class to create a table named `task_table`. It then returns an instance of the database.



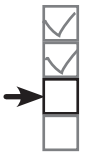

2. **TasksFragment gets the database instance's TaskDao object, and uses it to create a TasksViewModelFactory.**
The view model provider uses the newly created `TasksViewModelFactory` object to create a `TasksViewModel`.




3. **TasksFragment sets the layout's viewModel data binding variable to the TasksViewModel object.**

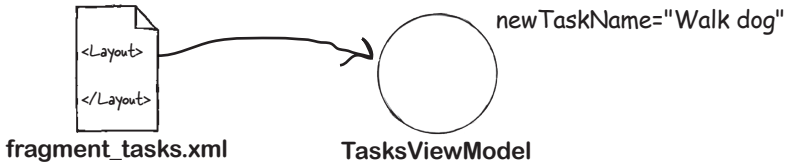


# The story continues

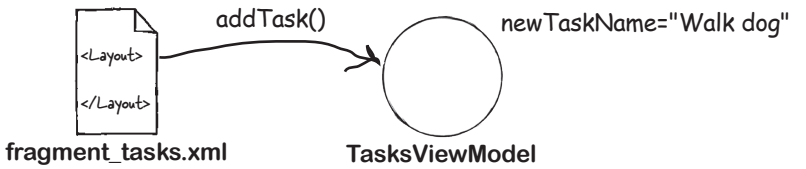


Setup
Database
**Insert**
Get tasks

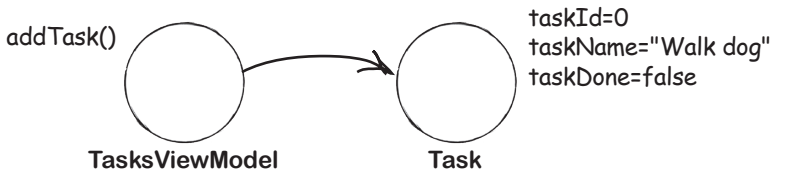
4 **The user enters a task name in the layout's edit text view.**
The layout uses data binding to set the view model's `newTaskName` property to this value.



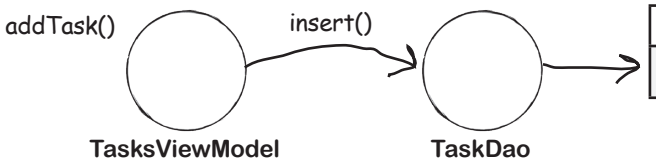

5 **The user clicks on the Save Task button.**
This uses data binding to call the `TaskViewModel` object's `addTask()` method.




6 **The addTask() method creates a new Task object, and sets its taskName property to the value of newTaskName.**

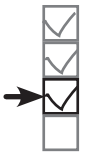



7 **The addTask() method calls TaskDao's insert() method.**
This inserts a record for the `Task` object's data into the database table. The table autogenerates a value for the `taskId` primary key.




| taskId | task_name | task_done |
|---|---|---|
| 1 | Walk dog | false |

Let's take the app for a test drive.

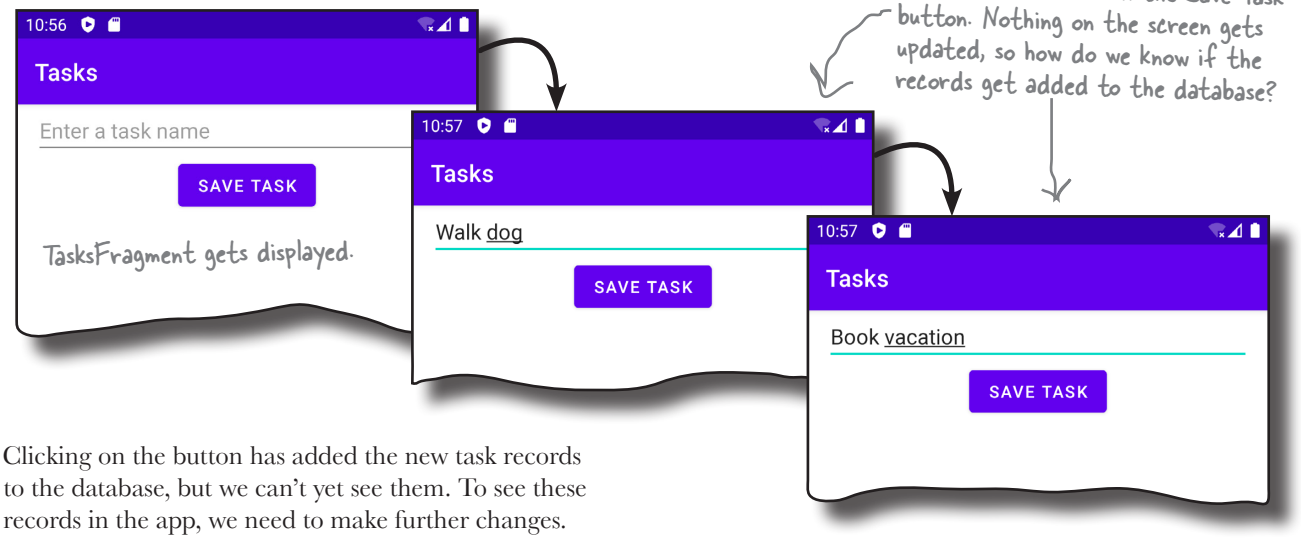


- Setup
- Database
- **Insert**
- Get tasks

# Test Drive

When we run the app, `TasksFragment` is displayed in `MainActivity`. When we try entering two new tasks, nothing seems to happen when we click on the Save Task button.

For each of these tasks, we enter a task name and click on the Save Task button. Nothing on the screen gets updated, so how do we know if the records get added to the database?




Clicking on the button has added the new task records to the database, but we can't yet see them. To see these records in the app, we need to make further changes.

## there are no Dumb Questions

**Q: When I tried to enter a task, the app crashed. Any idea why?**

A: This can happen if you update the database schema (the entities that the database contains) without updating the database version number. Room keeps track of the schema for each version of the database. If it spots any discrepancies, it adds an error message to the app's logcat, and the app crashes.

**Q: I see. How do I fix this problem?**

A: The simplest fix is to update the database version number in the file that defines the database (in this case *TaskDatabase.kt*).

**Q: How do I see the app's logcat?**

A: You should see an option for it in Android Studio. Any error messages get logged here, so it's a good place to look if you encounter any errors.

**Q: Are there any tools I can use to inspect the database and see what records and tables it contains?**

A: Android Studio includes a helpful Database Inspector you can use for this purpose. You can find out more here:

*https://developer.android.com/studio/inspect/database*

**Q: What other database features can I use with Room?**

A: You can define relationships between tables, create views, migrate data when you want to upgrade the database, and more. You can find more details of these features here:

*https://developer.android.com/training/data-storage/room*

**Q: In your layout code, you used "@=" to bind the edit text to the view model's property instead of "@". Why?**

A: Because we wanted the edit text to update the property's value, and not just read its value. Using `@=` instead of `@` enables this.

A bunch of components, in full costume, are playing a party game, "Who am I?" They'll give you a clue—you try to guess who they are based on what they say. Assume they always tell the truth about themselves. Fill in the blanks to the right to identify the attendees. Also, for each attendee, write down whether the component is part of the Model, View, or ViewModel layer of the app's architecture.

**Tonight's attendees:**

**Activity** **Fragment** **Layout** **ViewModel**
**Database** **Entity** **DAO**

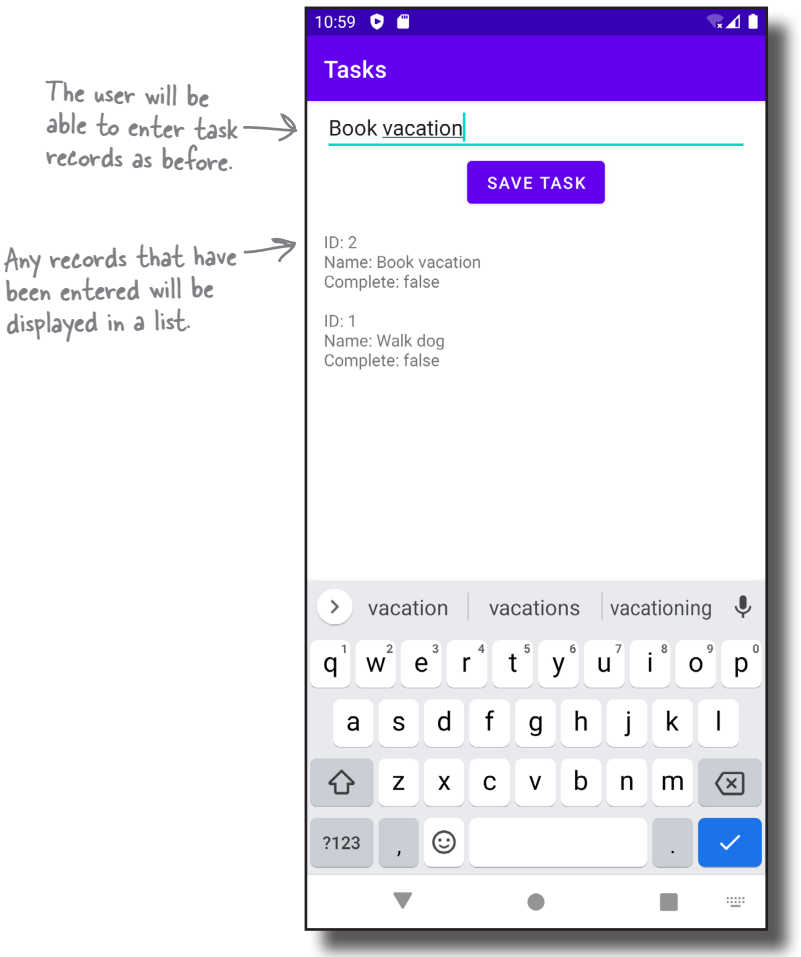
| | Name | Model, View, or ViewModel? |
|---|---|---|
| I take care of what the app's screens look like. | | |
| I persist data after the app's shut down. | | |
| I have a lifecycle, but I can't exist on my own. | | |
| I'm used for business logic. | | |
| Room uses me to create a table. | | |
| I have a lifecycle, and I'm a type of context. | | |
| I help you interact with the data in the database. | | |

→ Answers on page 619.

# TasksFragment needs to display records

You've now learned how to build an app that inserts records into a Room database. The next thing we'll do is update the app so that `TasksFragment` displays a list of all the records that have been inserted.

Here's what the new version of `TasksFragment` will look like:




To do this, we'll use `TaskDao`'s `getAll()` method to get all of the task records from the database. We'll then display them in `TasksFragment`'s text view by formatting them as a single `String`.

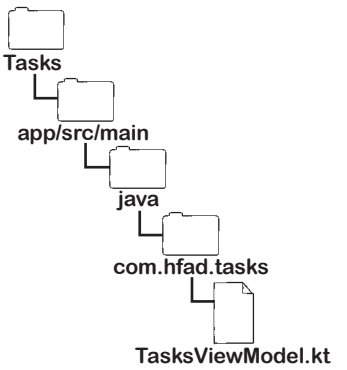
Let's get started.

# Use getAll() to get all tasks from the database

We'll begin by adding a property named `tasks` to `TasksViewModel`, which will hold a list of all the tasks in the database. We'll add the tasks to the property by setting it to `TaskDao`'s `getAll()` method like this:

```
class TasksViewModel(val dao: TaskDao) : ViewModel() {
    ...
    val tasks = dao.getAll()
    ...
}
```

This calls TaskDao's getAll() method, and assigns its return value to the tasks property.

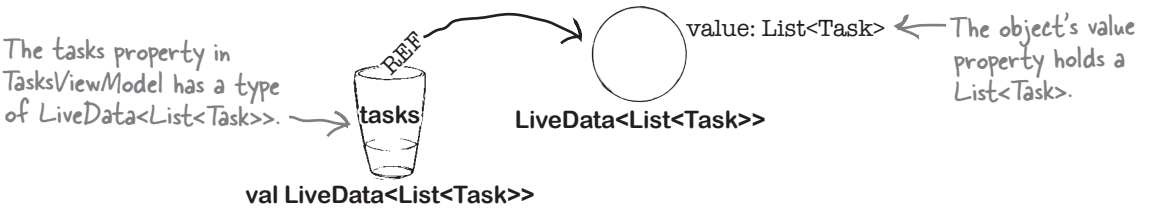



As you may remember, we defined the `getAll()` method in *TaskDao.kt* using the following code:

```
@Dao
interface TaskDao {
...
    @Query("SELECT * FROM task_table ORDER BY taskId DESC")
    fun getAll(): LiveData<List<Task>>
}
```

The getAll() method gets all records from the database. It returns a LiveData<List<Task>>.

As you can see, the method returns a `LiveData<List<Task>>` object: a live data list of `Task` objects. This means that `TasksViewModel`'s `tasks` property also has a type of `LiveData<List<Task>>`:




As the `tasks` property uses live data, it always includes the user's latest changes. If a new task is added to `task_table`, for example, the `tasks` property's value is automatically updated to include the new record.

Using live data in this way is good news for the Tasks app because we can use it to display a list of tasks in `TasksFragment` that's always up-to-date. Before we can do this, however, there's one more thing we need to learn about.

# A LiveData<List<Task>> is a more complex type

When we introduced you to data binding, we showed you how to bind a view to a `String` or number, including live data values. We were able to do this because `String`s and numbers are simple objects that can easily be displayed in a text view.

This time, however, the situation is different. The `tasks` property's type is `LiveData<List<Task>>`, which is much more complex than, say, `LiveData<String>`. Because of this, we can't simply bind the layout's text view to the `tasks` property directly because the text view wouldn't know how to display it.

So what's the solution?

## Use Transformations.map() to transform live data objects

Before we can use data binding to display the user's tasks, we first need to transform the `LiveData<List<Task>>` into something simpler that the text view knows how to display. To do this, we'll create a new property named `tasksString`, which will hold a `LiveData<String>` version of the `tasks` property.

We'll create the `LiveData<String>` using the **`Transformations.map()`** method. This method accepts a `LiveData` argument, and a lambda that specifies how the live data object should be transformed. It then returns a new `LiveData` object.

To transform the `tasks` property into a `LiveData<String>`, for example, we could use the code:

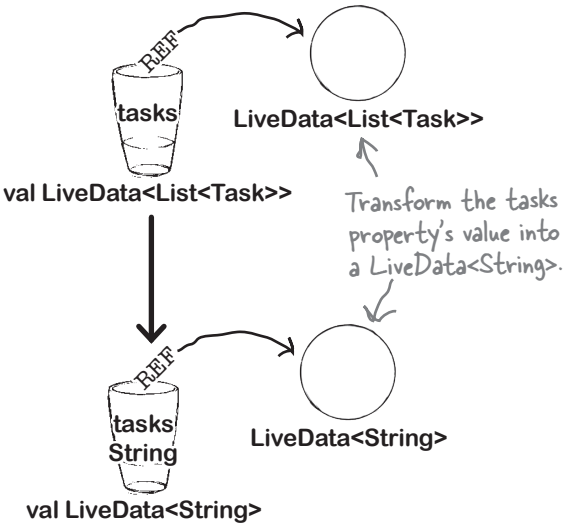
```
val tasksString = Transformations.map(tasks) {
    tasks -> formatTasks(tasks)
}
```

where `formatTasks()` is a method (which we need to write) that formats the `tasks` property's list of tasks as a `String`. The `Transformations.map()` method then wraps this `String` in a `LiveData` object, so that it returns a `LiveData<String>`.

The `Transformations.map()` method observes the `LiveData` object that's passed to it, and executes the lambda each time it's notified of changes. This means that in the above code, `tasksString` will ***automatically include any new task records the user enters***.

Over the next few pages, we're going to add the `tasksString` property to `TasksViewModel`, and then use data binding to display its value in `TasksFragment`'s layout. This approach means that we'll be able to display all of the user's tasks in a text view that always stays up-to-date.

**Transformations.map() observes a LiveData object and transforms it into another type of LiveData object.**

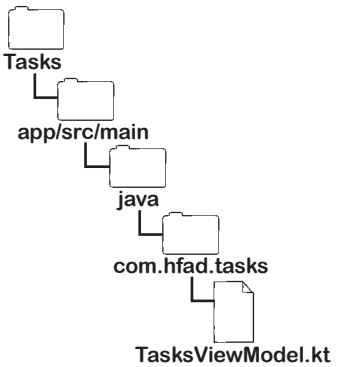


Setup
Database
Insert
**Get tasks**

# Let's update the TasksViewModel code

We'll begin by updating the `TasksViewModel` code so that it includes the new `tasks` and `tasksString` properties. We'll also add two methods—`formatTasks()` and `formatTask()`—to help us transform the `LiveData<List<Task>>` into a `LiveData<String>`.

Here's the full code for `TasksViewModel`, so update *TasksViewModel.kt* so that it includes the changes below (in bold):




```
package com.hfad.tasks

import androidx.lifecycle.Transformations
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch

class TasksViewModel(val dao: TaskDao) : ViewModel() {
    var newTaskName = ""

    private val tasks = dao.getAll()
    val tasksString = Transformations.map(tasks) {
        tasks -> formatTasks(tasks)
    }

    fun addTask() {
        viewModelScope.launch {
            val task = Task()
            task.taskName = newTaskName
            dao.insert(task)
        }
    }

    fun formatTasks(tasks: List<Task>): String {
        return tasks.fold("") {
                str, item -> str + '\n' + formatTask(item)
        }
    }

    fun formatTask(task: Task): String {
        var str = "ID: ${task.taskId}"
        str += '\n' + "Name: ${task.taskName}"
        str += '\n' + "Complete: ${task.taskDone}" + '\n'
        return str
    }
}
```

Import this class.

Get the records from the database.

Transforms the tasks into a LiveData<String>, which is assigned to tasksString. As this is a live data object, it will automatically get updated if the data changes.

This method uses pure Kotlin to format a list of tasks as a String. There's nothing Androidy about it.

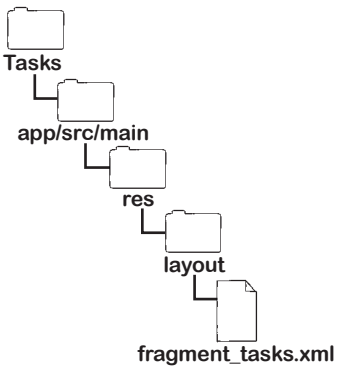
Format each task as a String.

# We'll bind the tasksString property to the layout's text view

Next, we'll use data binding to bind the `tasks` text view in `TasksFragment`'s layout to the `tasksString` property in `TasksViewModel`. Here's the code to do this; update *fragment_tasks.xml* to include this change (in bold):

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

    <data>
        <variable
            name="viewModel"
            type="com.hfad.tasks.TasksViewModel" />
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <EditText
            android:id="@+id/task_name"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:inputType="text"
            android:hint="Enter a task name"
            android:text="@={viewModel.newTaskName}" />

        <Button
            android:id="@+id/save_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:text="Save Task"
            android:onClick="@{() -> viewModel.addTask()}" />
```

You don't need to update any of the code on this page.

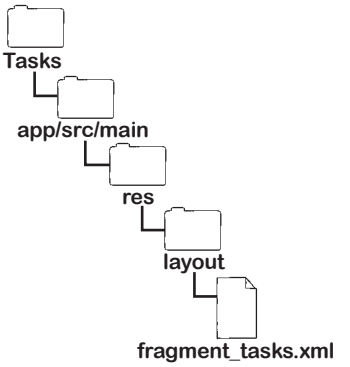



The code continues on the next page.

# The fragment_tasks.xml code continued...

```
        <TextView
            android:id="@+id/tasks"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@{viewModel.tasksString}" />
    </LinearLayout>
</layout>
```

Add this code.




That's the only change we need to make to *fragment_tasks.xml* in order for it to display a `String` of the user's tasks. Before we take the app for a test drive, however, there's just one more tweak we need to make.

## We need to make the layout respond to live data updates

As you already know, the `tasksString` property is a `LiveData<String>`, which means that it automatically includes any updates that are made to the records held in the database. If the user inserts a new record, for example, its data gets added to `tasksString`'s value.

As we're using data binding to display the value of the `tasksString` property in the `tasks` text view, we need to make sure that the layout gets notified whenever the `tasksString` property's value gets updated. Doing so means that the text view will display any new records as soon as they're inserted.

You've already learned how to make a layout respond to live data updates: by setting the layout's lifecycle owner in the fragment code like this:

```
binding.lifecycleOwner = viewLifecycleOwner
```

We therefore need to update the code for `TasksFragment` so that it includes this line. We'll show you the full code on the next page, and then go through what happens when the app runs.

### there are no Dumb Questions

**Q: What other uses does the `Transformations.map()` method have?**

A: As `Transformations.map()` gets reevaluated each time it receives new data, it's also useful for things like calculations.

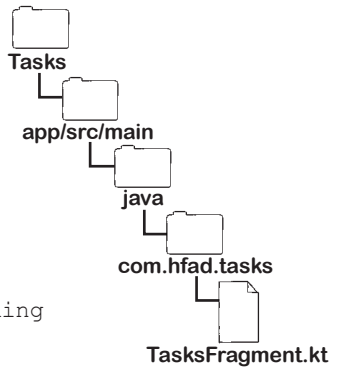
Suppose you have a live data `rate` property, and a calculation that uses it. You could, say, perform the calculation inside a call to `Transformations.map()` so that it's automatically recalculated each time the `rate` property changes.

**Q: Do you always display lists as formatted `Strings`?**

A: No. We did so here because it's a relatively simple way of displaying a bunch of records. You'll learn about a better approach in the next chapter.

# The full TasksFragment.kt code

Here's the full code for TasksFragment; update *TasksFragment.kt*: to include the extra line (in bold):




```
package com.hfad.tasks

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.lifecycle.ViewModelProvider
import com.hfad.tasks.databinding.FragmentTasksBinding

class TasksFragment : Fragment() {
    private var _binding: FragmentTasksBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentTasksBinding.inflate(inflater, container, false)
        val view = binding.root

        val application = requireNotNull(this.activity).application
        val dao = TaskDatabase.getInstance(application).taskDao
        val viewModelFactory = TasksViewModelFactory(dao)
        val viewModel = ViewModelProvider(
                this, viewModelFactory).get(TasksViewModel::class.java)
        binding.viewModel = viewModel
        binding.lifecycleOwner = viewLifecycleOwner

        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

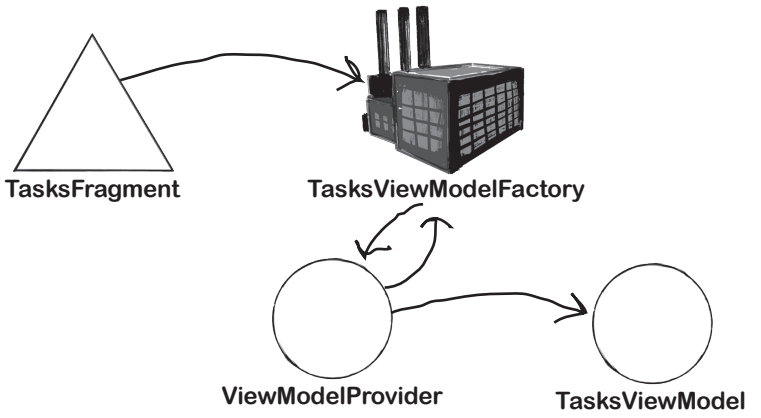
Set the layout's lifecycleOwner so that it can respond to live data updates.

Setup
Database
Insert
**Get tasks**

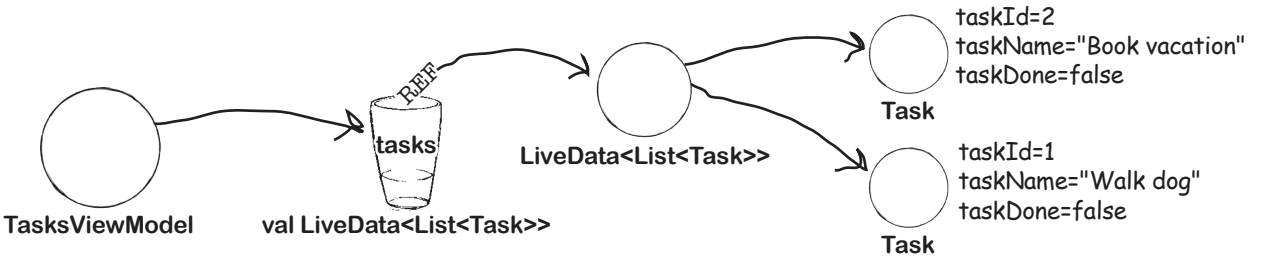
# What happens when the code runs

The following things happen when the app runs:

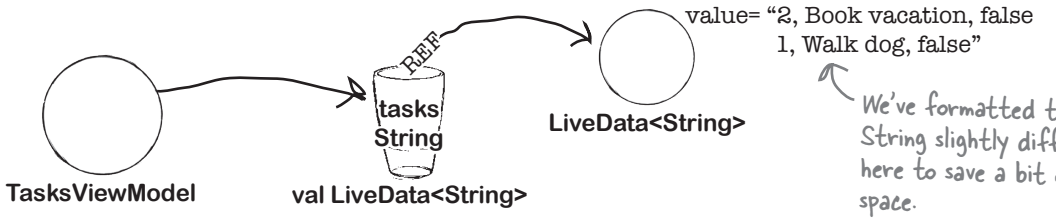
1. **TasksFragment creates a TasksViewModelFactory object, which the view model provider uses to create a TasksViewModel.**




2. **The TasksViewModel's tasks property is set to TaskDao's getAll() method, which returns a LiveData<List<Task>>.**
It holds a live data list of all the task records from the database.



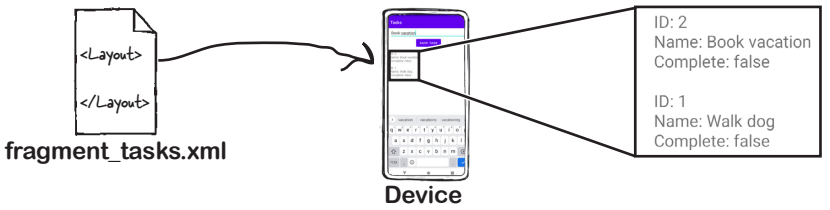

3. **TasksViewModel's tasksString property uses the Transformations.map() method to transform the tasks property's value into a LiveData<String>.**
It returns the `tasks` property's records formatted as a single `String`.




We've formatted the String slightly differently here to save a bit of space.

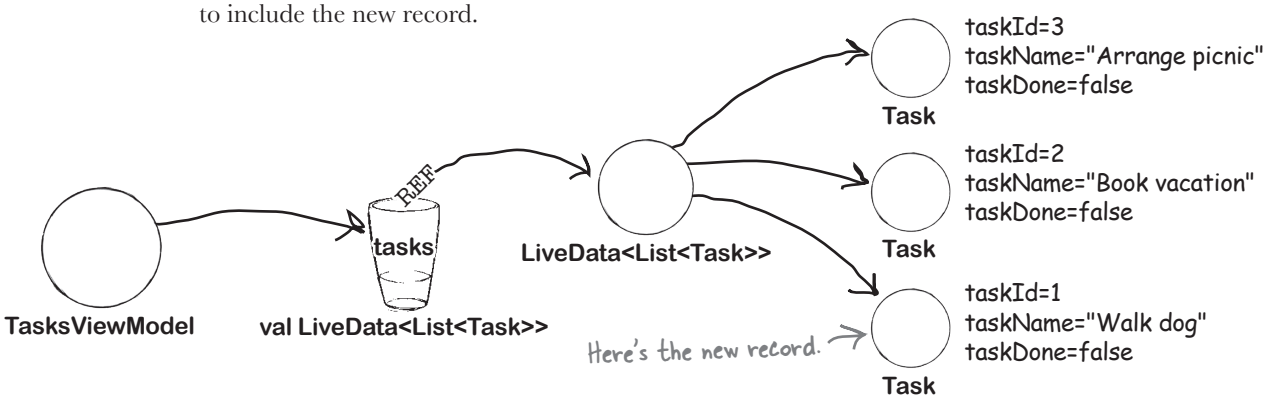
# The story continues

**4** **The layout's text view uses data binding to display tasksString's value.**



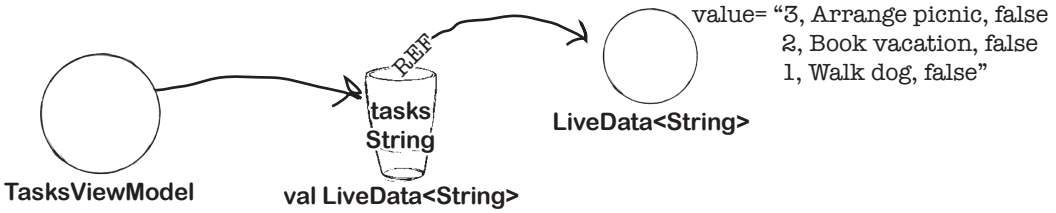

**5** **The user enters a new task record into the database.**

The `TasksViewModel`'s `tasks` property is automatically updated (using live data) to include the new record.



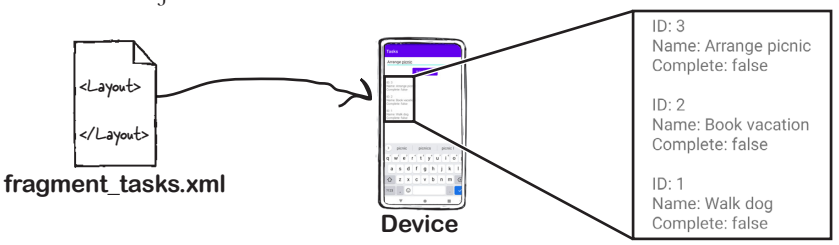

**6** **The tasksString property responds to the tasks property's update.**

As it uses live data, it automatically includes the new record in its `String`.




**7** **The layout responds to the tasksString property's update.**

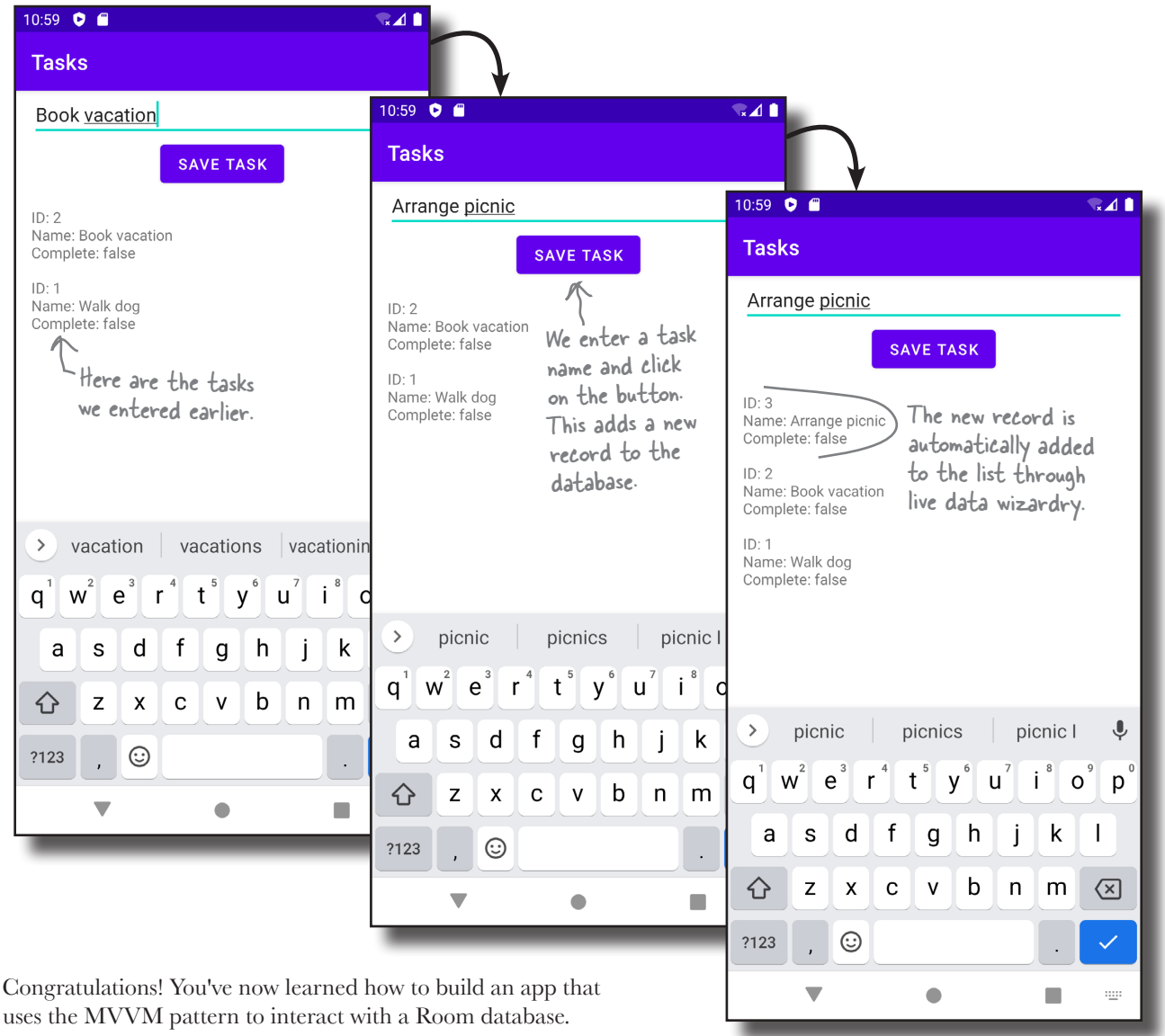
The record the user just entered is included in the text view.



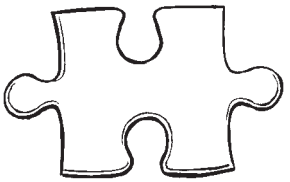
# Test Drive

When we run the app, `TasksFragment` is displayed as before, but this time, it shows us the tasks we entered earlier.

When we enter a new task, it gets added to the list of tasks as soon as we click on the Save Task button.




Congratulations! You've now learned how to build an app that uses the MVVM pattern to interact with a Room database. The user's records get saved to the database so that they persist when the app gets closed, and the app displays any new records as soon as they're inserted.



# Mixed Messages

Some view model code is listed below, but the `refresh()` method's code is missing. Your challenge is to match each candidate block of code (on the left) with the `result` property's final value if the candidate code were added to the `refresh()` method, and the method was called once. Not all the values will be used, and some values may be used more than once.

```
class MyViewModel() : ViewModel() {
    private val rate = MutableLiveData<Int>(1)
    private val a = MutableLiveData<Int>(0)
    private val b = MutableLiveData<Int>(0)
    val result = Transformations.map(rate) {
        rate -> rate * ((a.value ?: 0) + (b.value ?: 0))
    }

    fun refresh() {
        [                                    ]
    }
}
```

The candidate code goes here.

**Candidates:**

```
a.value = 2
```

```
rate.value = 2
b.value = 2
```

```
a.value = 2
rate.value = 2
```

```
a.value = 2
rate.value = 2
b.value = 2
```

```
a.value = 2
b.value = 2
rate.value = 2
```

**Possible result values:**

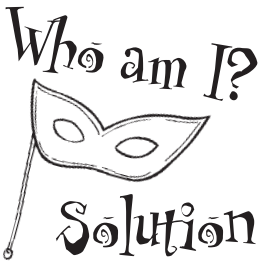
0

2

4

6

8

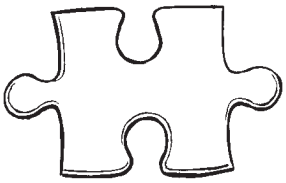



A bunch of components, in full costume, are playing a party game, "Who am I?" They'll give you a clue—you try to guess who they are based on what they say. Assume they always tell the truth about themselves. Fill in the blanks to the right to identify the attendees. Also, for each attendee, write down whether the component is part of the Model, View, or ViewModel layer of the app's architecture.

**Tonight's attendees:**

**Activity Fragment Layout ViewModel Database Entity DAO**

| | Name | Model, View, or ViewModel? |
|---|---|---|
| I take care of what the app's screens look like. | Layout | View |
| I persist data after the app's shut down. | Database | Model |
| I have a lifecycle, but I can't exist on my own. | Fragment | View |
| I'm used for business logic. | ViewModel | ViewModel |
| Room uses me to create a table. | Entity | Model |
| I have a lifecycle, and I'm a type of context. | Activity | View |
| I help others interact with the data in the database. | DAO | Model |



# Mixed Messages Solution

Some view model code is listed below, but the `refresh()` method's code is missing. Your challenge is to match each candidate block of code (on the left), with the `result` property's final value if the candidate code were added to the `refresh()` method, and the method was called once. Not all the values will be used, and some values may be used more than once.

```
class MyViewModel() : ViewModel() {
    private val rate = MutableLiveData<Int>(1)
    private val a = MutableLiveData<Int>(0)
    private val b = MutableLiveData<Int>(0)
    val result = Transformations.map(rate) {
        rate -> rate * ((a.value ?: 0) + (b.value ?: 0))
    }

    fun refresh() {
        [ The candidate code goes here. ]
    }
}
```

**Candidates:**

```
a.value = 2
```

```
rate.value = 2
b.value = 2
```

```
a.value = 2
rate.value = 2
```

```
a.value = 2
rate.value = 2
b.value = 2
```

```
a.value = 2
b.value = 2
rate.value = 2
```

**Possible result values:**

0

2

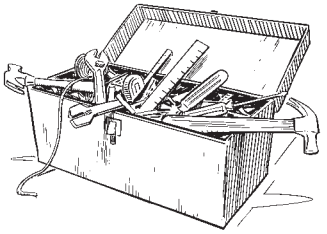
4

6

8

The value of result only gets updated when rate's value changes. If a or b change after rate is updated, their new values don't get picked up by the result property.

# Your Android Toolbox



**You've got Chapter 14 under your belt and now you've added the Room persistence library to your toolbox.**

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

## Bullet Points

- Room is a persistence library that sits on top of SQLite.
- Room apps are usually structured using the MVVM architectural design pattern. MVVM stands for Model-View-ViewModel.
- Define any database tables using an annotated data class. Use `@Entity` to tell Room to use the class to create a table. Use `@PrimaryKey` to define the primary key column. Use `@ColumnInfo` to specify column names.
- Define any data access methods in an annotated interface. Use `@Dao` to tell Room to use its methods for data access. Use `@Insert`, `@Update`, `@Delete`, and `@Query` to define the required data access.
- Define the database in an abstract class marked with `@Database`.
- If you update the database schema, you need to increment the database version number.
- Data access code needs to be run in the background. You can do this by turning any data access methods that don't return live data into coroutines.
- You can use the method `Transformations.map()` to transform live data objects. The method observes a live data object, executes a lambda when the object's value changes, and returns another live data object.

# 15 recycler views

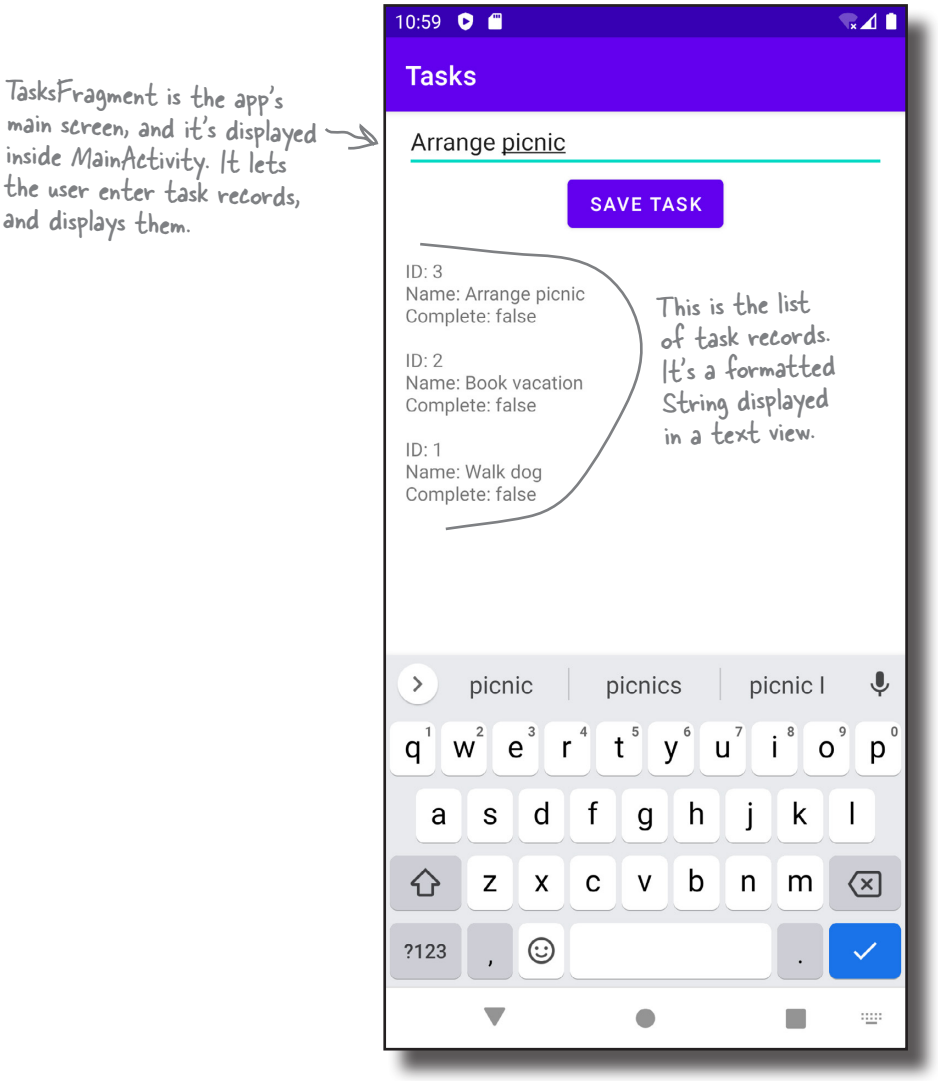
# Reduce, Reuse, Recycle

## 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**. Let's get recycling…

# What the Tasks app currently looks like

In the previous chapter, we built a Tasks app that lets the user enter task records into a Room database. The app displays the list of records as a formatted `String`, and it looks like this:

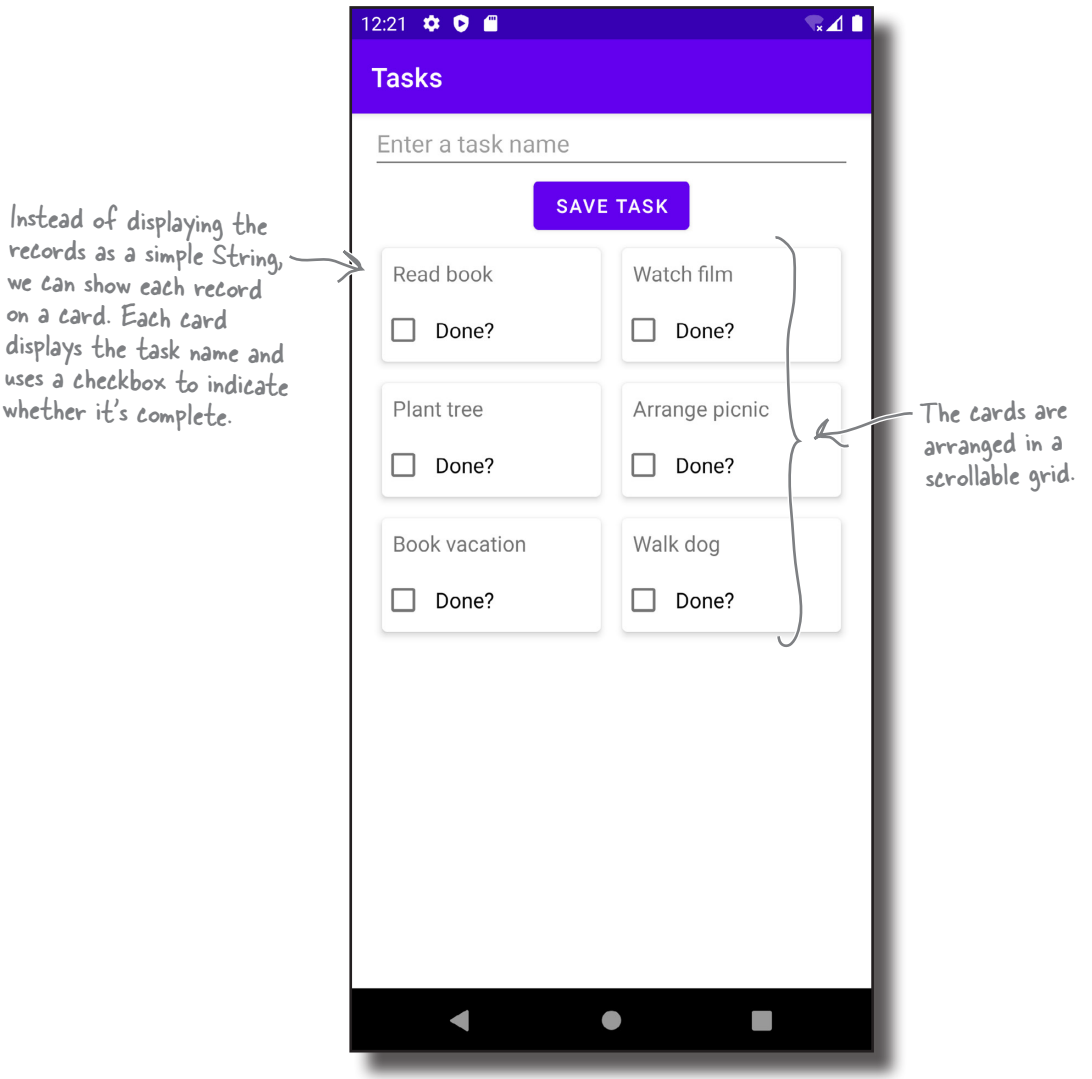



We decided to display the task records using a formatted `String` since it was a relatively quick and basic way of seeing the records that have been added to the database.

The list, however, looks a little bland. So how can we improve it?

# We can turn the list into a recycler view

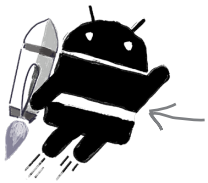
Instead of displaying the list of tasks as a formatted `String`, we can change it up so that it looks like this:




As you can see, each task record uses a text view and a checkbox for its data, instead of plain text. The items are also arranged on cards in a scrollable grid.

This kind of list is created using a **recycler view**. So what's that?

# Why use a recycler view?



Recycler views are another part of Android Jetpack.

A recycler view is a more advanced and flexible way of displaying a list of data than using a simple formatted `String`. It buys you the following benefits:

- **A rich UI for the list's items.**
  Each item is displayed in a layout, so you can use views such as text views, image views, and checkboxes to display its data.

- **A flexible way of positioning items.**
  Recycler views come with layout managers, which let you position views in a vertical or horizontal list, a grid, or a staggered grid where items have unequal heights.

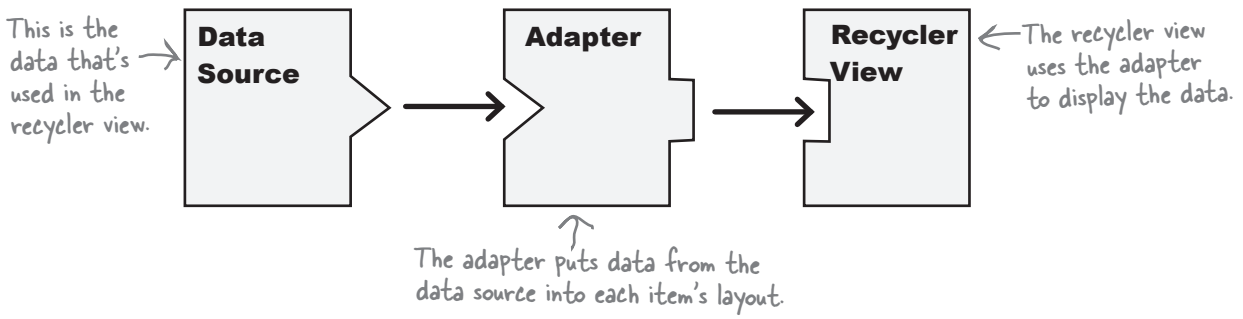
- **You can use it for navigation.**
  You can make items clickable so that they navigate to another fragment when clicked.

- **It's an efficient way to display large data sets.**
  Recycler views use a small number of views to give the appearance of a large collection of views that extend beyond the screen. When each item scrolls off the screen, it reuses—or *recycles*—its view for items that have scrolled onto the screen.

## Recycler views get their data from an adapter

Each recycler view you create uses an adapter to display its data. The adapter uses data from a data source (such as a database), and binds it to views in the item's layout. The recycler view then displays the items on the device screen as a scrollable list.

The data source, adapter, and recycler view fit together like this:



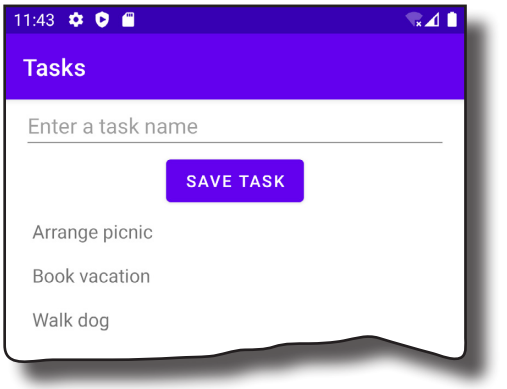

We're going to add a recycler view to the Tasks app. Let's go through the steps we'll take to do this.

# Here's what we're going to do

We'll go through the following steps to add a recycler view to the Tasks app:

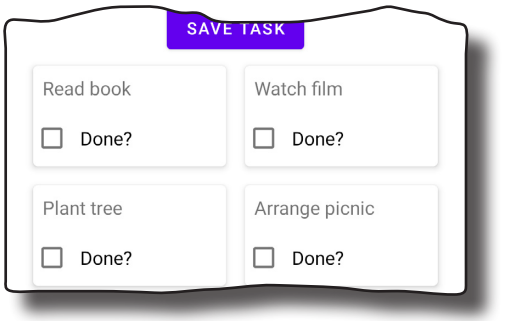
1. **Create a recycler view that displays a list of task names.**
We'll begin by creating a basic recycler view that just displays the name of each task. Keeping this first version relatively simple will make it easier to see how each part of the recycler view is constructed, and how they fit together.

The first version of the recycler view will display the tasks as a list of Strings.




2. **Update the recycler view to display a grid of cards.**
After we've got a basic recycler view working, we'll change it up so that it displays the name of each task and whether it's been completed in a grid of card views.

The second version of the recycler view will use a more complex layout.

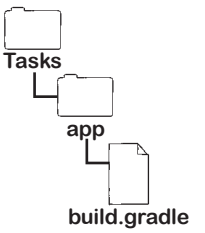



## Add a recycler view dependency to the app build.gradle file

Before we can start building the recycler view, we need to add a dependency to the recycler view library to the app's *build.gradle* file. Open the Tasks app, then open the file *Tasks/app/build.gradle*, and add the following line (in bold) to the `dependencies` section:

```
dependencies {
    ...
    implementation 'androidx.recyclerview:recyclerview:1.2.1'
    ...
}
```

This adds the recycler view library.




When prompted, make sure that you sync this change with the rest of the app.

Now that we've added this dependency, let's start building the recycler view.

**Don't forget!**

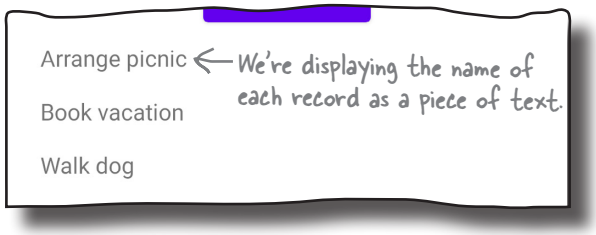
You're going to modify the Tasks app you created in Chapter 14, so make sure you open the project for this app.

**Task list**
**Grid of cards**

# Tell the recycler view how to display each item...

The first thing that we'll do is tell the recycler view how to display each task record.

For the first version of the app, we want to display the name of each task in the recycler view so that it looks like this:

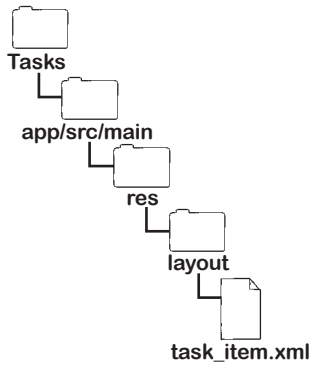



So how do we do this?

## ...by defining a layout file

You specify how each item in the recycler view should be laid out using a layout file. The recycler view uses this layout file when it needs to display each item. If the layout file is comprised of a single text view, for example, a text view will be displayed for each item in the recycler view's list.

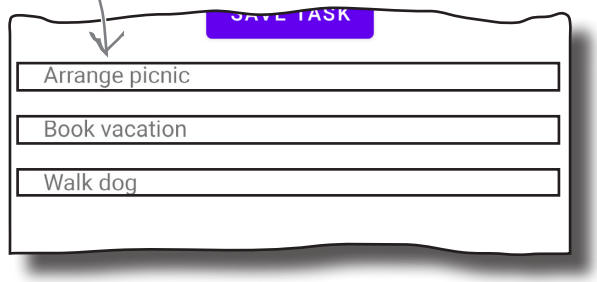
To create the layout file, highlight the *Tasks/app/src/main/res/layout* folder in the project explorer, and then choose File→New→Layout Resource File. When prompted, enter a file name of "task_item" and click on OK.




For the first version of the app, we want to display the name of each of the recycler view's tasks in a single text view, so we'll add a text view to the layout file we just created. To do this, update the code for *task_item.xml* so that it matches the code below:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textSize="16sp"
    android:padding="8dp" />
```

Each item is displayed using a text view.




That's all the code that we need to tell the recycler view how each item should be laid out. Next, we'll create the recycler view's adapter.

# The adapter adds data to the recycler view

As we said earlier, when you use a recycler view in your app, you need to create an adapter for it.

The recycler view's adapter has two main jobs: to create each of the views that are visible within the recycler view, and display a piece of data in each one. For the Tasks app, we need to define an adapter that uses *task_item.xml* to create a bunch of text views (one for each task record that's displayed), and puts a task name in each one.

We'll build the adapter over the next few pages. Here are the steps we'll go through to create it.

**An adapter acts as a bridge between a data source and a recycler view.**

1. **Specify what kind of data the adapter should work with.**
   We want the adapter to work with `Task` data, so we'll specify that it uses a `List<Task>`.

2. **Define the adapter's view holder.**
   This controls how each view in the item's layout should be populated.

3. **Inflate each item's layout.**
   When the recycler view needs to display each item, we'll inflate an instance of *task_item.xml* for that item.

4. **Display each item's data in the layout.**
   We'll do this by adding each `Task`'s `taskName` property value to the layout's text view.

We'll begin by creating a file for the adapter.

## Create the adapter file

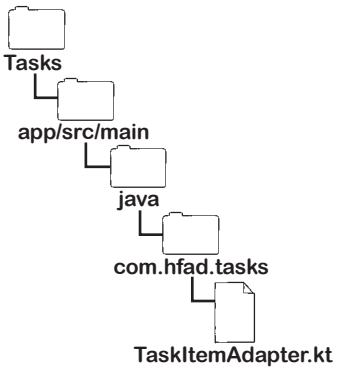
We're going to create an adapter for the recycler view named `TaskItemAdapter`. To do this, highlight the *com.hfad.tasks* package in the *app/src/main/java* folder, then go to File→New→Kotlin Class/File. Name the file "TaskItemAdapter" and choose the Class option.

Once you've created the file, update its code so it extends the `RecyclerView.Adapter` class like this:

```
package com.hfad.tasks

import androidx.recyclerview.widget.RecyclerView

class TaskItemAdapter : RecyclerView.Adapter() {
}
```

Import this class.

Extend RecyclerView.Adapter.




This turns the class into an adapter that can be used by a recycler view.

**Task list**
Grid of cards

# Tell the adapter what data it should work with

When you define a recycler view adapter, you need to tell it what sort of data should be added to the recycler view. We'll do this by adding a property to the adapter that specifies the data type.

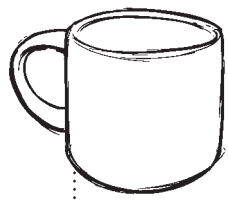
For the tasks app, we want the recycler view to display a list of task records, so we'll add a `List<Task>` property named `data` to the adapter. We'll also include a custom setter that calls `notifyDataSetChanged()` if the property gets updated; this tells the recycler view that the data has changed so it can redraw itself.

Here's the updated `TaskItemAdapter` code; update *TaskItemAdapter.kt* to include this change (in bold):

```
...
class TaskItemAdapter : RecyclerView.Adapter() {
    var data = listOf<Task>()
        set(value) {
            field = value
            notifyDataSetChanged()
        }
}
```

Add the data property and its setter.

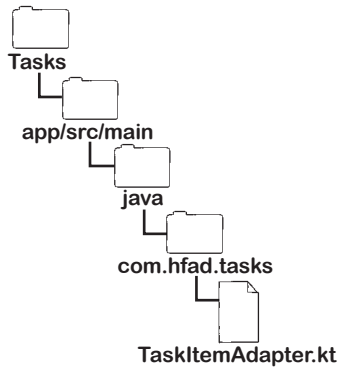
This tells the recycler view that the data has changed.



## Relax

**Don't worry if you get errors in your `TaskItemAdapter` code.**

The code isn't complete yet, and the error messages will go away when we finish writing it.




## Override the getItemCount() method

Next, we need to override the adapter's **`getItemCount()`** method. This tells the adapter how many data items there are so that the recycler view knows how many to display.

In the `TaskItemAdapter` code, we're using a `List<Task>` property named `data` for the recycler view's data items, so we can use `data.size` to say how many items there are.

Here's the `getItemCount()` method (in bold), which you can add to *TaskItemAdapter.kt*:

```
...
class TaskItemAdapter : RecyclerView.Adapter() {
    ...
    override fun getItemCount() = data.size
}
```

This is all the code we need for the getItemCount() method.

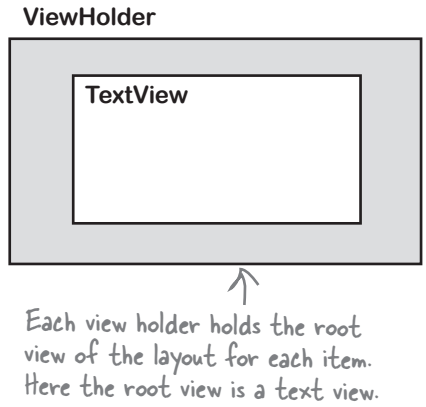
Now that we've specified the type of data the adapter works with, we'll use it to populate the layout's text view. We'll do this by defining the adapter's **view holder**.

# Define the adapter's view holder

The **view holder** contains information about how a view in the item's layout should be displayed, and its position in the recycler view. You can think of it as a holder for the root view of the item's layout—the layout that specifies how the recycler view should display each item.

In the Tasks app, we want the recycler view to use the layout file *task_item.xml* to display task records. This layout's root view is a `TextView`, so we need to define a view holder that works with text views.

You define a view holder by adding an inner class to the adapter file that extends `RecyclerView.ViewHolder`. It includes a constructor that specifies the type of the layout's root view (in this case `TextView`). The adapter's class definition also needs to be updated to specify the adapter's class name.



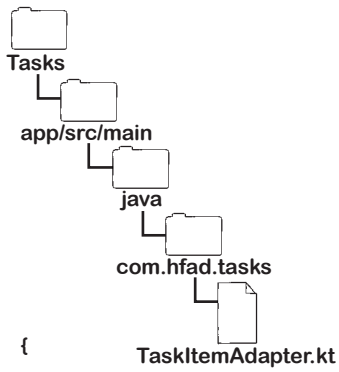

Here's the updated code for `TaskItemAdapter`; update *TaskItemAdapter.kt* so that it includes the changes shown here (in bold):

```
package com.hfad.tasks

import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView

class TaskItemAdapter : RecyclerView.Adapter<TaskItemAdapter.TaskItemViewHolder>() {
    var data = listOf<Task>()
        set(value) {
            field = value
            notifyDataSetChanged()
        }

    override fun getItemCount() = data.size

    class TaskItemViewHolder(val rootView: TextView)
                : RecyclerView.ViewHolder(rootView) {
    }
}
```

Import this class.

Use generics to say that TaskItemAdapter works with its TaskItemViewHolder inner class.

Add the TaskItemViewHolder inner class. We'll add more code to it over the next couple pages.



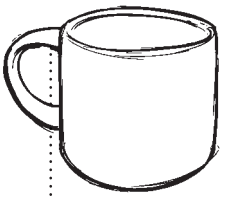

Now that we've defined a view holder, we need to specify which layout it uses by overriding the adapter's `onCreateViewHolder()` method.

**Task list**
Grey of cards

# Override the onCreateViewHolder() method

The adapter's **`onCreateViewHolder()`** method gets called each time the recycler view requires a new view holder. The recycler view calls the method repeatedly when it's first constructed to build the set of view holders that will be displayed on the screen.

The `onCreateViewHolder()` method needs to do two things: inflate the layout that's used for each item (in this case *task_item.xml*), and return a view holder. Here's the code to do this for `TaskItemAdapter`, so update the code for *TaskItemAdapter.kt* to include the changes shown here (in bold):



## Relax

**Writing adapter code can seem daunting at first, as it needs to do several different things.**

Take your time, keep going, and we know you'll be fine.

```
...
import android.view.LayoutInflater
import android.view.ViewGroup

class TaskItemAdapter : RecyclerView.Adapter<TaskItemAdapter.TaskItemViewHolder>() {
    ...
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                        : TaskItemViewHolder = TaskItemViewHolder.inflateFrom(parent)

    class TaskItemViewHolder(val rootView: TextView)
                        : RecyclerView.ViewHolder(rootView) {
        companion object {
            fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
                val layoutInflater = LayoutInflater.from(parent.context)
                val view = layoutInflater
                        .inflate(R.layout.task_item, parent, false) as TextView
                return TaskItemViewHolder(view)
            }
        }
    }
}
```

Import these classes.

Override this method.

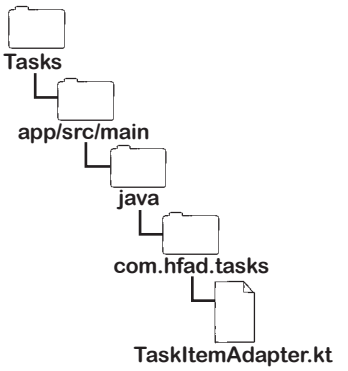
parent refers to the recycler view.

This lets you use different layouts for different items.

onCreateViewHolder() will call the view holder's inflateFrom() method.

Add an inflateFrom() method inside a companion object. This means that the method can be called without first creating a TaskItemViewHolder object.

The method inflates the item's layout, and uses it to create a TaskItemViewHolder.




As you can see, we've put the code to inflate *task_item.xml* in a new `inflateFrom()` method in `TaskItemViewHolder`, which the adapter's `onCreateViewHolder()` method calls using:

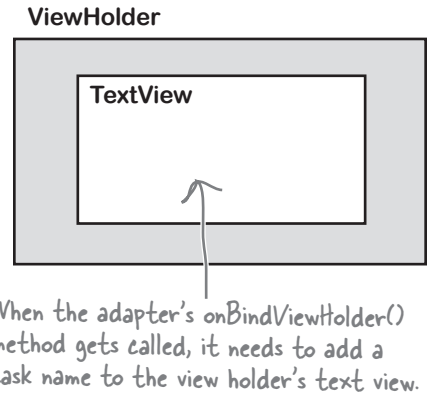
```
TaskItemViewHolder.inflateFrom(parent)
```

This approach passes responsibility for the view holder's layout to the view holder instead of inflating the layout in the main body of the adapter code.

# Add data to the layout's view

The final detail that we need to add to the adapter is how to display the task records in the view holder's layout. We'll do this by overriding the adapter's **`onBindViewHolder()`** method, which gets called whenever the recycler view needs to display data. It takes two parameters: the view holder the data needs to be bound to, and the data's position in the data set.

In the Tasks app, we want to take the `Task` object at a certain position in the adapter's `data` property (a `List<Task>`), and display its `taskName` in the view holder's layout. Here's the code to do this for `TaskItemAdapter`, so update the code for *TaskItemAdapter.kt* to include the changes shown here (in bold):




```
...
class TaskItemAdapter : RecyclerView.Adapter<TaskItemAdapter.TaskItemViewHolder>() {
    var data = listOf<Task>()
    ...
    override fun onBindViewHolder(holder: TaskItemViewHolder, position: Int) {
        val item = data[position]
        holder.bind(item)
    }

    class TaskItemViewHolder(val rootView: TextView)
                            : RecyclerView.ViewHolder(rootView) {
        ...
        fun bind(item: Task) {
            rootView.text = item.taskName
        }
    }
}
```

Override this method.

The view holder the method needs to add data to.

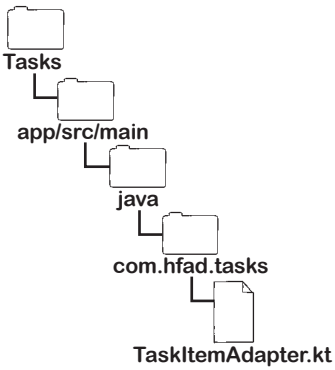
The item's position in the data.

Get the item that needs to be displayed.

Call the view holder's bind() method for that item.

Add this method to the view holder.

Add the task name to the layout's root view (a text view).

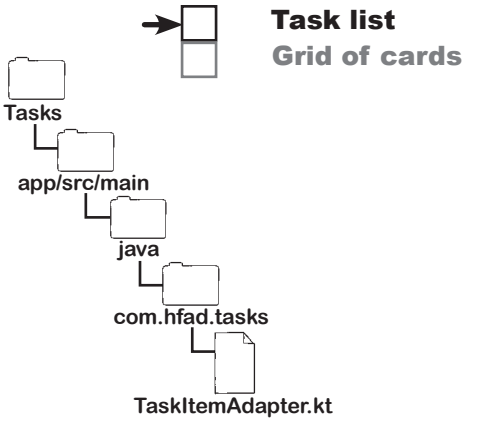



As you can see, we're setting the layout's text view text in a new `bind()` method, which we've added to `TaskItemViewHolder`. The adapter's `onBindViewHolder()` method then calls this each time it runs. We use this approach because it makes the view holder responsible for populating its layout, instead of the adapter.

That's all the code that we need to write for `TaskItemAdapter` and its `TaskItemViewHolder` inner class. Let's look at the full code.

# The full code for TaskItemAdapter.kt

Here's the full code for `TaskItemAdapter`; make sure that the code for *TaskItemAdapter.kt* includes everything shown here:




```
package com.hfad.tasks

import android.view.LayoutInflater
import android.view.ViewGroup
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView

class TaskItemAdapter : RecyclerView.Adapter<TaskItemAdapter.TaskItemViewHolder>() {
    var data = listOf<Task>()
        set(value) {
            field = value
            notifyDataSetChanged()
        }

    override fun getItemCount() = data.size

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                        : TaskItemViewHolder = TaskItemViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: TaskItemViewHolder, position: Int) {
        val item = data[position]
        holder.bind(item)
    }

    class TaskItemViewHolder(val rootView: TextView)
                            : RecyclerView.ViewHolder(rootView) {
        companion object {
            fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
                val layoutInflater = LayoutInflater.from(parent.context)
                val view = layoutInflater
                           .inflate(R.layout.task_item, parent, false) as TextView
                return TaskItemViewHolder(view)
            }
        }

        fun bind(item: Task) {
            rootView.text = item.taskName
        }
    }
}
```

This is for the recycler view's data.

The number of items

This gets called when a view holder needs to be created.

This gets called when data needs to be displayed in a view holder.

This defines the view holder.

This creates each view holder and inflates its layout.

This adds data to the view holder's layout.

# The adapter code is complete

We've now finished writing all of the code that's needed for `TaskItemAdapter`. It does the following four things:

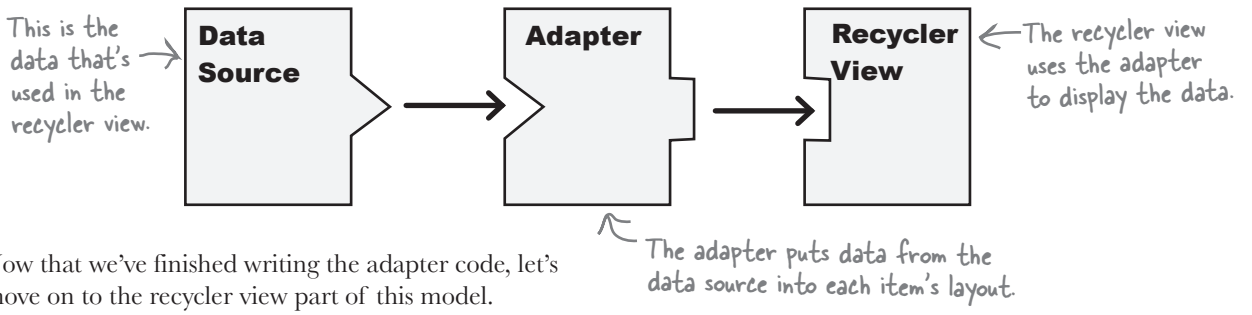
1. **Specifies that it works with Task data**
   We did this by defining a `List<Task>` property named `data`.

2. **Uses a view holder named TaskItemViewHolder**
   We added `TaskItemViewHolder` to `TaskItemAdapter` as an inner class.

3. **Inflates each item's layout**
   It does this when its `onCreateViewHolder()` method gets called.

4. **Displays each item's data in the layout**
   It does this using its `onBindViewHolder()` method.

As we said earlier, the adapter acts as a bridge between a data source and a recycler view. The data source, adapter, and recycler view fit together like this:




Now that we've finished writing the adapter code, let's move on to the recycler view part of this model.

## there are no Dumb Questions

**Q: Why is it called a recycler view?**

A: Because a recycler view reuses—or recycles—its views. This makes it a very efficient way of displaying a list of data.

**Q: You added a view holder inner class to the adapter as well. Can you remind me what that is?**

A: It's a holder for the root view of each item's layout. Every recycler view adapter that you create needs to use a view holder.

**Q: Does it have to be an inner class?**

A: No. As each view holder is usually tied to a single adapter, it's often defined as an inner class, but this isn't mandatory.

**Q: Why did you add `inflateFrom()` and `bind()` methods to the view holder? Couldn't you have included their code in the adapter's `onCreateViewHolder()` and `onBindViewHolder()` methods?**

A: We added these methods to the view holder because we wanted to keep all the code that manages view holders in one place. If we'd left the code in the main body of the adapter, we wouldn't have had such a clear separation of concerns.

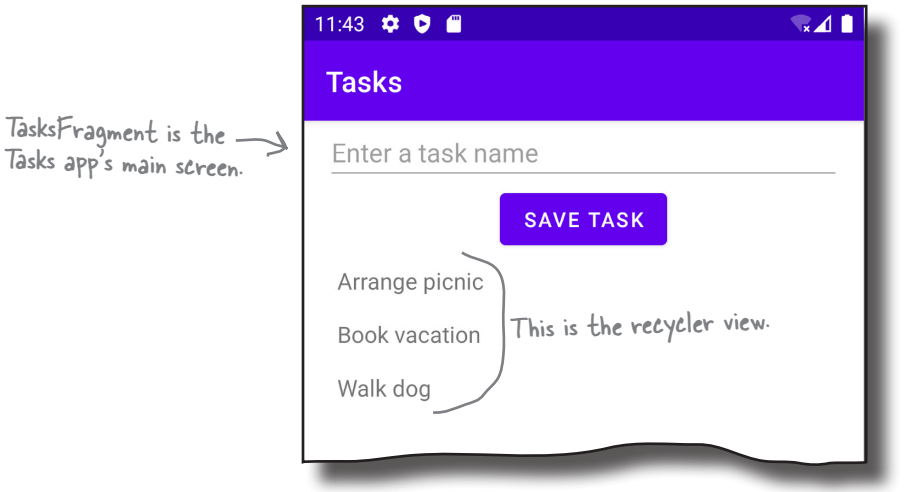
This approach is particularly useful when you want to do more than display a single text view for each item in the recycler view, as you'll find out later on.

**Task list**
Grid of cards

# We need to display the recycler view

The next thing that we need to do is display a recycler view in `TasksFragment` (the Tasks app's main screen), and get it to use the adapter that we've just created.

Here's a reminder of what the recycler view needs to look like:




## How to add a recycler view to a layout

You display a recycler view by adding an `<androidx.recyclerview.widget.RecyclerView>` element to a fragment's layout file. The code looks like this:

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/tasks_list"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager" />
```

This adds the recycler view to the layout

This displays items in a vertical list.

The line:

```
app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"
```

specifies that the recycler view uses a layout manager, which determines how the recycler view positions its items. Here, we're using a linear layout manager, which means that the recycler view will display its items in a vertical list with full-length rows.

You'll find out more about layout managers later in the chapter.

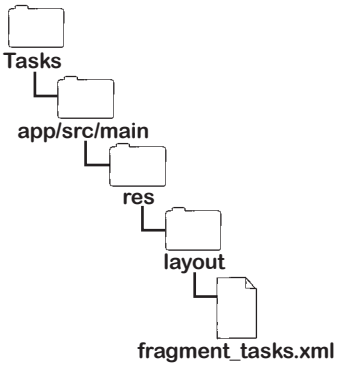
That's all the code you need to know to add a recycler view to `TasksFragment`'s layout, so let's see what the code looks like.

# The full code for fragment_tasks.xml

Below is the full code for *fragment_tasks.xml* (`TasksFragment`'s layout). As you can see, we've replaced the text view with a recycler view, so update the code for *fragment_tasks.xml* to include these changes (in bold):

```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".TasksFragment">

    <data>
        <variable
            name="viewModel"
            type="com.hfad.tasks.TasksViewModel" />
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <EditText
            android:id="@+id/task_name"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:inputType="text"
            android:hint="Enter a task name"
            android:text="@={viewModel.newTaskName}" />

        <Button
            android:id="@+id/save_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:text="Save Task"
            android:onClick="@{() -> viewModel.addTask()}" />
```

Add the app namespace.



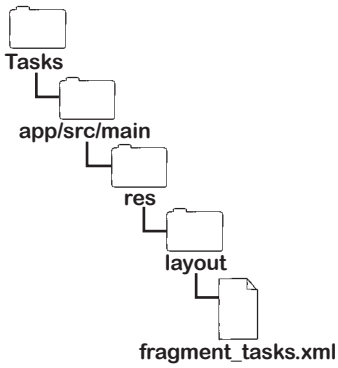

The code continues on the next page.

**Task list**
Grid of cards

# fragment_tasks.xml continued

Remove the text view.

```
<TextView
    android:id="@+id/tasks"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@{viewModel.tasksString}" />
```




Add the recycler view.

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/tasks_list"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1"
    android:gravity="top"
    app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager" />
</LinearLayout>
</layout>
```

That's all the code we need to write to add a recycler view to `TasksFragment`'s layout. Next, we'll tell the recycler view to use the adapter we created.

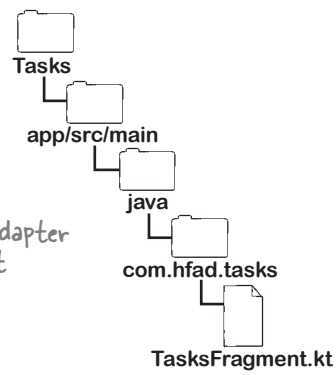
## Tell the recycler view to use the adapter

You make a recycler view use an adapter by creating an instance of the adapter, and attaching it to the recycler view. You do this in the fragment's Kotlin code.

In our case, we want to make the recycler view use `TaskItemAdapter`. This is done by adding the following code (in bold) to `TasksFragment`'s `onCreateView()` method:

```
...
override fun onCreateView(...): View? {
    ...
    val adapter = TaskItemAdapter()
    binding.tasksList.adapter = adapter
    ...
}
```

This adds the adapter to the tasksList recycler view.

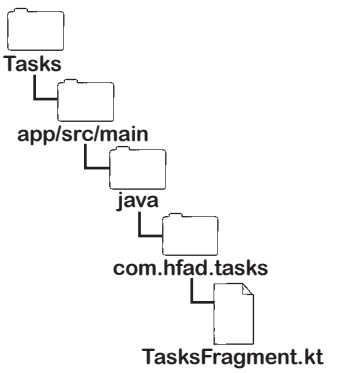



We'll add this code to `TasksFragment` on the next page.

# The updated code for TasksFragment.kt

Here's the code for `TasksFragment`; update *TasksFragment.kt* to include all the changes shown (in bold):

```
package com.hfad.tasks

...  <- We've left out the import statements because we don't need to add any new ones.

class TasksFragment : Fragment() {
    private var _binding: FragmentTasksBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentTasksBinding.inflate(inflater, container, false)
        val view = binding.root

        val application = requireNotNull(this.activity).application
        val dao = TaskDatabase.getInstance(application).taskDao

        val viewModelFactory = TasksViewModelFactory(dao)
        val viewModel = ViewModelProvider(
                this, viewModelFactory).get(TasksViewModel::class.java)
        binding.viewModel = viewModel
        binding.lifecycleOwner = viewLifecycleOwner

        val adapter = TaskItemAdapter()   <- Create a TaskItemAdapter.
        binding.tasksList.adapter = adapter   <- Attach the adapter to the recycler view.

        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```




So what's next?

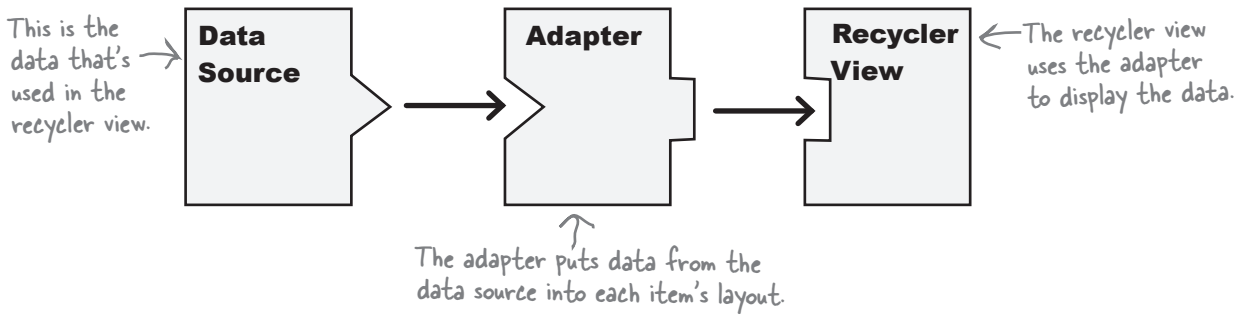
**Task list**
Grid of cards

# We've added a recycler view to TasksFragment's layout

We've now written all of the code that we need to display a recycler view in `TasksFragment`'s layout, and told it to use `TaskItemAdapter` as its adapter. But there's still one more thing left for us to do: we need to connect the adapter to a data source.

As you learned earlier, the adapter uses data from a data source (such as a database), and binds it to views in the item's layout. The recycler view then displays the items on the device screen.

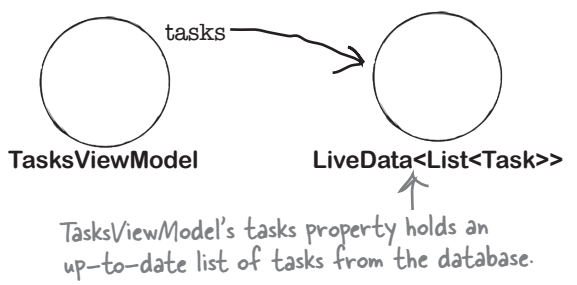
The data source, adapter, and recycler view fit together like this:

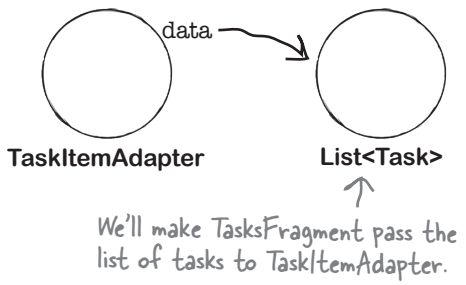



So in order to display data in the Tasks app's recycler view, we need to tell `TaskItemAdapter` what task data to use.

## We'll make TasksFragment add task data to TaskItemAdapter

We'll tell `TaskItemAdapter` what task data to use by getting `TasksFragment` to update its `data` property with a `List<Task>`. `TasksFragment` will get this list from `TasksViewModel`'s `tasks` property:




To do this, we first need to give `TasksFragment` access to `TasksViewModel`'s `tasks` property.

# Update the TasksViewModel.kt code

As you may recall, `TasksViewModel`'s `tasks` property is currently marked as private. We need to remove this modifier in order for `TasksFragment` to get the property's value.

We'll also remove all the code we added in the previous chapter to transform the task data into a formatted `String`: this code is no longer needed now that we're using a recycler view to display a `List<Task>`.

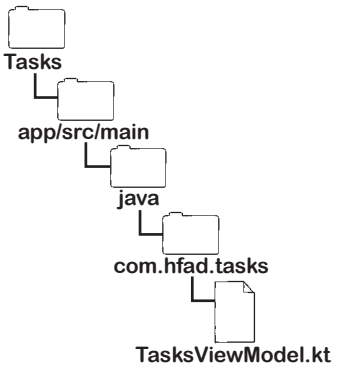
Here's the updated `TasksViewModel` code; update *TasksViewModel.kt* to include these changes:

```
package com.hfad.tasks

import androidx.lifecycle.Transformations
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch

class TasksViewModel(val dao: TaskDao) : ViewModel() {
    var newTaskName = ""
    private val tasks = dao.getAll()
    val tasksString = Transformations.map(tasks) {
        tasks -> formatTasks(tasks)
    }

    fun addTask() {
        viewModelScope.launch {
            val task = Task()
            task.taskName = newTaskName
            dao.insert(task)
        }
    }

    fun formatTasks(tasks: List<Task>): String {...}

    fun formatTask(task: Task): String {...}
}
```

Delete this line as it's no longer needed.

Remove the private modifier.

We no longer need the tasksString property, so remove it.




Delete these two methods.

Now that `TasksFragment` has access to the `tasks` property, let's make it pass the property's `List<Task>` to `TaskItemAdapter`.

**Task list**
Grid of cards

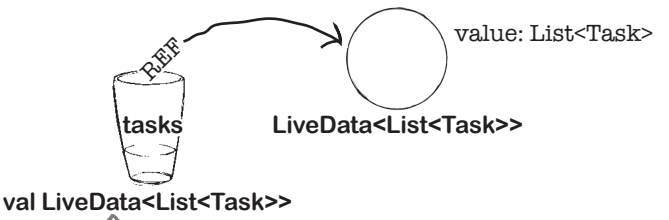
# TasksFragment needs to update TaskItemAdapter's data property

As you know, `TasksViewModel`'s `tasks` property holds a live data list of tasks, which it gets from the database using this code:

```
val tasks = dao.getAll()
```

This uses TaskDao's getAll() method to get a LiveData<List<Task>> from the database.

As this property uses live data, we can make `TasksFragment` observe it so that each time its value changes, the fragment is notified. `TasksFragment` will then be able to assign the latest version of the list to the adapter's `data` property, making sure that the data displayed in the recycler view is always up-to-date.



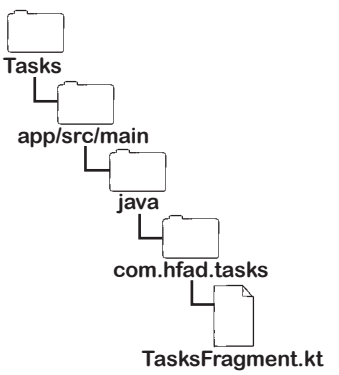

The tasks property holds a live data list of tasks, which it gets from the database.

You're already familiar with the code to observe live data properties, so here's the code we need to add to `TasksFragment` (in bold):

```
class TasksFragment : Fragment() {
    ...
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        ...

        val adapter = TaskItemAdapter()
        binding.tasksList.adapter = adapter

        viewModel.tasks.observe(viewLifecycleOwner, Observer {
            it?.let {
                adapter.data = it
            }
        })
        ...
    }
    ...
}
```

Observe the view model's tasks property.

When the list of tasks changes, assign it to the adapter's data property.



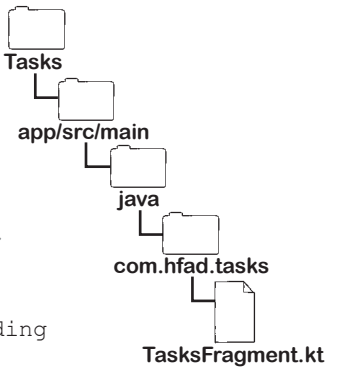

Let's update *TasksFragment.kt* to include this change.

# The full code for TasksFragment.kt

Here's the updated code for `TasksFragment`; update *TasksFragment.kt* to include all the changes shown (in bold)

```
package com.hfad.tasks

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.lifecycle.Observer
import androidx.lifecycle.ViewModelProvider
import com.hfad.tasks.databinding.FragmentTasksBinding

class TasksFragment : Fragment() {
    private var _binding: FragmentTasksBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentTasksBinding.inflate(inflater, container, false)
        val view = binding.root

        val application = requireNotNull(this.activity).application
        val dao = TaskDatabase.getInstance(application).taskDao

        val viewModelFactory = TasksViewModelFactory(dao)
        val viewModel = ViewModelProvider(
                this, viewModelFactory).get(TasksViewModel::class.java)
        binding.viewModel = viewModel
        binding.lifecycleOwner = viewLifecycleOwner

        val adapter = TaskItemAdapter()
        binding.tasksList.adapter = adapter
```

Import this class.



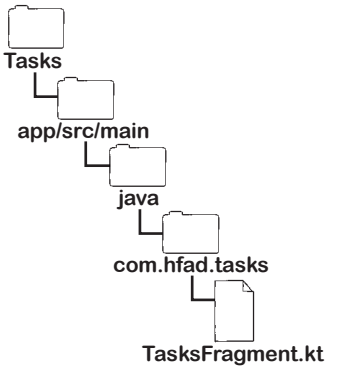

The code continues on the next page.

**Task list**
Grid of cards

# TasksFragment.kt continued

This passes the data to the adapter.

```
        viewModel.tasks.observe(viewLifecycleOwner, Observer {
            it?.let {
                adapter.data = it
            }
        })

        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```




## We've finished writing all the recycler view code

It took us a while, but we've now finished writing all the code that's needed to display a list of task names in a recycler view. We did this by:

1. **Creating an adapter named TaskItemAdapter**
   The adapter acts as a bridge between the recycler view and its data source. In the Tasks app, the data source is a Room database containing task records.

2. **Attaching TaskItemAdapter to a recycler view**
   We added a recycler view to `TasksFragment`'s layout, and told it to use `TaskItemAdapter` in its Kotlin code.

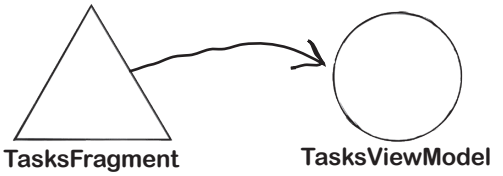
3. **Passing an up-to-date List<Task> to TaskItemAdapter**
   We did this by making `TasksFragment` set `TaskItemAdapter`'s data property each time `TasksViewModel`'s live data list of tasks gets updated.

Before we take the app for a test drive and see what the recycler view looks like, let's go through what happens when the code runs.

# What happens when the code runs

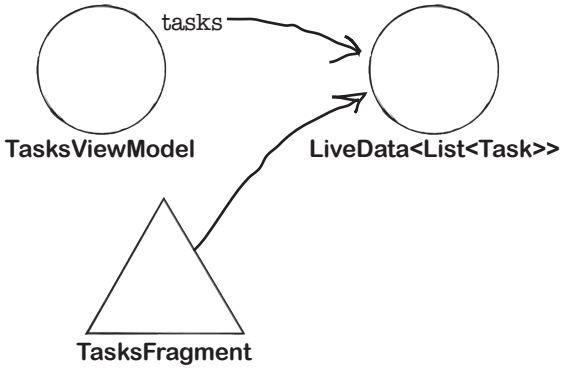
The following things happen when the app runs:

1. **When the app is launched, MainActivity displays TasksFragment.**
`TasksFragment` uses `TasksViewModel` as its view model.

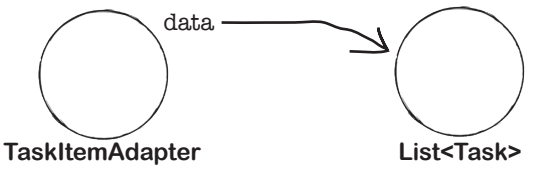



2. **TasksFragment creates a TaskItemAdapter object, and assigns it to the recycler view as its adapter.**

3. **TasksFragment observes TasksViewModel's tasks property.**
This property is a `LiveData<List<Task>>`, which holds an up-to-date list of records from the database.



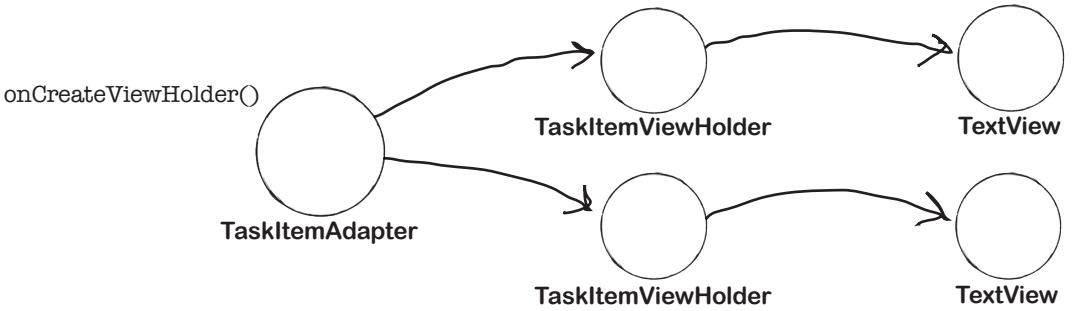

4. **TasksFragment sets the TaskItemAdapter's data property to the List<Task>.**



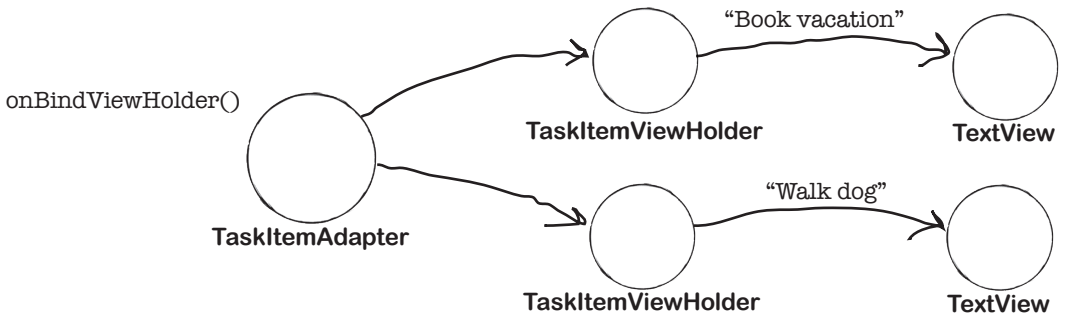
**Task list**
Grid of cards

# The story continues

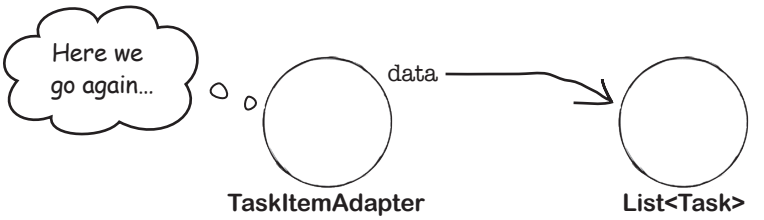
**5** **TaskItemAdapter's onCreateViewHolder() method is called for each item that needs to be displayed in the recycler view.**
This creates a `TaskItemViewHolder` for each item. A layout (defined by *task_item.xml*) is inflated for each view holder.




**6** **TaskItemAdapter's onBindViewHolder() is called for each TaskItemViewHolder.**
This binds data to the text view in each view holder's layout.




**7** **Each time TasksViewModel's tasks property gets updated, TasksFragment passes an updated List<Task> to the TaskItemAdapter.**
Steps 5 to 6 are repeated so the recycler view stays up-to-date.

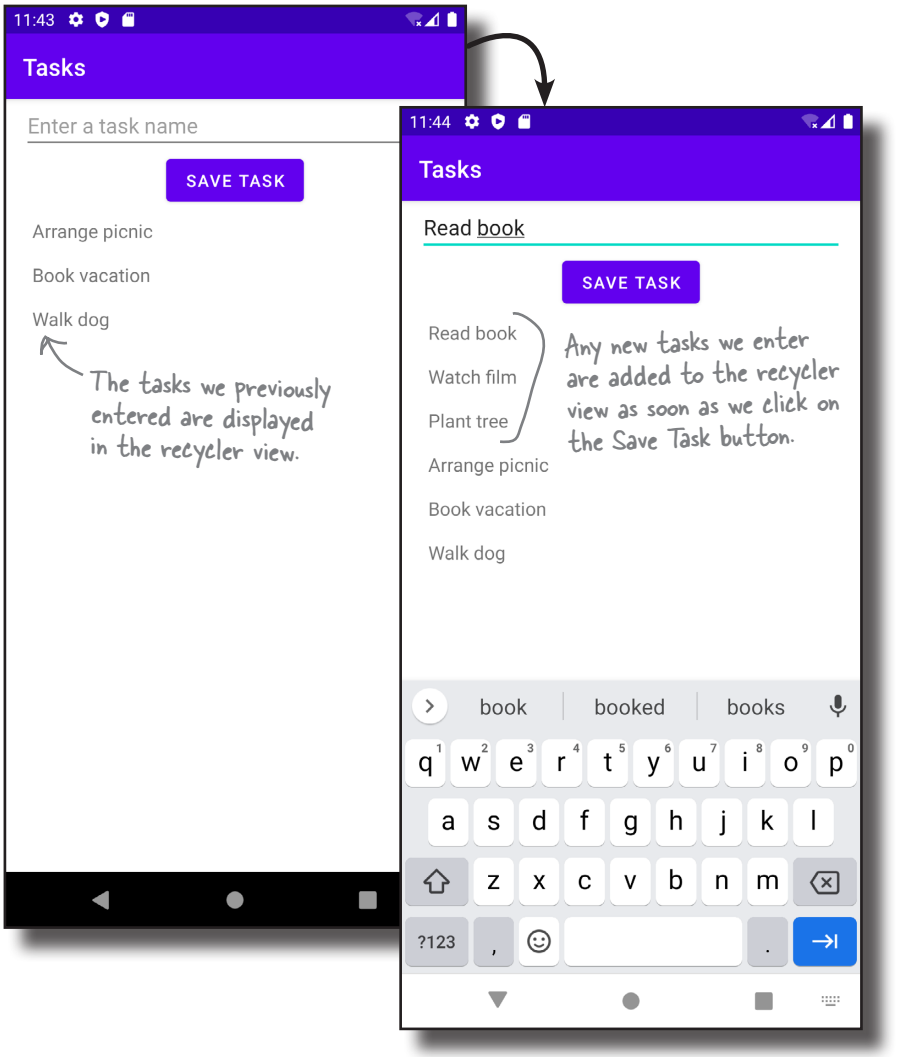



Let's take the app for a test drive.

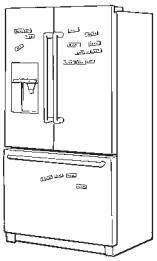
# Test Drive

When we run the app, `TasksFragment` displays the name of each task in a recycler view.

When we enter new tasks, these get added to the recycler view's list. The app works as planned.




You've now learned how to create a basic recycler view. Before we tweak it to display the task records in a different way, try the following exercise.



# Adapter Magnets

The Bits and Pizzas app includes a `Pizza` data class that looks like this:

```
package com.hfad.bitsandpizzas

data class Pizza(
    var pizzaId: Long = 0L,
    var pizzaName: String = "",
    var pizzaDescription: String = "",
    var pizzaImageId: Int = 0
)
```

The app needs to include a recycler view that displays the `pizzaName` for each `Pizza` item in the following layout (named *pizza_item.xml*):

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

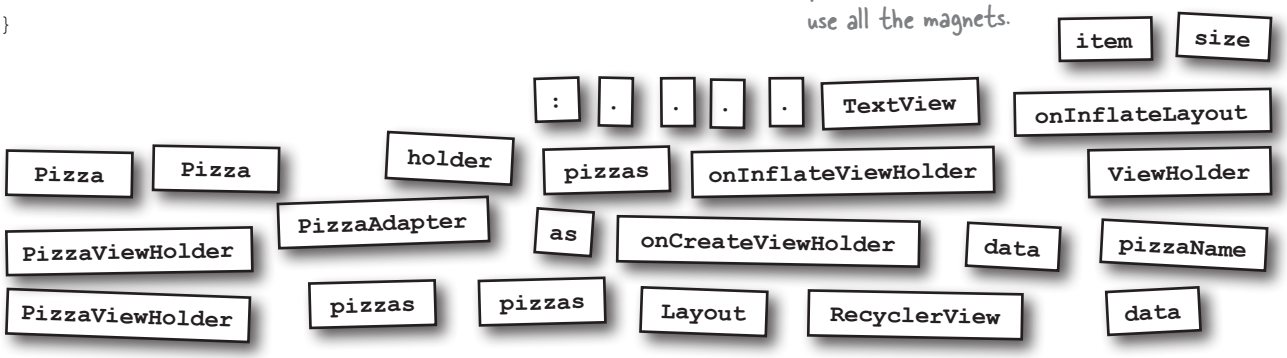
See if you can complete the recycler view's adapter code (below).

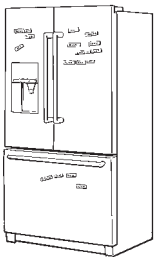
```
package com.hfad.bitsandpizzas

import android.view.LayoutInflater
import android.view.ViewGroup
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView

class PizzaAdapter : RecyclerView.Adapter<.....>() {

    var pizzas = listOf<.....>()
        set(value) {
            field = value
            notifyDataSetChanged()
        }

    override fun getItemCount() = .....
```

The exercise continues on the next page.

```
    override fun ____________________________(parent: ViewGroup, viewType: Int)
                        : PizzaViewHolder = PizzaViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(_____________________________________ , position: Int) {
        val item = ____________ [position]
        holder.bind(item)
    }

    class PizzaViewHolder(val rootView: TextView)
                            : ____________________________________(rootView) {
        companion object {
            fun inflateFrom(parent: ViewGroup): PizzaViewHolder {
                val layoutInflater = LayoutInflater.from(parent.context)
                val view = layoutInflater
                            .inflate(R.layout.pizza_item, parent, false) ____________
                return PizzaViewHolder(view)
            }
        }

        fun bind(item: ____________) {
            rootView.text = ____________________________
        }
    }
}
```

You won't need to use all the magnets.





# Adapter Magnets Solution

The Bits and Pizzas app includes a `Pizza` data class that looks like this:

```
package com.hfad.bitsandpizzas

data class Pizza(
    var pizzaId: Long = 0L,
    var pizzaName: String = "",
    var pizzaDescription: String = "",
    var pizzaImageId: Int = 0
)
```

The app needs to include a recycler view that displays the `pizzaName` for each `Pizza` item in the following layout (named *pizza_item.xml*):

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

See if you can complete the recycler view's adapter code (below).

```
package com.hfad.bitsandpizzas

import android.view.LayoutInflater
import android.view.ViewGroup
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView

class PizzaAdapter : RecyclerView.Adapter<PizzaAdapter.PizzaViewHolder>() {

    var pizzas = listOf<Pizza>()
        set(value) {
            field = value
            notifyDataSetChanged()
        }

    override fun getItemCount() = pizzas.size
```

The adapter needs to use the PizzaViewHolder inner class as its view holder.

The adapter uses a list of Pizzas.

Use this to specify how many items there are.

The solution continues on the next page.

Override this method.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                    : PizzaViewHolder = PizzaViewHolder.inflateFrom(parent)

override fun onBindViewHolder(holder : PizzaViewHolder, position: Int) {
    val item = pizzas[position]
    holder.bind(item)
}
```

Pass the type of view holder to onBindViewHolder.

Get the Pizza at the required position.

Extend this class.

```
class PizzaViewHolder(val rootView: TextView)
                    : RecyclerView.ViewHolder(rootView) {
    companion object {
        fun inflateFrom(parent: ViewGroup): PizzaViewHolder {
            val layoutInflater = LayoutInflater.from(parent.context)
            val view = layoutInflater
                        .inflate(R.layout.pizza_item, parent, false) as TextView
            return PizzaViewHolder(view)
        }
    }

    fun bind(item: Pizza) {
        rootView.text = item.pizzaName
    }
}
```

Cast the layout as a TextView to reflect the type of its root view.

bind() needs to accept a Pizza item.

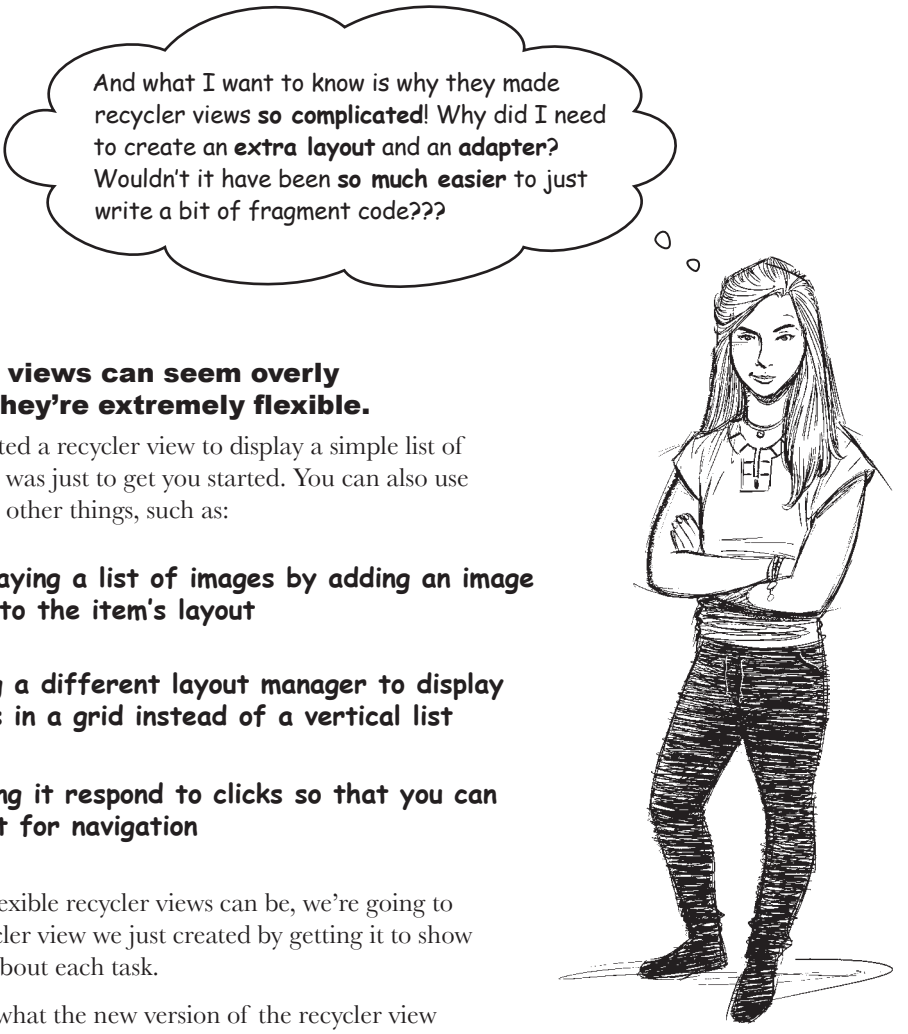
Display the pizzaName value in the layout's text view.

You didn't need to use these magnets.

**Task list**
**Grid of cards**

# Recycler views are very flexible

So far in this chapter, you've learned how to build a basic recycler view that displays a list of task names. You did this by creating a layout that's used by each item in the list, defining an adapter to populate it with data, and adding a recycler view to `TaskFragment`'s layout.




## While recycler views can seem overly complicated, they're extremely flexible.

In this app, we created a recycler view to display a simple list of task names, but this was just to get you started. You can also use recycler views to do other things, such as:

- **Displaying a list of images by adding an image view to the item's layout**
- **Using a different layout manager to display items in a grid instead of a vertical list**
- **Making it respond to clicks so that you can use it for navigation**

To show you how flexible recycler views can be, we're going to change up the recycler view we just created by getting it to show more information about each task.

Let's take a look at what the new version of the recycler view will look like.

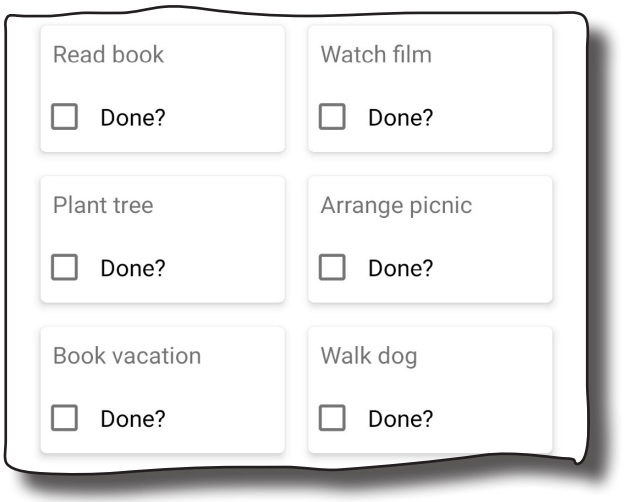
**Task list**
**Grid of cards**

# Recycler view 2.0

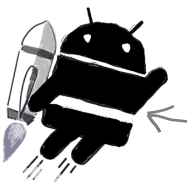
We're going to update the recycler view so that it shows the name of each task in a text view, and whether each task has been completed with a checkbox. We'll display each task record on a single card, which will look like it's slightly elevated. The cards will be arranged in a grid.

Here's what the new version of the recycler view will look like:

We're going to display the task items in a grid of cards. Each card contains a text view for the task name, and a checkbox to show whether the task has been completed.




We're going to create this version of the recycler view by changing *task_item.xml* (the layout used by the recycler view's items) so that it uses a **card view**. This is a type of frame layout that has rounded corners and shadows to make it look as though it's elevated above its background.



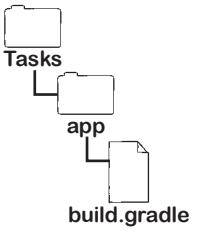
Card views are part of Android Jetpack.

## Add a card view dependency to the app build.gradle file

To use a card view, we first need to add a dependency to its library to the app's *build.gradle* file. Open the file *Tasks/app/build.gradle*, and add the following line (in bold) to the `dependencies` section:

```
...
dependencies {
    ...
    implementation 'androidx.cardview:cardview:1.0.0'
    ...
}
```

This adds the card view library to the app.




Make sure that you sync this change with the rest of the app.

**Task list**
**Grid of cards**

# How to create a card view

We're going to use a card view in *task_item.xml* that includes a text view and a checkbox.

You create a card view by adding an `<androidx.cardview.widget.CardView>` element to your layout code. The code for a typical card view looks like this:

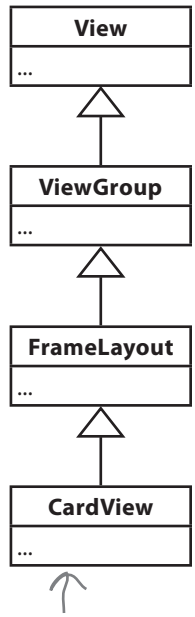
```
<androidx.cardview.widget.CardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="8dp"
    app:cardElevation="4dp"
    app:cardCornerRadius="4dp" >
    ...
</androidx.cardview.widget.CardView>
```

This defines the CardView.

Setting the card's elevation gives it a drop shadow.

This gives the CardView rounded corners.

You add any views you want to be displayed to the CardView.




A CardView is a type of FrameLayout. It includes all of FrameLayout's attributes, along with some extras.

As you can see, the above code includes an extra namespace of:

```
xmlns:app="http://schemas.android.com/apk/res-auto"
```

This lets you add attributes that give the card rounded corners and a drop shadow to make it look higher than its background. You add rounded corners using the `app:cardCornerRadius` attribute, and the `app:cardElevation` attribute sets its elevation and adds drop shadows:

There's also an app:cardBackgroundColor attribute that changes the card's background color.



This is an empty card view with rounded corners and elevation.

Once you've defined the card view, you add any views to it that you want it to include. In the Tasks app, we want to add a text view and a checkbox to the card view to display the name of each task and whether it's been completed. Let's see what the code for this looks like.

# The full code for task_item.xml

Here's the updated code for *task_item.xml*; update the code for this file so that it includes the changes below (in bold):

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.cardview.widget.CardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="8dp"
    app:cardElevation="4dp"
    app:cardCornerRadius="4dp" >

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical" >

        <TextView
            xmlns:android="http://schemas.android.com/apk/res/android"
            android:id="@+id/task_name"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textSize="16sp"
            android:padding="8dp" />

        <CheckBox
            android:id="@+id/task_done"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textSize="16sp"
            android:padding="8dp"
            android:clickable="false"
            android:text="Done?" />
    </LinearLayout>
</androidx.cardview.widget.CardView>
```

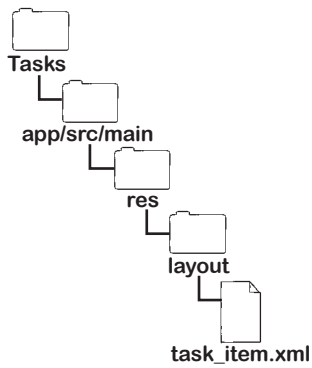
Add the card view.

We'll display the text view and checkbox in a linear layout inside the card view.

Move this line to the card view element.

Add an ID to the text view so that we can refer to it in the app's Kotlin code.

Add a checkbox.




Next, we need to update the adapter's view holder so that it works with the new layout, and populates the card's views.

**Task list**
**Grid of cards**

# The adapter's view holder needs to work with the new layout code

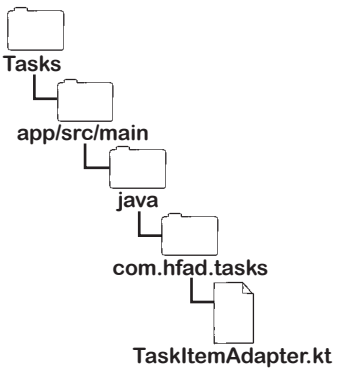
When we defined `TaskItemAdapter` (the recycler view's adapter), we included a `TaskItemViewHolder` inner class. We used it to inflate the layout (a text view) associated with each item in the recycler view, and populate it with the task's name.

Here's a reminder of the code we used for the original inner class:

```
class TaskItemViewHolder(val rootView: TextView)
                        : RecyclerView.ViewHolder(rootView) {
    companion object {
        fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
            val layoutInflater = LayoutInflater.from(parent.context)
            val view = layoutInflater
                       .inflate(R.layout.task_item, parent, false) as TextView
            return TaskItemViewHolder(view)
        }
    }

    fun bind(item: Task) {
        rootView.text = item.taskName
    }
}
```

This inflates an item's layout and uses it to create a view holder.

This adds data to the item's views.




Now that we've changed *task_item.xml*, we need to update `TaskItemViewHolder` so that it works with the new layout. To do this we need to make three changes:

1. **Update the view holder's constructor so that it uses a CardView instead of a TextView.**

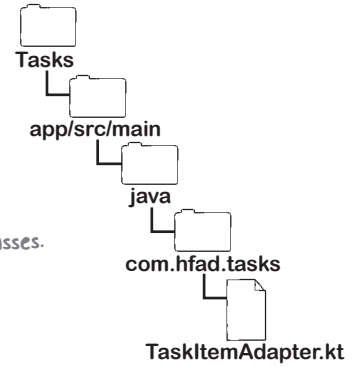
2. **Change the inflateFrom() method so that it inflates each item's layout as a CardView.**

3. **Update the bind() method so that it populates the layout's text view and checkbox with the item's taskName and taskDone property values.**

You're already familiar with the code that's needed to make these changes, so we'll show you the updated code for `TaskItemAdapter`—and its `TaskItemViewHolder` inner class—on the next page.

# The full code for TaskItemAdapter.kt

Here's the updated `TaskItemAdapter` code that works with the new layout code; update the code for *TaskItemAdapter.kt* so that it includes the changes below (in bold):




```
package com.hfad.tasks

import android.view.LayoutInflater
import android.view.ViewGroup
import android.widget.CheckBox
import android.widget.TextView
import androidx.cardview.widget.CardView
import androidx.recyclerview.widget.RecyclerView

class TaskItemAdapter : RecyclerView.Adapter<TaskItemAdapter.TaskItemViewHolder>() {
    var data = listOf<Task>()
        set(value) {
            field = value
            notifyDataSetChanged()
        }

    override fun getItemCount() = data.size

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                        : TaskItemViewHolder = TaskItemViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: TaskItemViewHolder, position: Int) {
        val item = data[position]
        holder.bind(item)
    }
```

Import these classes.

The code continues on the next page.

# TaskItemAdapter.kt continued

**Task list**
**Grid of cards**

```
class TaskItemViewHolder(val rootView: ~~TextView~~ CardView)
                                    : RecyclerView.ViewHolder(rootView) {
    val taskName = rootView.findViewById<TextView>(R.id.task_name)
    val taskDone = rootView.findViewById<CheckBox>(R.id.task_done)

    companion object {
        fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
            val layoutInflater = LayoutInflater.from(parent.context)
            val view = layoutInflater
                        .inflate(R.layout.task_item, parent, false) as ~~TextView~~ CardView
            return TaskItemViewHolder(view)
        }
    }

    fun bind(item: Task) {
        ~~rootView~~ taskName.text = item.taskName
        taskDone.isChecked = item.taskDone
    }
}
}
```

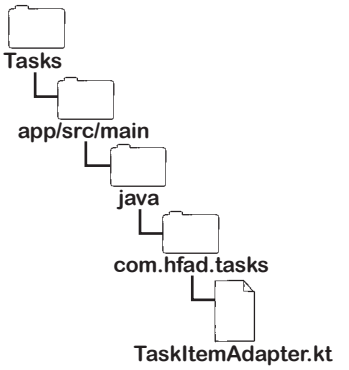
Change TextView to CardView.

Get references to the layout's text view and checkbox.

Change TextView to CardView here as well.

Display the task's name in the text view.

Use the checkbox to show whether the task is complete.




## there are no Dumb Questions

**Q: Can I display images inside a recycler view?**

A: Yes! To do this, add an image view to the item's layout, then populate it in the view holder's `bind()` method.

**Q: Do recycler views only work with database data?**

A: No, they can be used with data from other sources too, just so long as the adapter has access to the data.

**Q: I see that the view holder code uses `findViewById()`. I thought we'd stopped using that?**

A: The code uses this method to get a reference to each view in the item's layout that we want to populate with data. We've used this approach as it's a relatively simple way of adding data to each view.

In the next chapter, you'll learn how to replace these calls using data binding.

**Q: Can recycler views use multiple layouts?**

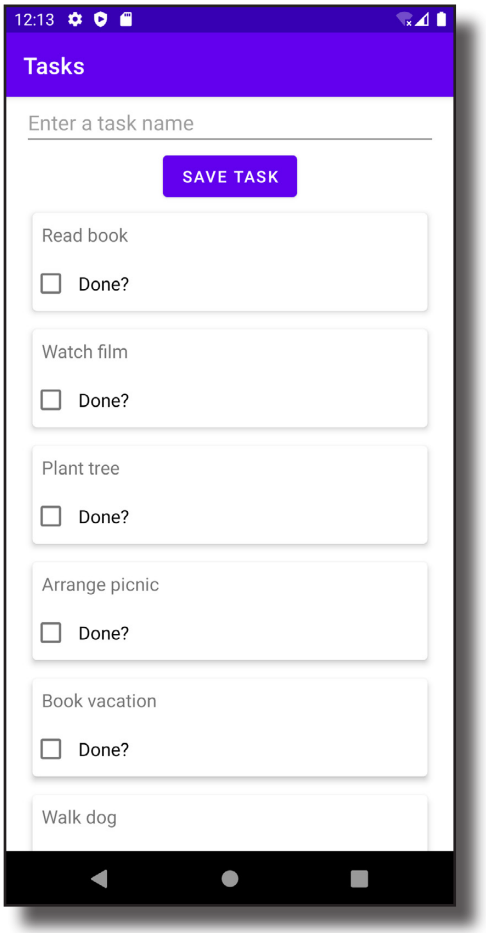
A: Yes! You can, say, make the first item in the list have a different layout to the rest of the items.

To do this, you create each layout you want to use, and define a view holder for each one. You then override the adapter's `getItemViewType()` method to say which view holder should be used for which position in the recycler view. Finally, you update `onCreateViewHolder()` to check its view type parameter, and create the correct corresponding view holder.

# What the recycler view looks like so far

If we were to run the app after updating the code for *task_item.xml* and *TaskItemAdapter.kt*, we'd see a recycler view that displays tasks in a vertical list of card views like this:

The changes we've made so far display the tasks in a vertical list of cards.




The recycler view arranges the cards in this way because we specified in *fragment_tasks.xml* that it must use a linear layout manager like this:

```
<androidx.recyclerview.widget.RecyclerView
    ...
    app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager" />
```

The current version of the app uses a linear layout manager to display items in a vertical list.

By default, this layout manager arranges items in a vertical list with full-length rows. You can, however, include extra options—or use a different type of layout manager—to change the way in which items are displayed.

Let's take a look at some of the possible options.

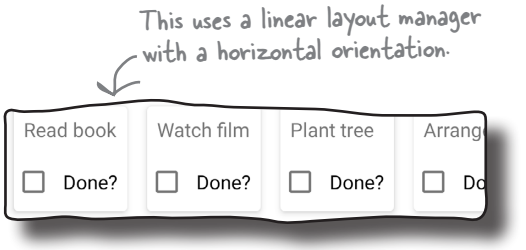
**Task list**
**Grid of cards**

# The layout manager gallery

Here are some of the other ways in which you might want to arrange items in a recycler view, and how to create each one.

## Display items in a horizontal row

A linear layout manager displays items in a vertical list by default. You can, however, display items in a horizontal row instead by changing the orientation to horizontal:



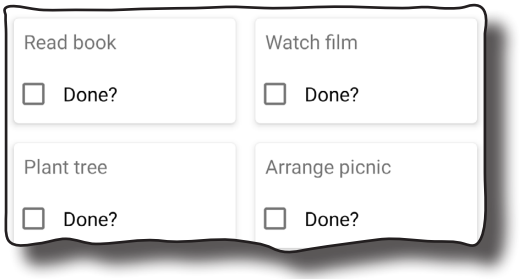

```
<androidx.recyclerview.widget.RecyclerView ...
    app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"
    android:orientation="horizontal" />
```

## Use GridLayoutManager to display items in a grid

If you want to arrange items in a grid, try using a `GridLayoutManager`. Use `app:spanCount` to specify how many columns the grid should have:

```
<androidx.recyclerview.widget.RecyclerView ...
    app:layoutManager="androidx.recyclerview.widget.GridLayoutManager"
    app:spanCount="2" />
```

This uses a grid layout manager with a span count of 2. The span count tells the layout manager how many columns there should be.

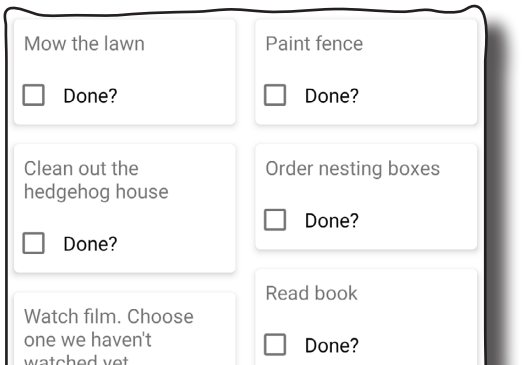



## Arrange items in a staggered grid

If your items have unequal sizes, you can use a `StaggeredGridLayoutManager` instead like this:

```
<androidx.recyclerview.widget.RecyclerView ...
    app:layoutManager="androidx.recyclerview.widget.StaggeredGridLayoutManager"
    app:spanCount="2" />
```

A staggered grid layout manager is good for items with unequal sizes.

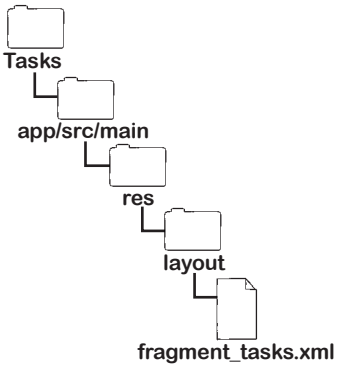



Let's apply one of these styles to the Tasks app's recycler view, and see what happens when the app runs.

# Update fragment_tasks.xml to arrange items in a grid

We're going to update the recycler view so that it arranges its items in a grid with two columns. The updated layout code for this change is shown below; update *fragment_tasks.xml* to include these changes (in bold):

```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".TasksFragment">

    <data>
        <variable
            name="viewModel"
            type="com.hfad.tasks.TasksViewModel" />
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <EditText
            ... />

        <Button
            ... />

        <androidx.recyclerview.widget.RecyclerView
            android:id="@+id/tasks_list"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1"
            android:gravity="top"
            app:layoutManager="androidx.recyclerview.widget.~~Linear~~GridLayoutManager"
            app:spanCount="2" />
    </LinearLayout>
</layout>
```




Use a grid layout manager instead of a linear one.

The grid will have two columns.

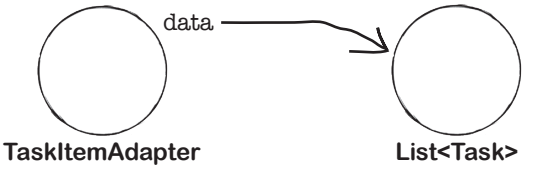
**Task list**
**Grid of cards**

# What happens when the code runs

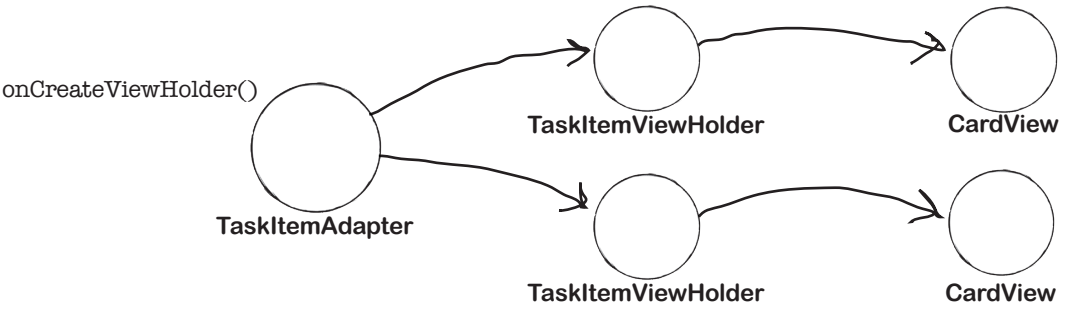
The following things happen when the app runs:

1. **TasksFragment creates a TaskItemAdapter object, and assigns it to the recycler view as its adapter.**

2. **TasksFragment sets the TaskItemAdapter's data property to a List<Task>.**
`TasksFragment` gets this `List<Task>` by observing `TasksViewModel`'s `tasks` property.

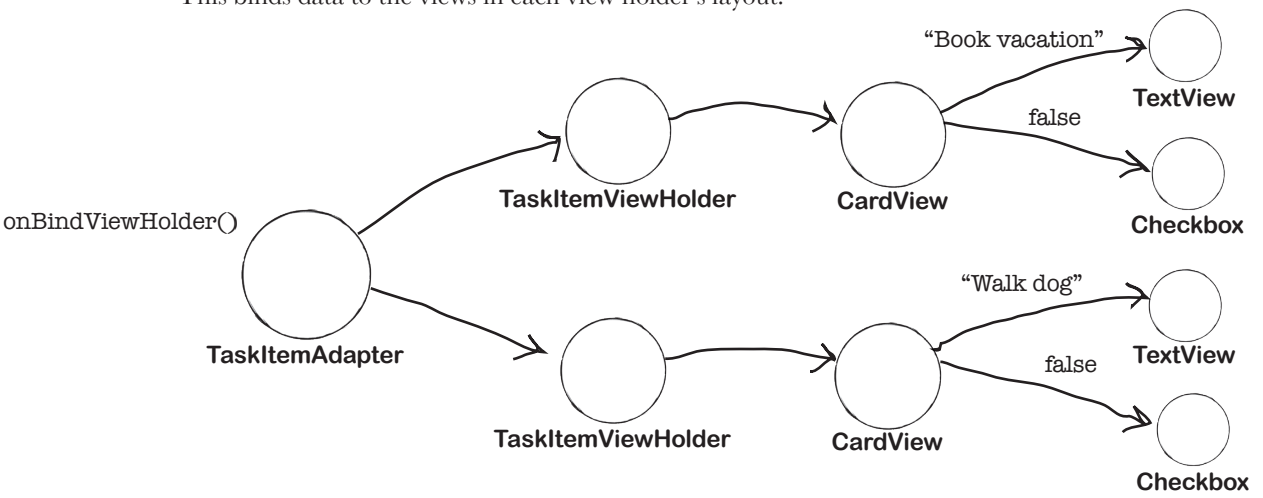



3. **TaskItemAdapter's onCreateViewHolder() method is called for each item that needs to be displayed in the recycler view.**
This creates a `TaskItemViewHolder` for each item. A layout (defined by *task_item.xml*) is inflated for each view holder.

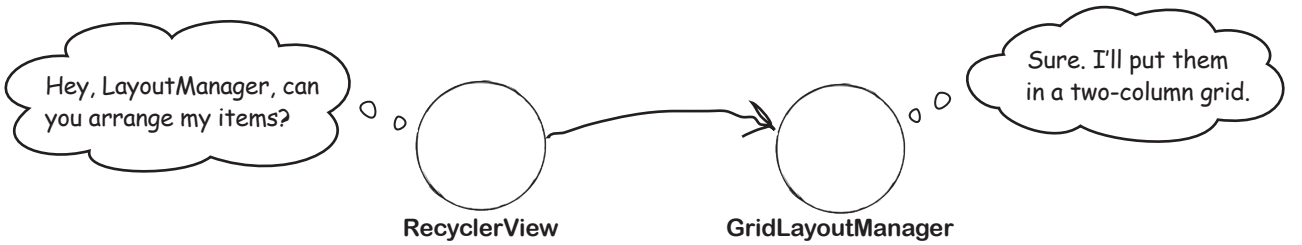


# The story continues

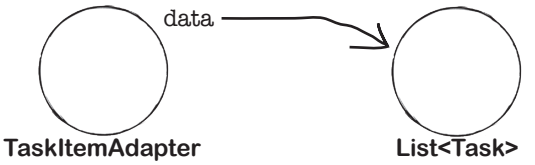
**4** **TaskItemAdapter's onBindViewHolder() is called for each TaskItemViewHolder.**
This binds data to the views in each view holder's layout.




**5** **The recycler view uses its layout manager to arrange its items.**
As the recycler view uses a `GridLayoutManager` with a `spanCount` of 2, it arranges items in a grid with two columns.




**6** **Each time TasksViewModel's tasks property gets updated, TasksFragment passes an updated List<Task> to the TaskItemAdapter.**
Steps 3 to 5 are repeated so the recycler view stays up-to-date.




Let's take the app for a test drive.

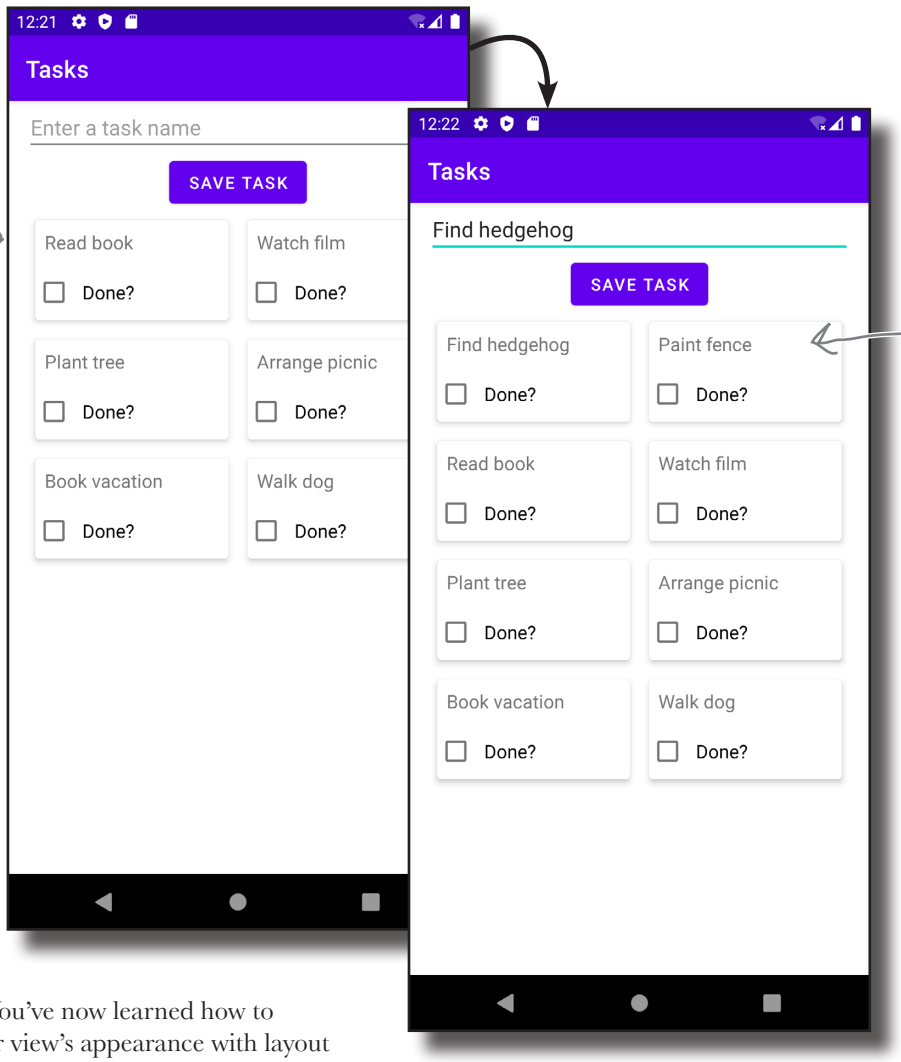
# Test Drive

**Task list**
**Grid of cards**

When we run the app, `TasksFragment`'s recycler view displays a grid of cards, each one showing a task name and whether the task has been completed.

When we enter new tasks, these get added to the recycler view as soon as we click on the Save Task button.

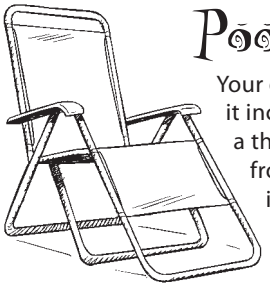
The tasks are displayed in a grid of cards.




Any new tasks we enter get added to the recycler view.

Congratulations! You've now learned how to control the recycler view's appearance with layout managers, and display data in a scrollable grid of cards.

In the next chapter, we'll build on this knowledge to make further improvements to the recycler view.

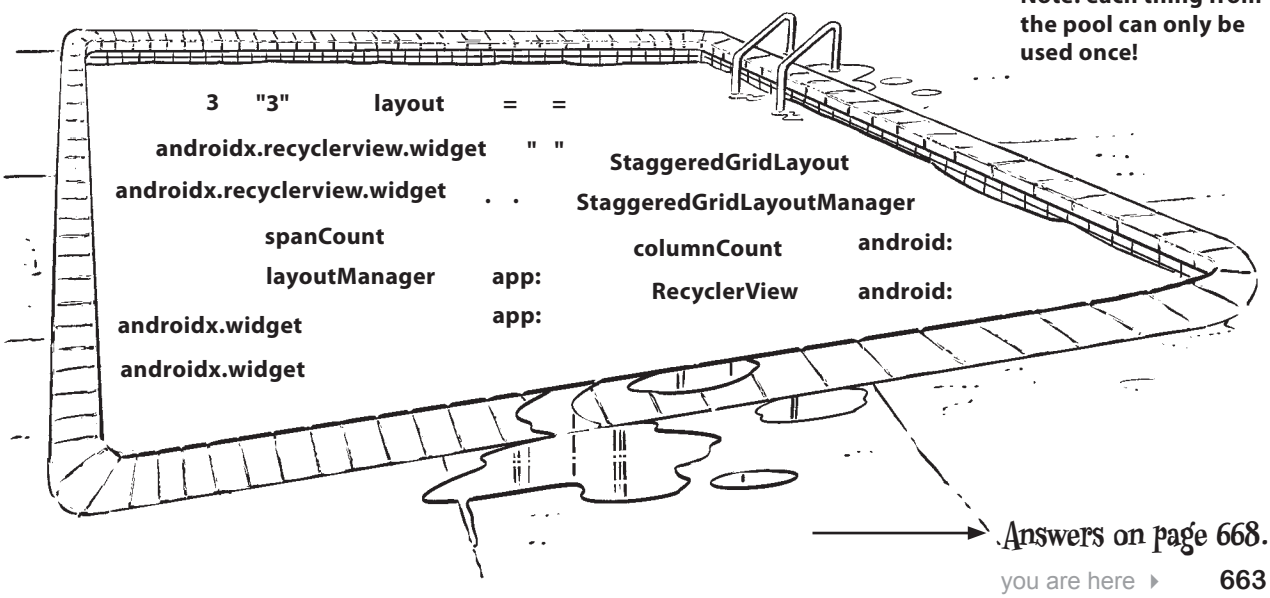


# Pool Puzzle

Your **goal** is to update the layout code below so that it includes a recycler view that displays its items in a three-column staggered grid. Take code snippets from the pool and place them into the blank lines in the code. You may **not** use the same snippet more than once, and you won't need to use all the snippets.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <.....
        android:id="@+id/recycler"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        ..........
        ..... />
</LinearLayout>
```

**Note: each thing from the pool can only be used once!**




Answers on page 668.

# Exercise

The Bits and Pizzas app includes a `Pizza` data class that looks like this:

```
package com.hfad.bitsandpizzas

data class Pizza(
    var pizzaId: Long = 0L,
    var pizzaName: String = "",
    var pizzaDescription: String = "",
    var pizzaImageId: Int = 0
)
```

The app needs to include a recycler view that displays the `pizzaName` and `PizzaDescription` properties for each `Pizza` item in the following linear layout (named *pizza_item.xml*):

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

    <TextView
        android:id="@+id/pizza_name"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <TextView
        android:id="@+id/pizza_description"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
 </LinearLayout>
```

See if you can update the adapter code on the next page so that it displays the correct properties in the above layout.

```
...
class PizzaAdapter : RecyclerView.Adapter<PizzaAdapter.PizzaViewHolder>() {
    var pizzas = listOf<Pizza>()
        set(value) {
            field = value
            notifyDataSetChanged()
        }

    override fun getItemCount() = pizzas.size

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                        : PizzaViewHolder = PizzaViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: PizzaViewHolder, position: Int) {
        val item = pizzas[position]
        holder.bind(item)
    }

    class PizzaViewHolder(val rootView: TextView)
                                        : RecyclerView.ViewHolder(rootView) {

        companion object {
            fun inflateFrom(parent: ViewGroup): PizzaViewHolder {
                val layoutInflater = LayoutInflater.from(parent.context)
                val view = layoutInflater.inflate(R.layout.pizza_item, parent, false)
                            as TextView
                return PizzaViewHolder(view)
            }
        }

        fun bind(item: Pizza) {
            rootView.text = item.pizzaName
        }
    }
}
```

This is the adapter code that you updated in an earlier exercise. Can you get it to work with the new version of pizza_item.xml?

# Exercise Solution

The Bits and Pizzas app includes a `Pizza` data class that looks like this:

```
package com.hfad.bitsandpizzas

data class Pizza(
    var pizzaId: Long = 0L,
    var pizzaName: String = "",
    var pizzaDescription: String = "",
    var pizzaImageId: Int = 0
)
```

The app needs to include a recycler view that displays the `pizzaName` and `PizzaDescription` properties for each `Pizza` item in the following linear layout (named *pizza_item.xml*):

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

    <TextView
        android:id="@+id/pizza_name"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <TextView
        android:id="@+id/pizza_description"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
 </LinearLayout>
```

See if you can update the adapter code on the next page so that it displays the correct properties in the above layout.

```
...
class PizzaAdapter : RecyclerView.Adapter<PizzaAdapter.PizzaViewHolder>() {
    var pizzas = listOf<Pizza>()
        set(value) {
            field = value
            notifyDataSetChanged()
        }

    override fun getItemCount() = pizzas.size

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                        : PizzaViewHolder = PizzaViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: PizzaViewHolder, position: Int) {
        val item = pizzas[position]
        holder.bind(item)
    }

    class PizzaViewHolder(val rootView: ~~TextView~~ LinearLayout)
                                        : RecyclerView.ViewHolder(rootView) {
        val pizzaName = rootView.findViewById<TextView>(R.id.pizza_name)
        val pizzaDescription = rootView.findViewById<TextView>(R.id.pizza_description)

        companion object {
            fun inflateFrom(parent: ViewGroup): PizzaViewHolder {
                val layoutInflater = LayoutInflater.from(parent.context)
                val view = layoutInflater.inflate(R.layout.pizza_item, parent, false)
                           as ~~TextView~~ LinearLayout
                return PizzaViewHolder(view)
            }
        }

        fun bind(item: Pizza) {
            ~~rootView~~ pizzaName.text = item.pizzaName
            pizzaDescription.text = item.pizzaDescription
        }
    }
}
```

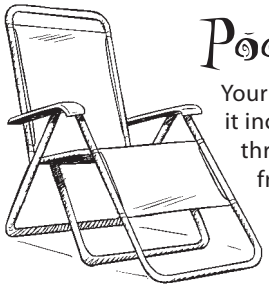
Don't worry if your code looks a little different from ours. This is just one possible correct answer.

The layout's root view is a LinearLayout.

Get references to the layout's views.

Cast view as a LinearLayout to match the layout's root view.

Add data to the layout's views.



# Pool Puzzle Solution

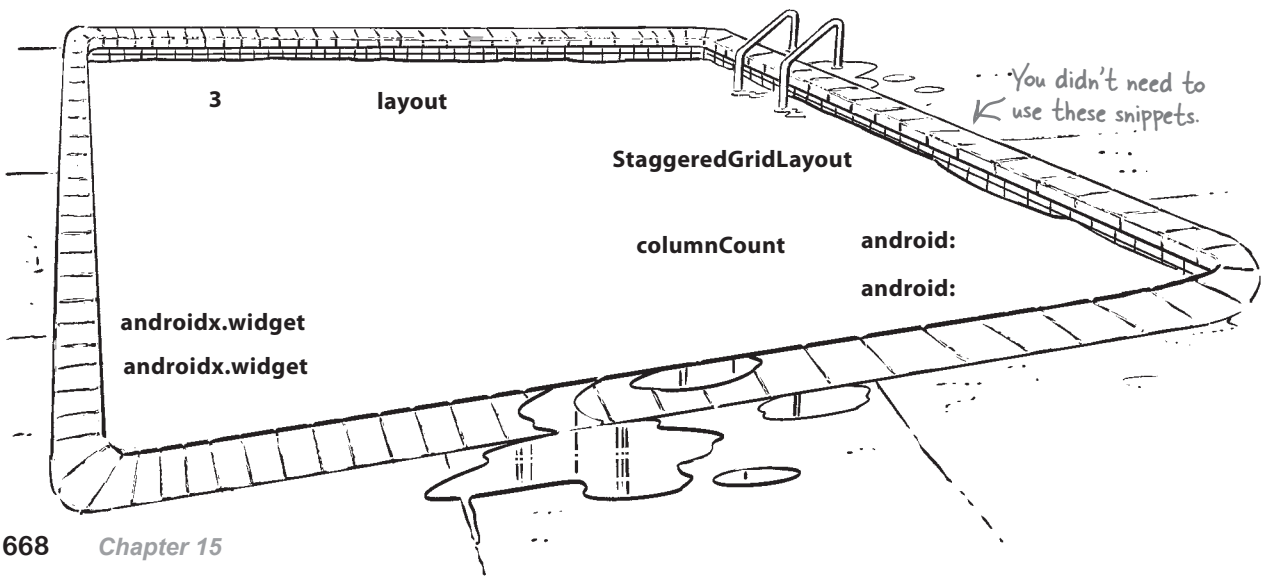
Your **goal** is to update the layout code below so that it includes a recycler view that displays its items in a three-column staggered grid. Take code snippets from the pool and place them into the blank lines in the code. You may **not** use the same snippet more than once, and you won't need to use all the snippets.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    < androidx.recyclerview.widget . RecyclerView .
        android:id="@+id/recycler"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app: layoutManager = " androidx.recyclerview.widget . StaggeredGridLayoutManager "
        app: spanCount = "3" />
</LinearLayout>
```

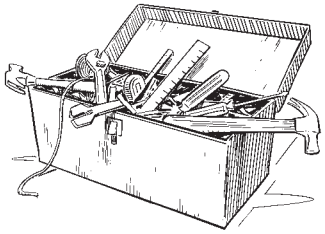
Use this element to add the recycler view.

Use this type of layout manager to display items in a staggered grid.

Make the staggered grid have three columns.



# Your Android Toolbox



**You've got Chapter 15 under your belt and now you've added recycler views to your toolbox.**

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

## Bullet Points

- A recycler view is a flexible way of displaying a scrollable list of data.
- Add a dependency to the recycler view library in the app's *build.gradle* file.
- For each recycler view, you define a layout for its items, and an adapter that puts data into each item's views. The recycler view then displays these items.
- Each adapter needs to extend `RecyclerView.Adapter`.
- The adapter uses a view holder that extends `RecyclerView.ViewHolder`. It contains information about how each item's layout should be displayed, and it's usually defined as an inner class of the adapter.
- The adapter's `onCreateViewHolder()` method inflates each item's layout, and creates view holders.
- The adapter's `onBindViewHolder()` method binds data to the views in each item's layout.
- Add a recycler view to a layout using the `<androidx.recyclerview.widget.RecyclerView>` element.
- A card view is a type of frame layout that has rounded corners and elevation.
- Add a dependency to the card view library in the app's *build.gradle* file.
- Use a layout manager to control how items are displayed.
- A `LinearLayoutManager` arranges items in a vertical list or horizontal row.
- A `GridLayoutManager` arranges items in a grid.
- A `StaggeredGridLayoutManager` is like a `GridLayoutManager` except the items can have unequal sizes.

# 16 diffutil and data binding

# Life in the Fast Lane

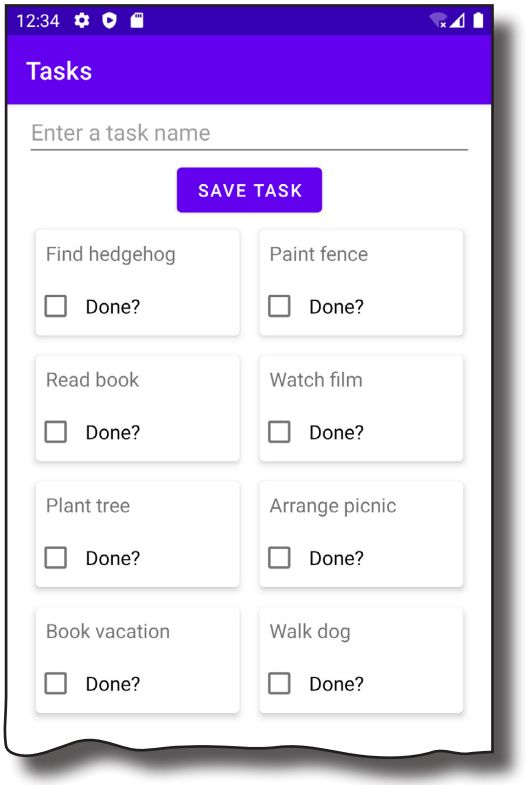
**Your app needs to run as smoothly and efficiently as possible.**

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**.

# The recycler view displays task data correctly...

In the previous chapter, we added a recycler view to the Tasks app that displays its data exactly how we want it to. Each task is shown in a separate card, and each card displays the task name and whether it has been completed. The cards are then laid out in a two-column grid like this:

This is the recycler view that we added to TasksFragment. It displays task records in a two-column grid.




## ...but when the data's updated, the recycler view jumps

Each time we add a new task, the recycler view gets redrawn so that it includes the new record and stays up-to-date. When it does this, however, the recycler view jumps and it's not very smooth.

Each time the recycler view needs to be updated, **its entire list gets redrawn**. There are no smooth transitions to indicate what's changed, and if the list is very long, the user might lose her place. It's also inefficient for large data sets, and can lead to performance problems.

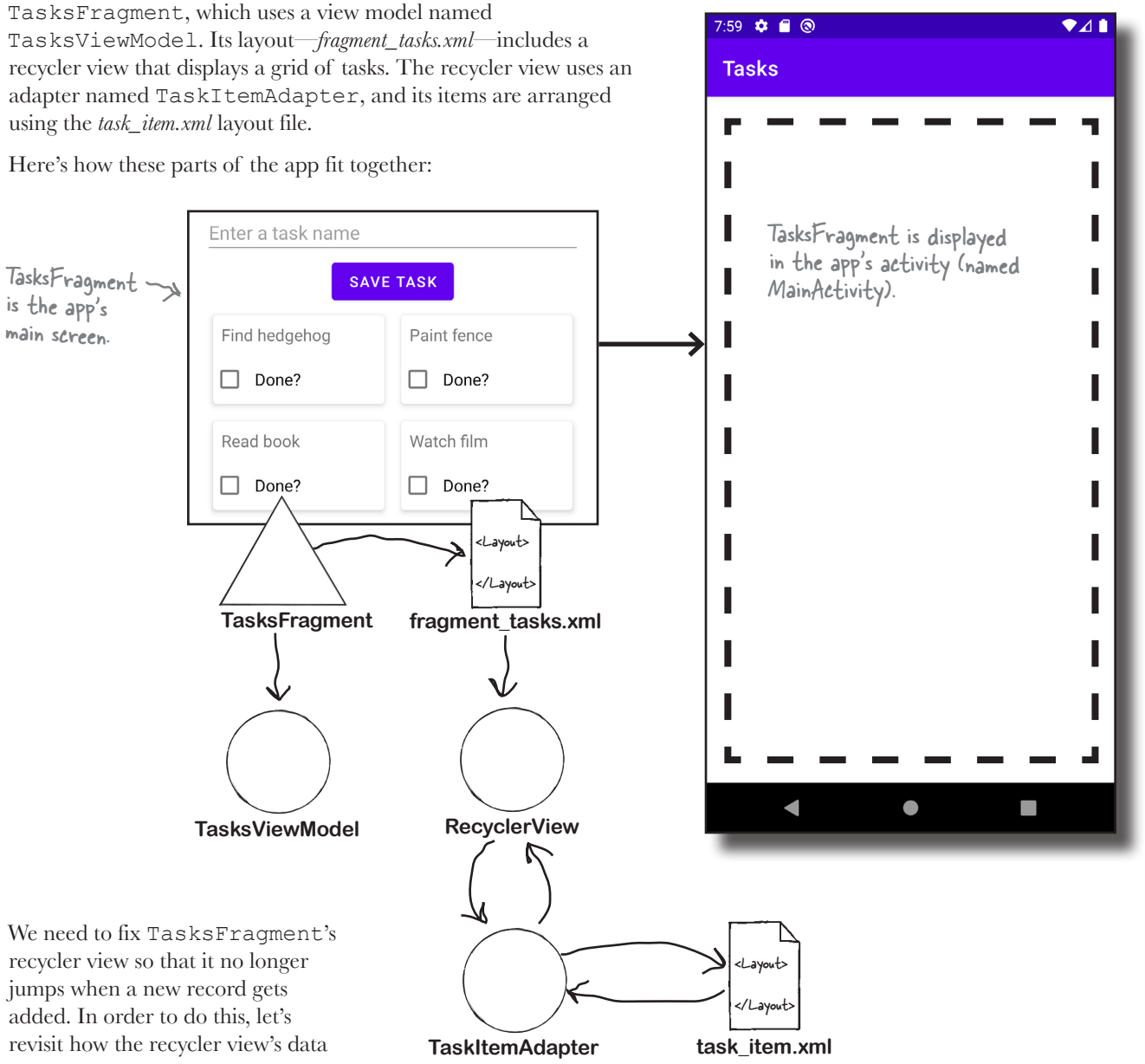
Before we solve these problems, let's have a quick reminder of how the Tasks app is structured.

# The Tasks app revisited

As we're sure you remember, the Tasks app lets the user enter task records, which are stored in a Room database. It includes a recycler view, which displays all of the records that have been entered.

The app's main screen is defined by a fragment named `TasksFragment`, which uses a view model named `TasksViewModel`. Its layout—*fragment_tasks.xml*—includes a recycler view that displays a grid of tasks. The recycler view uses an adapter named `TaskItemAdapter`, and its items are arranged using the *task_item.xml* layout file.

Here's how these parts of the app fit together:

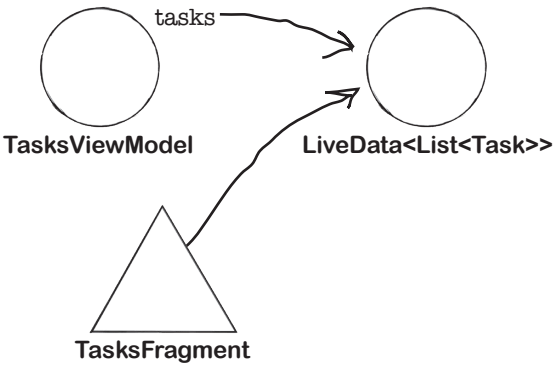



We need to fix `TasksFragment`'s recycler view so that it no longer jumps when a new record gets added. In order to do this, let's revisit how the recycler view's data gets set.

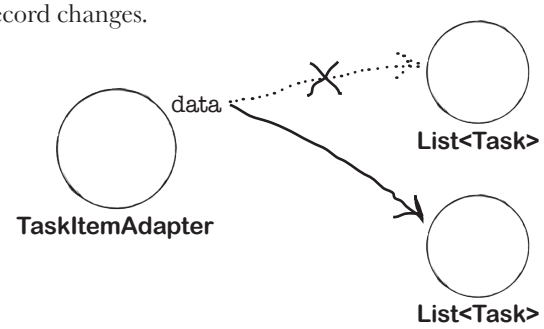
# How the recycler view gets its data

The following things happen each time the recycler view's data needs to be updated:

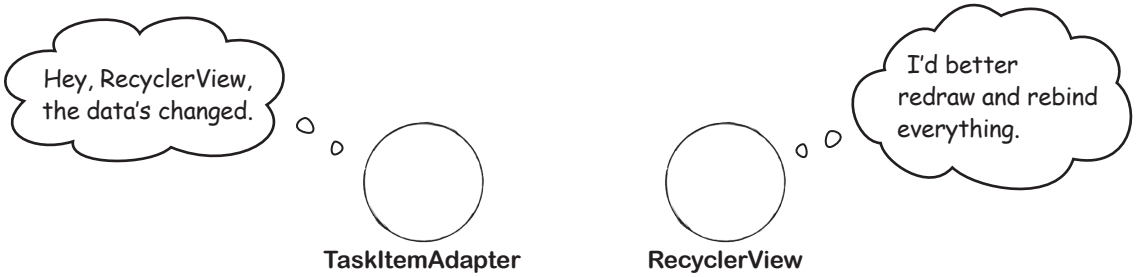
1. **TasksFragment is notified when a record is added to the database.**
This happens because it observes `TasksViewModel`'s `tasks` property: a `LiveData<List<Task>>` that gets its data from the database.




2. **TasksFragment sets TaskItemAdapter's data property, which holds the recycler view's data.**
It sets it to a new `List<Task>` (that it gets from the `tasks` property) that includes the latest record changes.




3. **TaskItemAdapter tells the recycler view that the data has changed.**
The recycler view responds by redrawing and rebinding each item in the list.



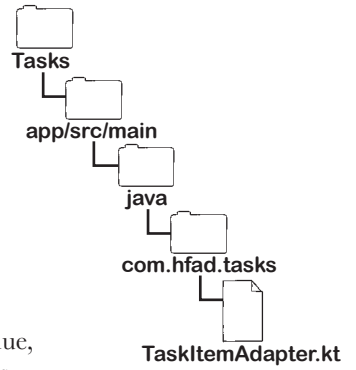
# The data property's setter calls notifyDataSetChanged()

The recycler view redraws and rebinds its entire list because of the setter, which we added to `TaskItemAdapter`'s `data` property. Here's a reminder of the code:

```
class TaskItemAdapter : RecyclerView.Adapter<TaskItemAdapter.TaskItemViewHolder>() {
    var data = listOf<Task>()
        set(value) {
            field = value
            notifyDataSetChanged()
        }
    ...
}
```

This is the data property's setter.

This method tells the recycler view when the data's changed.




The setter gets called each time the `data` property needs to be updated. As you can see, it sets the `data` property to the new value, and then calls `notifyDataSetChanged()`. This method tells any observers—including the recycler view—that the data set has changed, so the recycler view is redrawn to include the latest changes.

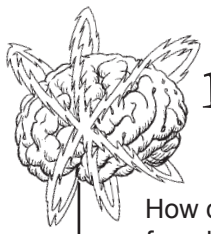
## notifyDataSetChanged() redraws the entire list

Using `notifyDataSetChanged()`, however, is problematic. Each time it's called, it tells the recycler view that the `data` property is out of date in some way, *without specifying how*. As the recycler view doesn't know what's changed, it responds by **rebinding and redrawing every item in its list**.

When the entire recycler view rebinds and redraws its items in this way, it loses track of the user's position in the list. If the list contains more than a few records, this can cause the list to jump.

It's also inefficient for large data sets. If the recycler view contains many items, rebinding and redrawing each item is a lot of unnecessary work and can lead to performance problems.

**Each time notifyDataSetChanged() gets called, the recycler view rebinds and redraws its entire list. This is inefficient, particularly for large data sets.**



## Brain Power

How do you think we could stop the recycler view's entire list from being redrawn each time the user enters a new record?

# Tell the recycler view what needs to change

A more efficient alternative to calling `notifyDataSetChanged()` is to tell the recycler which items in the list have changed so that it only updates those items. If a new task record gets added to the database, for example, it's more efficient for the recycler view to just add that item than it is to rebind and redraw the entire list.

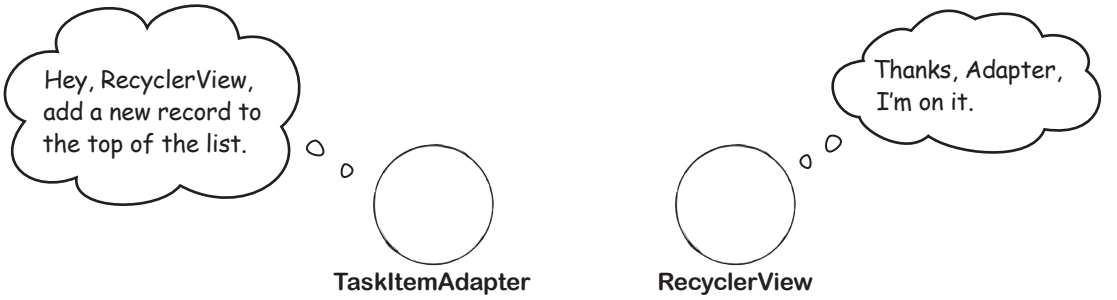
Manually working out these differences can be tricky and require a lot of code. The good news, however, is that the recycler view library includes a utility class named **`DiffUtil`** that takes care of all this hard work for you.

## DiffUtil works out differences between lists

The `DiffUtil` class specializes in finding out what the differences are between two lists so that you don't have to.

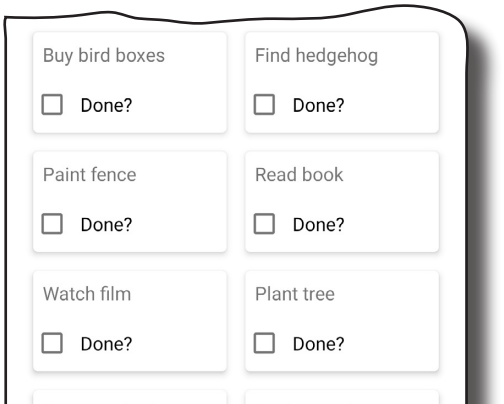
Each time an adapter is given a new version of the list its recycler view uses, `DiffUtil` compares it with the old version. It figures out what items have been added, removed, or updated, and it then tells the recycler view what needs to be changed in the most efficient way possible:

To be precise, it works out what the changes are using Eugene W. Myers's clever difference algorithm.




As the recycler view no longer needs to redraw and rebind its entire list, using `DiffUtil` is a much more efficient way of updating a recycler view's data. It also means that the user doesn't lose her place in the list, and the recycler view can even provide smooth transition animations to make it clear what changes there have been.

When the user enters a new record, it gets added to the list. The other items shuffle around to accommodate it.

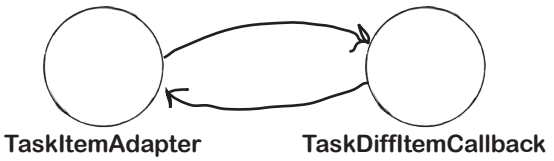


# Here's what we're going to do

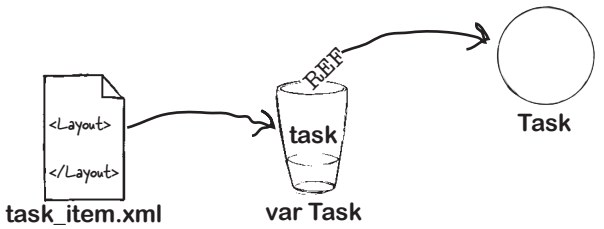
In this chapter, we're going to make improvements to the Tasks app's recycler view so that it uses `DiffUtil` and populates its views using data binding. These changes will make the recycler view more efficient, and improve the user's experience of it too.

Here are the steps we'll take:

1. **Make the recycler view use DiffUtil.**
We'll create a new class named `TaskDiffItemCallback` that uses `DiffUtil` to compare items in a list. We'll then update the `TaskItemAdapter` code so that it uses this new class. These changes will make the recycler view more efficient, and give the user a smoother experience when they use it.




2. **Implement data binding in the recycler view's layout.**
We'll remove the calls to `findViewById()` in the `TaskItemAdapter` code, and populate each item's views using data binding.




We'll begin by making the recycler view use `DiffUtil`.

**Don't forget!**

You're going to modify the Tasks app in this chapter, so make sure you open the project for this app.

**DiffUtil**
**Data binding**

# We need to implement DiffUtil.ItemCallback

In order to use `DiffUtil` with the Tasks app's recycler view, we need to create a new class (which we'll name `TaskDiffItemCallback`) that implements the **`DiffUtil.ItemCallback`** abstract class. This class is used to calculate the difference between two non-null items in a list, and will help make the recycler view more efficient.

When you implement `DiffUtil.ItemCallback`, you first need to specify the type of objects it works with. This is done using generics like so:

This makes the class work with Task objects.

```
class TaskDiffItemCallback : DiffUtil.ItemCallback<Task>() {...}
```

You also need to override two methods: `areItemsTheSame()` and `areContentsTheSame()`.

**`areItemsTheSame()`** is used to check whether two objects that are passed to it refer to the same *item*. We'll implement it using:

If oldItem and newItem refer to the same item, this method needs to return true.

```
override fun areItemsTheSame(oldItem: Task, newItem: Task)
    = (oldItem.taskId == newItem.taskId)
```

so that if both objects have the same `taskId`, it means they refer to the same item, and the method returns *true*.

**`areContentsTheSame()`** is used to check whether the two objects have the same *contents*, and it's only called if `areItemsTheSame()` is *true*. As `Task` is a data class, we can implement this method using:

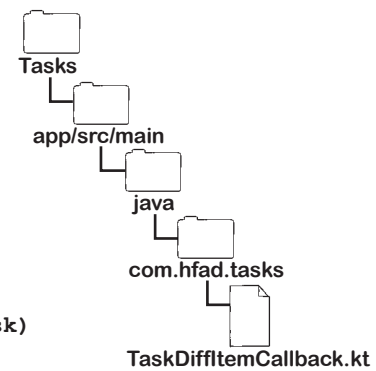
If oldItem and newItem have matching contents, this method needs to return true. As Task is a data class, oldItem == newItem if their property values match.

```
override fun areContentsTheSame(oldItem: Task, newItem: Task) = (oldItem == newItem)
```

## Create TaskDiffItemCallback.kt

To create the new class, highlight the *com.hfad.tasks* package in the *app/src/main/java* folder, then go to File→New→Kotlin Class/File. Name the file "TaskDiffItemCallback" and choose the Class option.

Once you've created the file, update its code so that it looks like this:




```
package com.hfad.tasks

import androidx.recyclerview.widget.DiffUtil

class TaskDiffItemCallback : DiffUtil.ItemCallback<Task>() {
    override fun areItemsTheSame(oldItem: Task, newItem: Task)
        = (oldItem.taskId == newItem.taskId)

    override fun areContentsTheSame(oldItem: Task, newItem: Task) = (oldItem == newItem)
}
```

# A ListAdapter accepts a DiffUtil.ItemCallback argument

Now that we've defined `TaskDiffItemCallback`, we need to use it in the adapter code. To do this, we'll update `TaskItemAdapter` so that it extends the **`ListAdapter`** class instead of `RecyclerView.Adapter`.

`ListAdapter` is a type of `RecyclerView.Adapter` that's designed to work with lists. It provides its own backing list so that you don't have to define your own, and it accepts a `DiffUtil.ItemCallback` in its constructor.

We're going to specify that `TaskItemAdapter` is a type of `ListAdapter` that provides its own `List<Task>`, and we'll pass it an instance of `TaskDiffItemCallback`. Here's the code to do this:

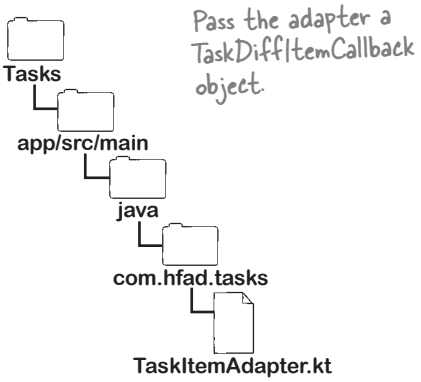
**A ListAdapter is a type of RecyclerView.Adapter that provides its own backing list. It works well with DiffUtil.**

```
...
import androidx.recyclerview.widget.ListAdapter

class TaskItemAdapter
        : ListAdapter<Task, TaskItemAdapter.TaskItemViewHolder>(TaskDiffItemCallback()) {
    ...
}
```

It uses the ListAdapter class from the androidx.recyclerview library.

This turns TaskItemAdapter into a List Adapter that works with Tasks and TaskItemViewHolders.

Pass the adapter a TaskDiffItemCallback object.




## We can simplify the rest of TaskItemAdapter's code

Once `TaskItemAdapter` has been changed to extend `ListAdapter`, we can remove its `List<Task> data` property, along with its setter. This property is no longer needed because `ListAdapter`s come with their own backing list, so you don't need to define your own.

We can also delete `TaskItemAdapter`'s `getItemCount()` method. This was needed when the adapter extended `RecyclerView.Adapter`, but `ListAdapter` provides its own implementation, so it's no longer required.

Finally, we need to update the adapter's `onBindViewHolder()` method so that instead of using:

```
val item = data[position]
```

to get the item at a certain position from the `data` property, it uses:

```
val item = getItem(position)
```

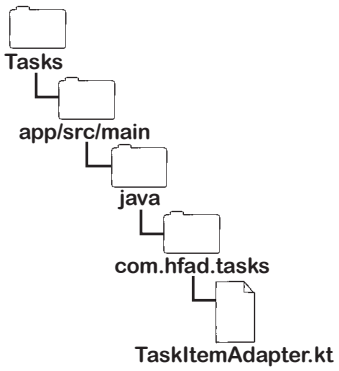
This gets the item at the specified position from the adapter's backing list.

We'll show you all of the code for this on the next page.

**DiffUtil**
**Data binding**

# The updated code for TaskItemAdapter.kt

Here's the updated `TaskItemAdapter` code; update the code in *TaskItemAdapter.kt* to include all the changes shown here:




```
package com.hfad.tasks

import android.view.LayoutInflater
import android.view.ViewGroup
import android.widget.CheckBox
import androidx.recyclerview.widget.ListAdapter
import android.widget.TextView
import androidx.cardview.widget.CardView
import androidx.recyclerview.widget.RecyclerView

class TaskItemAdapter : RecyclerView.Adapter<TaskItemAdapter.TaskItemViewHolder>() {
    : ListAdapter<Task, TaskItemAdapter.TaskItemViewHolder>(TaskDiffItemCallback()) {
    var data = listOf<Task>()
        set(value) {
            field = value
            notifyDataSetChanged()
        }

    override fun getItemCount() = data.size

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                        : TaskItemViewHolder = TaskItemViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: TaskItemViewHolder, position: Int) {
        val item = data[position] getItem(position)
        holder.bind(item)
    }

    class TaskItemViewHolder(val rootView: CardView) : RecyclerView.ViewHolder(rootView) {
        ...
    }
}
```

Import this class. Android has a few different ListAdapter classes, so make sure you import the correct one.

Remove the data property as it's no longer needed.

Change the class definition so that TaskItemAdapter extends ListAdapter instead of RecyclerView.Adapter.

We no longer need this method.

Get the item from the adapter's backing list.

We've left out details of the view holder code as it doesn't need to change.

# Populate the ListAdapter's list...

The final thing that we need to do is pass a list of `Task` records to `TaskItemAdapter`'s backing list.

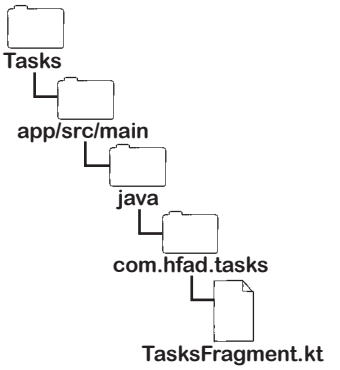
Previously, we did this by making `TasksFragment` observe `TasksViewModel`'s `tasks` property. Each time the property changes, the fragment updates the `TaskItemAdapter`'s `data` property to the `tasks` property's new value.

Here's a reminder of the code we used to do this:

```
viewModel.tasks.observe(viewLifecycleOwner, Observer {
    it?.let {
        adapter.data = it
    }
})
```

This sets the adapter's data property to a new list of tasks.

Now that the adapter uses a backing list instead of the `data` property, we need to use a slightly different approach.




## ...using submitList()

To pass a list of tasks to `TaskItemAdapter`'s backing list, we'll use a method called `submitList()`. This method is used to update a `ListAdapter`'s backing list with a new `List` object, so it's perfect for this situation.

Here's the new code that we need to add to `TasksFragment` (in bold), which we'll add on the next page:

```
viewModel.tasks.observe(viewLifecycleOwner, Observer {
    it?.let {
        ~~adapter.data = it~~
        adapter.submitList(it)
    }
})
```

We'll use the submitList() method to pass new data to the adapter's backing list.

When the adapter receives the new list, it uses the `TaskDiffItemCallback` class to compare it with the old version. It then updates the recycler view with the differences instead of replacing the entire list. This approach is more efficient, and leads to a much smoother user experience.

Let's see what the updated `TasksFragment` code looks like.

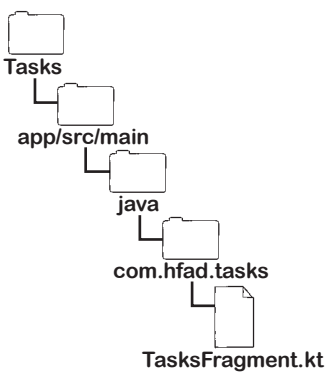
**DiffUtil**
**Data binding**

# The updated code for TasksFragment.kt

Here's the updated `TasksFragment` code; update the code in *TasksFragment.kt* to include all the changes shown here (in bold):

```
...
class TasksFragment : Fragment() {
    private var _binding: FragmentTasksBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
            inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentTasksBinding.inflate(inflater, container, false)
        val view = binding.root

        val application = requireNotNull(this.activity).application
        val dao = TaskDatabase.getInstance(application).taskDao
        val viewModelFactory = TasksViewModelFactory(dao)
        val viewModel = ViewModelProvider(
                this, viewModelFactory).get(TasksViewModel::class.java)
        binding.viewModel = viewModel
        binding.lifecycleOwner = viewLifecycleOwner
        val adapter = TaskItemAdapter()
        binding.tasksList.adapter = adapter

        viewModel.tasks.observe(viewLifecycleOwner, Observer {
            it?.let {
                adapter.~~data = it~~ submitList(it)
            }
        })
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Update this line.

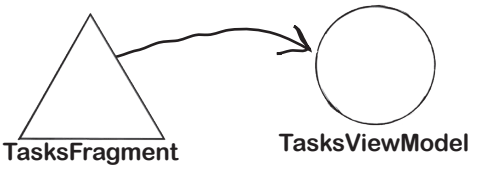



Let's run through what happens when the app runs.

# What happens when the code runs

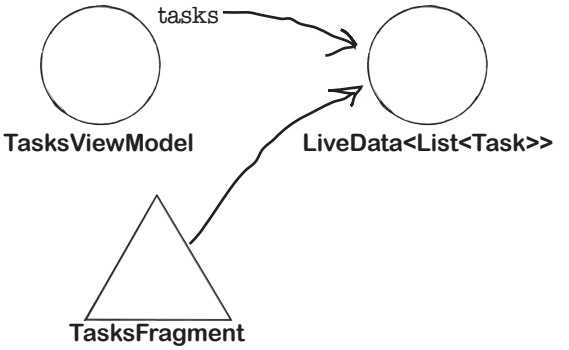
The following things happen when the app runs:

1. **When the app launches, MainActivity displays TasksFragment.**
`TasksFragment` uses `TasksViewModel` as its view model.




2. **TasksFragment creates a TaskItemAdapter object, and assigns it to the recycler view as its adapter.**

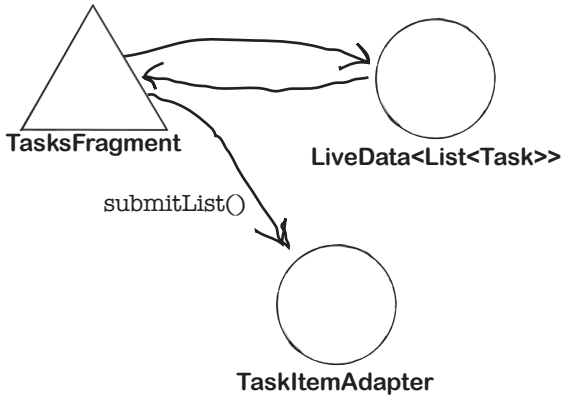
3. **TasksFragment observes TasksViewModel's tasks property.**
This property is a `LiveData<List<Task>>` that holds an up-to-date list of records from the database.



**DiffUtil**
**Data binding**

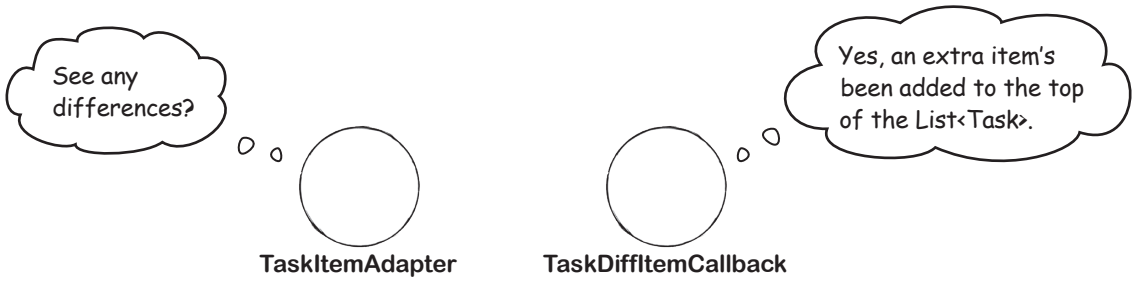
# The story continues

**4** **Each time the tasks property gets a new value, TasksFragment submits its List<Task> to the TaskItemAdapter.**



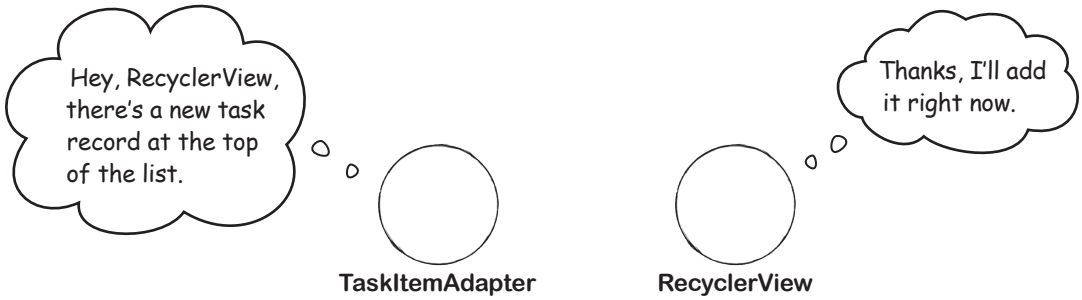

**5** **TaskItemAdapter uses TaskDiffItemCallback to compare its old data with the new.**

It uses `TaskDiffItemCallback`'s `areItemsTheSame()` and `areContentsTheSame()` methods to figure out what's changed.




**6** **TaskItemAdapter tells the recycler view what's changed.**

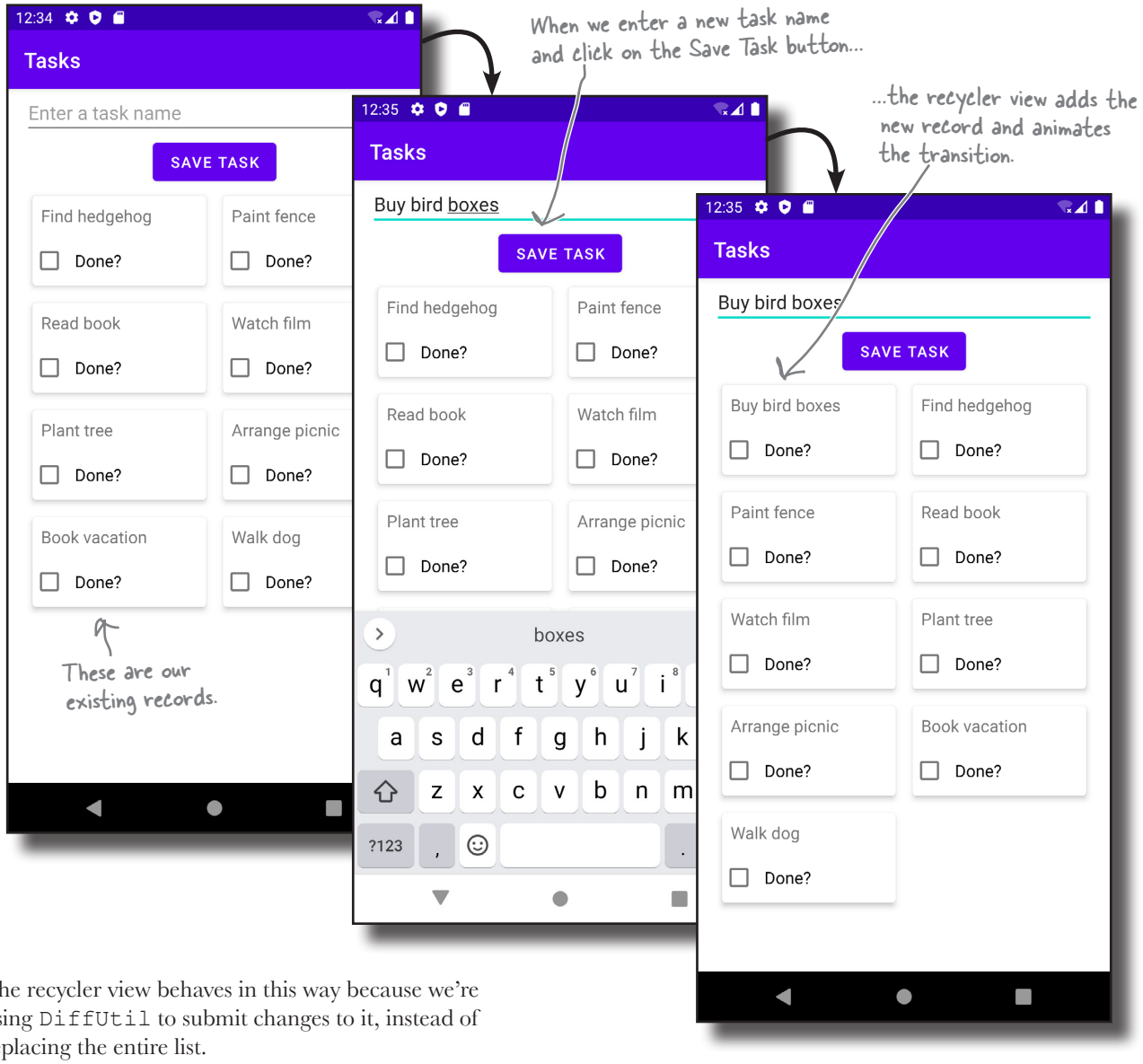
The recycler view rebinds and redraws the necessary items.



# Test Drive

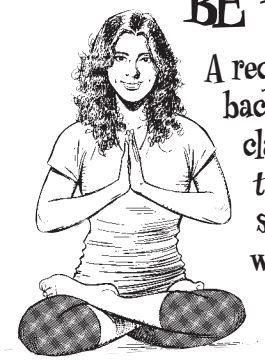
When we run the app, `TasksFragment` displays a grid of cards in the recycler view as before.

When we enter a new task name and click on the button, a new task card is added to the recycler view, and the existing cards shuffle along to accommodate it.




The recycler view behaves in this way because we're using `DiffUtil` to submit changes to it, instead of replacing the entire list.

# BE the ListAdapter



A recycler view's ListAdapter class has a backing list of Drinks. It uses the Drink class shown on the right. Your job is to play like you're the ListAdapter, and say if the ItemCallback classes below will correctly spot any changes when the ListAdapter is given a new list. Why? Why not?

Hint: the drinkId property is a unique identifier for each Drink.

```
class Drink(
    var drinkId: Long = 0L,
    var drinkName: String = ""
)
```

**A**

```
class DrinkDiffItemCallback : DiffUtil.ItemCallback<Drink>() {
    override fun areItemsTheSame(oldItem: Drink, newItem: Drink)
        = (oldItem.drinkId == newItem.drinkId)

    override fun areContentsTheSame(oldItem: Drink, newItem: Drink)
        = (oldItem == newItem)
}
```

**B**

```
class DrinkDiffItemCallback : DiffUtil.ItemCallback<Drink>() {
    override fun areItemsTheSame(oldItem: Drink, newItem: Drink)
        = (oldItem == newItem)

    override fun areContentsTheSame(oldItem: Drink, newItem: Drink)
        = ((oldItem.drinkId == newItem.drinkId) &&
           (oldItem.drinkName == newItem.drinkName))
}
```

**C**

```
class DrinkDiffItemCallback : DiffUtil.ItemCallback<Drink>() {
    override fun areItemsTheSame(oldItem: Drink, newItem: Drink)
        = (oldItem.drinkId == newItem.drinkId)

    override fun areContentsTheSame(oldItem: Drink, newItem: Drink)
        = (oldItem.drinkName == newItem.drinkName)
}
```

Answers on page 702.

# Recycler views can use data binding

Another way in which we can improve the Tasks app's recycler view is to make it use data binding.

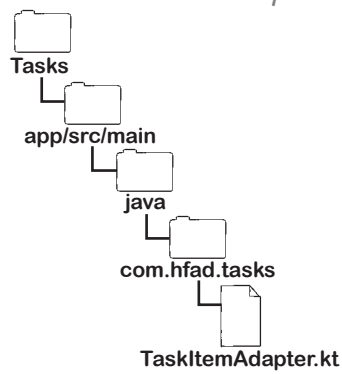
As you may recall, `TaskItemAdapter`'s `TaskItemViewHolder` inner class uses `findViewById()` to get a reference to the views for each item in the recycler view. The view holder's `bind()` method then uses these references to add data to each view.

Here's a reminder of this code:

```
class TaskItemViewHolder(val rootView: CardView)
                                        : RecyclerView.ViewHolder(rootView) {
    val taskName = rootView.findViewById<TextView>(R.id.task_name)
    val taskDone = rootView.findViewById<CheckBox>(R.id.task_done)
    ...

    fun bind(item: Task) {
        taskName.text = item.taskName
        taskDone.isChecked = item.taskDone
    }
}
```

This gets references to the views in the item's layout.

This populates the views with data.




If we change the recycler view to use data binding, we can remove the calls to `findViewById()`, and make each view fetch its own data.

## How we'll implement data binding

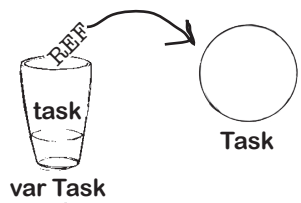
We'll make the recycler view use data binding in a similar way to how it's implemented with fragments. We'll go through the following steps:

1. **Add a data binding variable to task_item.xml.**
   We'll add a `<layout>` element to the layout's root, and create a data binding variable named `task` whose type is `Task`. This will generate a binding class named `TaskItemBinding`.

2. **Set the data binding variable in TaskItemAdapter.**
   We'll use `TaskItemBinding` to inflate each item's layout, and set its data binding variable to the `Task` object for that item.

3. **Use the data binding variable to set the views data.**
   Finally, we'll update *task_item.xml* so that each view fetches its data from the layout's `Task` object.

Let's begin by defining the data binding variable.




We'll make the recycler view use data binding by adding a Task data binding variable (named task) to the layout used by its items. Each view will then be able to get data from the layout's Task object.

**DiffUtil**
**Data binding**

# Add a data binding variable to task_item.xml

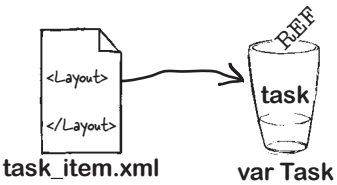
We'll begin by adding a `<layout>` element to *task_item.xml*'s root, and specifying a data binding variable. Instead of using it to bind views to a view model, however, we'll specify that its type is `Task`.

Here's the code to do this: update *task_item.xml* to include these changes (in bold):

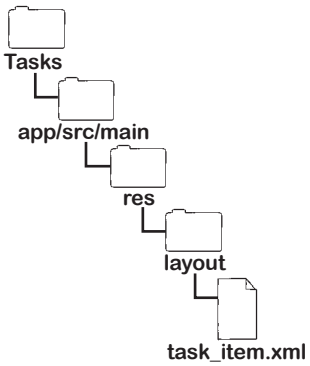
```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <data>
        <variable
            name="task"
            type="com.hfad.tasks.Task" />
    </data>

    <androidx.cardview.widget.CardView
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto">
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="8dp"
        app:cardElevation="4dp"
        app:cardCornerRadius="4dp" >

        ...

    </androidx.cardview.widget.CardView>
</layout>
```

Add a <layout> element.

Define a data binding variable named task.




Move these lines to the <layout>.




The <layout> element's closing tag

Making *task_item.xml*'s root element a `<layout>` tells Android that you want to use data binding with it, so it generates a new binding class named `TaskItemBinding`. We'll use this class to inflate the above layout, and set its `task` data binding variable to a `Task` object.

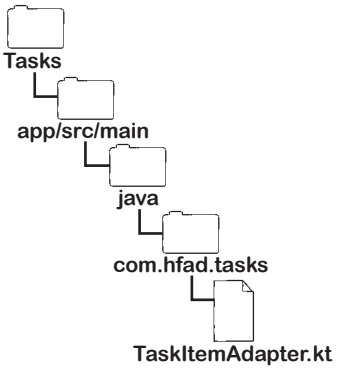
# The layout gets inflated in the adapter's view holder code

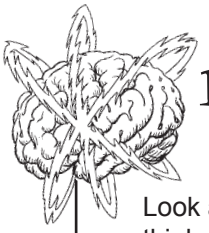
When we first created the recycler view, we inflated the layout file *task_item.xml* in `TaskItemAdapter`'s `TaskItemViewHolder` inner class. We now need to change this code so that it works with the binding class `TaskItemBinding`. Before we begin, here's a reminder of the current code.

```
class TaskItemAdapter...{
    ...
    class TaskItemViewHolder(val rootView: CardView)
                                        : RecyclerView.ViewHolder(rootView) {
        val taskName = rootView.findViewById<TextView>(R.id.task_name)
        val taskDone = rootView.findViewById<CheckBox>(R.id.task_done)

        companion object {
            fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
                val layoutInflater = LayoutInflater.from(parent.context)
                val view = layoutInflater
                           .inflate(R.layout.task_item, parent, false) as CardView
                return TaskItemViewHolder(view)
            }
        }

        fun bind(item: Task) {
            taskName.text = item.taskName
            taskDone.isChecked = item.taskDone
        }
    }
}
```

This inflates the layout and returns a TaskItemViewHolder object.

This adds data to each view.



## Brain Power

Look at the above `TaskItemViewHolder` code. What changes do you think we need to make in order to bind *task_item.xml* to a `Task` object?

**DiffUtil**
**Data binding**

# Use the binding class to inflate the layout

The first change that we'll make to `TaskItemViewHolder` is to inflate *task_item.xml* using the `TaskItemBinding` class. We'll do this in the view holder's `inflateFrom()` method like so:

```
fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
    val layoutInflater = LayoutInflater.from(parent.context)
    val view = layoutInflater.inflate(R.layout.task_item, parent, false) as CardView
    val binding = TaskItemBinding.inflate(layoutInflater, parent, false)
    return TaskItemViewHolder(view binding)
}
```

Use the binding variable to create a TaskItemViewHolder.

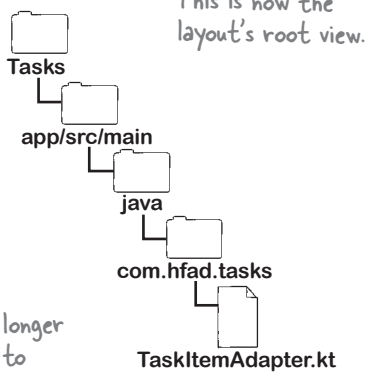
Inflate the layout using TaskItemBinding.

Notice that we're now passing the `binding` variable—a `TaskItemBinding` object—to `TaskItemViewHolder`'s constructor. This means that we also need to update `TaskItemViewHolder`'s class definition so that it looks like this:

```
class TaskItemViewHolder(val rootView: CardView) : RecyclerView.ViewHolder(rootView) {
class TaskItemViewHolder(val binding: TaskItemBinding)
        : RecyclerView.ViewHolder(binding.root) {
    ...
}
```

TaskItemViewHolder needs to accept a TaskItemBinding object.

This is now the layout's root view.




## Set the layout's data binding variable to a Task

Now that we've used the `TaskItemBinding` class to inflate *task_item.xml*, we can use it to set the `task` data binding variable. To do this, we'll change `TaskItemViewHolder`'s `bind()` method so that it sets `task` to the recycler view's current `Task` item like so:

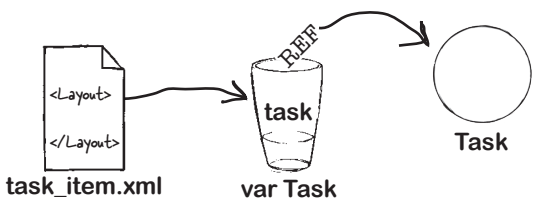
```
fun bind(item: Task) {
    taskName.text = item.taskName
    taskDone.isChecked = item.taskDone
    binding.task = item
}
```

The view holder no longer needs to add data to the layout's views.

Set the data binding variable to a Task.

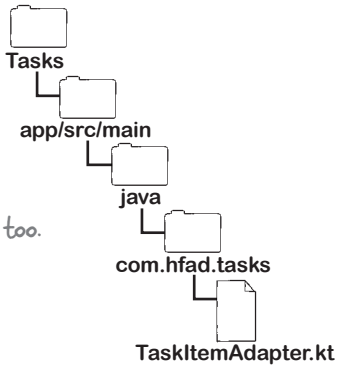
We've removed the lines that set the layout's `task_name` and `task_done` views because with data binding, these are no longer needed. This means we can also remove the `taskName` and `taskDone` properties from the view holder.

We'll show you the full code for `TaskItemAdapter` (including its `TaskItemViewHolder` inner class) on the next page.



# The full code for TaskItemAdapter.kt

Here's the updated `TaskItemAdapter` code; make sure that the code in *TaskItemAdapter.kt* includes all the changes shown here (in bold):




```
package com.hfad.tasks

import android.view.LayoutInflater
import android.view.ViewGroup
~~import android.widget.CheckBox~~
import androidx.recyclerview.widget.ListAdapter
~~import android.widget.TextView~~
~~import androidx.cardview.widget.CardView~~
import androidx.recyclerview.widget.RecyclerView
import com.hfad.tasks.databinding.TaskItemBinding

class TaskItemAdapter
      : ListAdapter<Task, TaskItemAdapter.TaskItemViewHolder>(TaskDiffItemCallback()) {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                          : TaskItemViewHolder = TaskItemViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: TaskItemViewHolder, position: Int) {
        val item = getItem(position)
        holder.bind(item)
    }

    class TaskItemViewHolder(val ~~rootView~~ binding: ~~CardView~~ TaskItemBinding)
                                  : RecyclerView.ViewHolder(~~rootView~~ binding.root) {
        ~~val taskName = rootView.findViewById<TextView>(R.id.task_name)~~
        ~~val taskDone = rootView.findViewById<CheckBox>(R.id.task_done)~~
```

Delete this line.

Delete these lines too.

Add this import.

We're now passing a TaskItemBinding to TaskItemViewHolder instead of a CardView.

Rename this.

Change this.

Delete these lines.

The code continues on the next page.

**DiffUtil**
**Data binding**

# TaskItemAdapter.kt continued

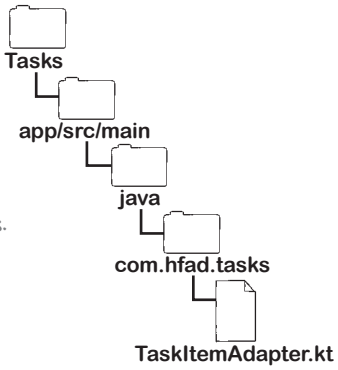
```
        companion object {
            fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
                val layoutInflater = LayoutInflater.from(parent.context)
                val view = layoutInflater
                        .inflate(R.layout.task_item, parent, false) as CardView
                val binding = TaskItemBinding.inflate(layoutInflater, parent, false)
                return TaskItemViewHolder(view binding)
            }
        }

        fun bind(item: Task) {
            taskName.text = item.taskName
            taskDone.isChecked = item.taskDone
            binding.task = item
        }
    }
}
```

Inflate the layout using TaskItemBinding.

Pass the binding variable to TaskItemViewHolder's constructor.

Delete these lines.

Set the layout's data binding variable.




## Use data binding to set the layout's views

Now that we've set *task_item.xml*'s `task` data binding variable to the view holder's `Task` item, we can use data binding to populate the layout's views.

You're already familiar with the code to do this. To set the `task_name` view's text to the name of the task, for example, we can use:

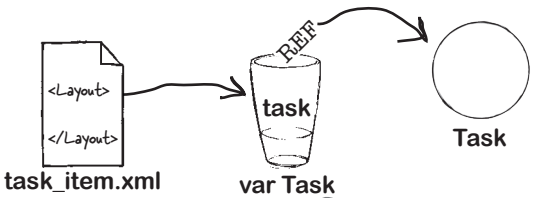
```
<TextView
    android:id="@+id/task_name"
    ...
    android:text="@{task.taskName}" />
```

Set the view's text to the task's name.

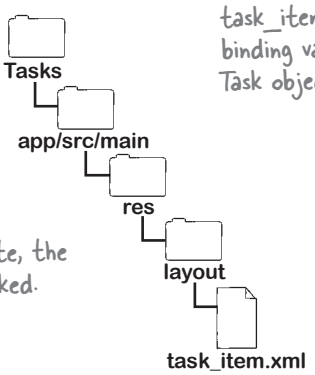
and to set the `task_done` checkbox, we can use:

```
<CheckBox
    android:id="@+id/task_done"
    ...
    android:checked="@{task.taskDone}" />
```

If the task is complete, the checkbox will be checked.




We've now set task_item.xml's data binding variable to a Task object.

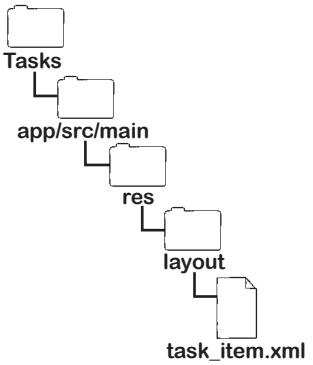



We'll show you the full code on the next page.

# The full code for task_item.xml

Here's the updated code for *task_item.xml*; update the code for this file so that it includes all the changes shown here (in bold):

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

    <data>
        <variable
            name="task"
            type="com.hfad.tasks.Task" />
    </data>

    <androidx.cardview.widget.CardView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="8dp"
        app:cardElevation="4dp"
        app:cardCornerRadius="4dp" >

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="vertical" >

            <TextView
                android:id="@+id/task_name"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:textSize="16sp"
                android:padding="8dp"
                android:text="@{task.taskName}" />
```




Display the name of the task.

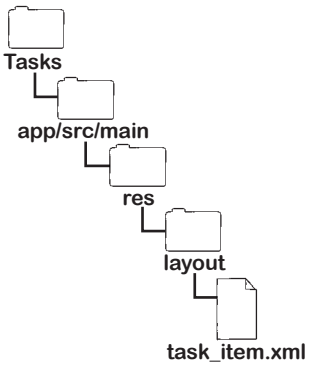
The code continues on the next page.

DiffUtil
Data binding

# The task_item.xml code continued

```
            <CheckBox
                android:id="@+id/task_done"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:textSize="16sp"
                android:padding="8dp"
                android:clickable="false"
                android:text="Done?"
                android:checked="@{task.taskDone}" />
        </LinearLayout>
    </androidx.cardview.widget.CardView>
</layout>
```

Display whether the task is complete.




Let's go through what happens when the code runs, and take it for a test drive.

## there are no Dumb Questions

**Q: Why did we use a `RecyclerView.Adapter` in the previous chapter? Could we have used `ListAdapter` instead?**

A: We used a `RecyclerView.Adapter` in the previous chapter so that you could get a feel for how recycler views fit together and work. Using a `ListAdapter` is an extra refinement that works well with `DiffUtil`, which is why we introduced it in this chapter.

**Q: Can you use `DiffUtil` without implementing a `ListAdapter`?**

A: You can, but it's more complicated. It's easier to use `ListAdapter` instead.

**Q: Why are we replacing those calls to `findViewById()`?**

A: `findViewById()` has a number of disadvantages, as we discussed in Chapter 10.

First, it can be quite inefficient. Each time the method's called, Android has to search the layout's hierarchy for a view with a matching ID. If it doesn't find one, the app will crash.

`findViewById()` isn't type-safe either. If you specify an incorrect type for the view you want to find, the app may crash when you try to use it.

**Q: Oh yes, I think I remember these disadvantages. And data binding doesn't have these problems?**

A: No. Using data binding is safer because it means you're much less likely to experience these kinds of runtime errors.

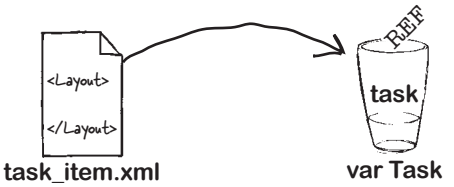
**Q: In *task_item.xml* you defined a data binding variable whose type is `Task`. Why?**

A: Because we wanted to display values from each item's `Task` object, and creating a variable of this type lets us set these values using data binding.

# What happens when the code runs

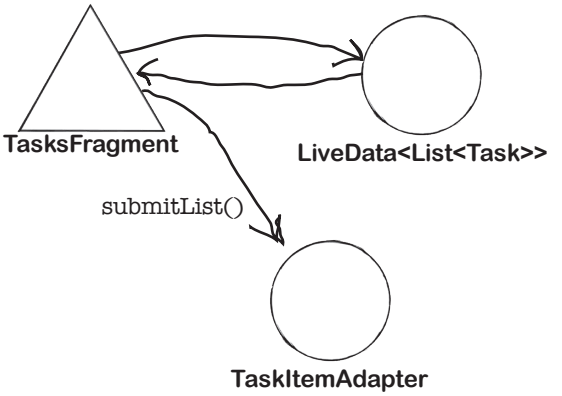
The following things happen when the app runs:

1. **task_item.xml defines a Task data binding variable named task.**
As *task_item.xml* has a `<layout>` element at its root, a binding class named `TaskItemBinding` is generated for this layout.




2. **TasksFragment creates a TaskItemAdapter object, and assigns it to the recycler view as its adapter.**

3. **TasksFragment submits a List<Task> to the TaskItemAdapter.**
The `List<Task>` contains an up-to-date list of records from the database.

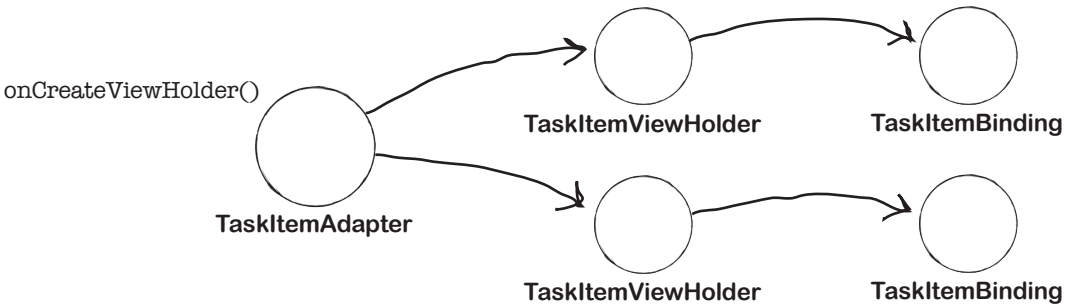


DiffUtil
Data binding

# The story continues

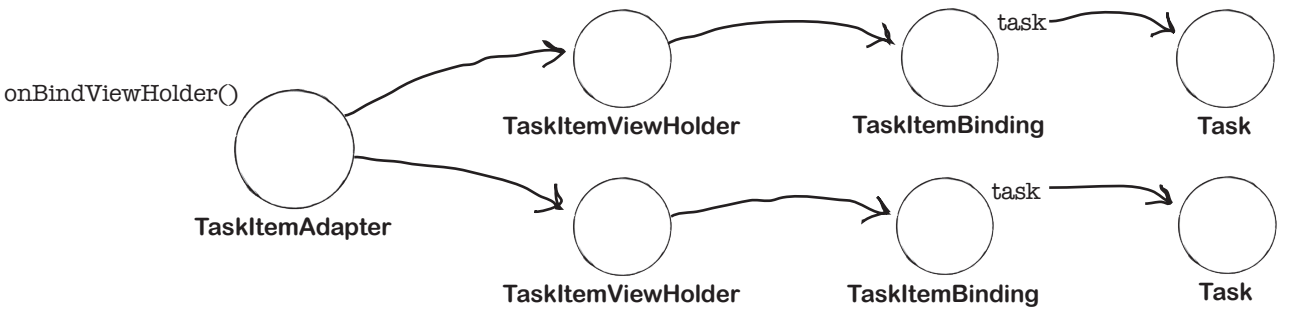
**4** **TaskItemAdapter's onCreateViewHolder() method is called for each item that needs to be displayed in the recycler view.**

`onCreateViewHolder()` calls `TaskItemViewHolder.inflateFrom()`, which creates a `TaskItemBinding` object. It inflates the object's layout, and uses it to create a `TaskItemViewHolder`.



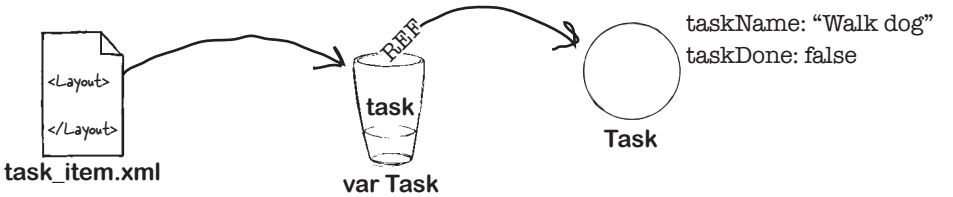

**5** **TaskItemAdapter's onBindViewHolder() method is called for each TaskItemViewHolder.**

This calls the `TaskItemViewHolder`'s `bind()` method, which uses the `TaskItemBinding` object to set the layout's `task` variable to the item's `Task`.



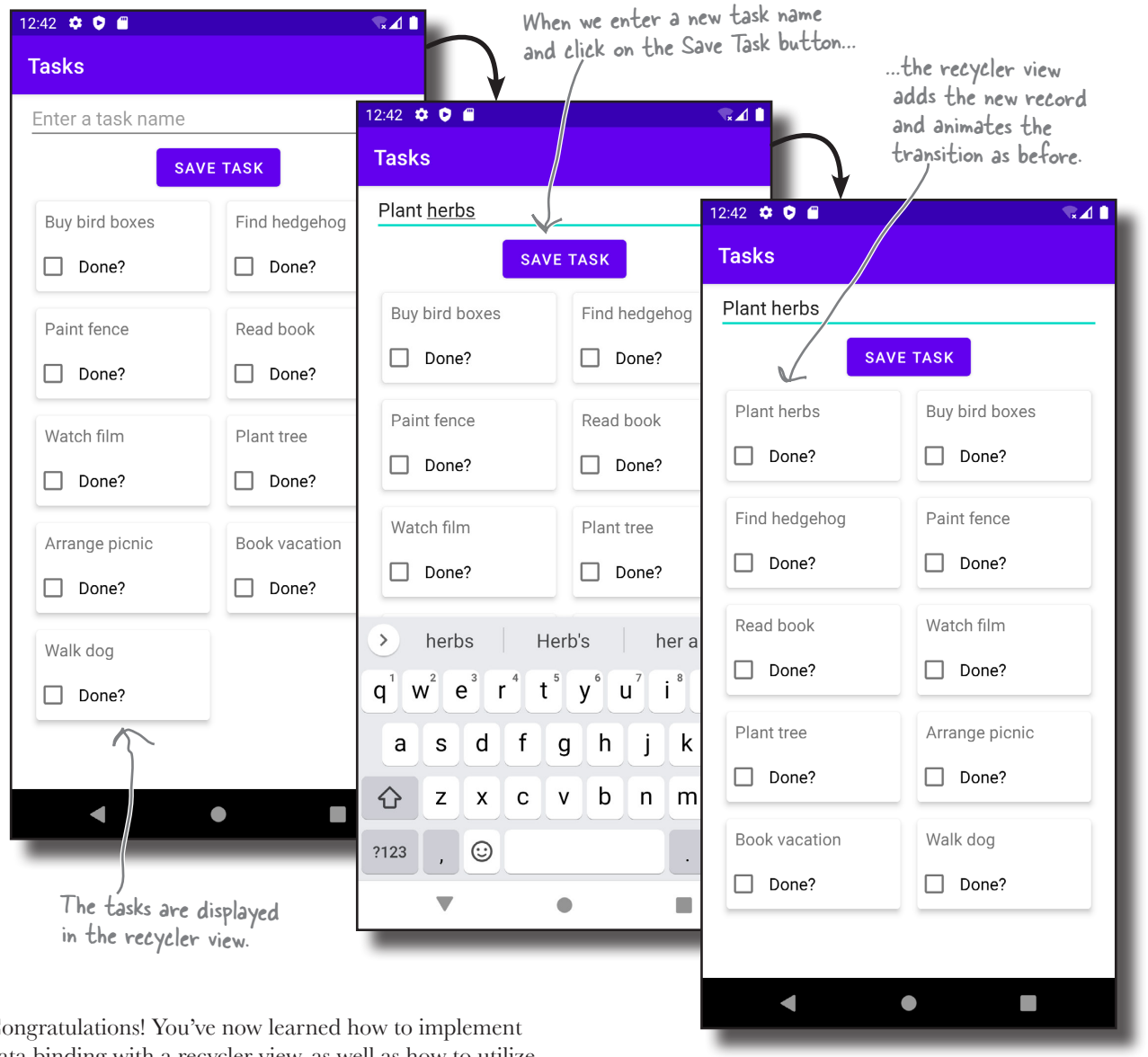

**6** **Data binding code in task_item.xml uses the task property to set the views for each item.**

The `task_name` view's `text` property is set to `task.taskName`, and the `task_done` view's `checked` property is set to `task.taskDone`.



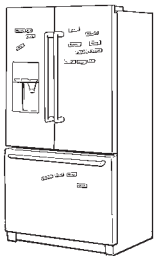
# Test Drive

When we run the app, `TasksFragment` displays a grid of cards in the recycler view. It behaves in the same way as before, but this time, we're using data binding.




Congratulations! You've now learned how to implement data binding with a recycler view, as well as how to utilize `DiffUtil`.

In the next chapter, we'll build on this knowledge to make the recycler view navigate to individual records.



# Adapter Magnets

The Bits and Pizzas app includes a recycler view that uses a layout named *pizza_item.xml* for its items. The layout defines a data binding variable (named `pizza`) like this:

```
<layout...>
    <data>
        <variable
            name="pizza"
            type="com.hfad.bitsandpizzas.Pizza" />
    </data>
    ...
</layout>
```

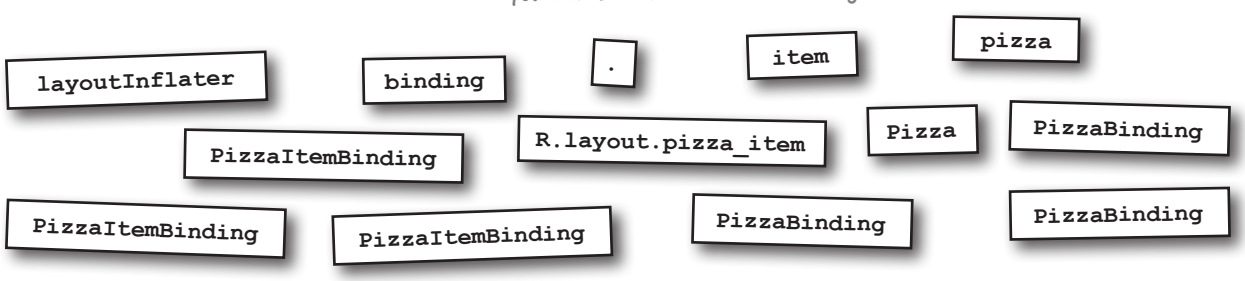
The recycler view uses an adapter named `PizzaAdapter`, shown below. See if you can complete the code for this adapter so that it sets the layout's `pizza` data binding variable.

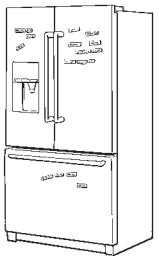
```
package com.hfad.bitsandpizzas

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.ListAdapter
import androidx.recyclerview.widget.RecyclerView

import com.hfad.bitsandpizzas.databinding.……..

class PizzaAdapter
    : ListAdapter<Pizza, PizzaAdapter.PizzaViewHolder>(PizzaDiffItemCallback) {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                        : PizzaViewHolder = PizzaViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: PizzaViewHolder, position: Int) {
        val item = getItem(position)
        holder.bind(item)
    }
```

The exercise continues on the next page.

```
    class PizzaViewHolder(val binding: ..........)
                                      : RecyclerView.ViewHolder(binding.root) {

        companion object {
            fun inflateFrom(parent: ViewGroup): PizzaViewHolder {
                val layoutInflater = LayoutInflater.from(parent.context)

                val binding = ..........

                              .inflate( ..........., parent, false)

                return PizzaViewHolder(binding)
            }
        }

        fun bind(item: Pizza) {

            ..........= ..........
        }
    }
}
```

You won't need to use all the magnets.





# Adapter Magnets Solution

The Bits and Pizzas app includes a recycler view that uses a layout named *pizza_item.xml* for its items. The layout defines a data binding variable (named `pizza`) like this:

```
<layout...>
    <data>
        <variable
            name="pizza"
            type="com.hfad.bitsandpizzas.Pizza" />
    </data>
    ...
</layout>
```

The recycler view uses an adapter named `PizzaAdapter`, shown below. See if you can complete the code for this adapter so that it sets the layout's `pizza` data binding variable.

```
package com.hfad.bitsandpizzas

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.ListAdapter
import androidx.recyclerview.widget.RecyclerView

import com.hfad.bitsandpizzas.databinding.PizzaItemBinding

class PizzaAdapter
    : ListAdapter<Pizza, PizzaAdapter.PizzaViewHolder>(PizzaDiffItemCallback) {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                        : PizzaViewHolder = PizzaViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: PizzaViewHolder, position: Int) {
        val item = getItem(position)
        holder.bind(item)
    }
```

Import the layout's binding class.

The solution continues on the next page.

PizzaViewHolder needs to accept a PizzaItemBinding object.

```
class PizzaViewHolder(val binding: PizzaItemBinding)
                                   : RecyclerView.ViewHolder(binding.root) {

    companion object {
        fun inflateFrom(parent: ViewGroup): PizzaViewHolder {
            val layoutInflater = LayoutInflater.from(parent.context)

            val binding = PizzaItemBinding.inflate(layoutInflater, parent, false)

            return PizzaViewHolder(binding)
        }
    }

    fun bind(item: Pizza) {
        binding . pizza = item
    }
}
```

Use PizzaItemBinding to inflate the layout.

Set the layout's data binding variable.

You didn't need to use these magnets.

PizzaBinding

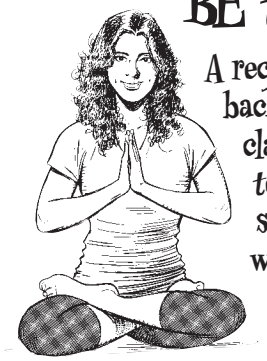
R.layout.pizza_item

PizzaBinding

Pizza

PizzaBinding

# BE the ListAdapter Solution



A recycler view's ListAdapter class has a backing list of Drinks. It uses the Drink class shown on the right. Your job is to play like you're the ListAdapter, and say if the ItemCallback classes below will correctly spot any changes when the ListAdapter is given a new list. Why? Why not?

Hint: the drinkId property is a unique identifier for each Drink.

```
class Drink(
    var drinkId: Long = 0L,
    var drinkName: String = ""
)
```

**A**

```
class DrinkDiffItemCallback : DiffUtil.ItemCallback<Drink>() {
    override fun areItemsTheSame(oldItem: Drink, newItem: Drink)
        = (oldItem.drinkId == newItem.drinkId)

    override fun areContentsTheSame(oldItem: Drink, newItem: Drink)
        = (oldItem == newItem)
}
```

✗ As Drink isn't a data class, this will only return true if oldItem and newItem are the same object.

**B**

```
class DrinkDiffItemCallback : DiffUtil.ItemCallback<Drink>() {
    override fun areItemsTheSame(oldItem: Drink, newItem: Drink)
        = (oldItem == newItem)
    override fun areContentsTheSame(oldItem: Drink, newItem: Drink)
        = ((oldItem.drinkId == newItem.drinkId) &&
           (oldItem.drinkName == newItem.drinkName))
}
```

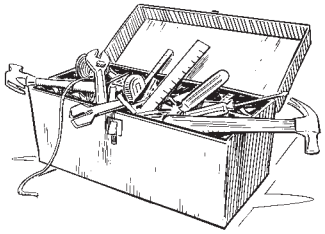
✗ This needs to test whether the two items have matching IDs. Instead it checks if they're the same object.

**C**

```
class DrinkDiffItemCallback : DiffUtil.ItemCallback<Drink>() {
    override fun areItemsTheSame(oldItem: Drink, newItem: Drink)
        = (oldItem.drinkId == newItem.drinkId)

    override fun areContentsTheSame(oldItem: Drink, newItem: Drink)
        = (oldItem.drinkName == newItem.drinkName)
}
```

✓ This correctly tests if two items have the same ID and matching contents.

# Your Android Toolbox



**You've got Chapter 16 under your belt and now you've added DiffUtil and recycler view data binding to your toolbox.**

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

## Bullet Points

- Each time `notifyDataSetChanged()` is called, the recycler rebinds and redraws every item.
- `DiffUtil` figures out which items have changed, and updates the recycler view in the most efficient way possible.
- A `ListAdapter` is a type of `RecyclerView.Adapter` that works with lists. It provides its own backing list, and accepts a `DiffUtil.ItemCallback` in its constructor.
- `DiffUtil.ItemCallback` calculates the difference between two non-null items in a list.
- Override the `DiffUtil.ItemCallback`'s `areItemsTheSame()` method to specify whether two objects refer to the same item.
- Override the `DiffUtil.ItemCallback`'s `areContentsTheSame()` method to specify whether two objects have the same contents.
- Use the `submitList()` method to submit a new version of a list to a `ListAdapter`.
- Recycler views can use data binding.
- Define a data binding variable in the layout used for the recycler view's items.
- Set the data binding variable in the recycler view's adapter code.

# 17 recycler view navigation

# Pick a Card

**Some apps rely on the user selecting an item from a list.**

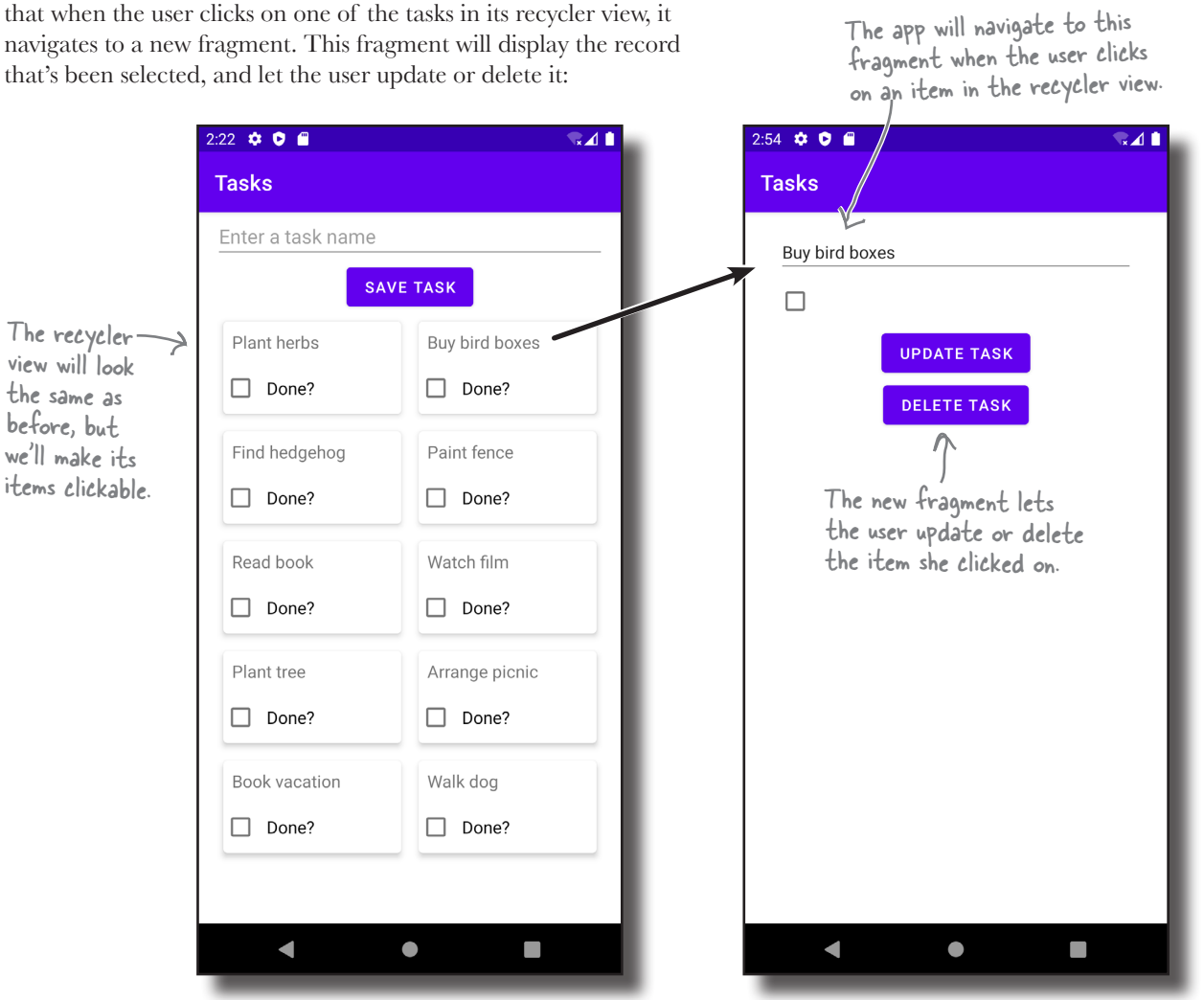
And in this chapter, you'll learn **how to make recycler views a core part of your app design** by **making their items clickable**. You'll discover how to **implement recycler view navigation** by making the app navigate to a new screen each time the user clicks on a record. You'll find out **how to show the user extra information** about their chosen record, and update it in the database. By the end of this chapter, you'll have all the tools you need to **transform your awesome ideas into the app of your dreams**…

# Recycler views can be used for navigation

In the previous two chapters, you learned how to build a recycler view that displays a scrollable list of data, and used `DiffUtil` to make it more efficient. But this isn't the end of the story.

Recycler views are a major component of many Android apps because, in addition to displaying lists of data, you can use them to navigate through the app. When the user clicks on an item in the recycler view, you can make the app navigate to a new fragment, which displays further details of that record.

To see how this works, we're going to change up the Tasks app so that when the user clicks on one of the tasks in its recycler view, it navigates to a new fragment. This fragment will display the record that's been selected, and let the user update or delete it:



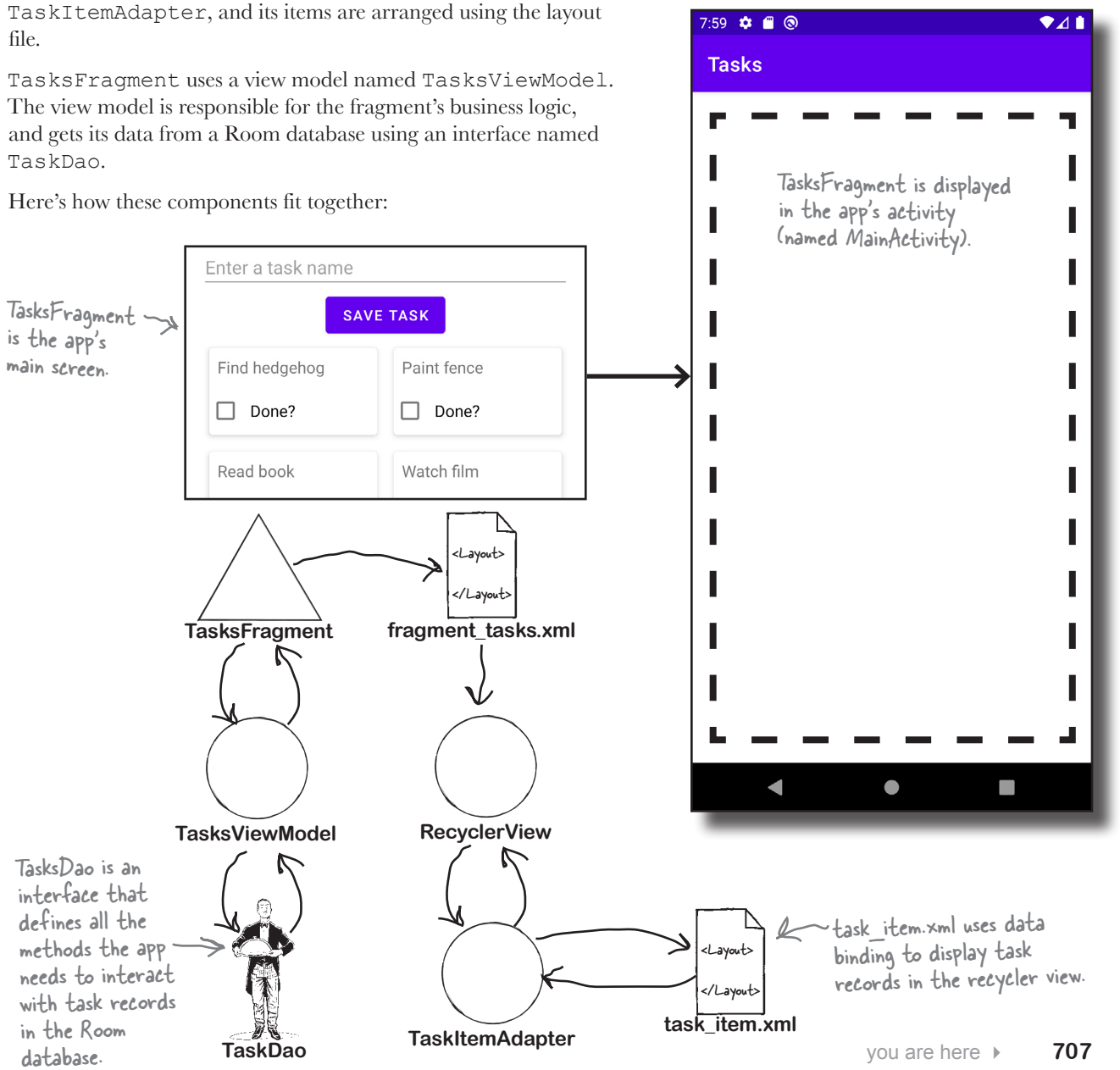
# How the Tasks app is currently structured

Before we look at how we need to change the Tasks app, let's remind ourselves of its current structure.

The app is comprised of a single activity (`MainActivity`), which displays a fragment named `TasksFragment`. This fragment is the app's main screen, and its layout includes a recycler view that displays a grid of tasks. The recycler view uses an adapter named `TaskItemAdapter`, and its items are arranged using the layout file.

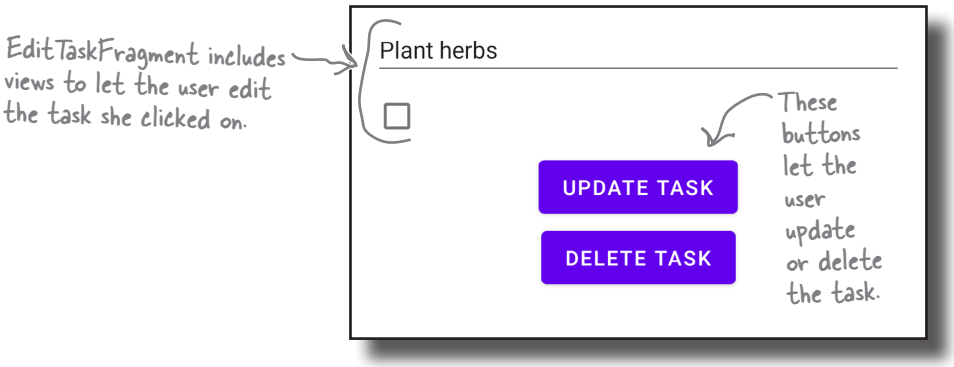
`TasksFragment` uses a view model named `TasksViewModel`. The view model is responsible for the fragment's business logic, and gets its data from a Room database using an interface named `TaskDao`.

Here's how these components fit together:



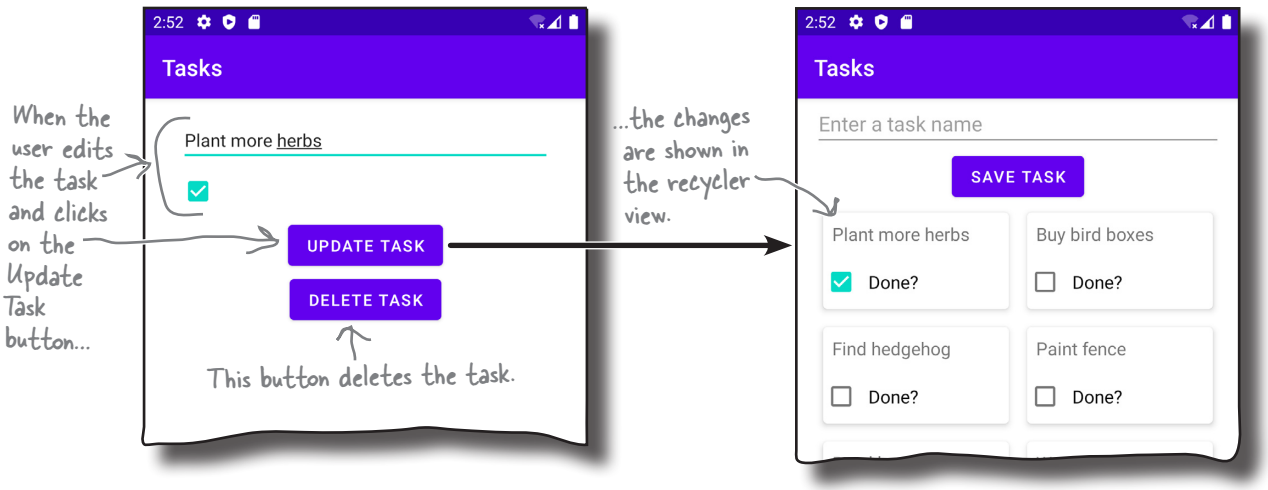
# We're going to make the recycler view navigate to a new fragment

We're going to update the Tasks app so that when the user clicks on a task in the recycler view, it displays a fragment named `EditTaskFragment`. Here's what the new fragment will look like:




As you can see, `EditTaskFragment` includes an edit text and a checkbox that let the user edit the task. The edit text displays the name of the task, and the checkbox shows whether the task has been completed.

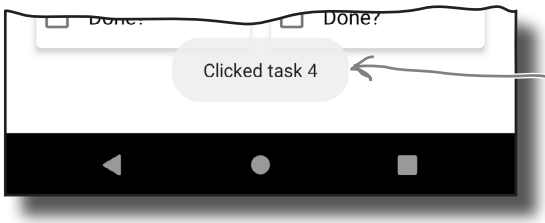
The fragment also includes an Update Task button, which, when clicked, updates the record in the database, and a Delete Task button, which deletes the record. When either of these are clicked, the app navigates back to `TasksFragment`, which displays an updated list of tasks in its recycler view:



# Here's what we're going to do

We're going to build the new version of the app in the following three stages:

1. **Make items in the recycler view respond to clicks.**
We'll update the app so that when we click on a task in the recycler view, its ID is displayed in a toast.

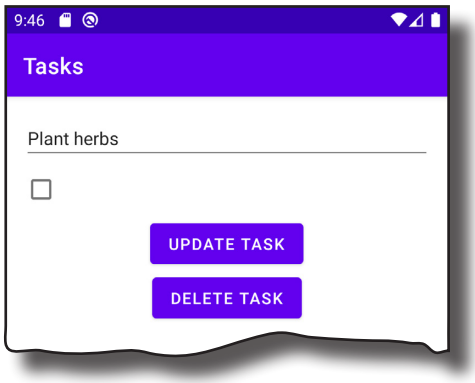



When we click on the task with an ID of 4, a toast will display this message.

2. **Navigate to EditTaskFragment when an item is clicked.**
We'll create `EditTaskFragment`, and use the Navigation component to navigate to it when the user clicks on a task record. We'll display the task's ID in the new fragment.

We'll display the task's ID in EditTaskFragment so that we know that the new fragment has picked up the task's ID.

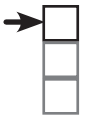
3. **Display the task record in EditTaskFragment, and let the user update or delete the record.**
We'll create a view model for `EditTaskFragment`, which will use the `TaskDao` interface to interact with the database.




**Don't forget!**

You're going to modify the Tasks app in this chapter, so make sure you open the project for this app.

We'll begin by making the recycler view respond to clicks.



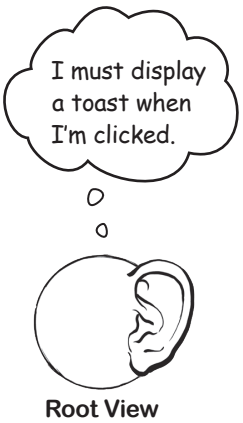
**Respond to clicks**
Navigate
Show record

# Make each item clickable

The first change we're going to make to the Tasks app is to make it display a toast when the user clicks on one of the items in the recycler view.

We can make each item respond to clicks by adding an `OnClickListener` to each item's root view. To do this, we'll call each item's `setOnClickListener()` method immediately after each item's data gets added to its layout.

The best place to add each `OnClickListener` is in the `TaskItemViewHolder`'s `bind()` method, since this is where the layout's data binding variable gets set to a `Task` item. As you may recall, the `bind()` method gets called by `TaskItemAdapter`'s `onBindViewHolder()` method, which fires each time the recycler view needs to display an item's data.




Here's the code to add an `OnClickListener` to the root view of each item's layout; we'll add this to *TaskItemAdapter.kt* a few pages ahead:

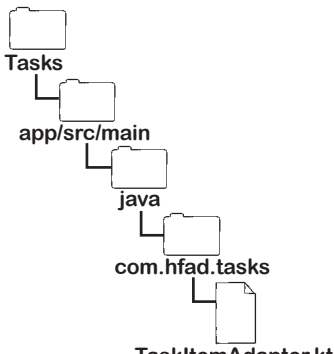
```
class TaskItemAdapter : ListAdapter<...>(TaskDiffItemCallback()) {
    ...
    override fun onBindViewHolder(holder: TaskItemViewHolder, position: Int) {
        val item = getItem(position)
        holder.bind(item)
    }
    class TaskItemViewHolder(val binding: TaskItemBinding)
                            : RecyclerView.ViewHolder(binding.root) {
        ...
        fun bind(item: Task) {
            binding.task = item
            binding.root.setOnClickListener {
                //Code that runs when an item is clicked
            }
        }
    }
}
```

onBindViewHolder() is called each time the recycler view needs to display an item's data.

onBindViewHolder() calls TaskItemViewHolder's bind() method, passing it an item.

This sets the layout's data binding variable for the item.

Make the item respond to clicks.




Now that you know how to add an `OnClickListener` to each item, let's make it display a toast when an item is clicked.

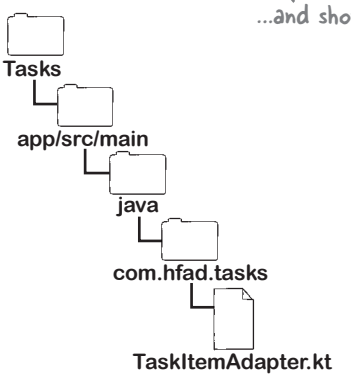
# Where should we create the toast?

To display a toast each time an item's clicked, we *could* simply add the following code (in bold) to the view holder's `setOnClickListener()` method:

```
class TaskItemViewHolder(val binding: TaskItemBinding)
                        : RecyclerView.ViewHolder(binding.root) {
    ...
    fun bind(item: Task) {
        binding.task = item
        binding.root.setOnClickListener {
            Toast.makeText(binding.root.context, "Clicked task ${item.taskId}",
                                                 Toast.LENGTH_SHORT).show()
        }
    }
}
```

Set the toast's text...

...and show the toast.




This approach, however, would mean that we'd be putting code that describes the app's behavior in the view holder code. This code is responsible for binding data to each item's layout, so it's not an appropriate place for us to put this kind of code.

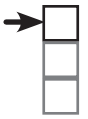
Adding the toast code to the view holder would also make the view holder code less flexible. It would mean that each time the user clicks on an item, it could only display a toast, and couldn't be reused elsewhere.

So what's the alternative?

## We'll make TasksFragment pass the toast code in a lambda

An alternative is to define the code each item needs to execute in `TasksFragment`, and pass it to `TaskItemViewHolder`—via `TaskItemAdapter`—in a lambda. Doing so means that the *fragment* controls what happens when an item is clicked instead of the view holder.

Before we look at the code, let's run through how it will work.

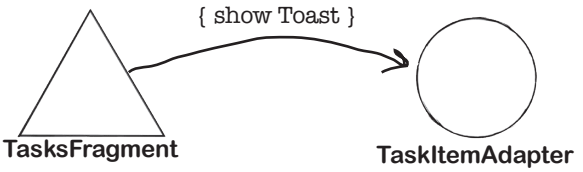


**Respond to clicks**
Navigate
Show record

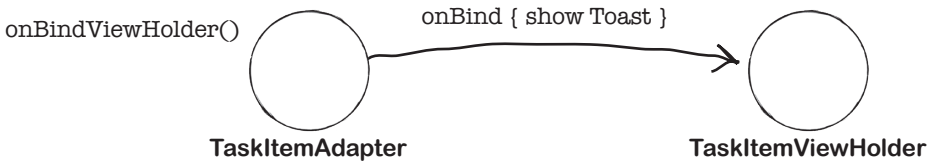
# How the code will work

Here's what the code we're going to write will do:

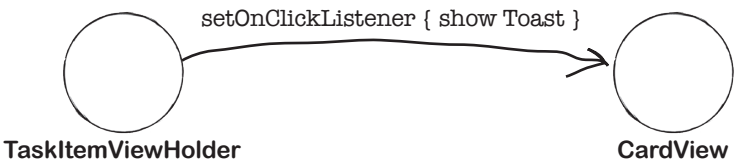
1. **TasksFragment will pass a lambda to TaskItemAdapter's constructor.**
The lambda includes the code to display a toast.




2. **When the TaskItemAdapter's onBindViewHolder() method is called, it calls the TaskItemViewHolder's bind() method, and passes it the lambda.**



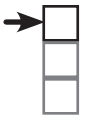

3. **The TaskItemViewHolder adds the lambda to each item's OnClickListener.**
When the user clicks on each item (a `CardView`), it executes the lambda and displays the toast.




So that's how the code will work.

To implement this, we need to update the `TasksFragment`, `TaskItemAdapter`, and `TaskItemViewHolder` code. We'll begin by updating the `TaskItemAdapter` and `TaskItemViewHolder` code so that the adapter is able to accept the lambda, and pass it to the view holder.

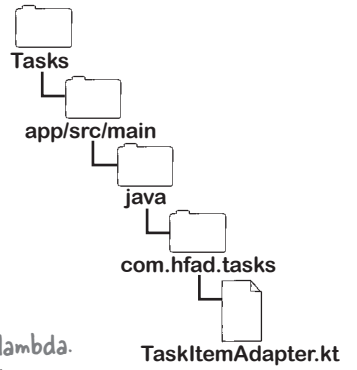
We'll show you the code for this on the next page.



**Respond to clicks**
Navigate
Show record

# The full code for TaskItemAdapter.kt

Here's the `TaskItemAdapter` and `TaskItemViewHolder` code; update *TaskItemAdapter.kt* to include all the changes shown here (in bold):




```
package com.hfad.tasks

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.ListAdapter
import androidx.recyclerview.widget.RecyclerView
import com.hfad.tasks.databinding.TaskItemBinding

class TaskItemAdapter(val clickListener: (taskId: Long) -> Unit)
        : ListAdapter<Task, TaskItemAdapter.TaskItemViewHolder>(TaskDiffItemCallback()) {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                            : TaskItemViewHolder = TaskItemViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: TaskItemViewHolder, position: Int) {
        val item = getItem(position)
        holder.bind(item, clickListener)
    }

    class TaskItemViewHolder(val binding: TaskItemBinding)
                            : RecyclerView.ViewHolder(binding.root) {

        companion object {
            fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
                val layoutInflater = LayoutInflater.from(parent.context)
                val binding = TaskItemBinding.inflate(layoutInflater, parent, false)
                return TaskItemViewHolder(binding)
            }
        }

        fun bind(item: Task, clickListener: (taskId: Long) -> Unit) {
            binding.task = item
            binding.root.setOnClickListener { clickListener(item.taskId) }
        }
    }
}
```

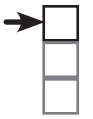
Make TaskItemAdapter accept a lambda.

Pass the lambda to TaskItemViewHolder's bind() method.

Make the bind() method accept a lambda.

Make the item clickable.

Execute the lambda when an item's clicked.



**Respond to clicks**
Navigate
Show record

# We'll pass a lambda to TaskItemAdapter

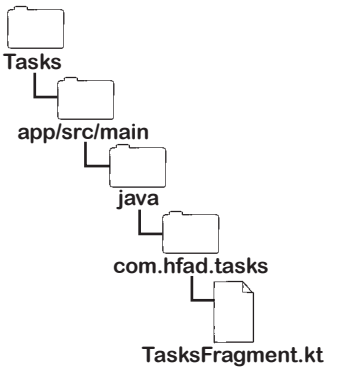
Now that `TaskItemAdapter` includes a lambda parameter in its constructor, we need to pass one to it in the `TasksFragment` code where it gets created.

As you may recall, `TasksFragment` creates a `TaskItemAdapter` object in its `onCreateView()` method, which it then assigns to the recycler view like so:

```
override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
): View? {
    ...
    val adapter = TaskItemAdapter()
    binding.tasksList.adapter = adapter
    ...
}
```

Create the adapter...

...and assign it to the recycler view.



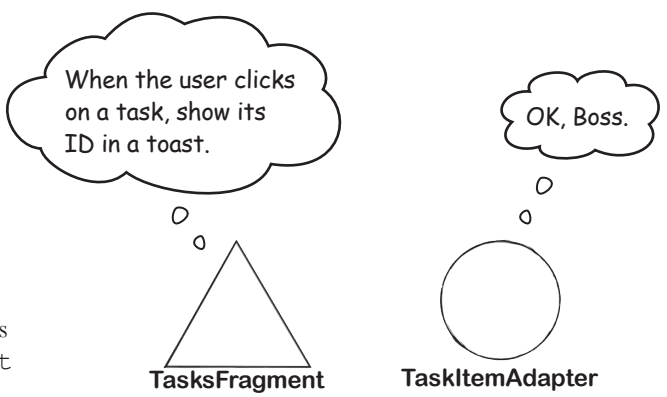

As we want to display a toast each time an item in the recycler view is clicked, we can update this code so that it passes the following lambda to `TaskItemAdapter`'s constructor:

```
override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
): View? {
    ...
    val adapter = TaskItemAdapter { taskId ->
        Toast.makeText(context, "Clicked task $taskId", Toast.LENGTH_SHORT).show()
    }
    binding.tasksList.adapter = adapter
    ...
}
```

This passes a lambda to TaskItemAdapter, which, when clicked, displays a toast.

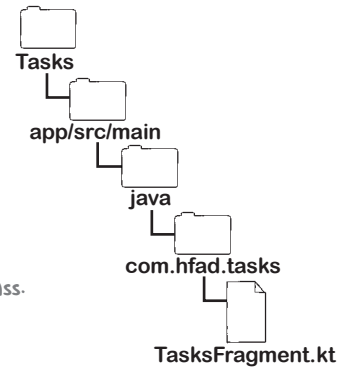
The adapter passes the lambda to `TaskItemViewHolder`'s `bind()` method, which uses the lambda in the `OnClickListener` code it assigns to each item's root view. When the user clicks on an item in the recycler view, the lambda is executed.

You've now learned everything that you need to know to make an item in the recycler view display a toast when it's clicked. Let's see what the full code for `TasksFragment` looks like.



# The full code for TasksFragment.kt

Here's the updated `TasksFragment` code; make sure that the code in *TasksFragment.kt* includes all the changes shown here (in bold):

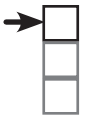



```
package com.hfad.tasks

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Toast
import androidx.lifecycle.Observer
import androidx.lifecycle.ViewModelProvider
import com.hfad.tasks.databinding.FragmentTasksBinding

class TasksFragment : Fragment() {
    private var _binding: FragmentTasksBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
            inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentTasksBinding.inflate(inflater, container, false)
        val view = binding.root

        val application = requireNotNull(this.activity).application
        val dao = TaskDatabase.getInstance(application).taskDao

        val viewModelFactory = TasksViewModelFactory(dao)
        val viewModel = ViewModelProvider(
                this, viewModelFactory).get(TasksViewModel::class.java)
        binding.viewModel = viewModel
        binding.lifecycleOwner = viewLifecycleOwner
```

Import this class.

The code continues on the next page.

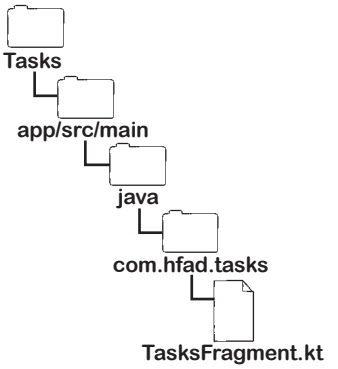
# The TasksFragment.kt code continued



**Respond to clicks**
Navigate
Show record

```
        val adapter = TaskItemAdapter() { taskId ->
            Toast.makeText(context, "Clicked task $taskId", Toast.LENGTH_SHORT).show()
        }
        binding.tasksList.adapter = adapter

        viewModel.tasks.observe(viewLifecycleOwner, Observer {
            it?.let {
                adapter.submitList(it)
            }
        })

        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Pass a lambda to TaskItemAdapter's constructor.




Let's see what happens when the code runs.

## there are no Dumb Questions

**Q: In the `TaskItemAdapter` code, you defined the `clickListener` argument's type as `(taskId: Long) -> Unit`. Can you remind me what that means?**

A: It means that the argument can be passed a piece of code called a lambda. The code accepts a `Long` argument named `taskId`, and returns `Unit`.

**Q: I see. So it's really just a way of passing code to a method?**

A: Yes. It means that we can make `TasksFragment` specify what code should run, instead of hardcoding it in the `TaskItemAdapter` code.

**Q: Would adding it to the `TaskItemAdapter` code really be so bad?**

A: It would make the adapter code less flexible, and the app's components wouldn't have such a clear separation of concerns.

**Q: I think I've seen other ways of making a recycler view respond to clicks. Are these ways wrong?**

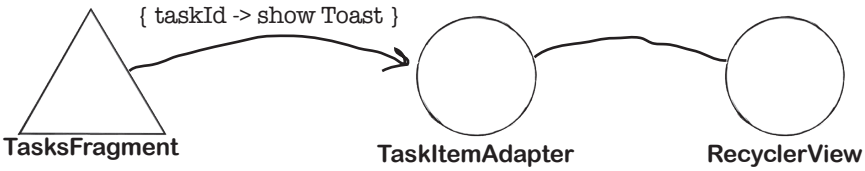
A: There are many different ways of making a recycler view respond to clicks. We chose this approach because it's pretty flexible, and easier to learn than some other methods.

We suggest that you get to grips with the approach we're using in this chapter before exploring other approaches.

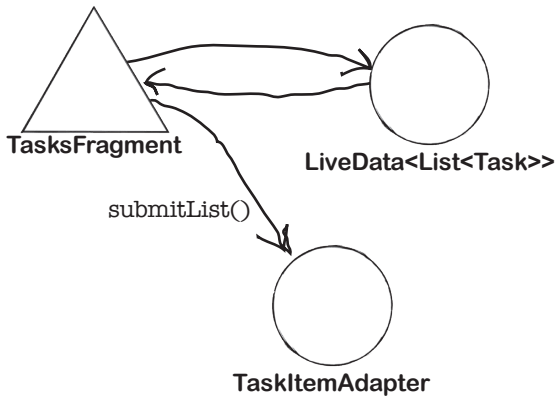
# What happens when the code runs

The following things happen when the app runs:

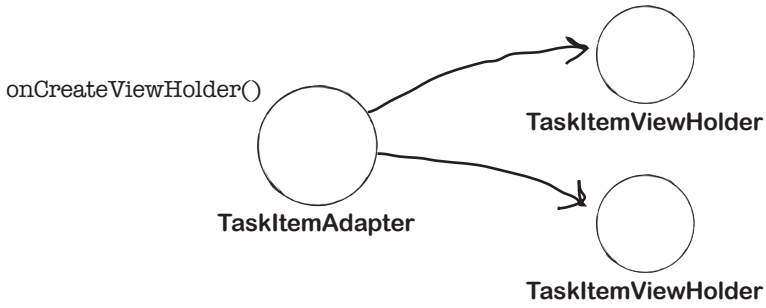
1. **TasksFragment creates a TaskItemAdapter object, and assigns it to the recycler view as its adapter.**
The fragment passes a lambda (named `clickListener`) to the adapter, telling it to display a toast when it's executed.

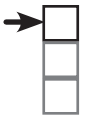



2. **TasksFragment submits a List<Task> to the TaskItemAdapter.**
The `List<Task>` contains an up-to-date list of records from the database.




3. **TaskItemAdapter's onCreateViewHolder() method is called for each item that needs to be displayed in the recycler view.**
This creates a set of `TaskItemViewHolders`.

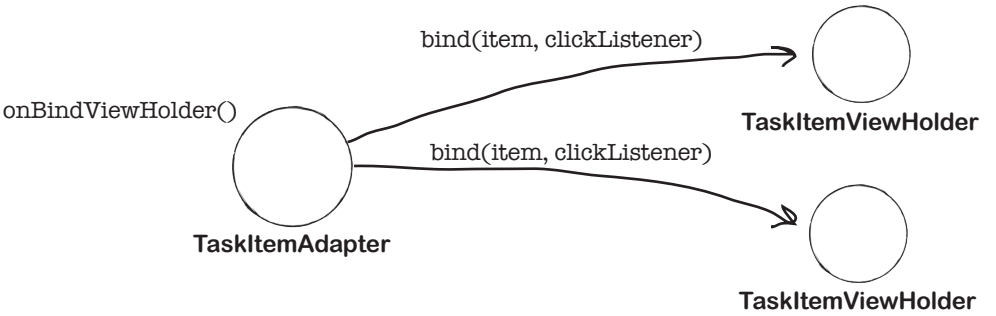




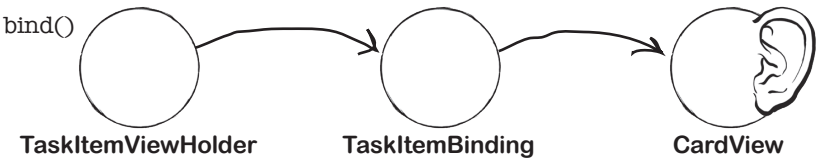
**Respond to clicks**
Navigate
Show record

# The story continues

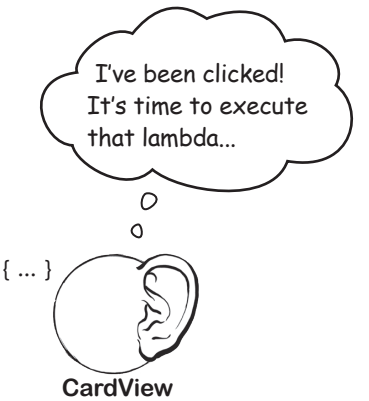
4 **TaskItemAdapter's onBindViewHolder() method is called for each TaskItemViewHolder.**
This calls the `TaskItemViewHolder`'s `bind()` method, passing it the item that was clicked and the `clickListener` lambda.




5 **The TaskItemViewHolder's bind() method adds an OnClickListener to the root view of each view holder's layout.**
In this example, the root view is a `CardView`.




6 **When the user clicks on an item in the recycler view, the OnClickListener registers the click.**
It executes the `clickListener` lambda, which displays a toast.

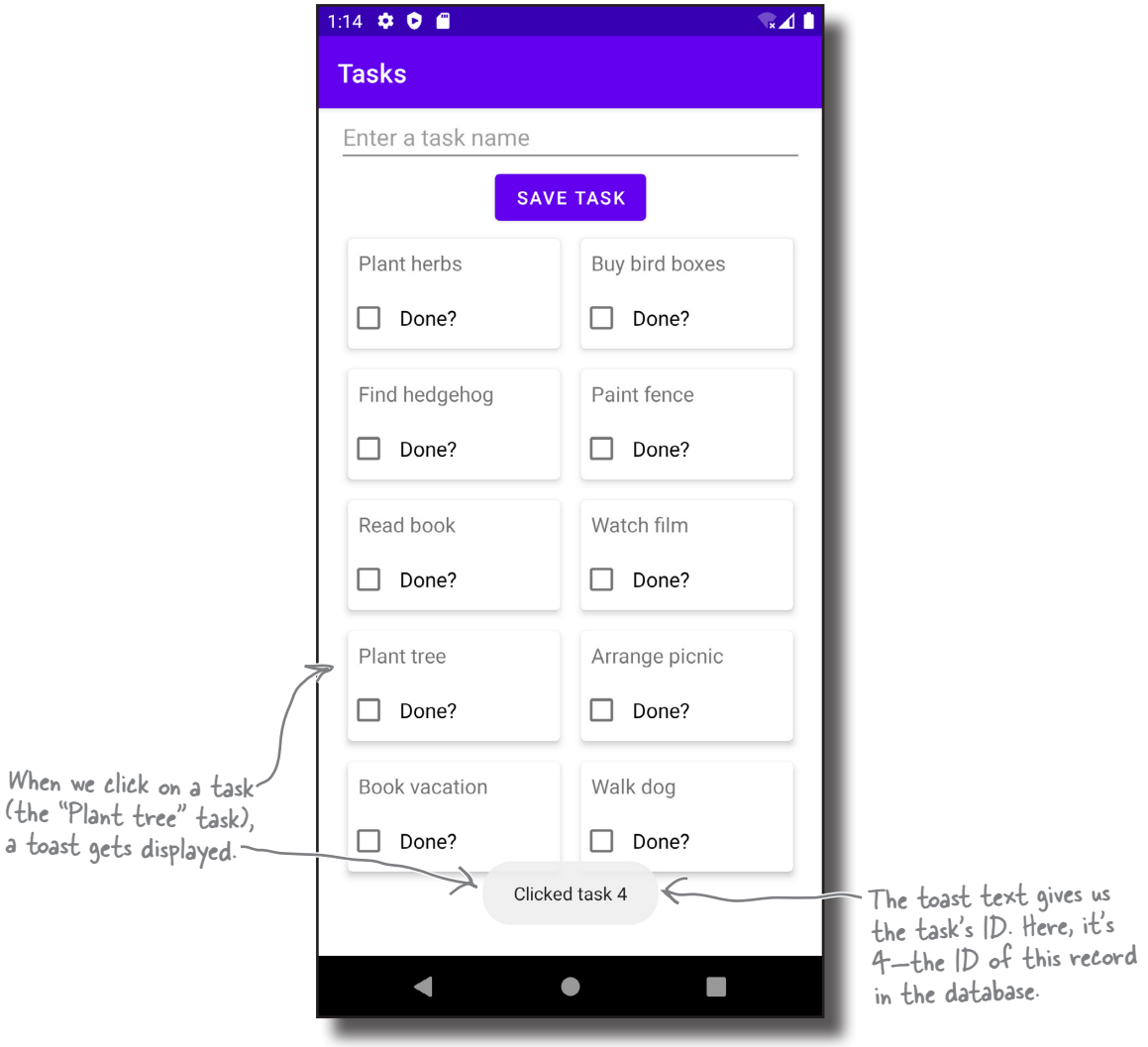



Let's take the app for a test drive.

# Test Drive

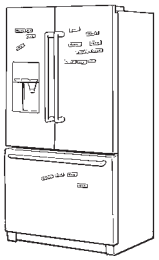
**Respond to clicks**
Navigate
Show record

When we run the app, `TasksFragment` displays a grid of cards in the recycler view as before. If we click on one of the tasks, the app displays its ID in a toast.




You've now learned how to make a recycler view's items respond to click events. We'll use this knowledge to make the app navigate to a new fragment when an item is clicked.

Before we do this, try the following exercise.



# Adapter Magnets

The Bits and Pizzas app includes a recycler view that uses a layout named *pizza_item.xml* to display `Pizza` objects. The `Pizza` data class looks like this:

```
package com.hfad.bitsandpizzas

data class Pizza(
    var pizzaId: Long = 0L,
    var pizzaName: String = "",
    var pizzaDescription: String = "",
    var pizzaImageId: Int = 0
)
```

The recycler view uses an adapter named `PizzaAdapter`, shown below. See if you can complete the code for this adapter so that when one of its items is clicked, it executes a lambda that's passed to the adapter in its constructor.

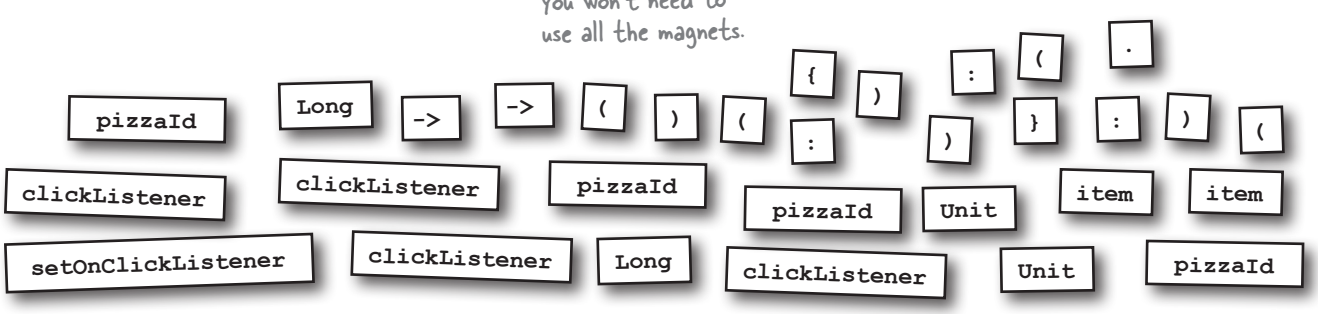
Hint: The lambda should accept a `pizzaId` argument, and return `Unit`.

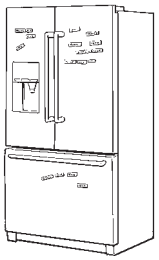
```
package com.hfad.bitsandpizzas

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.ListAdapter
import androidx.recyclerview.widget.RecyclerView
import com.hfad.bitsandpizzas.databinding.PizzaItemBinding

class PizzaAdapter(val clickListener: ..........)

    : ListAdapter<Pizza, PizzaAdapter.PizzaViewHolder>(PizzaDiffItemCallback) {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                        : PizzaViewHolder = PizzaViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: PizzaViewHolder, position: Int) {
        val item = getItem(position)

        holder.bind(item, ..........)
    }
```

The exercise continues on the next page.

```
    class PizzaViewHolder(val binding: PizzaItemBinding)
                                        : RecyclerView.ViewHolder(binding.root) {

        companion object {
            fun inflateFrom(parent: ViewGroup): PizzaViewHolder {
                val layoutInflater = LayoutInflater.from(parent.context)

                val binding = PizzaItemBinding.inflate(layoutInflater, parent, false)
                return PizzaViewHolder(binding)
            }
        }

        fun bind(item: Pizza,

                 ..........) {

            binding.pizza = item

            binding.root. ..........

                ..........

              ..........

        }
    }
}
```

You won't need to use all the magnets.





# Adapter Magnets Solution

The Bits and Pizzas app includes a recycler view that uses a layout named *pizza_item.xml* to display `Pizza` objects. The `Pizza` data class looks like this:

```
package com.hfad.bitsandpizzas

data class Pizza(
    var pizzaId: Long = 0L,
    var pizzaName: String = "",
    var pizzaDescription: String = "",
    var pizzaImageId: Int = 0
)
```

The recycler view uses an adapter named `PizzaAdapter`, shown below. See if you can complete the code for this adapter so that when one of its items is clicked, it executes a lambda that's passed to the adapter in its constructor.

Hint: The lambda should accept a `pizzaId` argument, and return `Unit`.

```
package com.hfad.bitsandpizzas

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.ListAdapter
import androidx.recyclerview.widget.RecyclerView
import com.hfad.bitsandpizzas.databinding.PizzaItemBinding

class PizzaAdapter(val clickListener: ( pizzaId : Long ) -> Unit )
    : ListAdapter<Pizza, PizzaAdapter.PizzaViewHolder>(PizzaDiffItemCallback) {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                        : PizzaViewHolder = PizzaViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: PizzaViewHolder, position: Int) {
        val item = getItem(position)

        holder.bind(item, clickListener )
    }
```

This specifies that PizzaAdapter accepts a lambda argument in its constructor. The lambda accepts a Long, and returns Unit.

Pass the lambda to PizzaViewHolder's bind() method.

The solution continues on the next page.

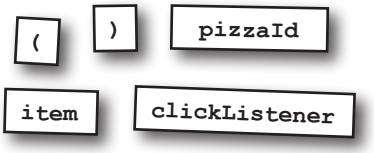
```
class PizzaViewHolder(val binding: PizzaItemBinding)
                                    : RecyclerView.ViewHolder(binding.root) {

    companion object {
        fun inflateFrom(parent: ViewGroup): PizzaViewHolder {
            val layoutInflater = LayoutInflater.from(parent.context)

            val binding = PizzaItemBinding.inflate(layoutInflater, parent, false)
            return PizzaViewHolder(binding)
        }
    }

    fun bind(item: Pizza,
             clickListener : ( pizzaId : Long ) -> Unit ) {

        binding.pizza = item

        binding.root. setOnClickListener {
            clickListener ( item . pizzaId )
        }
    }
}
```

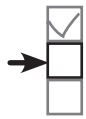
The bind() method accepts a lambda argument.

Make items clickable.

Execute the lambda when an item's clicked.

You didn't need to use these magnets.



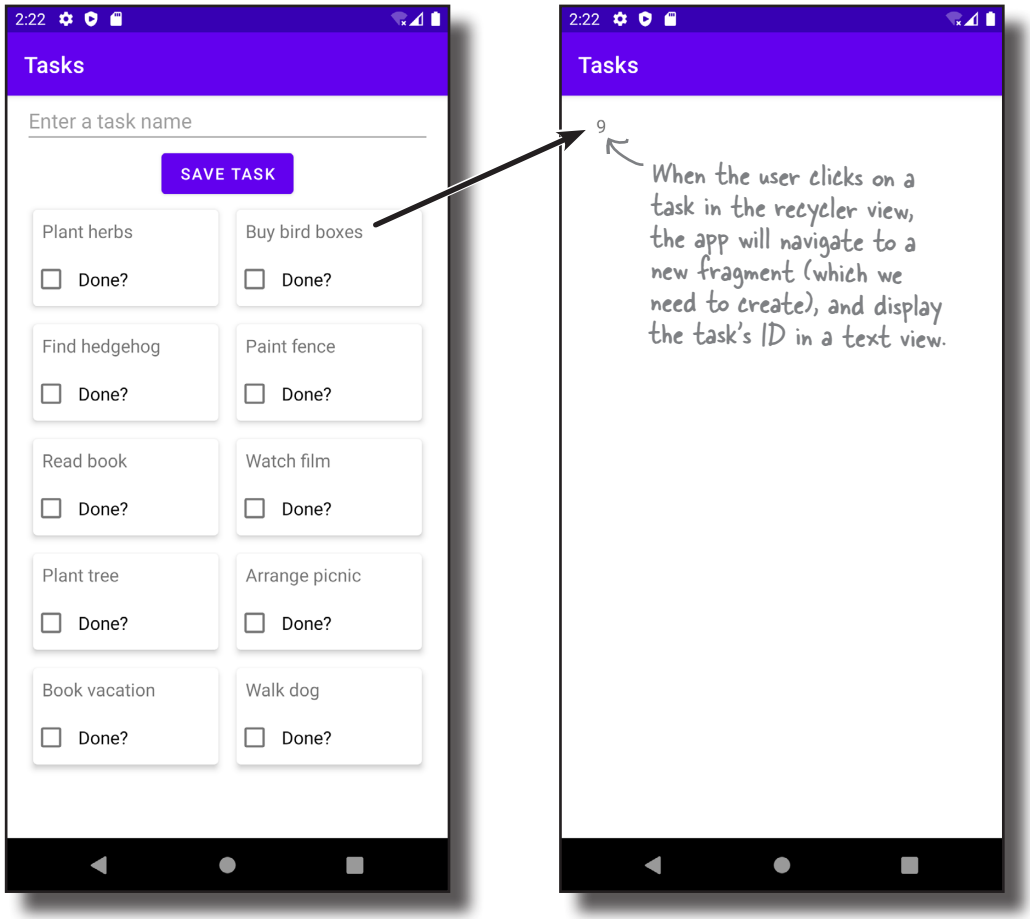


**Respond to clicks**
**Navigate**
**Show record**

# We want to use the recycler view to navigate to a new fragment

So far, you've learned how to make items in a recycler view respond to click events. When the user clicks on a task in the Tasks app's recycler view, for example, it displays a toast.

Next, we're going to change this behavior so that when the user clicks on an item, the app navigates to a new fragment (which we'll create) and displays the task's ID. Here's what the new version of the app will look like:




To get this working, we'll use the Navigation component to navigate to the new fragment, and the Safe Args plug-in to pass the fragment the task's ID. This means that we needs to update the project and the app's *build.gradle* files to include these components.

# Update the project build.gradle file...

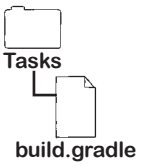
We'll begin by updating the project's *build.gradle* file so that it specifies which version of the Navigation component we want to use, and adds a classpath for the Safe Args plug-in.

To do this, open the file *Tasks/build.gradle*, and add the following lines (in bold) to the relevant sections:

```
buildscript {
    ext.nav_version = "2.3.5"
    ...
    dependencies {
        ...
        classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version"
    }
}
```

We're adding the Navigation component's version to a variable to make sure that we use the same version for the Navigation component and Safe Args plug-in.

Add a classpath for Safe Args.




## ...then update the app build.gradle file

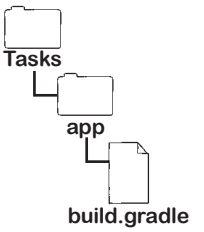
We also need to add the Safe Args plug-in to the app's *build.gradle* file, along with a dependency for the Navigation component.

Open the file *Tasks/app/build.gradle*, and add the following lines (in bold) to the appropriate sections

```
plugins {
    ...
    id 'androidx.navigation.safeargs.kotlin'
}
...
dependencies {
    ...
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
    ...
}
```

Include the Safe Args plug-in.

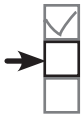
Add the Navigation component.




Once you've made these changes, click on the Sync Now option to sync the changes you've made with the rest of your project.

Now that we've enabled the Navigation component and the Safe Args plug-in, let's create the new fragment that we want the app to be able to navigate to.

# Create EditTaskFragment...



**Respond to clicks**
**Navigate**
**Show record**

We're going to create a new fragment named `EditTaskFragment` that the app will navigate to when the user clicks an item in the recycler view.

To do this, highlight the *com.hfad.tasks* package in the *app/src/main/java* folder, then choose File→New→Fragment→Fragment (Blank). Name the fragment "EditTaskFragment", name its layout "fragment_edit_task", and make sure the language is set to Kotlin.

We'll update the code for `EditTaskFragment` and its layout a few pages ahead. Before that, let's create a navigation graph that will tell the app how to navigate between the app's fragments.

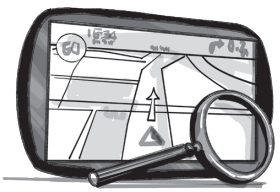
This is EditTaskFragment. In the first instance, it will just display which task the user clicked on.

```
9
```

## ...and create a navigation graph

We'll add a navigation graph to the project in the same way that we did for the other apps we created.

Select the *Tasks/app/src/main/res* folder in the project explorer, and then choose File→New→Android Resource File. When prompted, enter a file name of "nav_graph", choose a resource type of "Navigation", and click on the OK button. This creates a navigation graph named *nav_graph.xml*.



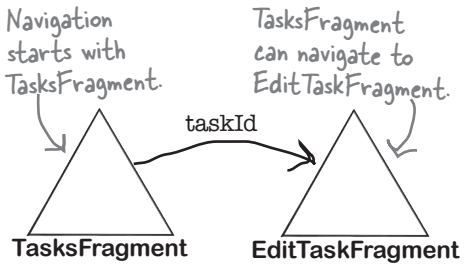
Navigation Graph

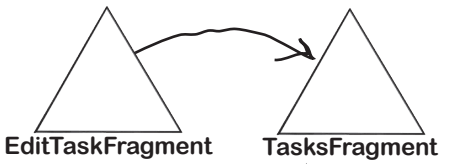
The navigation graph needs to describe how the user will navigate between `TasksFragment` and `EditTaskFragment`. Here's how the navigation needs to work:

1. **The app displays TasksFragment.**
   This is the first fragment the user needs to see, so it needs to be the navigation graph's start destination.

2. **When the user clicks an item in TasksFragment's recycler view, the app navigates to EditTaskFragment.**
   `TasksFragment` will pass `EditTaskFragment` a `Long` parameter containing the task ID for the clicked item.

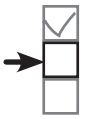
3. **When the user clicks a button in EditTaskFragment (which we'll add to the fragment later in the chapter), the app navigates back to TasksFragment.**




EditTaskFragment can navigate to TasksFragment.

We'll show you the full code for this on the next page.



**Respond to clicks**
**Navigate**
**Show record**

# Update the navigation graph

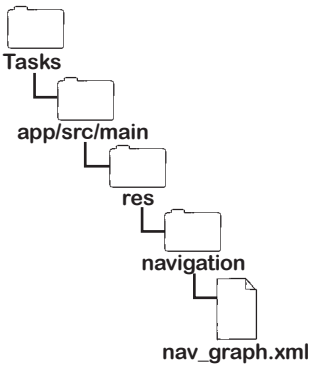
Here's the full code for the navigation graph; update *nav_graph.xml* to include the changes shown here (in bold):

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/tasksFragment">
    <fragment
        android:id="@+id/tasksFragment"
        android:name="com.hfad.tasks.TasksFragment"
        android:label="fragment_tasks"
        tools:layout="@layout/fragment_tasks" >
        <action
            android:id="@+id/action_tasksFragment_to_editTaskFragment"
            app:destination="@id/editTaskFragment" />
    </fragment>
    <fragment
        android:id="@+id/editTaskFragment"
        android:name="com.hfad.tasks.EditTaskFragment"
        android:label="fragment_edit_task"
        tools:layout="@layout/fragment_edit_task" >
        <argument
            android:name="taskId"
            app:argType="long" />
        <action
            android:id="@+id/action_editTaskFragment_to_tasksFragment"
            app:destination="@id/tasksFragment"
            app:popUpTo="@id/tasksFragment"
            app:popUpToInclusive="true" />
    </fragment>
</navigation>
```

Make sure the file includes this line.

The app will start with TasksFragment.

This action lets TasksFragment navigate to EditTaskFragment.



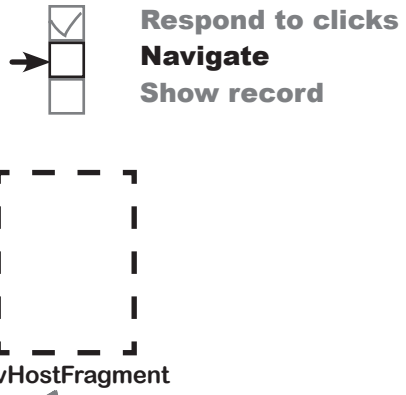

EditTaskFragment accepts a Long argument.

This action lets EditTaskFragment navigate to TasksFragment.

This pops fragments off the back stack, up to and including TasksFragment.

Next, let's link the navigation graph to `MainActivity` so that it displays each fragment when we navigate to it.

# Add a NavHostFragment to MainActivity's layout




To link the navigation graph we just created to `MainActivity`, we need to add a navigation host to its layout, and tell it to use *nav_graph.xml* as its navigation graph. This will let `MainActivity` display the correct fragment as the user navigates through the app.

We'll add the navigation host to the layout in the same way that we did in previous chapters: by adding a `NavHostFragment` to *activity_main.xml*'s `FragmentContainerView`. Update *activity_main.xml* so that it includes the changes below (in bold):

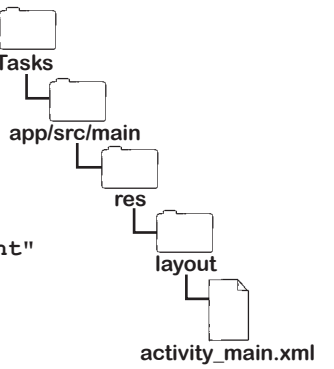
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/fragment_container_view"
    android:id="@+id/nav_host_fragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:name="com.hfad.tasks.TasksFragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    app:navGraph="@navigation/nav_graph"
    app:defaultNavHost="true"
    tools:context=".MainActivity" />
```

Add this extra namespace.

Change the ID.

Update the name attribute so that it refers to NavHost Fragment.

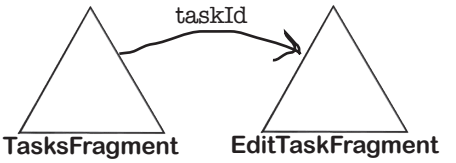
Add these attributes.




That's all the code we need to change for `MainActivity`'s layout. Next, let's make `TasksFragment` navigate to `EditTaskFragment` when the user clicks an item in its recycler view.

# Make TasksFragment navigate to EditTaskFragment

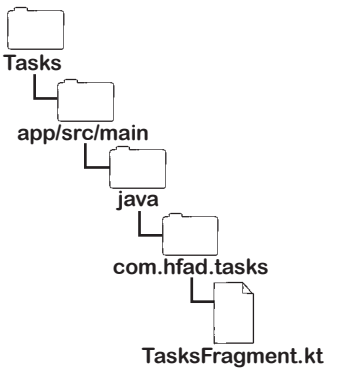
Each time the user clicks on an item in `TasksFragment`'s recycler view, we want the app to navigate to `EditTaskFragment`, passing it the ID of the `Task` that was clicked.




One way of doing this would be to update the lambda that `TasksFragment` passes to its `TaskItemAdapter` so that it includes all the necessary navigation code like this:

```
override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
): View? {
    ...
    val adapter = TaskItemAdapter { taskId ->
        val action = TasksFragmentDirections
                .actionTasksFragmentToEditTaskFragment(taskId)
        this.findNavController().navigate(action)
    }
    binding.tasksList.adapter = adapter
    ...
}
```

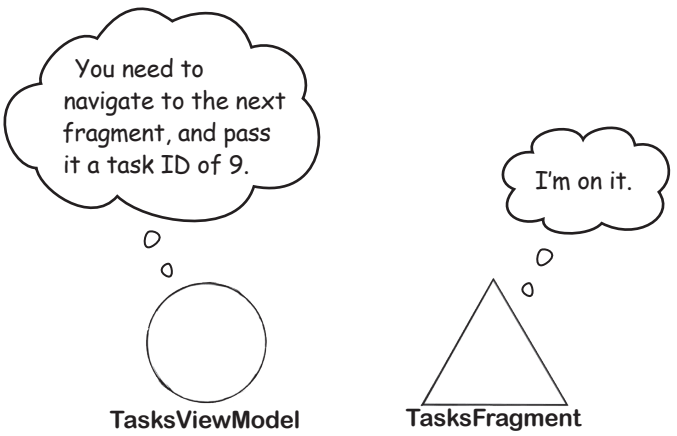
When this lambda is executed, it passes the task ID to EditTaskFragment, and navigates to that fragment.

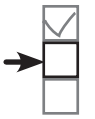



As you can see, this uses the `TasksFragmentDirections` class (generated by the Safe Args plug-in) to pass the item's task ID to `EditTaskFragment`, and navigate to that fragment.

This approach, however, means that we're adding business logic—deciding when `TasksFragment` should navigate to `EditTaskFragment`—to the fragment code, instead of adding it to `TasksViewModel`. As you learned in Chapter 13, the view model code should make this kind of decision instead of the fragment.

To address this, we'll use a similar approach to the one we used for the Guessing Game app in Chapter 13. We'll add a new live data property to `TasksViewModel`, which will hold the ID for the task the user clicks on. When this property changes value, `TasksFragment` will respond by navigating to `EditTaskFragment`, passing it the ID.





**Respond to clicks**
**Navigate**
**Show record**

# Add a new property to TasksViewModel

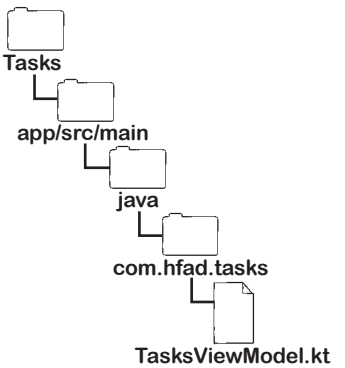
We'll begin by adding a new live data property to `TasksViewModel` that specifies the task ID that `TasksFragment` needs to pass to `EditTaskFragment`. We'll name this property `navigateToTask`, and define it using the following code (we'll add this to *TasksViewModel.kt* on the next page):

```
class TasksViewModel(val dao: TaskDao) : ViewModel() {
    ...
    private val _navigateToTask = MutableLiveData<Long?>()
    val navigateToTask: LiveData<Long?>
        get() = _navigateToTask
    ...
}
```

Add a private backing property named _navigateToTask.

navigateToTask provides other classes with read-only access to its backing property's value.

As you can see, the `navigateToTask` property uses a mutable backing property marked as `private`, which means that only `TasksViewModel` can set it. This protects the property from unwanted updates by other classes.




## Add methods to update the new property

Each time the user clicks on a task in the recycler view, we want `TasksFragment` to navigate to `EditTaskFragment`, and pass it the task's ID.

To enable this, we'll add two methods to the `TasksViewModel`—`onTaskClicked()` and `onTaskNavigated()`—which will be used to set the value of `navigateToTask`'s backing property. `onTaskClicked()` will set the property to the task's ID, and `onTaskNavigated()` will set it to `null`.

Here's the code for the two methods:

```
fun onTaskClicked(taskId: Long) {
    _navigateToTask.value = taskId
}

fun onTaskNavigated() {
    _navigateToTask.value = null
}
```

This sets _navigateToTask to the task ID that TasksFragment needs to pass to EditTaskFragment.

Those are all the changes we need to make to `TasksViewModel`. We'll show you the full code on the next page.

# The full code for TasksViewModel.kt

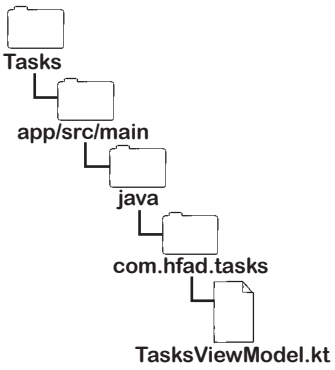
Here's the updated `TasksViewModel` code; make sure that the code in *TasksViewModel.kt* includes all the changes shown here (in bold):

```
package com.hfad.tasks

import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch

class TasksViewModel(val dao: TaskDao) : ViewModel() {
    var newTaskName = ""
    val tasks = dao.getAll()
    private val _navigateToTask = MutableLiveData<Long?>()
    val navigateToTask: LiveData<Long?>
        get() = _navigateToTask

    fun addTask() {
        viewModelScope.launch {
            val task = Task()
            task.taskName = newTaskName
            dao.insert(task)
        }
    }

    fun onTaskClicked(taskId: Long) {
        _navigateToTask.value = taskId
    }

    fun onTaskNavigated() {
        _navigateToTask.value = null
    }
}
```

Import these classes.

Add the navigateToTask property, along with its backing property _navigateToTask.

Add these two methods.




Now that we've updated `TasksViewModel`, let's see what changes we need to make to the `TasksFragment` code.

# Make TasksFragment navigate to EditTaskFragment

We need to update the code for `TasksFragment` so that when the user clicks on a task, it navigates to `EditTaskFragment`, passing it the task's ID.

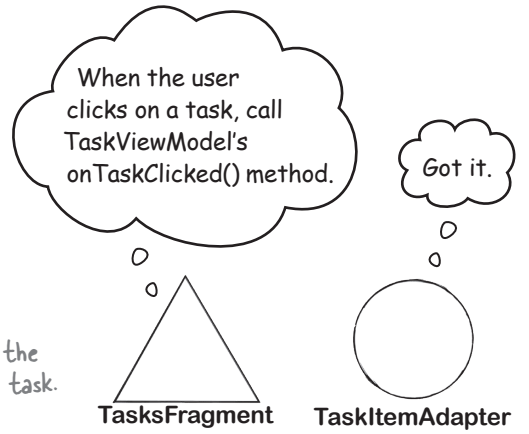
To do this, we'll call `TasksViewModel`'s `onTaskClicked()` method when the user clicks on a task, and navigate to `EditTaskFragment` when its `navigateToTask` property gets updated to a new task ID.

## Call onTaskClicked() when the user clicks a task

To call the `onTaskClicked()` method, we'll pass the lambda below (in bold) to `TaskItemAdapter`'s constructor:

```
val adapter = TaskItemAdapter { taskId ->
    viewModel.onTaskClicked(taskId)
}
binding.tasksList.adapter = adapter
```

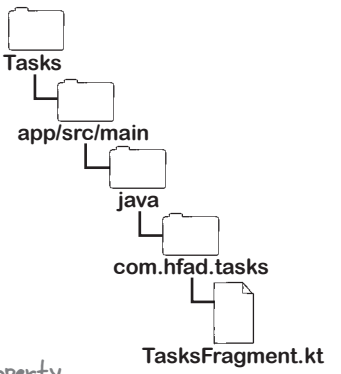
This will get executed when the user clicks on a task.




Each time the user clicks on a task, the lambda will get executed: it will call `TasksViewModel`'s `onTaskClicked()` method, setting the `navigateToTask` property to the task's ID.

## Navigate to EditTaskFragment when navigateToTask gets updated

To make `TasksFragment` navigate to `EditTaskFragment`, we'll make it observe `TaskViewModel`'s `navigateToTask` property. When this property is set to a `Long` task ID, the fragment will navigate to `EditTaskFragment`, passing it the ID. We'll then set the `navigateToTask` property back to null by calling the view model's `onTaskNavigated()` method.




Here's the code to do this:

Observe the navigateToTask property.

```
viewModel.navigateToTask.observe(viewLifecycleOwner, Observer { taskId ->
    taskId?.let {
        val action = TasksFragmentDirections
                .actionTasksFragmentToEditTaskFragment(taskId)
        this.findNavController().navigate(action)
        viewModel.onTaskNavigated()
    }
})
```

The let block will only run if taskId is not null.

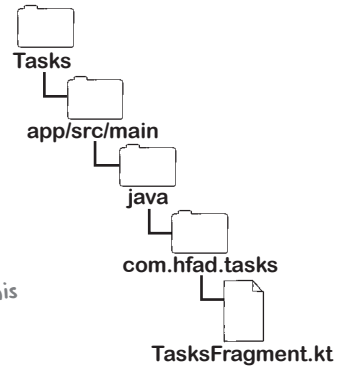
Navigate to EditTaskFragment and pass it the taskId.

Call onTaskNavigated() to set the navigateToTask property to null.

Let's update the `TasksFragment` code.

# The full code for TasksFragment.kt

Here's the updated `TasksFragment` code; make sure that the code in includes all the changes shown here (in bold):



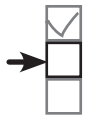

```
package com.hfad.tasks

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Toast
import androidx.lifecycle.Observer
import androidx.lifecycle.ViewModelProvider
import androidx.navigation.fragment.findNavController
import com.hfad.tasks.databinding.FragmentTasksBinding

class TasksFragment : Fragment() {
    private var _binding: FragmentTasksBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
            inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentTasksBinding.inflate(inflater, container, false)
        val view = binding.root

        val application = requireNotNull(this.activity).application
        val dao = TaskDatabase.getInstance(application).taskDao

        val viewModelFactory = TasksViewModelFactory(dao)
        val viewModel = ViewModelProvider(
                this, viewModelFactory).get(TasksViewModel::class.java)
        binding.viewModel = viewModel
        binding.lifecycleOwner = viewLifecycleOwner
```

We no longer need this import so remove it.

Import this class.

The code continues on the next page.

# The TasksFragment.kt code continued



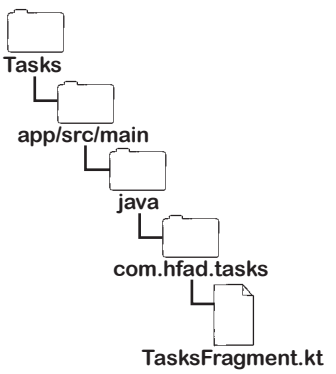
**Respond to clicks**
**Navigate**
**Show record**

```
        val adapter = TaskItemAdapter { taskId ->
            Toast.makeText(context, "Clicked task $taskId", Toast.LENGTH_SHORT).show()
            viewModel.onTaskClicked(taskId)
        }
        binding.tasksList.adapter = adapter

        viewModel.tasks.observe(viewLifecycleOwner, Observer {
            it?.let {
                adapter.submitList(it)
            }
        })

        viewModel.navigateToTask.observe(viewLifecycleOwner, Observer { taskId ->
            taskId?.let {
                val action = TasksFragmentDirections
                        .actionTasksFragmentToEditTaskFragment(taskId)
                this.findNavController().navigate(action)
                viewModel.onTaskNavigated()
            }
        })

        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Call this method instead of displaying a toast.

Observe the navigateToTask property.

Run this code when navigateToTask is set to a new taskId that's not null.



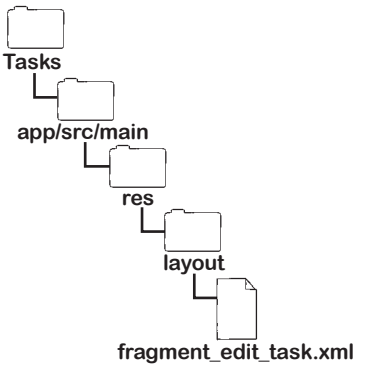

We've now updated the code for `TasksViewModel` and `TasksFragment` so that when the user clicks on a task in the recycler view, `TasksFragment` navigates to `EditTaskFragment` and passes it the task's ID.

The next thing we need to do is display the task ID in `EditTaskFragment`'s layout. Let's do this now.

# Make EditTaskFragment display the task ID

We're going to make `EditTaskFragment` display the task ID by updating the fragment's layout and Kotlin code. We'll add a text view to the layout, and then use Kotlin code to retrieve the task's ID and add it to the text view.

We'll begin by adding the text view to the fragment's layout; update *fragment_edit_task.xml* so that it matches the code shown here:




```
<?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:orientation="vertical"
    android:padding="16dp"
    tools:context=".EditTaskFragment">

    <TextView
        android:id="@+id/task_id"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="16sp" />
</LinearLayout>
```

The layout includes a text view, which will be used to display a task ID.

## We also need to update EditTaskFragment.kt

Now that we've added a text view to `EditTaskFragment`'s layout, we need to set its text to the task ID. To do this, we'll add the following code to the fragment's `onCreateView()` method:

```
val textView = view.findViewById<TextView>(R.id.task_id)
val taskId = EditTaskFragmentArgs.fromBundle(requireArguments()).taskId
textView.text = taskId.toString()
```

Get a reference to the text view.

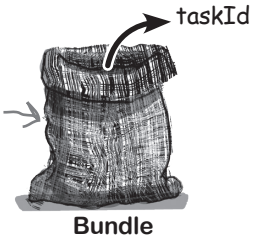
Get the value of taskId.

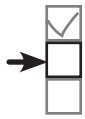
Set the text view's text.

As you can see, this uses the `EditTaskFragmentArgs` class (generated by the Safe Args plug-in) to get the value of the `taskId` argument that's been passed to `EditTaskFragment`. It then uses this to set the text view's text.

Let see what the full code for `EditTaskFragment` looks like.

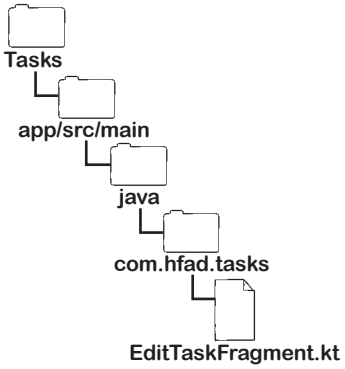
As you learned in Chapter 7, you use the fromBundle() method to get any arguments that have been passed to the fragment.






# The full code for EditTaskFragment.kt

Here's the full code for `EditTaskFragment`; replace the code in *EditTaskFragment.kt* so that it matches the code shown here:




```
package com.hfad.tasks

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.TextView

class EditTaskFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        val view = inflater.inflate(R.layout.fragment_edit_task, container, false)
        val textView = view.findViewById<TextView>(R.id.task_id)
        val taskId = EditTaskFragmentArgs.fromBundle(requireArguments()).taskId
        textView.text = taskId.toString()
        return view
    }
}
```

Display the task ID in the layout's text view.

Let's go through what happens when the code runs.

## there are no Dumb Questions

**Q: In the `EditTaskFragment` code, I see that you're using the `findViewById()` method to get a reference to the layout's text view. Why aren't you using view binding or data binding?**

A: We're using `findViewById()` here because it's the quickest way of testing whether the task ID has been successfully passed to `EditTaskFragment`.

We'll update the code to use data binding once we're sure this version of the code works.

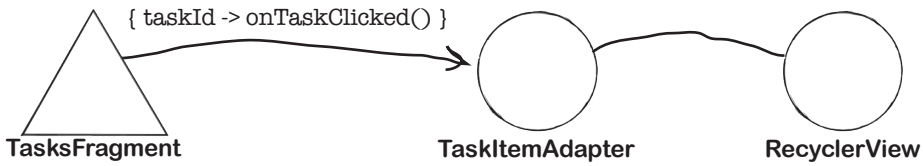
**Q: When I tried to update my Kotlin code, Android Studio told me that it couldn't find the `TasksFragmentDirections` or `EditTaskFragmentArgs` classes. Any idea why that might be?**

A: These classes are generated by the Safe Args plug-in, so make sure you've updated your *build.gradle* files to include it. Then double-check your navigation graph to make sure `EditTaskFragment` accepts an argument.

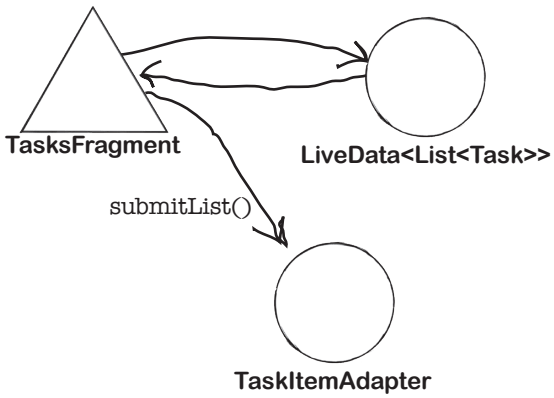
# What happens when the code runs

The following things happen when the app runs:

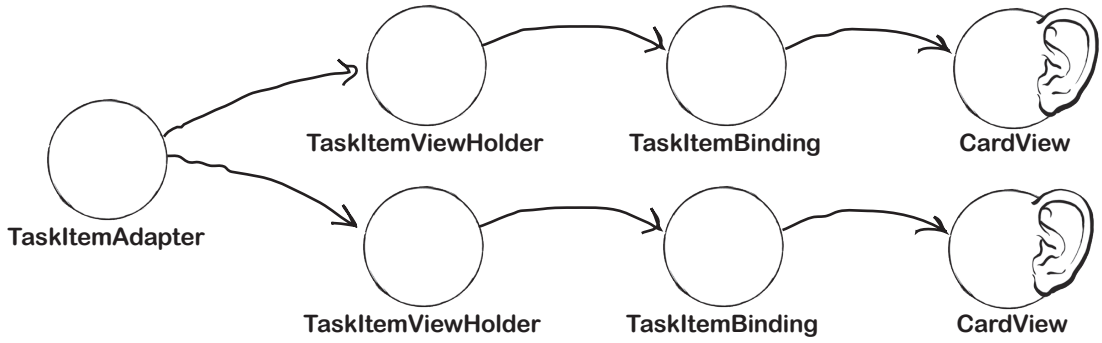
1. **TasksFragment creates a TaskItemAdapter object, and assigns it to the recycler view as its adapter.**
The fragment passes a `clickListener` lambda to the adapter, telling it to call `TasksViewModel`'s `onTaskClicked()` method when it's executed.



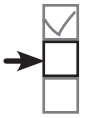

2. **TasksFragment submits a List<Task> to the TaskItemAdapter.**
The `List<Task>` contains an up-to-date list of records from the database.




3. **TaskItemAdapter creates a set of TaskItemViewHolders, and an OnClickListener is set to each view holder's root view.**
In this example, the root view is a `CardView`.

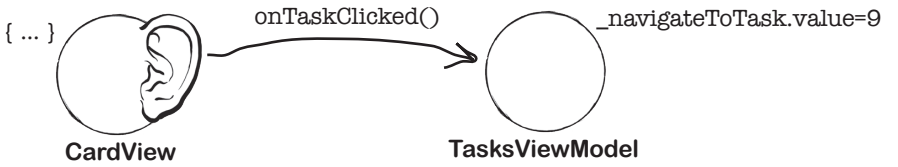


# The story continues

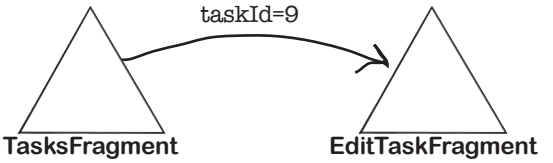


**Respond to clicks**
**Navigate**
**Show record**

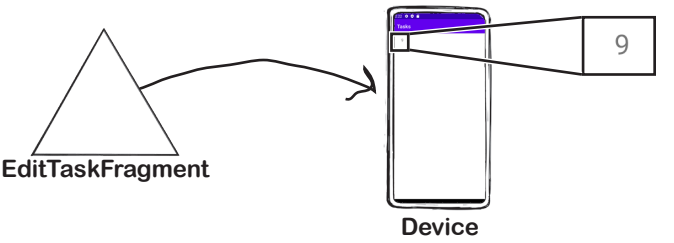
4 **When the user clicks on a task in the recycler view, the OnClickListener registers the click, and executes the lambda.**
It calls `TasksViewModel`'s `onTaskClicked()` method, which sets its `_navigateToTask` property to the ID of the task that was clicked.




5 **TasksFragment is notified that TasksViewModel's navigateToTask property—which uses _navigateToTask as its backing property—has been updated.**
It navigates to `EditTaskFragment`, passing it the value of the `navigateToTask` property.




6 **EditTaskFragment gets the value of the taskId that's been passed to it, and displays it in its layout.**

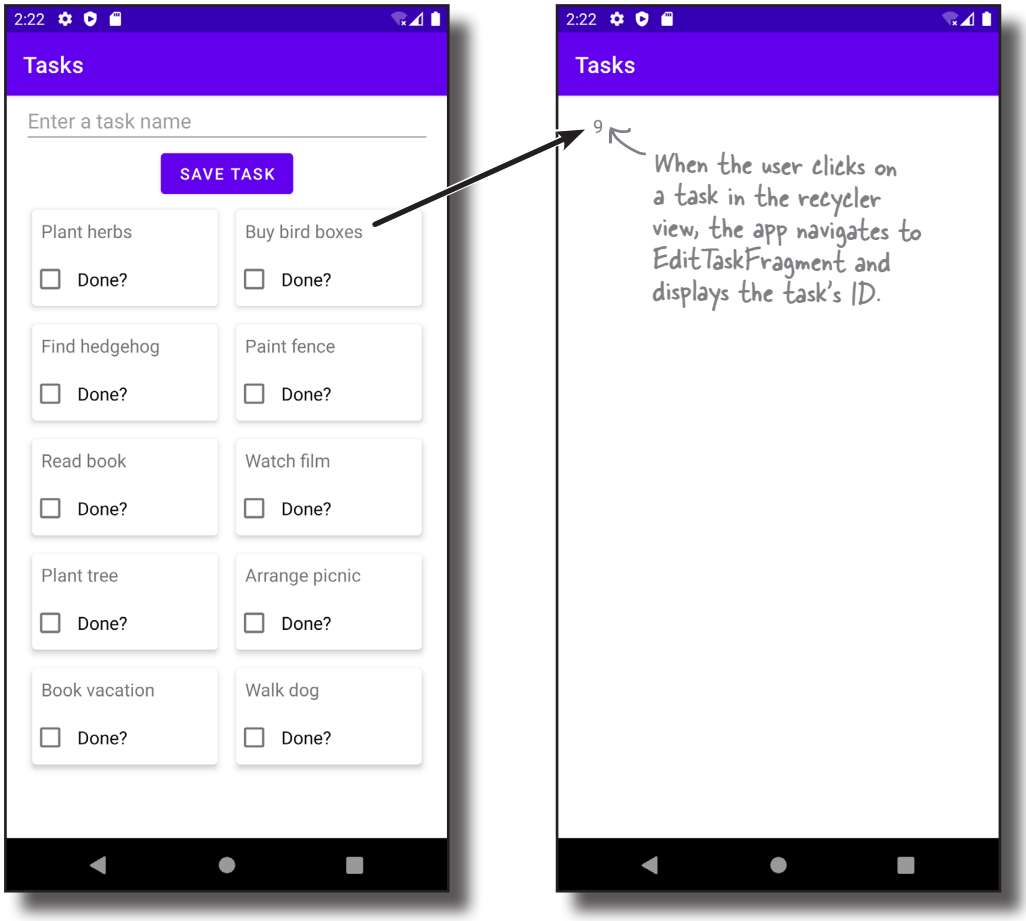



Let's take the app for a test drive.

# Test Drive

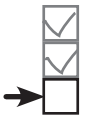
When we run the app, `TasksFragment` displays a grid of cards in the recycler view as before.

When we click on one of the tasks, the app navigates to `EditTaskFragment`, which displays the task's ID.




You've now learned how to use a recycler view to navigate to a new fragment, and tell it which item was clicked.

Next, we'll update the Tasks app so that when the user clicks on an item, `EditTaskFragment` displays the task's complete details, and lets her update or delete the record in the database.



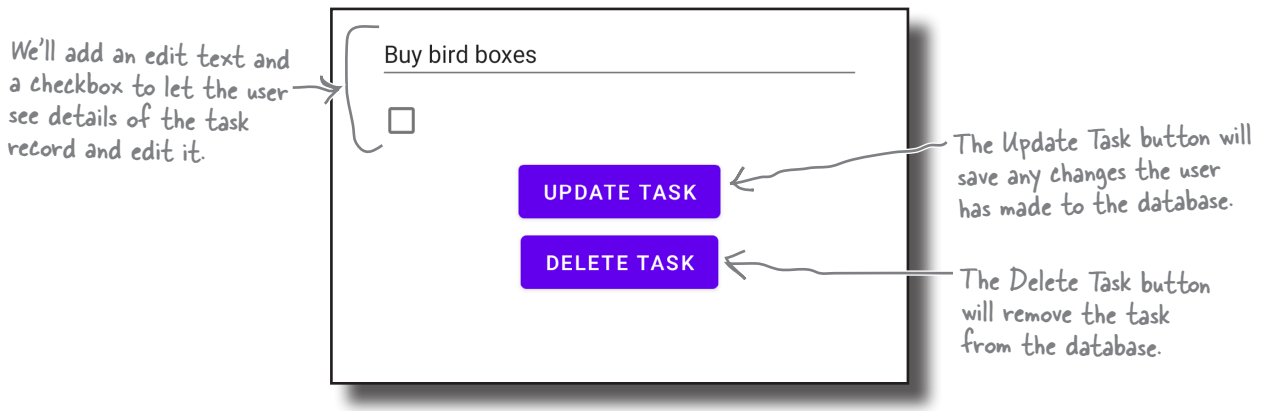
**Respond to clicks**
**Navigate**
**Show record**

# We want to use EditTaskFragment to update task records

We've now updated the Tasks app so that when the user clicks on a task in the recycler view, it navigates to `EditTaskFragment`, which displays the task's ID.

What we *really* want, however, is for `EditTaskFragment` to display the complete task record, and let the user update or delete it in the database.

To do this, we're going to update `EditTaskFragment` so that it looks like this:




The new version of the fragment will work as follows:

1. **When the user clicks on a task in the recycler view, EditTaskFragment displays its details.**
   It gets the task record from the database, and displays the task name and whether it has been completed.

2. **When the user updates the task's details and clicks on the Update Task button, the changes are saved.**
   It updates the record in the database, and navigates back to `TasksFragment`.

3. **When the user clicks on the Delete Task button, the task is deleted.**
   It deletes the task record from the database, and navigates to `TasksFragment`.

For each of these actions, the fragment needs to interact with the app's Room database, which means that it needs to use the `TaskDao` interface we defined in Chapter 14. Before we update the app, let's have a quick reminder of what `TaskDao` does.

# Use TaskDao to interact with database records

As you learned in Chapter 14, you interact with records held in a Room database using a DAO interface. The Tasks app, for example, includes a DAO named `TaskDao` that lets us interact with task records.

Here's a reminder of what the `TaskDao` code looks like:

```
...
@Dao
interface TaskDao {
    @Insert
    suspend fun insert(task: Task)

    @Update
    suspend fun update(task: Task)

    @Delete
    suspend fun delete(task: Task)

    @Query("SELECT * FROM task_table WHERE taskId = :key")
    fun get(key: Long): LiveData<Task>

    @Query("SELECT * FROM task_table ORDER BY taskId DESC")
    fun getAll(): LiveData<List<Task>>
}
```

Insert a task record into the database.

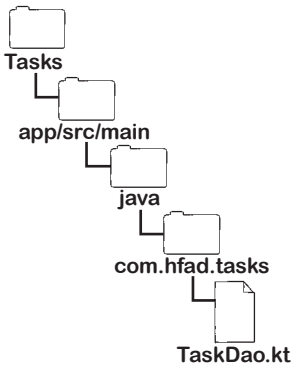
These methods are marked with suspend, which means that they are coroutines.

Update a task record.

Delete this task.

This gets the task record with a matching ID, and returns a live data version of it.

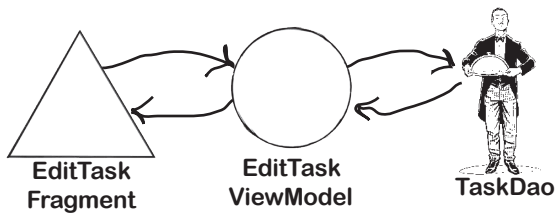
This returns a live data list of all tasks.




As you can see, the interface includes methods to get one or more records from the database, and suspendable coroutines to insert, update, and delete records in a background thread.

## We'll create a view model to access TaskDao's methods

We need to use `TaskDao`'s methods to let `EditTaskFragment` get a task record from the database, update its details, or delete it. Instead of adding the code for this to `EditTaskFragment`, we'll create a new view model (named `EditTaskViewModel`), which will take care of the fragment's business logic and data. `EditTaskViewModel` will access `TaskDao`'s methods, and pass the results to `EditTaskFragment`.




Let's go and create `EditTaskViewModel`.

**Respond to clicks**
**Navigate**
**Show record**

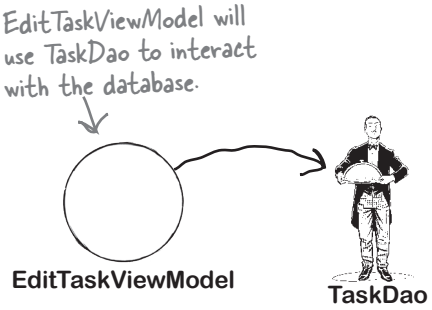
# Create EditTaskViewModel

To create `EditTaskViewModel`, highlight the *com.hfad.tasks* package in the *app/src/main/java* folder, then go to File→New→Kotlin Class/File. Name the file "EditTaskViewModel" and choose the Class option.

## The view model needs to get a task record...

The first thing that `EditTaskViewModel` needs to do is get a task record from the app's database so that it can be displayed in `EditTaskFragment`'s layout. To enable this, we'll pass the view model two things in its constructor: a task ID to tell it which task to get, and a `TaskDao` object that it will use to interact with the database.

We'll also add a `LiveData<Task>` property to the view model (named `task`), which we'll set using `TaskDao`'s `get()` method. This will set the property to the task record the user wants to see.




Here's the code to do this; we'll show you the full `EditTaskViewModel` code a couple pages ahead:

EditTaskViewModel will accept a task ID and TaskDao object.

```
class EditTaskViewModel(taskId: Long, val dao: TaskDao) : ViewModel() {
    val task = dao.get(taskId)
}
```

This sets the task property to a LiveData<Task>.

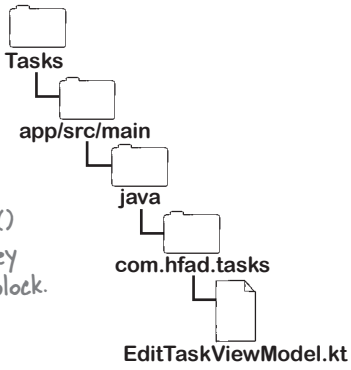
## ...and include methods to update and delete the task

We'll also add `updateTask()` and `deleteTask()` methods to `EditTaskViewModel`, which `EditTaskFragment` will use to update or delete task records. These methods will call `TaskDao`'s `update()` and `delete()` coroutines like so:

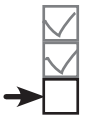
```
fun updateTask() {
    viewModelScope.launch {
        dao.update(task.value!!)
    }
}

fun deleteTask() {
    viewModelScope.launch {
        dao.delete(task.value!!)
    }
}
```

TaskDao's update() and delete() methods are coroutines, so they need to be called in a launch block.

If the Task is null, this throws a null pointer exception.



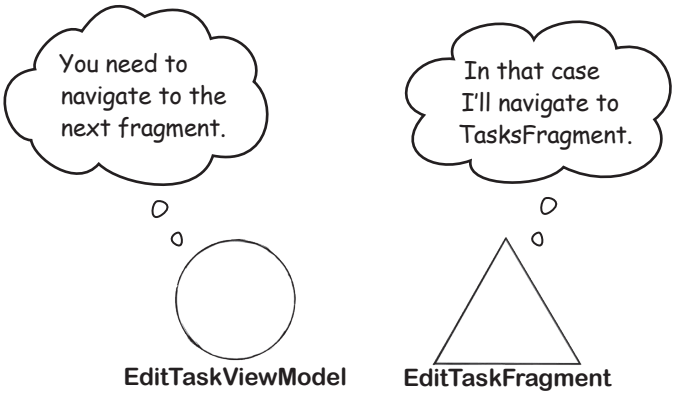

Before we add this code to `EditTaskViewModel`, let's see what else the view model code needs to do.



**Respond to clicks**
**Navigate**
**Show record**

# EditTaskViewModel will tell EditTaskFragment when to navigate

The final thing `EditTaskViewModel` needs to do is tell `EditTaskFragment` when it should navigate back to `TasksFragment`. To enable this, we'll add a new `LiveData<Boolean>` property (named `navigateToList`) to the view model, along with a backing property named `_navigateToList`. `EditTaskFragment` will observe `navigateToList` so that when it's value becomes *true*, it navigates to `TasksFragment`.

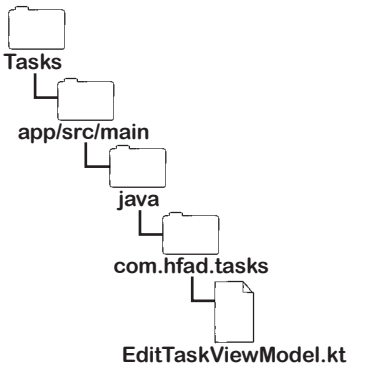



We'll set `_navigateToList` to *true* in the view model's `updateTask()` and `deleteTask()` methods. This means that as soon as a task record has been updated or deleted, the app will navigate to `TasksFragment`.

Here's the updated code for these methods:

```
fun updateTask() {
    viewModelScope.launch {
        dao.update(task.value!!)
        _navigateToList.value = true
    }
}

fun deleteTask() {
    viewModelScope.launch {
        dao.delete(task.value!!)
        _navigateToList.value = true
    }
}
```

When this property is set to true, EditTaskFragment needs to navigate to TasksFragment.




We'll also add a new method to `EditTaskViewModel` named `onNavigatedToList()`, which sets `_navigateToList` back to *false*.

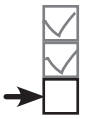
Here's the code for this method:

```
fun onNavigatedToList() {
    _navigateToList.value = false
}
```

EditTaskFragment will call this method when it's done navigating.

We'll show you the entire code for `EditTaskViewModel` on the next page.

# The full code for EditTaskViewModel.kt



**Respond to clicks**
**Navigate**
**Show record**

Here's the full `EditTaskViewModel` code; make sure that the code in *EditTaskViewModel.kt* includes all the changes shown here (in bold):

```
package com.hfad.tasks

import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch

class EditTaskViewModel(taskId: Long, val dao: TaskDao) : ViewModel() {
    val task = dao.get(taskId)
    private val _navigateToList = MutableLiveData<Boolean>(false)
    val navigateToList: LiveData<Boolean>
        get() = _navigateToList

    fun updateTask() {
        viewModelScope.launch {
            dao.update(task.value!!)
            _navigateToList.value = true
        }
    }

    fun deleteTask() {
        viewModelScope.launch {
            dao.delete(task.value!!)
            _navigateToList.value = true
        }
    }

    fun onNavigatedToList() {
        _navigateToList.value = false
    }
}
```

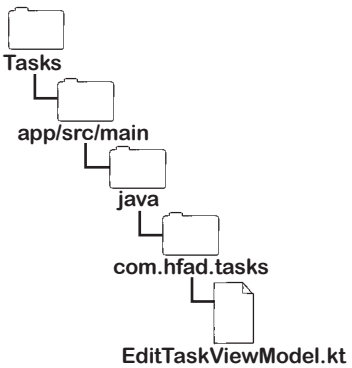
Import these classes.

The view model accepts these arguments.

Add these properties.

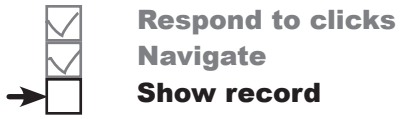
Add methods to update and delete the task.

Add a method to set _navigateToList back to false.

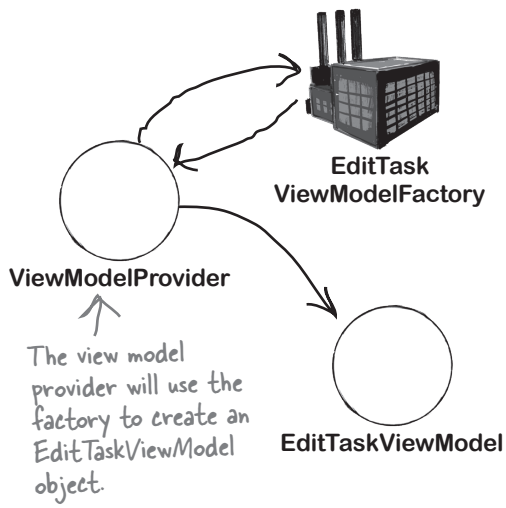



That's all the code we need for `EditTaskViewModel`. So what's next?

# EditTaskViewModel needs a view model factory



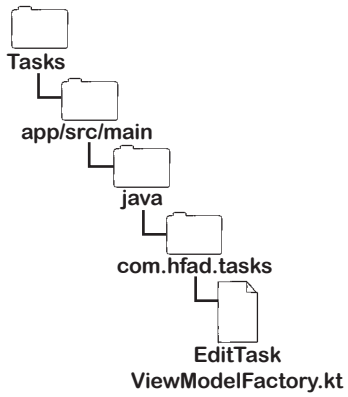

The next thing we'll do is define a view model factory named `EditTaskViewModelFactory`, which `EditTaskFragment` will use to create an instance of `EditTaskViewModel`. As you learned in Chapter 11, a view model factory is needed for all view models that—like `EditTaskViewModel`—don't include a no-argument constructor.




## Create EditTaskViewModelFactory

To create the factory, highlight the *com.hfad.tasks* package in the *app/src/main/java* folder, then go to File→New→Kotlin Class/File. Name the file "EditTaskViewModelFactory" and choose the Class option.

Once you've created the file, update the code in *EditTaskViewModelFactory.kt* so that it looks like this:




```
package com.hfad.tasks

import androidx.lifecycle.ViewModel
import androidx.lifecycle.ViewModelProvider

class EditTaskViewModelFactory(private val taskId: Long,
                               private val dao: TaskDao)
            : ViewModelProvider.Factory {

    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(EditTaskViewModel::class.java)) {
            return EditTaskViewModel(taskId, dao) as T
        }
        throw IllegalArgumentException("Unknown ViewModel")
    }
}
```

Import these classes.

The factory needs these arguments for it to create an EditTaskViewModel.

Create an EditTaskViewModel object.

This code is virtually identical to all the other view model factory classes you've created for other apps.

Now that we've written the code for `EditTaskViewModel` and its factory, let's update the code for `EditTaskFragment` and its layout. We'll begin with the layout.

# fragment_edit_task.xml needs to display the task

We're going to update *fragment_edit_task.xml* so that it includes an edit text and a checkbox, which will be used to display `Task` data. We'll use data binding to bind these views to the `taskName` and `taskDone` properties in `EditTaskViewModel`'s `task`.

We'll also add two buttons to the layout, which will call `EditTaskViewModel`'s `deleteTask()` and `updateTask()` methods, and let the user update or delete the task record.

Here's the updated code for the layout; update the code for *fragment_edit_task.xml* so that it includes these changes (in bold):

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

    <data>
        <variable
            name="viewModel"
            type="com.hfad.tasks.EditTaskViewModel" />
    </data>

    <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:orientation="vertical"
        android:padding="16dp"
        ~~tools:context=".EditTaskFragment"~~ >

        <~~TextView~~ EditText
            ~~android:id="@+id/task_id"~~
            android:id="@+id/task_name"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textSize="16sp"
            android:inputType="text"
            android:text="@={viewModel.task.taskName}" />
```

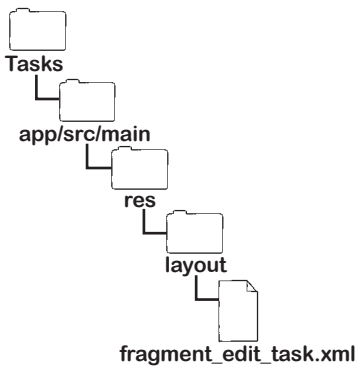
Add a <layout> element to the layout's root.

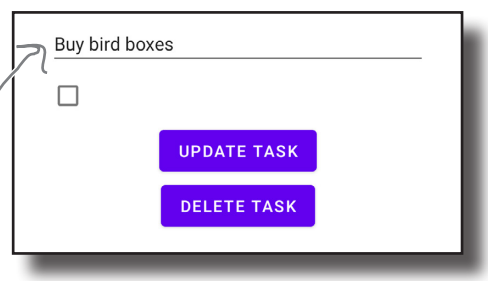
Define a data binding variable named viewModel.

Move these lines to the <layout> element.

Change the TextView to an EditText, and give it a different ID.

Bind the text to the task's name.




Display the task name in an edit text.

The code continues on the next page.

# fragment_edit_task.xml continued

Add a checkbox, and bind it to the task's taskDone property.

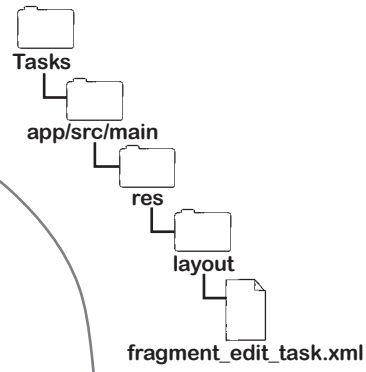
```
<CheckBox
    android:id="@+id/task_done"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textSize="16sp"
    android:checked="@={viewModel.task.taskDone}" />
```

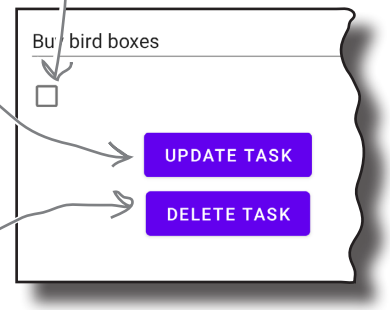
Add a button that calls the view model's updateTask() method.

```
<Button
    android:id="@+id/update_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:text="Update Task"
    android:onClick="@{() -> viewModel.updateTask()}" />
```

Add a second button that calls the view model's deleteTask() method.

```
<Button
    android:id="@+id/delete_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:text="Delete Task"
    android:onClick="@{() -> viewModel.deleteTask()}" />
    </LinearLayout>
</layout>
```



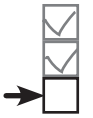

## We need to update EditTaskFragment.kt too

The final change that we need to make to the Tasks app is to update `EditTaskFragment`'s Kotlin code. The code needs to do three things:

1. **Set the layout's viewModel data binding variable.**
   We'll set it to an instance of `EditTaskViewModel`, which the fragment will create.
2. **Set the layout's lifecycle owner.**
   This is so that the layout can interact with live data properties.
3. **Observe the view model's navigateToList property.**
   When this becomes *true*, the fragment will navigate to `TasksFragment`, and call `EditTaskViewModel`'s `onNavigatedToList()` method.

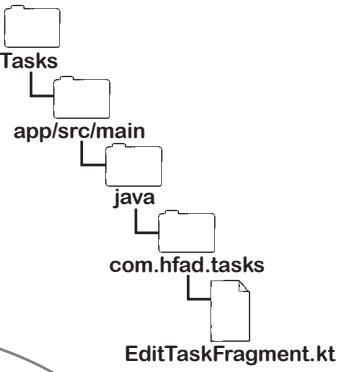
As you're already familiar with all the code to do this, we'll show you the updated code for `EditTaskFragment` on the next page.

# The full code for EditTaskFragment.kt



**Respond to clicks**
**Navigate**
**Show record**

Here's the full `EditTaskFragment` code: make sure that the code in *EditTaskFragment.kt* includes all the changes shown here (in bold):




```
package com.hfad.tasks

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
~~import android.widget.TextView~~
import androidx.lifecycle.Observer
import androidx.lifecycle.ViewModelProvider
import androidx.navigation.findNavController
import com.hfad.tasks.databinding.FragmentEditTaskBinding

class EditTaskFragment : Fragment() {
    private var _binding: FragmentEditTaskBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        ~~val view = inflater.inflate(R.layout.fragment_edit_task, container, false)~~
        _binding = FragmentEditTaskBinding.inflate(inflater, container, false)
        val view = binding.root

        val taskId = EditTaskFragmentArgs.fromBundle(requireArguments()).taskId
        ~~val textView = view.findViewById<TextView>(R.id.task_id)~~
        ~~textView.text = taskId.toString()~~

        val application = requireNotNull(this.activity).application
        val dao = TaskDatabase.getInstance(application).taskDao
```

Delete this line.

Import these classes.

The fragment now uses data binding, so add these properties.

Remove this line.

Add these lines.

Delete these lines.

Add these lines. They're needed to create an EditTaskViewModelFactory.

The code continues on the next page.

# EditTaskFragment continued

Respond to clicks
Navigate
**Show record**

```
        val viewModelFactory = EditTaskViewModelFactory(taskId, dao)
        val viewModel = ViewModelProvider(this, viewModelFactory)
                                        .get(EditTaskViewModel::class.java)
        binding.viewModel = viewModel
        binding.lifecycleOwner = viewLifecycleOwner

        viewModel.navigateToList.observe(viewLifecycleOwner, Observer { navigate ->
            if (navigate) {
                view.findNavController()
                    .navigate(R.id.action_editTaskFragment_to_tasksFragment)
                viewModel.onNavigatedToList()
            }
        })
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

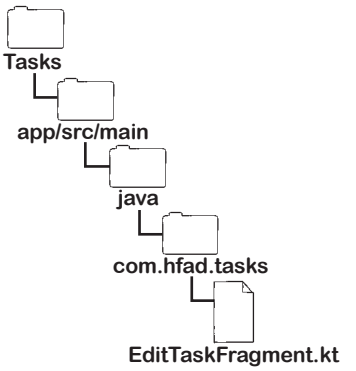
Use a view model factory to get a reference to a view model.

Set the layout's data binding variable.

This lets the layout interact with live data properties.

Navigate to TasksFragment when the navigateToList property is set to true, and call the view model's onNavigatedToList() method.

Add this method.




That's everything that we need to make `EditTaskFragment` display a `Task` in its layout, and let the user update or delete it. Let's go through what the code does when it runs.

## there are no Dumb Questions

**Q: You said that the `updateTask()` and `deleteTask()` methods in `EditTaskViewModel` use coroutines to update and delete task records in a background thread. Is this necessary?**

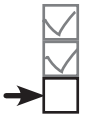
A: Yes. If you try to run potentially time-consuming database code like this in the main thread, you'll get compiler errors.

**Q: Could I add Up button navigation to the app too?**

A: Yes! This is a great idea, as it will give the user a simple way of navigating from `EditTaskFragment` to `TasksFragment` without making any record updates.

We're not including the code to do this here, but by all means have a go at adding it yourself. If you need a reminder of how to implement Up navigation, we suggest referring back to Chapter 8.

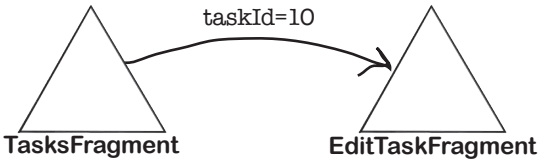
# What happens when the code runs



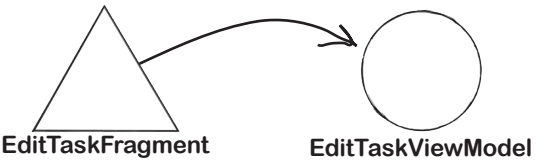
**Respond to clicks**
**Navigate**
**Show record**

The following things happen when the app runs:

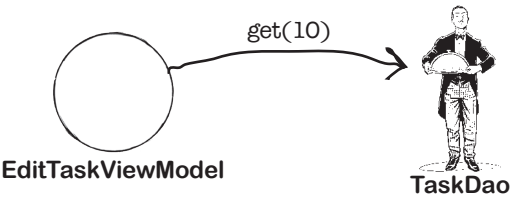
1. **When the user clicks on a task, TasksFragment navigates to EditTaskFragment, passing it the task ID.**



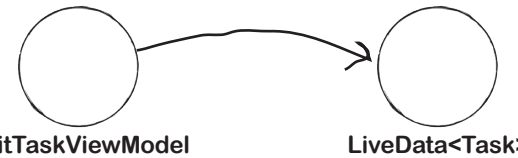

2. **EditTaskFragment gets a reference to its EditTaskViewModel object.**



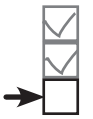

3. **EditTaskViewModel calls the TaskDao object's get() method, passing it the task ID.**




4. **The TaskDao's get() method returns a LiveData<Task>, which is assigned to EditTaskViewModel's task property.**



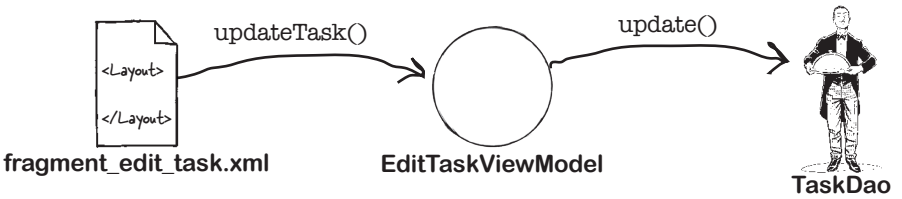
# The story continues



**Respond to clicks**
**Navigate**
**Show record**

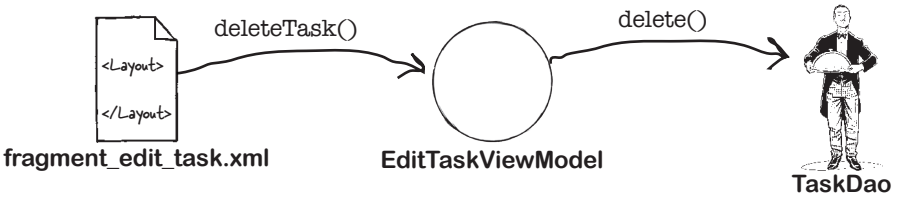
**5a** **When the user clicks on the Update Task button, it calls EditTaskViewModel's updateTask() method.**

This method uses `TaskDao`'s `update()` method to update the record in the database.



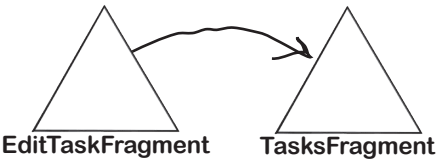

**5b** **When the user clicks on the Delete Task button, it calls EditTaskViewModel's deleteTask() method.**

This method uses `TaskDao`'s `delete()` method to delete the record.




**6** **The app navigates to TasksFragment.**

Any changes that have been made to the task record are reflected in the recycler view.




Let's take the app for a test drive.

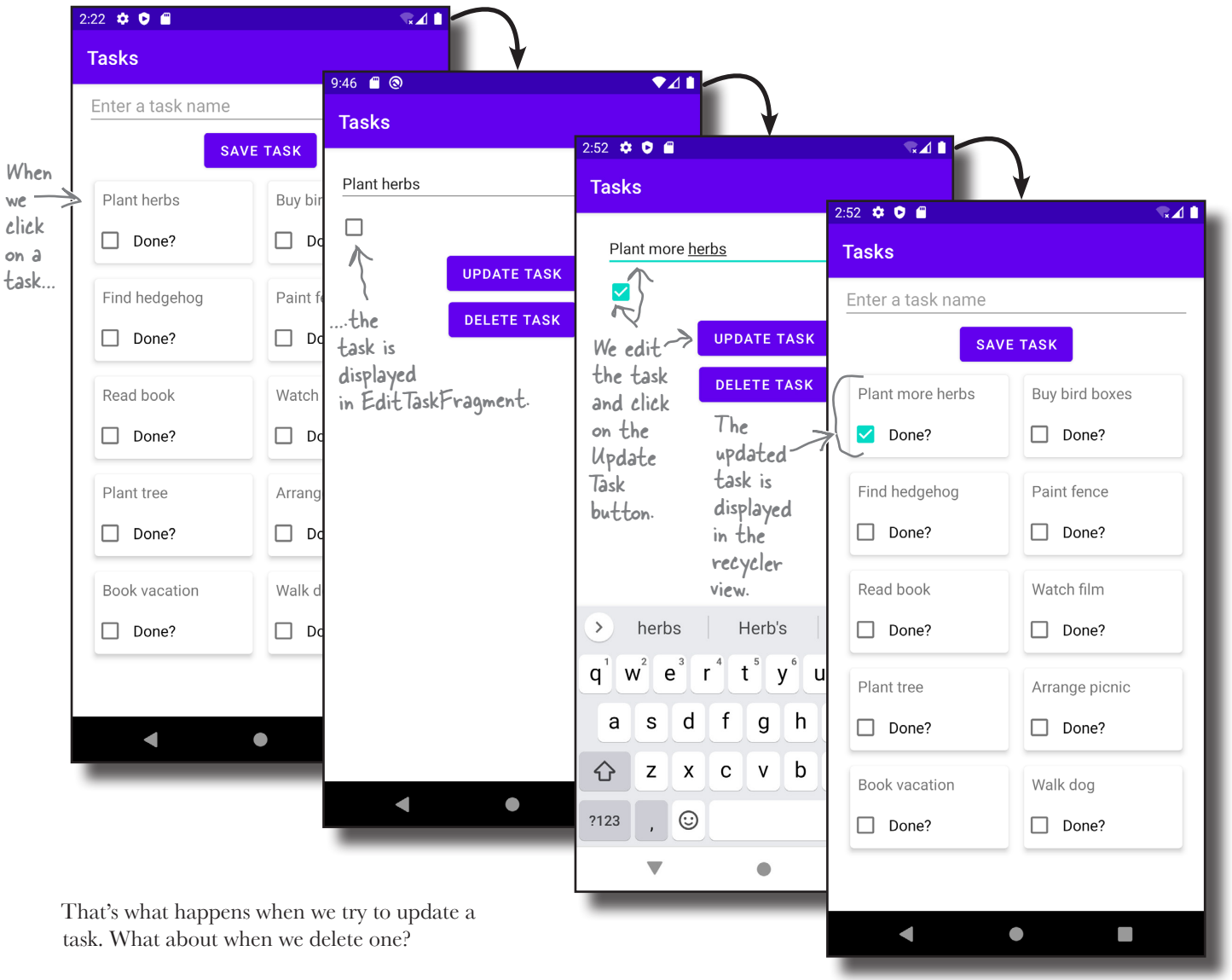
# Test Drive

**Respond to clicks**
**Navigate**
**Show record**

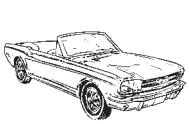
When we run the app, `TasksFragment` displays a grid of cards in the recycler view as before.

When we click on one of the tasks, the app navigates to `EditTaskFragment`, which displays the task's record.

When we make changes to the task and click on the Update Task button, the changes are saved to the database and displayed in `TasksFragment`'s recycler view.




That's what happens when we try to update a task. What about when we delete one?

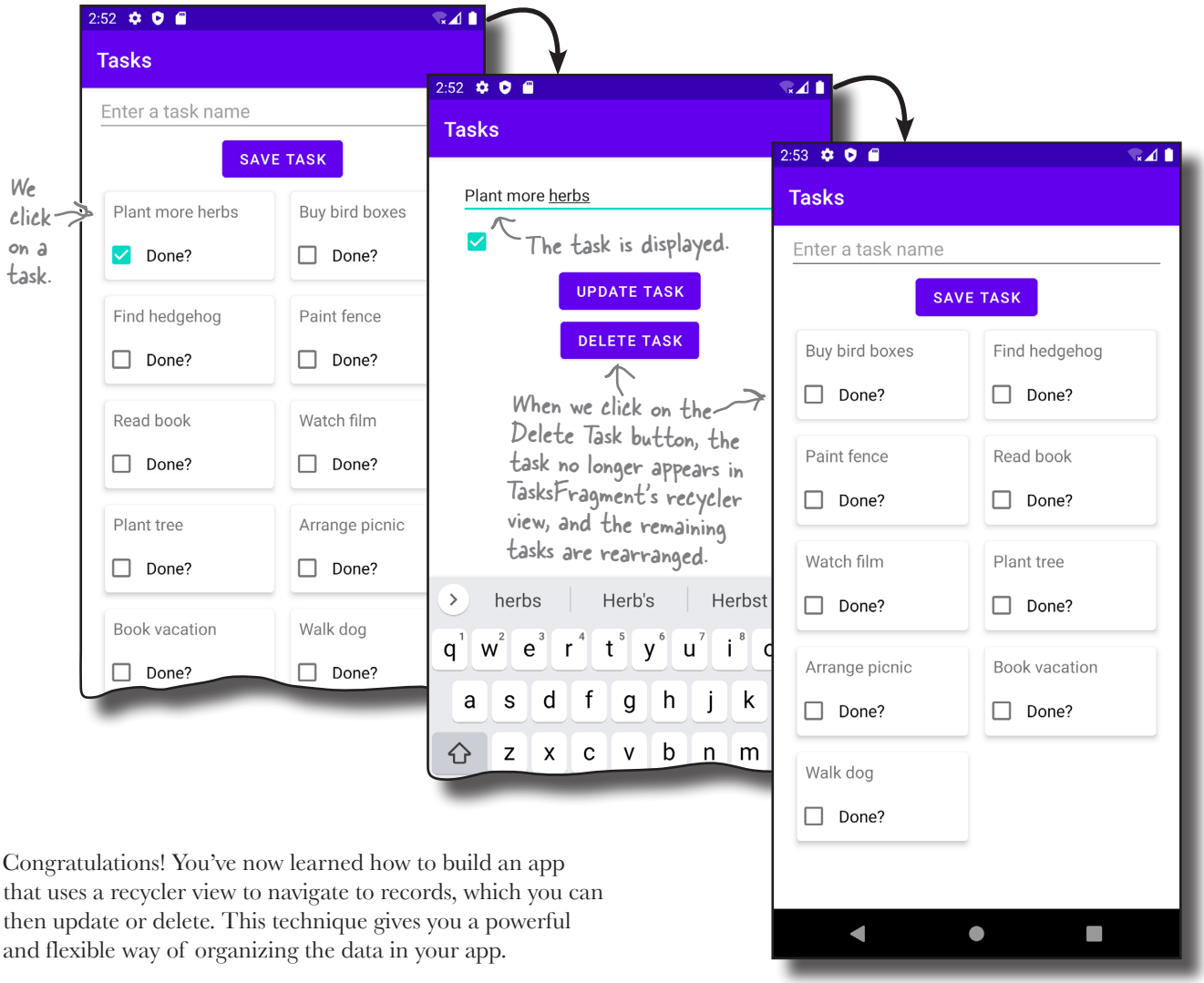


# Test Drive (continued)

**Respond to clicks**
**Navigate**
**Show record**

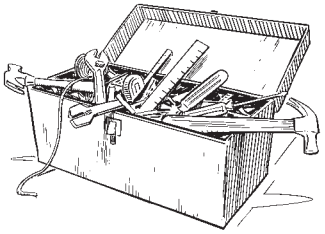
When we click on a task, the app navigates to `EditTaskFragment` and displays the task record as before.

When we click on the Delete Task button, the record is deleted from the database. When the app navigates to `TasksFragment`, the record no longer appears in the recycler view.




Congratulations! You've now learned how to build an app that uses a recycler view to navigate to records, which you can then update or delete. This technique gives you a powerful and flexible way of organizing the data in your app.

# Your Android Toolbox



**You've got Chapter 17 under your belt and now you've added recycler view navigation to your toolbox.**

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

## Bullet Points

- You can make a recycler view's items clickable. You can use this feature to make the recycler view navigate to another fragment, which shows details of the clicked item.
- You can make a recycler view respond to clicks by adding an `OnClickListener` to the root view of the layout it uses for its items.
- You can tell the `OnClickListener` how to respond to clicks by passing it a lambda. The lambda will get executed when the `OnClickListener` registers a click.
- The lambda can be defined in fragment code, and passed to the `OnClickListener` via the adapter's constructor.

# 18 jetpack compose

# 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. Turn the page, and let's get composing…

# UI components don't have to be Views

So far in this book, you've learned how to use layout files and views to build swish, interactive UIs. But while we've focused on this approach, it's not your only option.

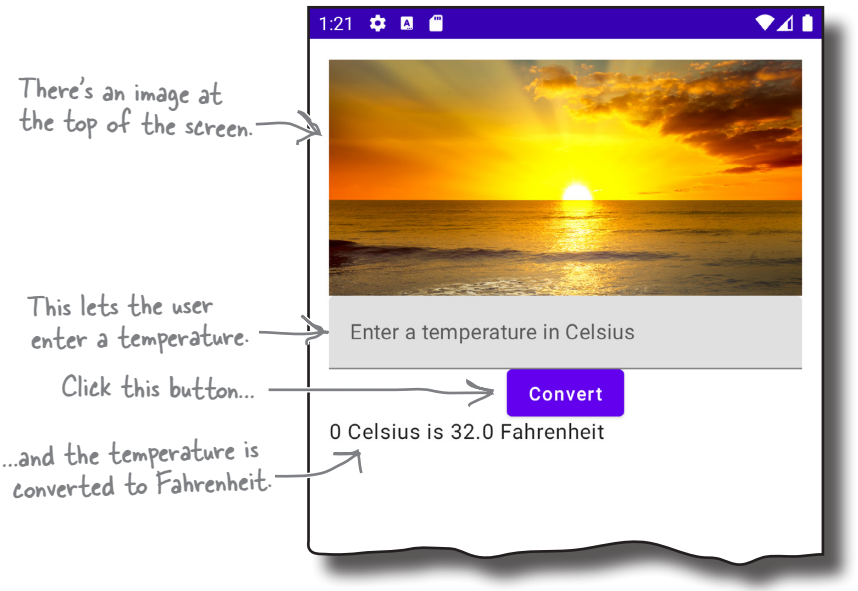
An alternative is to build your UI using **Jetpack Compose**. Compose is part of Android Jetpack; it's an entire toolkit of libraries, tools, and APIs designed to help you build native UIs with pure Kotlin code.

The great news is that using Jetpack Compose builds on your existing Android knowledge. You can use view models and live data with Compose, for example, and even add Compose components to existing UIs.

You'll find out more about this in the next chapter.

## We're going to build a Compose app

In this chapter, we're going to introduce you to Compose by building a new Temperature Converter app that converts temperatures from Celsius to Fahrenheit. Here's what the app will look like:




As you can see, the app uses several familiar looking components. The main difference is that it's written using Compose.

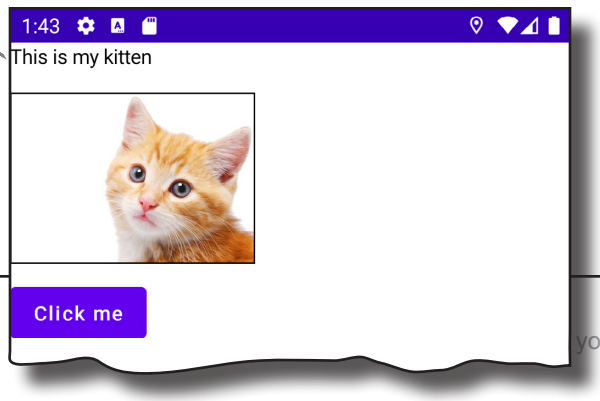
Before we start building the app, see if you can tell what Compose code does by giving the following exercise a try.

# Sharpen your pencil

We know we've not shown you any Compose code yet, but see if you can figure out what each line in the code below does. We've completed the first few to get you started.

```
class MainActivity : ComponentActivity() {        Define an activity that extends ComponentActivity
    override fun onCreate(savedInstanceState: Bundle?) {        Override onCreate()
        super.onCreate(savedInstanceState)        Call onCreate() in the superclass
        setContent {
            Column {
                Text("This is my kitten")

                Spacer(Modifier.height(16.dp))

                Image(
                    painter = painterResource(R.drawable.kitten),
                    contentDescription = "Kitten image",
                    modifier = Modifier.border(1.dp, Color.Black)
                )

                Spacer(Modifier.height(16.dp))

                Button(onClick = { }) {
                    Text("Click me")
                }
            }
        }
    }
}
```

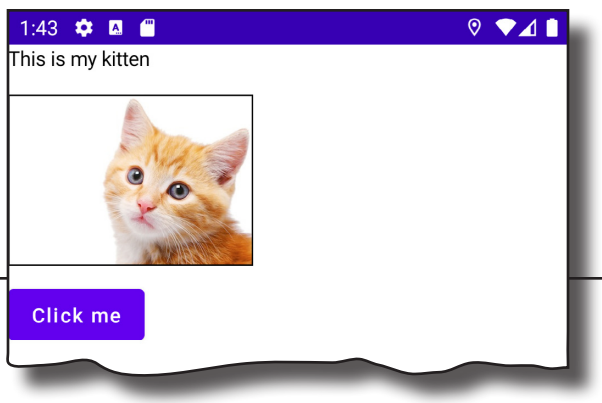
This is what the app looks like when the code runs.



# Sharpen your pencil Solution

We know we've not shown you any Compose code yet, but see if you can figure out what each line in the code below does. We've completed the first few to get you started.

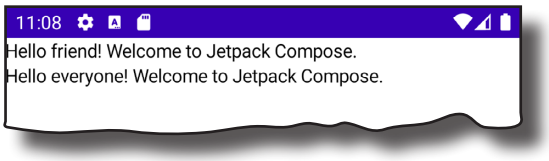
```
class MainActivity : ComponentActivity() {   Define an activity that extends ComponentActivity
    override fun onCreate(savedInstanceState: Bundle?) {   Override onCreate()
        super.onCreate(savedInstanceState)   Call onCreate() in the superclass
        setContent {   Set the activity's content
            Column {   Arrange components in a column
                Text("This is my kitten")   Display some text

                Spacer(Modifier.height(16.dp))   Add a 16dp space

                Image(   Display an image
                    painter = painterResource(R.drawable.kitten),   The image file to use
                    contentDescription = "Kitten image",   An image description
                    modifier = Modifier.border(1.dp, Color.Black)   Add a black border
                )   The Image closing parenthesis

                Spacer(Modifier.height(16.dp))   Add a 16dp space

                Button(onClick = { }) {   Display a button that does nothing when clicked
                    Text("Click me")   Display text on the button
                }   The Button closing parenthesis
            }
        }
    }
}
```



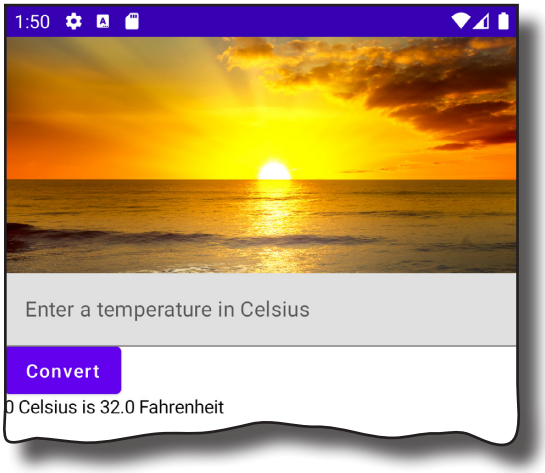
# Here's what we're going to do

Now that you've had a glimpse of some Compose code and seen what it does, let's run through what we'll do in this chapter.

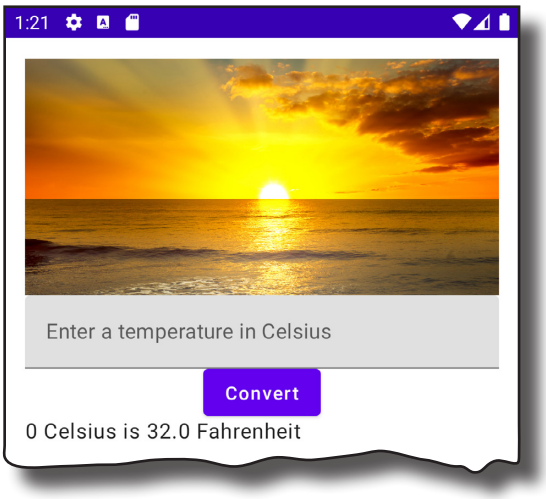
1. **Create an app that displays two text items in a column.**
You'll create a new project that uses Compose to display some hardcoded text. You'll then convert the code into a composable function, and learn how to preview it.



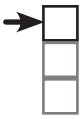

2. **Make the app convert a temperature from Celsius to Fahrenheit.**
In this step, you'll build a UI that lets you enter a temperature in degrees Celsius. When you click a button, it converts the temperature to Fahrenheit.




3. **Modify the app's appearance.**
Finally, you'll learn how to center components, use padding, and apply a theme.



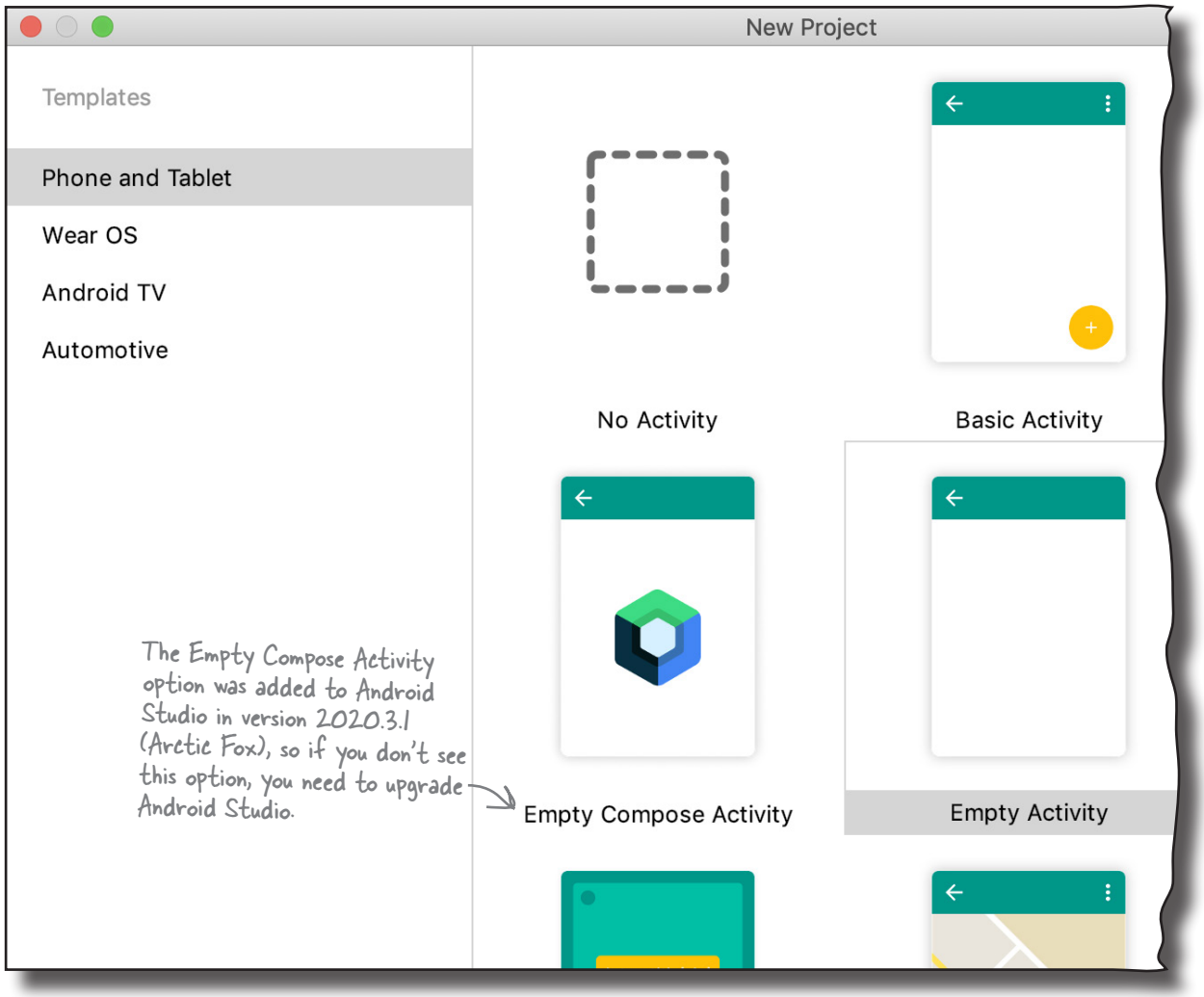

Let's begin by creating a new project for the app.



**Text**
**Converter**
**Modify**

# Create a new Compose project

We're going to create a new Android Studio project that will use Compose for its UI. Create this project now by selecting the **Empty Compose Activity** option:




Choosing this option adds a bunch of Compose libraries and code to your project. It's the best type of project to create if you want to build an Android app from scratch with a Compose UI.

Once you've selected the Empty Compose Activity option, click on the Next button to configure the project.

# Configure the project

The next screen should look familiar to you, as it includes the same options that you've used to configure projects throughout this book.

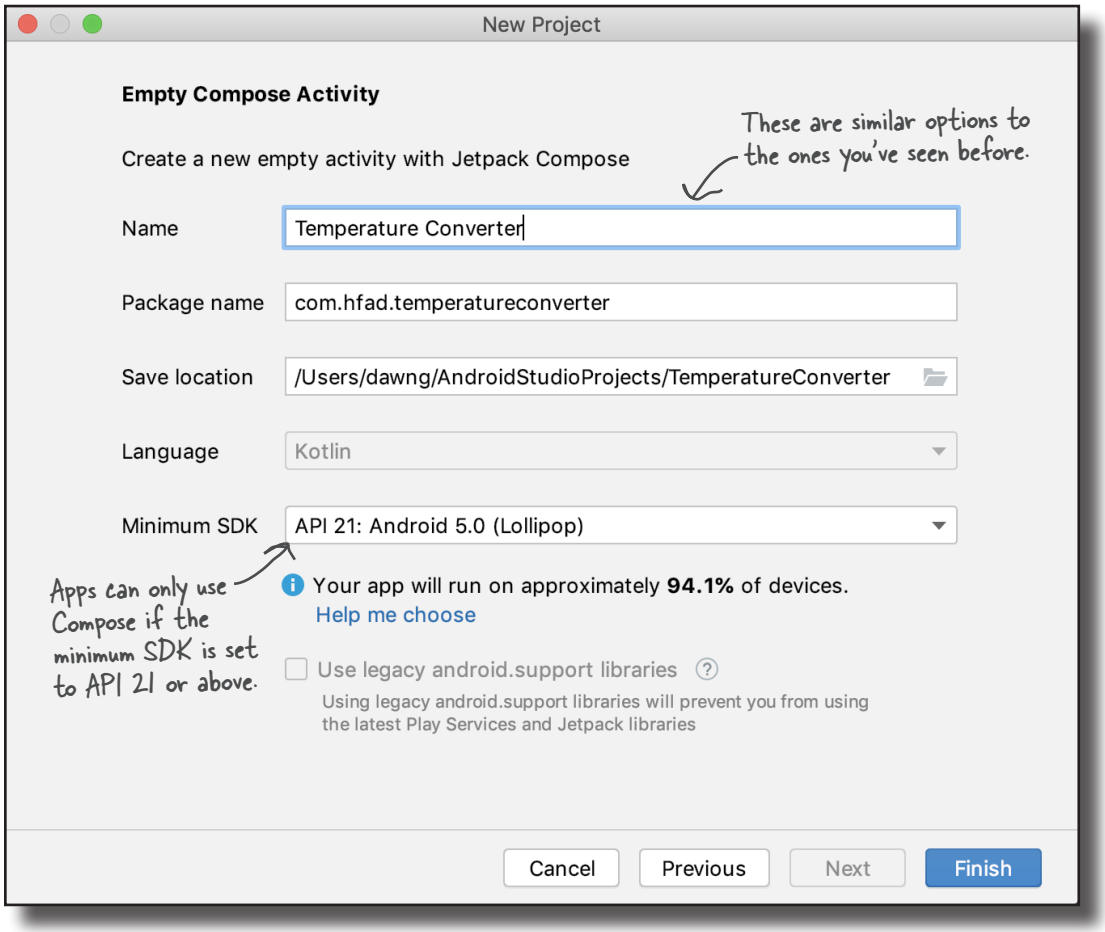
Enter a name of "Temperature Converter" and a package name of "com.hfad.temperatureconverter" and accept the default save location.

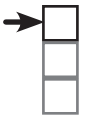
Notice that the language is set to Kotlin, and can't be updated. Compose apps can *only* be created using Kotlin, so you can't select a different language.

Choose a minimum SDK of API 21 so that the app will run on most Android devices. This is the oldest SDK that's available for this type of project, because Compose is only compatible with API 21 and above.

Once you've selected these options, click on the Finish button.

**Compose UIs can only be created using Kotlin.**

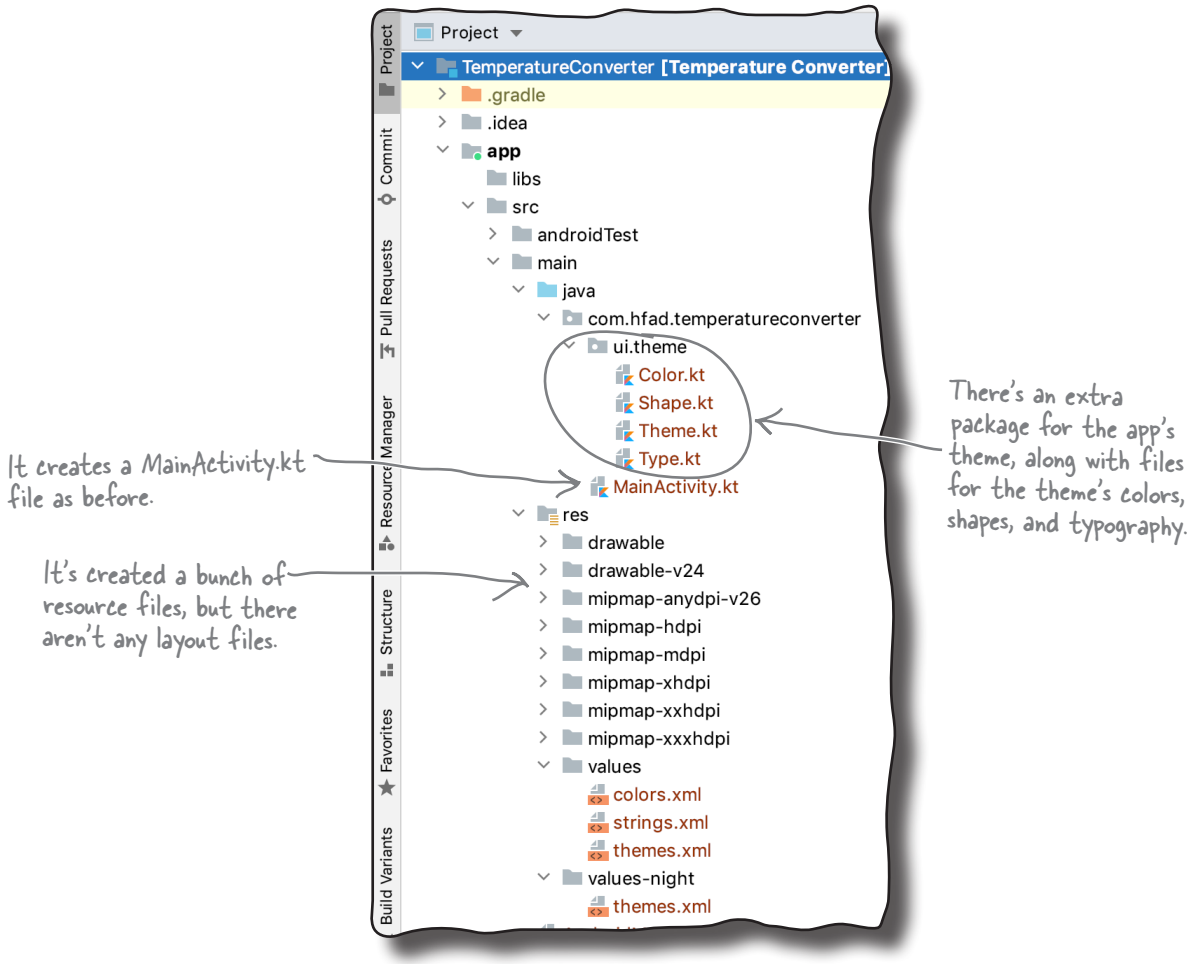




**Text**
**Converter**
**Modify**

# Compose projects have no layout files

When you create a project using the Empty Compose Activity option, Android Studio creates a folder structure for you, populated with all the files the new project needs. The folder structure looks like this:




Many of the files and folders should look familiar to you, as they're the same ones that get generated for projects that don't use Compose. It includes an activity file named *MainActivity.kt*, for example, and a `String` resource file named *strings.xml*.

The big difference is that **Android Studio doesn't generate any layout files for you**. This is because Compose projects use activity code to define the screen's appearance instead of layouts.

# What Compose activity code looks like

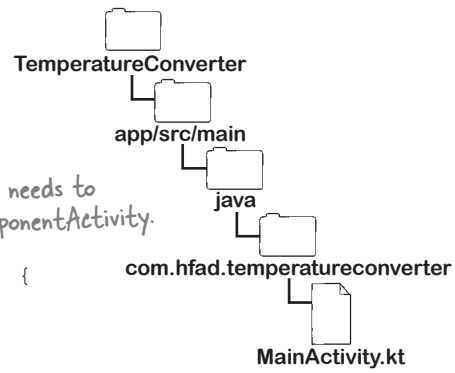
When you use Compose, the activity code is responsible for the app's behavior *and* its appearance. For this reason, it looks slightly different from the activity code you're used to working with.

Let's see what basic Compose activity code looks like. Open the package *com.hfad.temperatureconverter* in the *app/src/main/java* folder, and open the file *MainActivity.kt* (if it's not already open). Then **replace the code** that Android Studio generated with the code shown here:

```
package com.hfad.temperatureconverter

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
        }
    }
}
```

The activity needs to extend ComponentActivity.

The onCreate() method needs to call setContent().

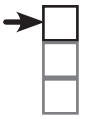



## Compose activities extend ComponentActivity

As you can see, the above activity doesn't extend `AppCompatActivity`: it uses **`ComponentActivity`** instead. `androidx.activity.ComponentActivity` is a subclass of `Activity`, and it's used to define a basic activity that uses Compose for its UI instead of a layout file.

Just like all the other activities you've seen, the activity overrides the `onCreate()` method. Instead of calling `setContentView()` to inflate the activity's layout, however, it uses **`setContent()`**. This is an extension function that's used to add Compose components—called **composables**—to an activity's UI so they run when the activity gets created.

Let's see how this works by using Compose to add some text to the activity's UI..



**Text**
Converter
Modify

# Use a Text composable to display text

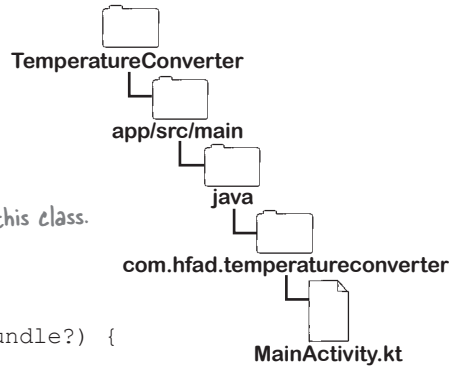
We're going to make `MainActivity` display some text by adding a **`Text`** composable to the call to `setContent()`. You can think of `Text` as being the Compose equivalent of a text view. Simply specify the text you want to show, and the activity displays it.

Here's the code to add some text to `MainActivity`; update the code for *MainActivity.kt* to include the changes below (in bold):

```
package com.hfad.temperatureconverter

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.material.Text

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text("Hello friend! Welcome to Jetpack Compose.")
        }
    }
}
```

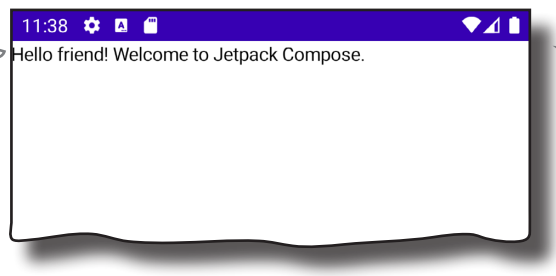
Import this class.

This displays some text in the activity's UI.




When this code runs, it displays the text at the top of the screen like this:

This is the text we just added to MainActivity.




There's no app bar because Android Studio has applied a theme that doesn't have one.

Now that you've learned how to display some hardcoded text with Compose, let's make it more flexible by adding the `Text` to a **composable function**.

# Use composables in composable functions

A composable function is a function that uses one or more composables to define part of the UI.

To see how this works, we'll define a composable function named `Hello` that accepts a `String` argument for the user's name. When the function runs—or is **composed**—it will add the `String` to a `Text` composable, which displays the text in the UI.

The code for the new `Hello` function looks like this:

```
@Composable
fun Hello(name: String) {
    Text("Hello $name! Welcome to Jetpack Compose.")
}
```

This annotation turns it into a composable function.

We'll add this function to MainActivity.kt a couple pages ahead.

As you can see, the function is annotated with **`@Composable`**. This annotation is required for *all* composable functions. If you omit the annotation, the code won't compile.

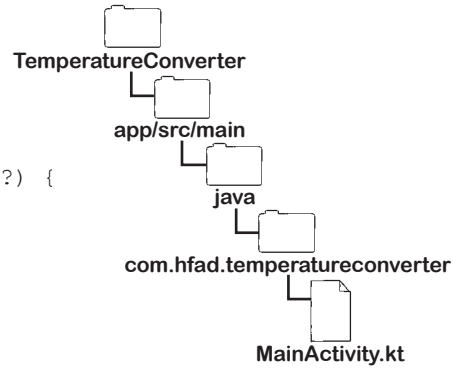
## Call the Hello composable from setContent()

Marking the function with `@Composable` doesn't just specify that it uses composables; **it makes the function itself a composable**, which you can use in your code like any other sort of composable.

In the app we're building, we want to display the `Hello` text in `MainActivity`'s UI. We can do so by calling `Hello` from `setContent()` like this:

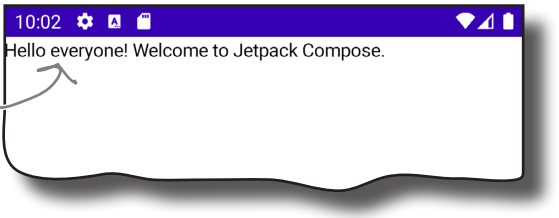
```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Hello("everyone")
        }
    }
}
```

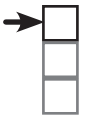
This passes the String "everyone" to the Hello composable function.




When this code runs, it displays the text like this:

The Hello text includes the String that's passed to it.






# Most UIs have multiple composables

So far, you've seen how to run a single composable, but most of the time, you'll want to use multiple composables in the UI, or call the same composable multiple times. If you wanted the app to say hello twice, for example, you could run the `Hello` composable function twice with different arguments like so:

```
Hello("friend")
Hello("everyone")
```

When you have a UI that includes multiple composables, you need to specify how they should be arranged. If you don't, Compose will stack the composables on top of one another like this:

So how do you arrange composables?

## You can arrange composables using Row or Column

Most of the time, you'll probably want to arrange composables in rows or columns, and Compose includes **`Row`** and **`Column`** composables that let you do this. To arrange two `Hello` composables in a column, for example, you'd simply add them to a `Column` composable like this:

```
Column {
    Hello("friend")
    Hello("everyone")
}
```

When the code runs, the composables are arranged in a column like this:

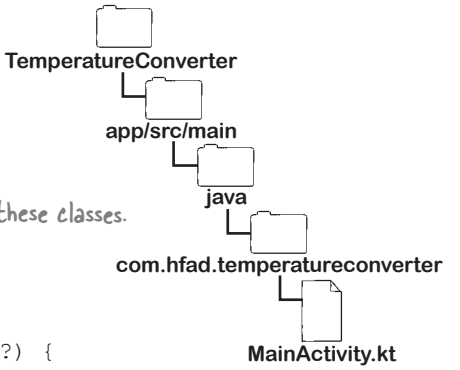
Let's update `MainActivity` so that it produces this UI.

# The full code for MainActivity.kt

We're going to add the `Hello` composable function to *MainActivity.kt*, run it twice, and arrange the results in a column.

Here's the complete code for the file; update *MainActivity.kt* to include the changes below (in bold):

```
package com.hfad.temperatureconverter

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Column
import androidx.compose.material.Text
import androidx.compose.runtime.Composable

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text("Hello friend! Welcome to Jetpack Compose.")
            Column {
                Hello("friend")
                Hello("everyone")
            }
        }
    }
}

@Composable
fun Hello(name: String) {
    Text("Hello $name! Welcome to Jetpack Compose.")
}
```




Import these classes.

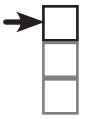
Delete this line.

Display two greetings in a column.

Add the Hello composable function. Make sure that it goes after the MainActivity class definition.

Let's take the app for a test drive and see what it looks like.

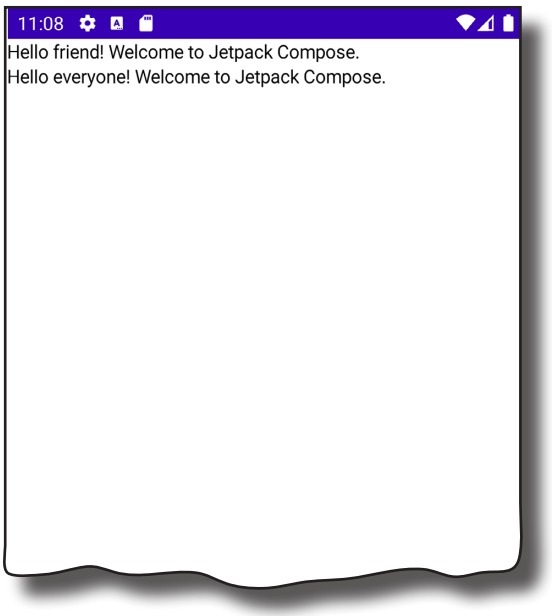
# Test Drive



**Text**
Converter
Modify

When we run the app, `MainActivity` is displayed. It includes two `Hello` composables arranged in a single column.

The two components are arranged just how we want.




Congratulations! You've now learned how to use Compose to create `MainActivity`'s UI instead of adding views to a layout file.

## there are no Dumb Questions

**Q: Do I have to use Compose? Can't I stick to using views and layouts?**

A: Even though Jetpack Compose is still relatively new, we think it has a big future. It may even become the default way in which you build Android apps, so we definitely recommend you learn how to use it.

**Q: You said every activity that uses composables needs to extend `ComponentActivity` or one of its subclasses. Does `ComponentActivity` extend `AppCompatActivity`?**

A: No. This might surprise you, but `AppCompatActivity` is a subclass of `ComponentActivity`, not the other way around.

**Q: I see. So does that mean that I can add composables to my existing activity code?**

A: Yes. You can even use composables and views in the same UI. You'll find out how this is done in the next chapter.

# You can preview composable functions

Another feature of using composable functions is that you can preview them within Android Studio without having to load the app onto a device. You can preview any composable function so long as it doesn't have any arguments, and you can even use this technique to preview entire **compositions**—UIs made up of composables.

A composition is a UI made up of composables.

You say you want to preview a composable function by annotating it with **`@Preview`**. The following code, for example, specifies a composable function named `PreviewMainActivity` that lets you preview two `Hello` composables arranged in a column:

```
@Preview(showBackground = true)
@Composable
fun PreviewMainActivity() {
    Column {
        Hello("friend")
        Hello("everyone")
    }
}
```

Adding an @Preview annotation lets you preview the function without loading it onto a device. By default, its background is transparent, and setting showBackground to true overrides this.

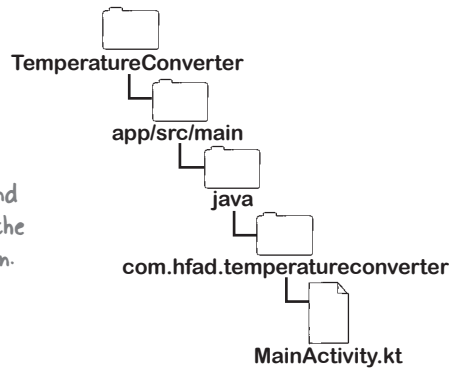
## Add MainActivityPreview to MainActivity.kt

To see how previews work, let's add the `MainActivityPreview` composable function to *MainActivity.kt*. Update the file so that it includes the changes shown here:

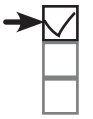
```
...
import androidx.compose.ui.tooling.preview.Preview

...

@Preview(showBackground = true)
@Composable
fun PreviewMainActivity() {
    Column {
        Hello("friend")
        Hello("everyone")
    }
}
```

Import this class.

Add this function to the end of the file so it's outside the MainActivity class definition.




We'll show you how to see the preview on the next page.



Text
Converter
Modify

# Preview composables with the Design or Split option

You can preview any composable functions marked with `@Preview` by selecting the Split or Design view of the activity file. Choosing Split lets you see the code and preview side by side, and choosing Design just displays the preview.

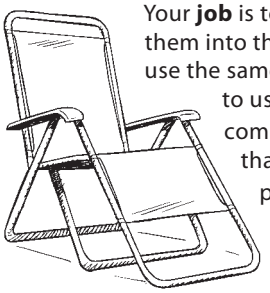
Here's what `MainActivityPreview` looks like when we select the Split option:

If you make changes to the composable function you're previewing, you need to refresh the preview before you can see what it looks like. Simply click on the Build Refresh button in the preview's top menu and you should see the effect of your changes.

Now that you've learned how to preview composable functions, try the following exercise.

**You can preview any composable function that has no arguments.**

# Pool Puzzle

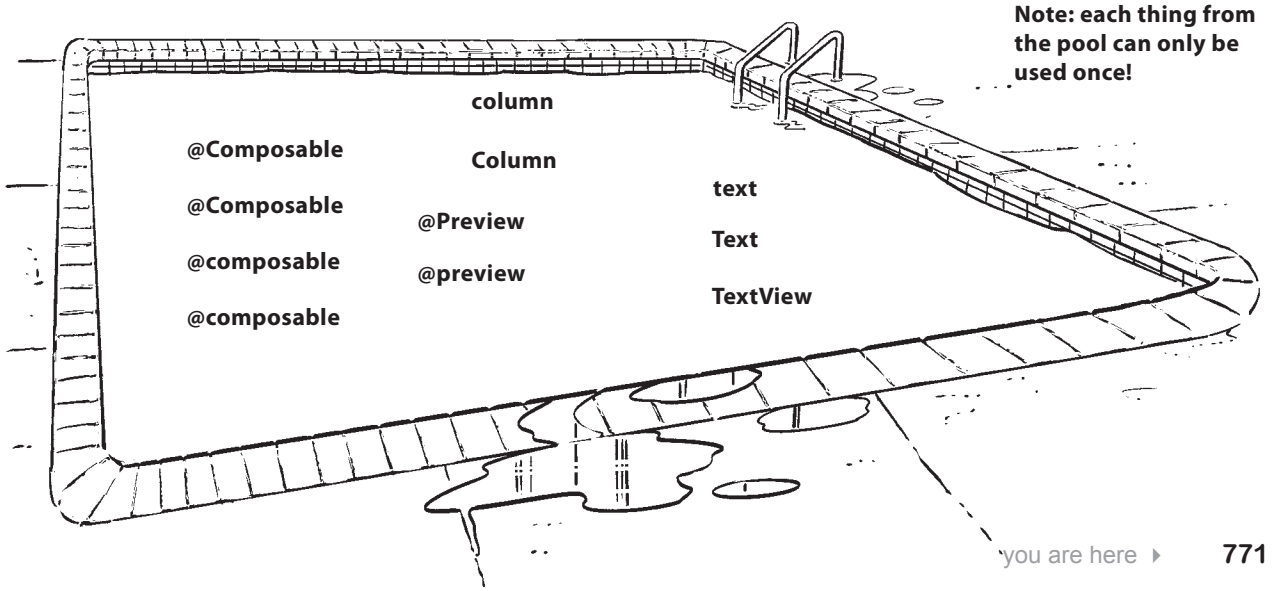


Your **job** is to take code snippets from the pool and place them into the blank lines in the code below. You may **not** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to create two composable functions: one named `TeamHello` that accepts a list of names and says hello to each person, and one named `HelloPreview` that previews `TeamHello` and arranges its text in a column.

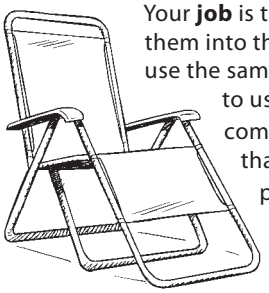
This is what the preview of HelloPreview needs to look like.

```
.....
fun TeamHello(names: List<String>) {
    for (name in names) {
        .....("Hello $name!")
    }
}

.....(showBackground = true)
.....
fun HelloPreview() {
    .....{
        TeamHello(listOf("Virginia", "Zan", "Katie"))
    }
}
```

**Note: each thing from the pool can only be used once!**



# Pool Puzzle Solution



Your **job** is to take code snippets from the pool and place them into the blank lines in the code below. You may **not** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to create two composable functions: one named `TeamHello` that accepts a list of names and says hello to each person, and one named `HelloPreview` that previews `TeamHello` and arranges its text in a column.

The preview of HelloPreview looks like this.

Mark TeamHello as a composable function.

```
@Composable
fun TeamHello(names: List<String>) {
    for (name in names) {
        Text("Hello $name!")
    }
}
```

The function needs to display a Text for each item.

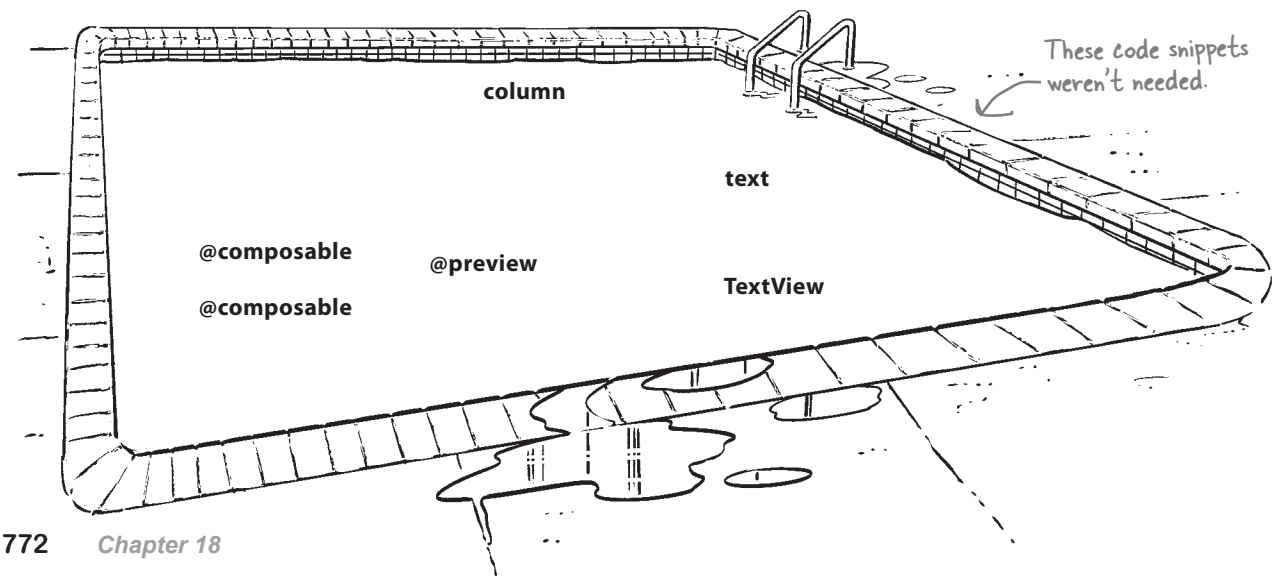
We want to be able to preview this function.

```
@Preview(showBackground = true)
```

HelloPreview needs to be annotated with @Composable too.

```
@Composable
fun HelloPreview() {
    Column {
        TeamHello(listOf("Virginia", "Zan", "Katie"))
    }
}
```

The preview needs to arrange the text items in a column.

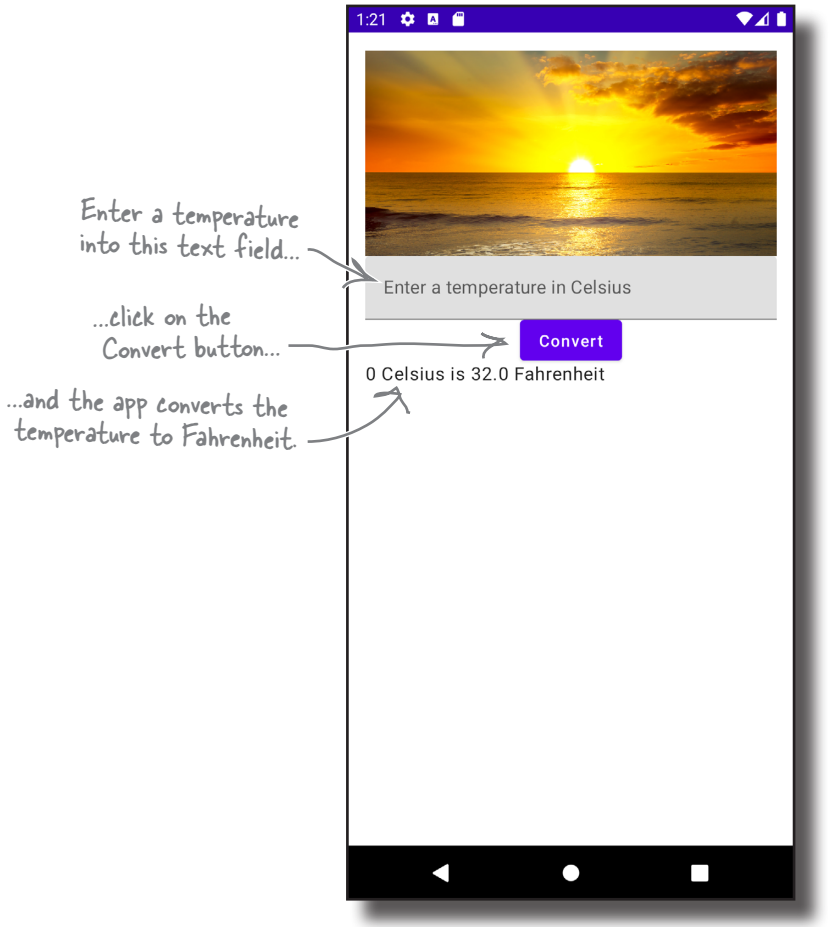


# Let's make the app convert temperatures

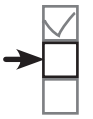
So far, you've used Compose to display text, written some composable functions, and learned how to preview them. But that's not the end of the story.

For the rest of this chapter, we're going to take you deeper into Compose by changing the app we've just built into one that converts temperatures from Celsius to Fahrenheit. Instead of saying hello, it will ask the user for a temperature in Celsius, and convert it when she clicks on a button.

Here's what the app will look like; as you can see, it includes an image, a text field that lets you enter data, a button, and some text:




What are you waiting for? Let's get composing.



Text
**Converter**
Modify

# Add a MainActivityContent composable function

We'll begin by adding a new composable function to *MainActivity.kt* named `MainActivityContent` that we'll use for the activity's main content. We'll add all the composables needed for `MainActivity`'s UI to this function, and call it from `setContent()` and `PreviewMainActivity`. This approach means that the activity's composition will be displayed both when we run the app, and in the preview.

We'll also remove the `Hello` composable function from the code as this is no longer needed.

Here's the updated code for *MainActivity.kt*; update the file to include these changes (in bold):

```
...
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MainActivityContent()
            Column {
                Hello("friend")
                Hello("everyone")
            }
        }
    }
}

@Composable
fun Hello(name: String) {...}

@Composable
fun MainActivityContent() {
}

@Preview(showBackground = true)
@Composable
fun PreviewMainActivity() {
    MainActivityContent()
    Column {
        Hello("friend")
        Hello("everyone")
    }
}
```

Use MainActivityContent for MainActivity's UI.

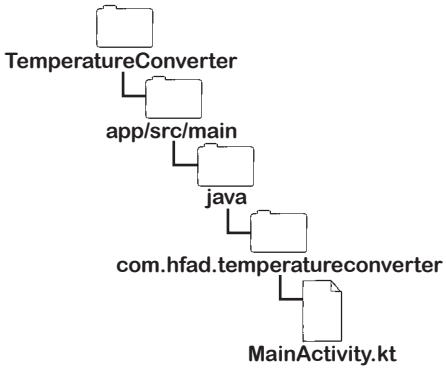
Delete these lines.

Remove the Hello composable function.

Add MainActivityContent.

Add MainActivityContent to the preview.

Remove these lines.



# Display the header image...

The first component we'll add to `MainActivityContent` is an image that appears at the top of the screen.

First, make sure that your project includes the folder *app/src/main/res/drawable* (you'll need to create it if it's not there). Then download the file *sunrise.webp* from *tinyurl.com/hfad3*, and add it to the *drawable* folder.

The header displays this image.

## ...with an Image composable

You display an image in Compose using an **`Image`** composable. The basic code looks like this:

```
Image(
    painter = painterResource(R.drawable.sunrise),
    contentDescription = "sunrise image"
)
```

This specifies what image to use.

This is a description of the image.

The `Image` composable requires two arguments: `painter` and `contentDescription`.

The `painter` argument specifies the image that should be displayed. Here, it uses `painterResource(R.drawable.sunrise)` to display the *sunrise.webp* drawable resource.

The `contentDescription` argument is a description of the image that's used for accessibility.

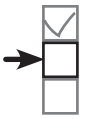
There are many other optional arguments you can use to control the image appearance, and how it's displayed. The following code, for example, sets the image height to 180dp, makes it fill the available width, and scales the image:

```
Image(
    painter = painterResource(R.drawable.sunrise),
    contentDescription = "sunrise image",
    modifier = Modifier
        .height(180.dp)
        .fillMaxWidth(),
    contentScale = ContentScale.Crop
)
```

Modifiers are used to add extra options to composables, such as setting the width and height.

This argument scales the image.

Now that you've seen how to add an image with Compose, let's add one to `MainActivity`.



Text
Converter
Modify

# Add an Image to MainActivity.kt

We're going to add an image to `MainActivity` by defining a new composable function (named `Header`) to *MainActivity.kt* that creates one. We'll run this from the `MainActivityContent` composable function so that the image is added to the UI and the preview.

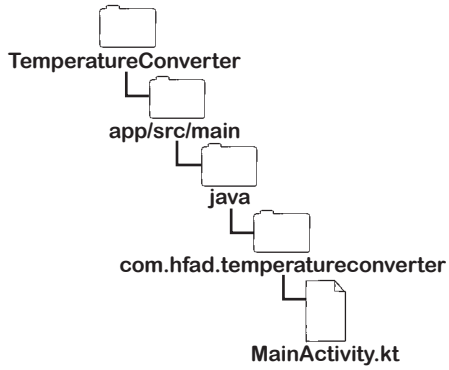
Here's the updated code for *MainActivity.kt*; update this file to include the changes below (in bold):

```
...
import androidx.compose.foundation.Image
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.ui.Modifier
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.res.painterResource
import androidx.compose.ui.unit.dp
...

@Composable
fun Header(image: Int, description: String) {
    Image(
        painter = painterResource(image),
        contentDescription = description,
        modifier = Modifier
            .height(180.dp)
            .fillMaxWidth(),
        contentScale = ContentScale.Crop
    )
}

@Composable
fun MainActivityContent() {
    Header(R.drawable.sunrise, "sunrise image")
}
...
```

Add these imports.

Add the Header composable function.

Run Header from MainActivityContent.



This is what the Header composable looks like when the code runs.

That's everything we need to display an image in the composition, so let's move on to the next component.

# Let's display the temperature text

The next thing we'll include is a composable function (named `TemperatureText`) that converts a Celsius temperature to Fahrenheit, and displays the result. We'll call this function from `MainActivityContent` so that it's included in both the UI and the preview.

You're already familiar with the code to do this, so update *MainActivity.kt* to include the changes below (in bold):

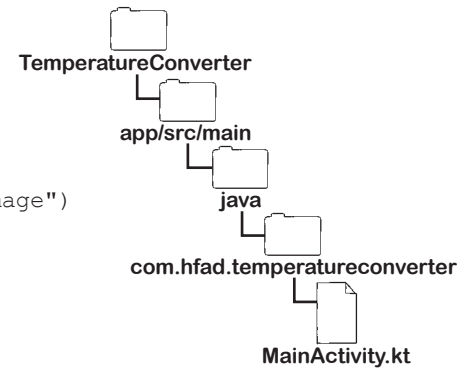
```
...
@Composable
fun TemperatureText(celsius: Int) {
    val fahrenheit = (celsius.toDouble()*9/5)+32
    Text("$celsius Celsius is $fahrenheit Fahrenheit")
}

@Composable
fun MainActivityContent() {
    Column {
        Header(R.drawable.sunrise, "sunrise image")
        TemperatureText(0)
    }
}
...
```

Add the TemperatureText composable function.

This formula converts a temperature from Celsius to Fahrenheit.

Arrange the composables in a column.

Run TemperatureText from MainActivityContent.

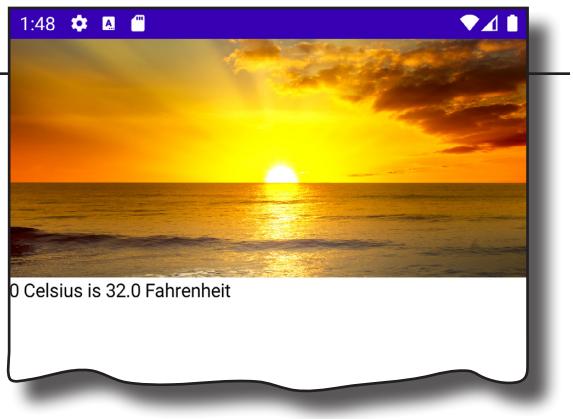


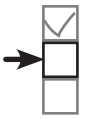

Let's take the app for a test drive.

## Test Drive

When we run (or preview) the app, it displays an `Image` and `Text` in a column. The `Text` correctly shows the value of 0° Celsius in Fahrenheit.

Now that we've made sure the `TemperatureText` function works with a hardcoded temperature, let's update it to a new temperature when the user clicks a button.





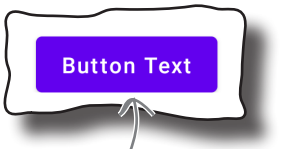

# Use a Button composable to add a button

You add a button in Compose using a **Button** composable. Button code looks like this:

```
Button(onClick = {
    //Code that runs when clicked
}) { Text("Button Text") }
```

Pass onClick a lambda to specify what happens when the button is clicked.

This is the text that appears on the button.




When you use a `Button` composable, you need to specify two things: its click behavior, and what should be displayed on the button.

You specify its click behavior using the `Button`'s `onClick` argument. This accepts a lambda that runs each time the user clicks the button.

You say what should be displayed on the `Button` using a separate lambda that includes a composable. When the code runs, it adds the composable to the `Button`. The above code, for example, passes the `Button` a `Text` composable, so it creates a button with text on it.

## Let's write a ConvertButton composable function

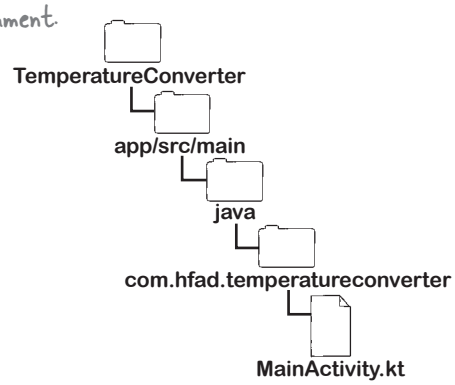
We're going to add a `Button` composable to the Temperature Converter app that, when clicked, changes the temperature that `TemperatureText` converts to Fahrenheit. To do this, we'll write a new composable function (named `ConvertButton`) that displays a `Button`. We'll also specify that it accepts a lambda argument that the function will use for the `Button`'s click behavior.

Here's what the code for the `ConvertButton` composable function looks like; we'll add it to *MainActivity.kt* a few pages ahead:

```
@Composable
fun ConvertButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Convert")
    }
}
```

ConvertButton accepts a lambda argument.

This adds a Button to the UI.




Now that we've written the `ConvertButton` function, let's add it to `MainActivityContent` so that the `Button` composable gets added to the UI.

# We need to pass a lambda to ConvertButton

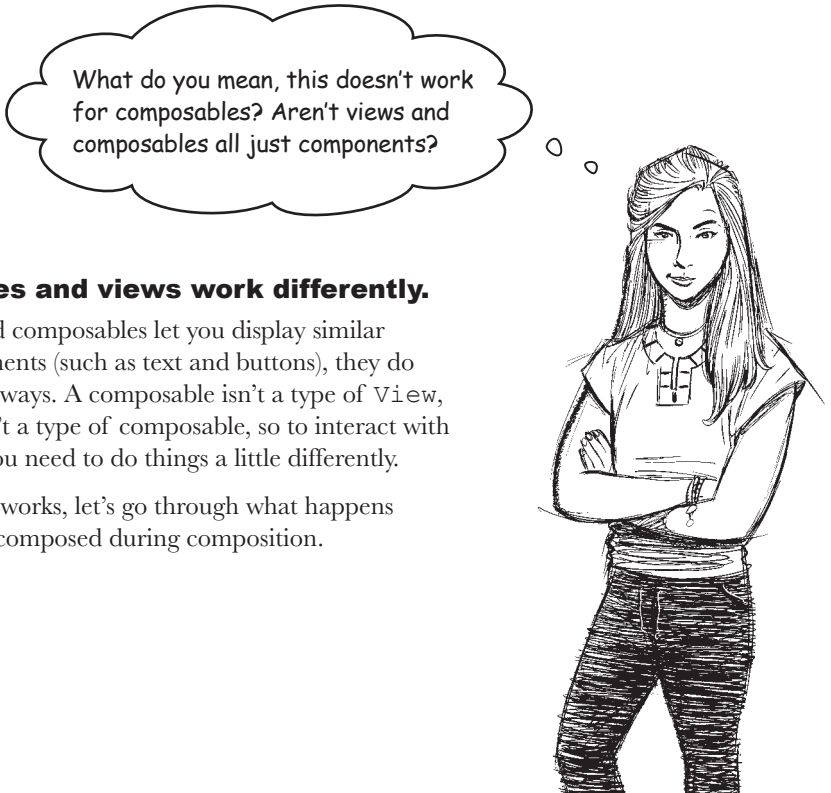
To run `ConvertButton` from `MainActivityContent`, we need to pass it a lambda that specifies what should happen when it's clicked. The code needs to look like this:

```
@Composable
fun MainActivityContent() {
    ...
    ConvertButton {
        //Code that runs when the button is clicked
    }
    ...
}
```

When `ConvertButton` is clicked, we want it to update the text displayed by `TemperatureText`. If `TemperatureText` was a *view*, we could update its text using code like this:

```
binding.textView.text = "This is the new text"
```

While this approach works with views, however, it doesn't work for *composables*. With composables, you need a different approach.




### Composables and views work differently.

While views and composables let you display similar looking components (such as text and buttons), they do this in different ways. A composable isn't a type of `View`, and a `View` isn't a type of composable, so to interact with composables, you need to do things a little differently.

To see how this works, let's go through what happens when the UI is composed during composition.

## Composition Up Close

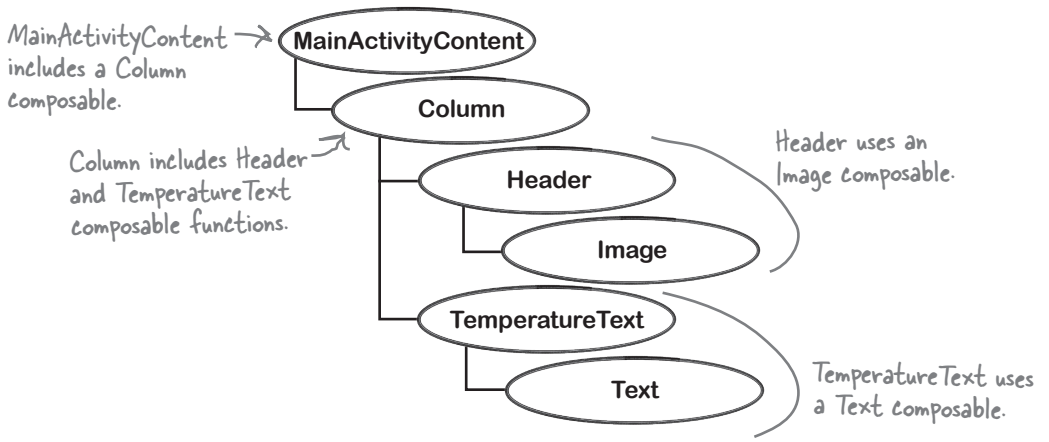
When Compose first builds the UI, it creates a hierarchical tree of all the composables that get called. When the following code runs, for example:

```
@Composable
fun MainActivityContent() {
    Column {
        Header(R.drawable.sunrise, "sunrise image")
        TemperatureText(0)
    }
}
```

Header displays an Image.

TemperatureText displays a Text.

Compose builds a tree of composables that looks like this:




Once a composable has been composed, it only gets redrawn—or **recomposed**—when its inputs change. `TemperatureText`, for example, is defined using the code:

```
@Composable
fun TemperatureText(celsius: Int) {
    val fahrenheit = (celsius.toDouble()*9/5)+32
    Text("$celsius Celsius is $fahrenheit Fahrenheit")
}
```

Because `TemperatureText` accepts a single argument named `celsius`, it will only be recomposed when the value of `celsius` gets updated. If you want `TemperatureText` to be redrawn with new text, you need to update the value of its argument.

# We need to change the value of TemperatureText's argument

As you've just seen, composables are recomposed when the values they depend on get updated. This means that if we want `TemperatureText` to display different text when the user clicks on `ConvertButton`'s button, the lambda we pass to `ConvertButton` needs to update the value of `TemperatureText`'s argument.

To do this, we'll add a new `celsius` variable to `MainActivityContent`, and pass its value to `TemperatureText`. When the user clicks on the `ConvertButton` composable, we'll get it to update the value of `celsius` so that `TemperatureText` is recomposed.

**A composable is recomposed when any of its inputs change their value.**

## Use remember to store celsius in memory

We'll define the `celsius` variable by adding the following code to `MainActivityContent`:

```
val celsius = remember { mutableStateOf(0) }
```

This stores the value of celsius in the composable's memory.

This creates an object whose type is **`MutableState`**, sets its value to `0`, and stores it in memory. You can think of `celsius` as working a bit like a live data object. Each time it gets set to a new value, any composables that use it are notified, and recomposed.

The object is stored in memory using **`remember`**. `remember` stores an object when the composable that calls it—in this case `MainActivityContent`—is first composed, and forgets it when the composable is removed from the UI. This can happen when the user rotates the device, and the activity—including its UI—is destroyed and recreated.

Just like a `MutableLiveData` object, you set a `MutableState` object's value by updating its **`value`** property. To set the value of `celsius` to 20 when the user clicks on the `ConvertButton` composable, for example, you'd use the following code:

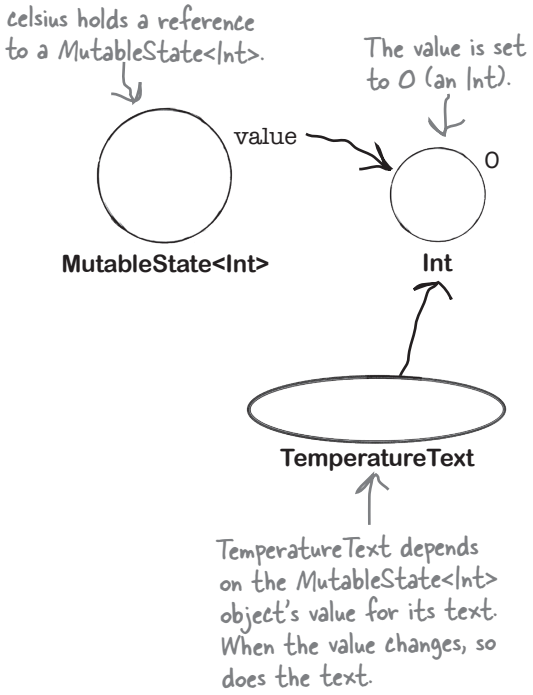
```
ConvertButton { celsius.value = 20 }
```

In order for `TemperatureText` to respond to this value change, its argument needs to be set to `celsius.value` like this:

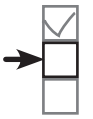
```
TemperatureText(celsius.value)
```

Each time `celsius.value` gets updated, `TemperatureText` is recomposed with the new value, which changes the text that's displayed.

We'll show you the full code over the next couple pages.

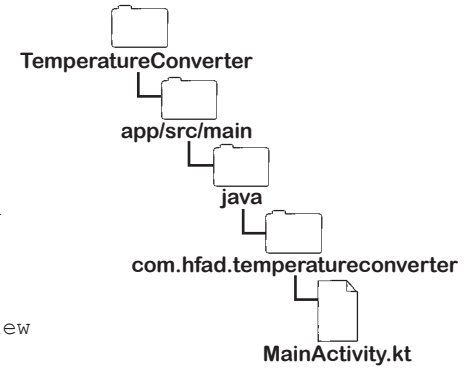


# The full code for MainActivity.kt



**Text**
**Converter**
**Modify**

Here's the complete code so far for *MainActivity.kt*; update the file to include the changes below (in bold):




```
package com.hfad.temperatureconverter

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Column
import androidx.compose.material.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.foundation.Image
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.ui.Modifier
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.res.painterResource
import androidx.compose.ui.unit.dp
import androidx.compose.material.Button
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MainActivityContent()
        }
    }
}

@Composable
fun TemperatureText(celsius: Int) {
    val fahrenheit = (celsius.toDouble()*9/5)+32
    Text("$celsius Celsius is $fahrenheit Fahrenheit")
}
```

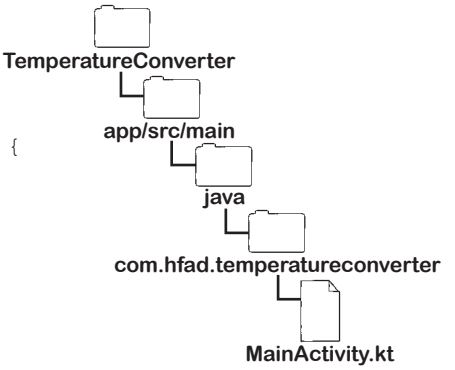
Import these extra classes.

The code continues on the next page.

# MainActivity.kt continued

```
@Composable
fun ConvertButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Convert")
    }
}
```

Add the ConvertButton composable function.



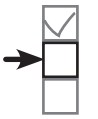

```
@Composable
fun Header(image: Int, description: String) {
    Image(
        painter = painterResource(image),
        contentDescription = description,
        modifier = Modifier
            .height(180.dp)
            .fillMaxWidth(),
        contentScale = ContentScale.Crop
    )
}

@Composable
fun MainActivityContent() {
    val celsius = remember { mutableStateOf(0) }

    Column {
        Header(R.drawable.sunrise, "sunrise image")
        ConvertButton { celsius.value = 20 }
        TemperatureText(~~0~~ celsius.value)
    }
}

@Preview(showBackground = true)
@Composable
fun PreviewMainActivity() {
    MainActivityContent()
}
```

Add the celsius variable.

Make ConvertButton update celsius when it's clicked.

Pass celsius.value to TemperatureText so the text changes when celsius is updated.

Let's run through what happens when the code runs, and take the app for a test drive.

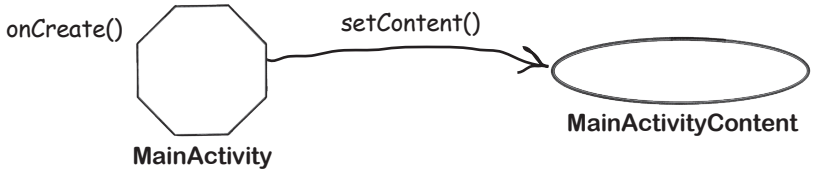


Text
**Converter**
Modify

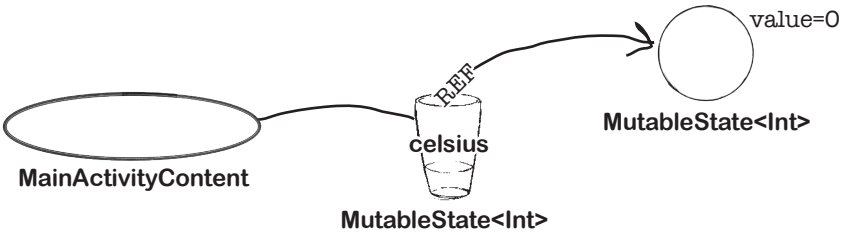
# What happens when the app runs

The following things happen when the app runs:

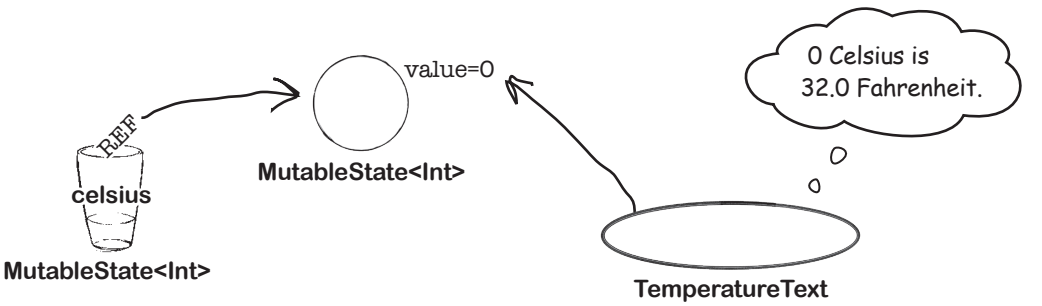
1. **MainActivity launches, and its onCreate() method runs.**
It calls `setContent()`, which runs the `MainActivityContent` composable function.




2. **MainActivityContent creates a MutableState<Int> variable named celsius, sets its value to 0, and stores it in memory.**

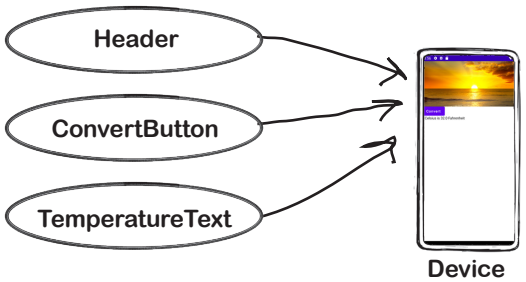



3. **MainActivityContent runs the Header, ConvertButton, and TemperatureText composable functions.**
It passes the value of `celsius` to the `TemperatureText` composable function, which converts it to Fahrenheit.

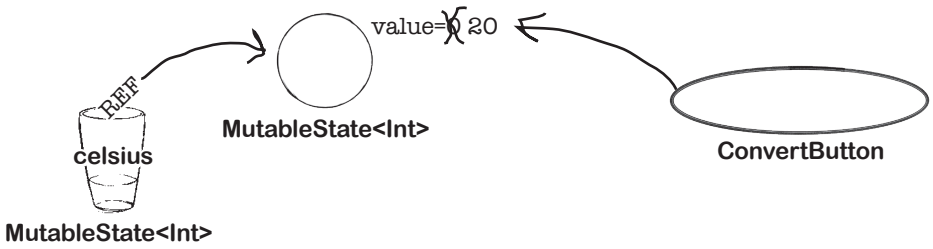


# The story continues

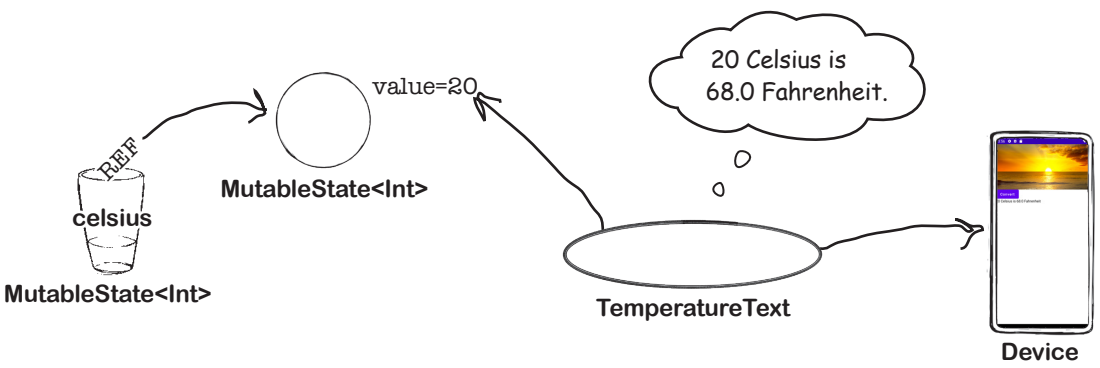
**4** **The Header, ConvertButton, and TemperatureText composable functions add an image, a button, and some text to the UI.**




**5** **The user clicks on the ConvertButton composable in the UI.**
This sets the value of `celsius` to 20.

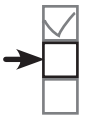



**6** **TemperatureText is recomposed.**
It converts the new value of `celsius` to Fahrenheit, and displays the result.



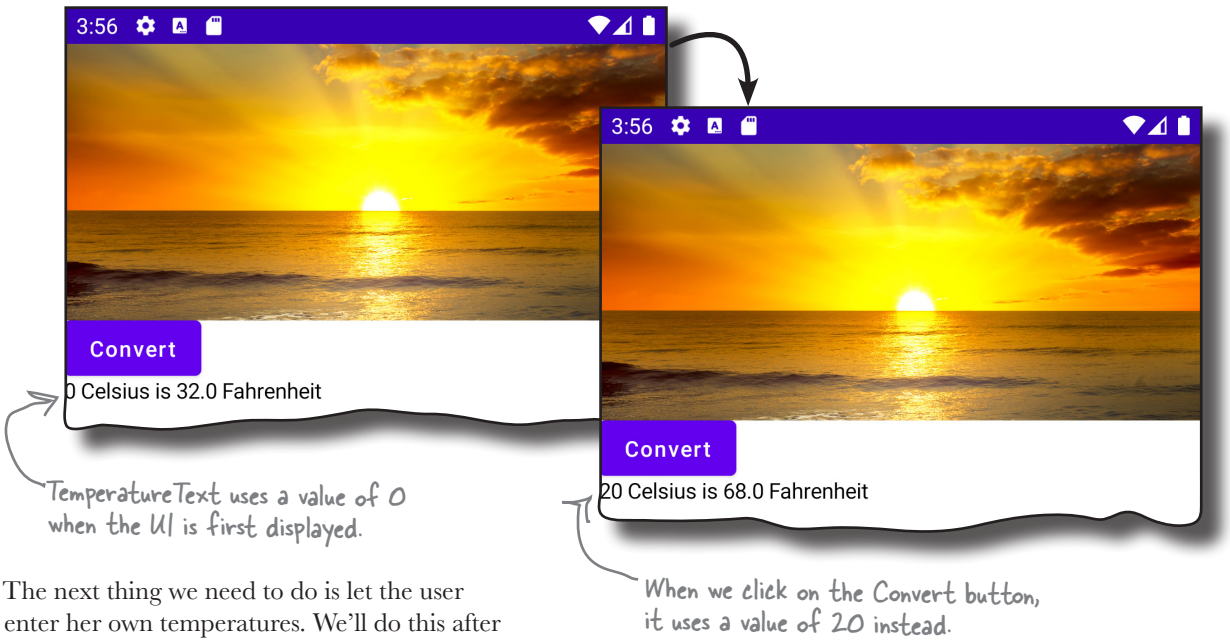

Let's take the app for a test drive.

# Test Drive




When we run the app, `MainActivity` is displayed. It includes an image, a button, and some text that gives the value of 0° Celsius in Fahrenheit.

When we click on the Convert button, the text is updated to give the value of 20° Celsius in Fahrenheit instead.




The next thing we need to do is let the user enter her own temperatures. We'll do this after you've tried the exercise on the next page.

## there are no Dumb Questions

**Q: Did you say that `remember` stores an object in memory?**

A: Yes—well remembered!

`remember` stores an object in memory when the composable that calls it is first composed, and forgets about it when the composable is removed from the UI. This can happen when the user rotates the device screen because Android destroys and recreates the activity, along with any composables.

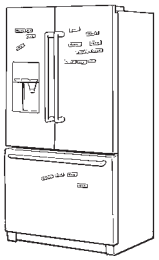
**Q: Can I persist a composable's state when the screen is rotated?**

A: For simple values, you can use `rememberSaveable` instead of `remember`. This automatically saves any value that can be saved to a `Bundle`.

Another option is to use a view model. We'll show you how to use composables with a view model in the next chapter.

**Q: I've seen syntax like: `var x by remember { ... }` What does this do?**

A: It's the same as using `val x = remember { ... }` except you can replace `x.value` in your code with `x`. If you use this approach, make sure you import `androidx.compose.runtime.getValue` and `androidx.compose.runtime.setValue` or the code won't compile.



# Compose Magnets

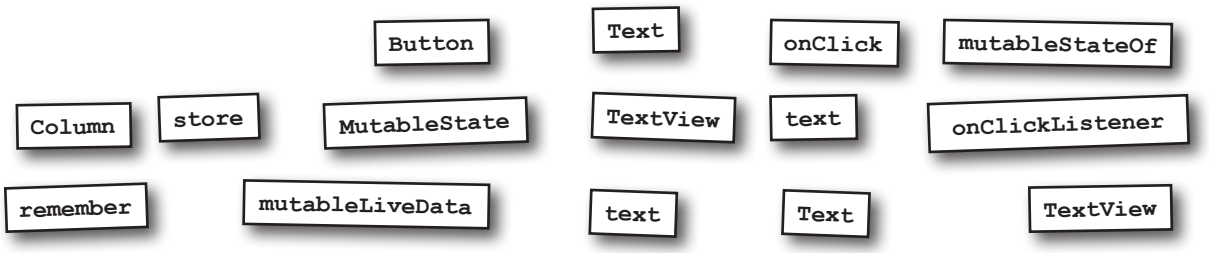
Someone used fridge magnets to create a new composable function named `ChangeHello`. The function displays some text that says "Hello friend", and a button that, when clicked, changes the text to "Hello everyone".

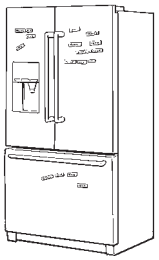
Unfortunately a stray cat ran into the kitchen and dislodged some of the magnets. Can you piece them back together again?

When run, the composable function produces this UI.

```
@Composable
fun ChangeHello() {

    val name = ..........{ ..........("friend") }

    ..........{

        ..........("Hello ${name.value}")

        ..........(

            .......... = { name.value = "everyone" }

        ) {
            ..........("Say hello everyone")
        }
    }
}
```

You won't need to use all the magnets.





# Compose Magnets Solution

Someone used fridge magnets to create a new composable function named `ChangeHello`. The function displays some text that says "Hello friend", and a button that, when clicked, changes the text to "Hello everyone".

Unfortunately a stray cat ran into the kitchen and dislodged some of the magnets. Can you piece them back together again?

When run, the composable function produces this UI.

```
@Composable
fun ChangeHello() {
    val name = remember { mutableStateOf("friend") }
    Column {
        Text("Hello ${name.value}")
        Button(
            onClick = { name.value = "everyone" }
        ) {
            Text("Say hello everyone")
        }
    }
}
```

This stores the value of the name variable in memory.

Arrange the composables in a column.

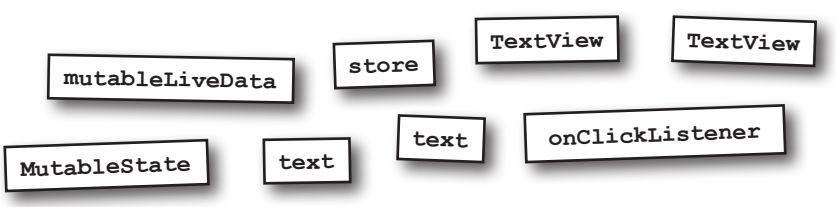
Add text to the UI.

Add a button.

Change the value of name when the button is clicked.

Display this text on the button.

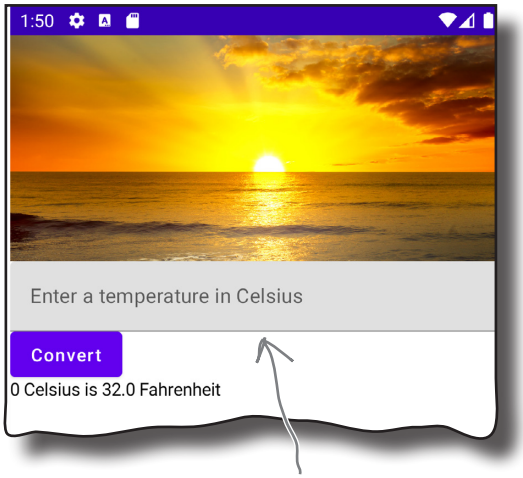
You didn't need to use these magnets.



# Let the user enter a temperature

So far, you've built a version of the Temperature Converter app that converts a hardcoded temperature from Celsius to Fahrenheit. When the user clicks on a button, it changes the temperature that's being converted to another value.

What we *really* want to do is let the user convert her own temperatures, so the next thing we'll do is add a text field to the UI. The user will enter a Celsius temperature into the text field, and when she clicks the button, the app will convert it to Fahrenheit and display the result:




We want the user to enter a Celsius temperature in the TextField. When she clicks the button, the app will convert it to Fahrenheit.

## We'll use a TextField composable

We'll add a text field to the UI using a `TextField` composable. You can think of this type of composable as the Compose equivalent of an `EditText`.

You add a `TextField` using code like this:

```
@Composable
fun ExampleTextField() {
    val text = remember { mutableStateOf("") }
    TextField(
        value = text.value,
        onValueChange = { text.value = it },
        label = { Text("What is your name?") }
    )
}
```

Remember the TextField's state.

Add the TextField.

Use the text variable's value for the TextField's value.

Update text.value when the user updates the TextField.

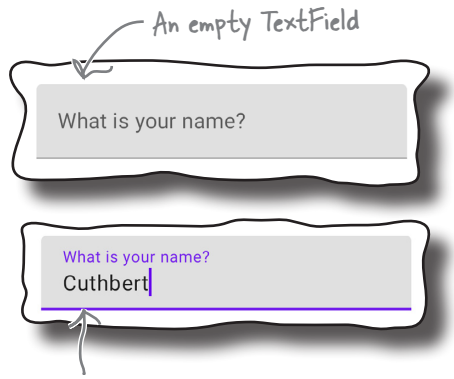
Use this text for the TextField's label.

The `value` attribute is used for the `TextField`'s value, and in this example, it uses a `MutableState` variable named `text` to store this value in memory.

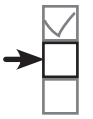
The `onValueChange` attribute uses a lambda to update the value of the `text` variable when the user enters text.

The `label` attribute provides the `TextField`'s label. It's displayed in the text area when the `TextField` is empty, and moves out of the way when the user enters text.

Now that you've seen what text field code looks like, let's add one to the app.




When the user enters text, the TextField's label moves out the way.




# Add a TextField to a composable function

We'll add a `TextField` to the UI by creating a new `EnterTemperature` composable function, which we'll call from `MainActivityContent`.

Here's what the code for the `EnterTemperature` function looks like:

We'll pass arguments for the TextField's value, and its onValueChange lambda.

```
@Composable
fun EnterTemperature(temperature: String, changed: (String) -> Unit) {
    TextField(
        value = temperature,
        label = { Text("Enter a temperature in Celsius") },
        onValueChange = changed,
        modifier = Modifier.fillMaxWidth()
    )
}
```

We'll make the TextField as wide as possible.

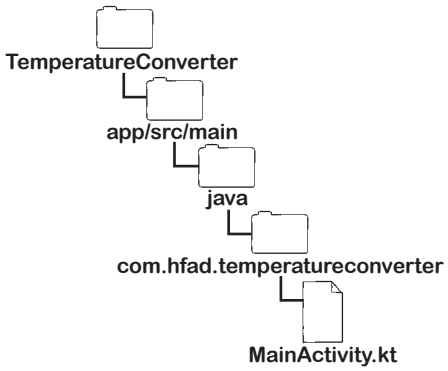
As you can see, the function accepts two arguments: a `String` for the `TextField`'s value, and a lambda that specifies what should happen when the user enters a new value.

## Call the function in MainActivityContent

`MainActivityContent` needs to pass both arguments to `EnterTemperature` when it runs the function, so we'll use a new `MutableState` object named `newCelsius` for its value, and update it when the user enters text.

We'll also change the lambda that's passed to `ConvertButton` so that it updates the value of `celsius` when the user enters a valid `Int`; this will recompose `TemperatureText` when the user enters a new temperature.

Here's what the code looks like; we'll update *MainActivity.kt* on the next page:




```
@Composable
fun MainActivityContent() {
    val celsius = remember { mutableStateOf(0) }
    val newCelsius = remember { mutableStateOf("") }
    ...
        EnterTemperature(newCelsius.value) { newCelsius.value = it }
        ConvertButton {
            newCelsius.value.toIntOrNull()?.let {
                celsius.value = it
            }
        }
    ...
}
```

Store the temperature the user enters.

Use newCelsius.value for the temperature the user enters, and update it when she enters a new value.

If the user enters a valid Int, update celsius.value. This will recompose TemperatureText.

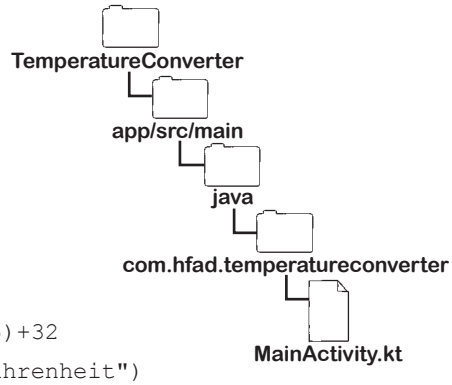
# The full code for MainActivity.kt

Here's the code for *MainActivity.kt*; update the file to include the changes below (in bold):

```
...
import androidx.compose.material.TextField
```

Import this class.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MainActivityContent()
        }
    }
}

@Composable
fun TemperatureText(celsius: Int) {
    val fahrenheit = (celsius.toDouble()*9/5)+32
    Text("$celsius Celsius is $fahrenheit Fahrenheit")
}

@Composable
fun ConvertButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Convert")
    }
}
```

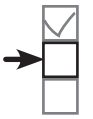



Add the EnterTemperature composable function.

```
@Composable
fun EnterTemperature(temperature: String, changed: (String) -> Unit) {
    TextField(
        value = temperature,
        label = { Text("Enter a temperature in Celsius") },
        onValueChange = changed,
        modifier = Modifier.fillMaxWidth()
    )
}
```

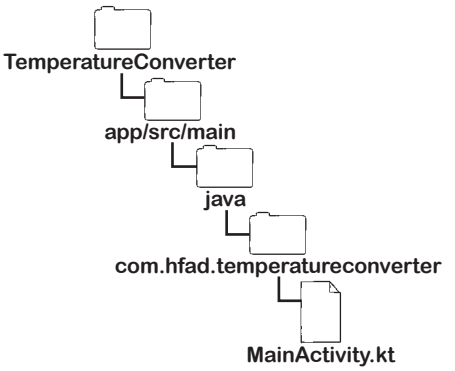
The code continues on the next page.

# MainActivity.kt continued



Text
**Converter**
Modify

```
@Composable
fun Header(image: Int, description: String) {
    Image(
        painter = painterResource(image),
        contentDescription = description,
        modifier = Modifier
            .height(180.dp)
            .fillMaxWidth(),
        contentScale = ContentScale.Crop
    )
}

@Composable
fun MainActivityContent() {
    val celsius = remember { mutableStateOf(0) }
    val newCelsius = remember { mutableStateOf("") }

    Column {
        Header(R.drawable.sunrise, "sunrise image")
        EnterTemperature(newCelsius.value) { newCelsius.value = it }
        ConvertButton {
            newCelsius.value.toIntOrNull()?.let {
                celsius.value = 20 it
            }
        }
        TemperatureText(celsius.value)
    }
}

@Preview(showBackground = true)
@Composable
fun PreviewMainActivity() {
    MainActivityContent()
}
```




Add newCelsius, which is used for EnterTemperature's state.

Add EnterTemperature to the UI.

Add these lines too.

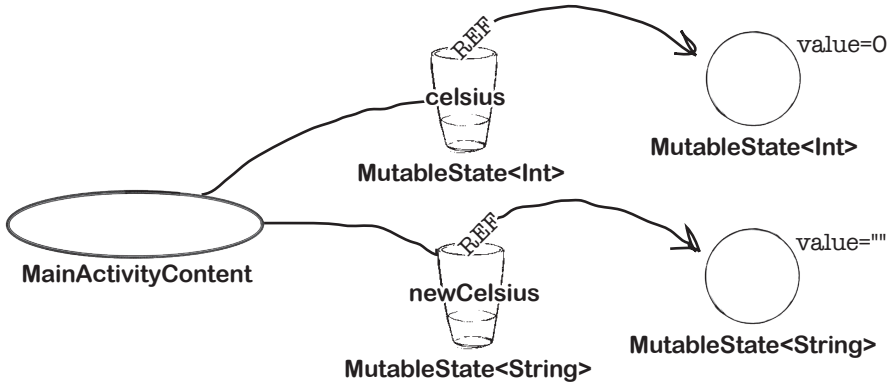
If newCelsius.value is an Int, assign it to celsius.value.

Let's see what happens when the code runs.

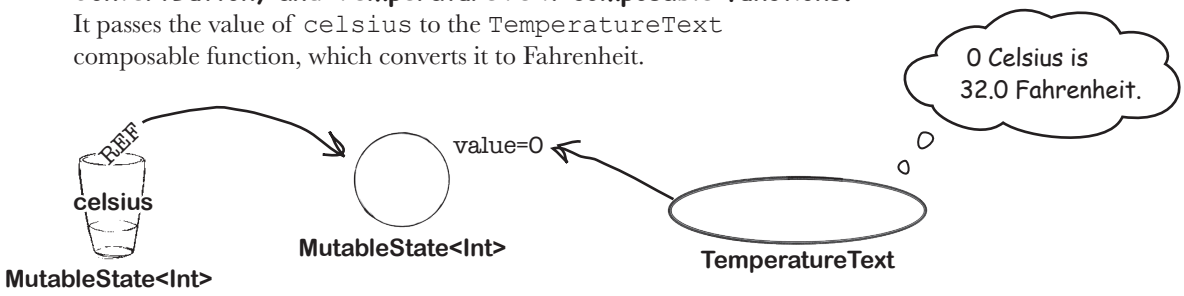
# What happens when the app runs

The following things happen when the app runs:

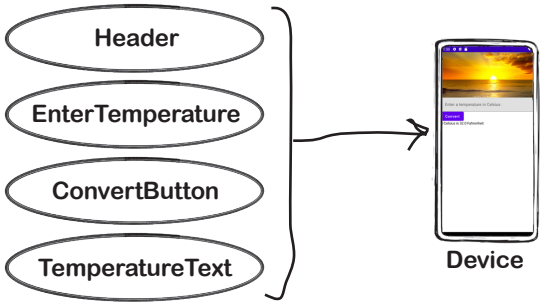
1. **When MainActivityContent runs, it creates a MutableState<Int> variable named celsius, and a MutableState<String> named newCelsius.**
It sets `celsius` to `0` and `newCelsius` to `""` and stores both in memory.



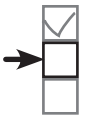

2. **MainActivityContent runs the EnterTemperature, Header, ConvertButton, and TemperatureText composable functions.**
It passes the value of `celsius` to the `TemperatureText` composable function, which converts it to Fahrenheit.



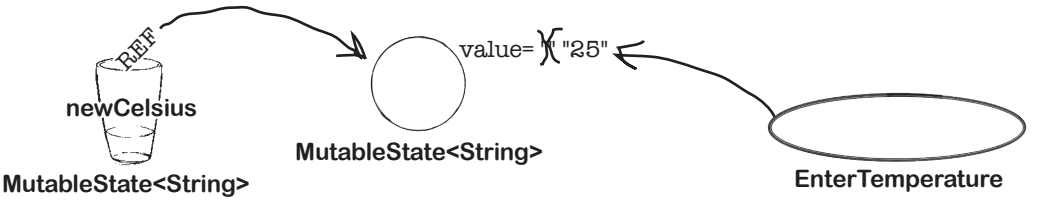

3. **The Header, EnterTemperature, ConvertButton, and TemperatureText composable functions add an image, a text field, a button, and some text to the UI.**



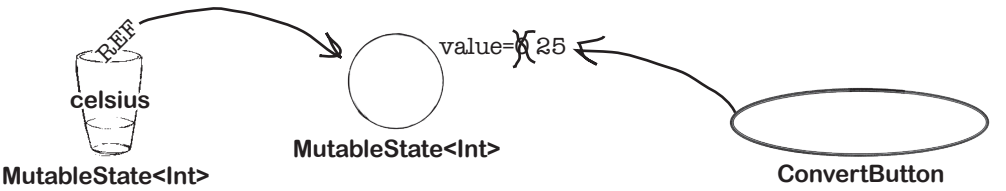
# The story continues



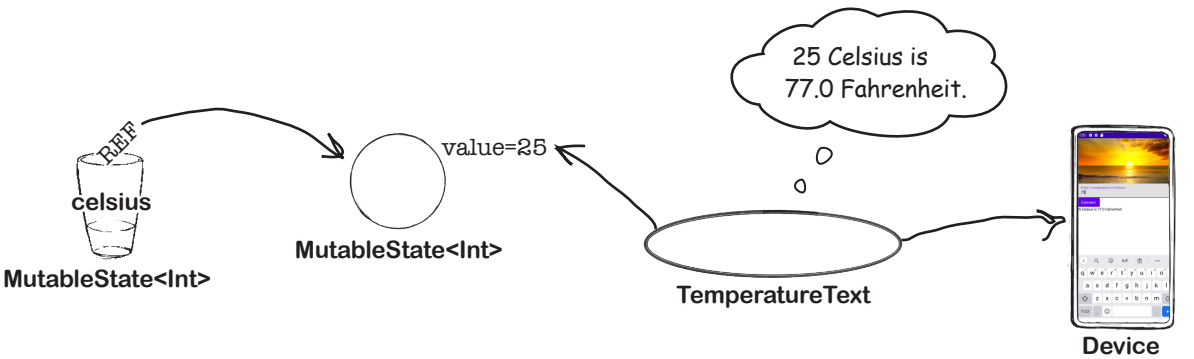

**4** **The user enters a new value into EnterTemperature (in this example "25").**
`EnterTemperature` sets `newCelsius` to this value.




**5** **The user clicks on the ConvertButton composable.**
It converts the value of `newCelsius` to an `Int`, and assigns it to `celsius.value`.




**6** **TemperatureText is recomposed.**
It converts the new value of `celsius` to Fahrenheit, and displays the result.

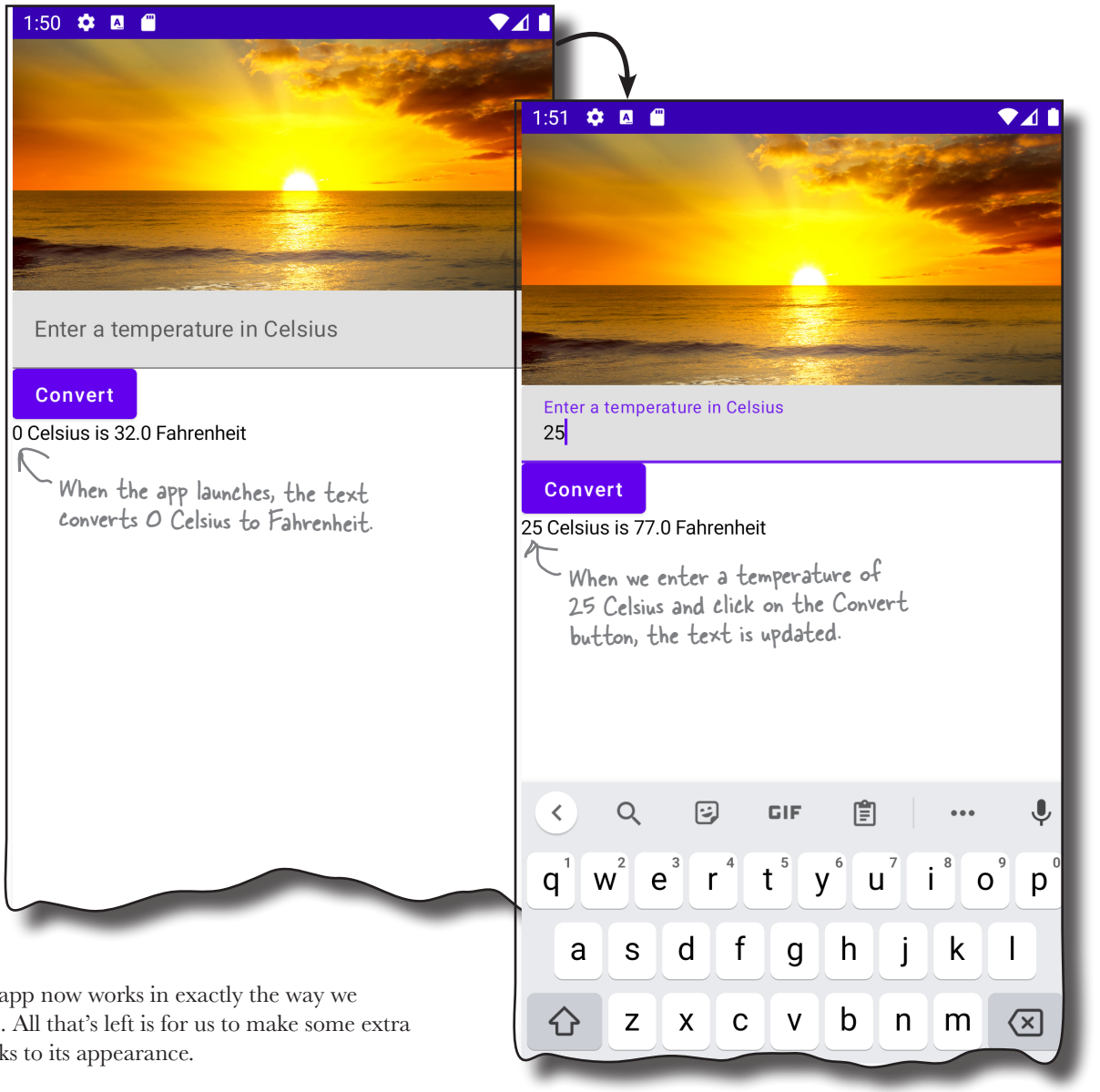



Let's take the app for a test drive.

# Test Drive

When we run the app, `MainActivity` includes a text field.

When we enter a temperature and click on the Convert button, the app converts the temperature to Fahrenheit and updates the temperature text.



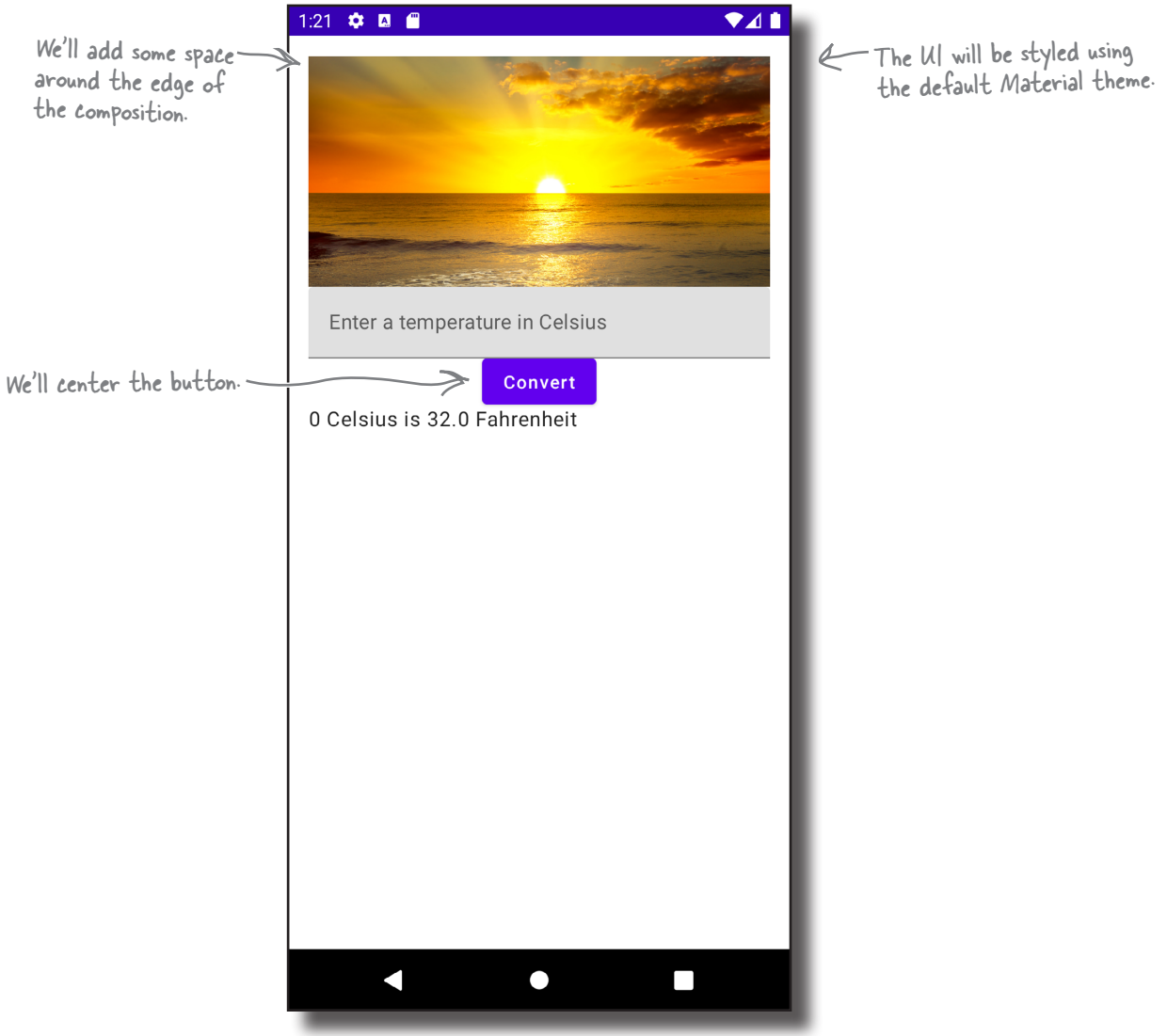

The app now works in exactly the way we want. All that's left is for us to make some extra tweaks to its appearance.

Text
Converter
**Modify**

# We'll tweak the app's appearance

We're going to change the app so that there's a gap between the edge of the screen and the UI components, and the button is centered horizontally. We'll also style the composables so that they use the default Material theme.

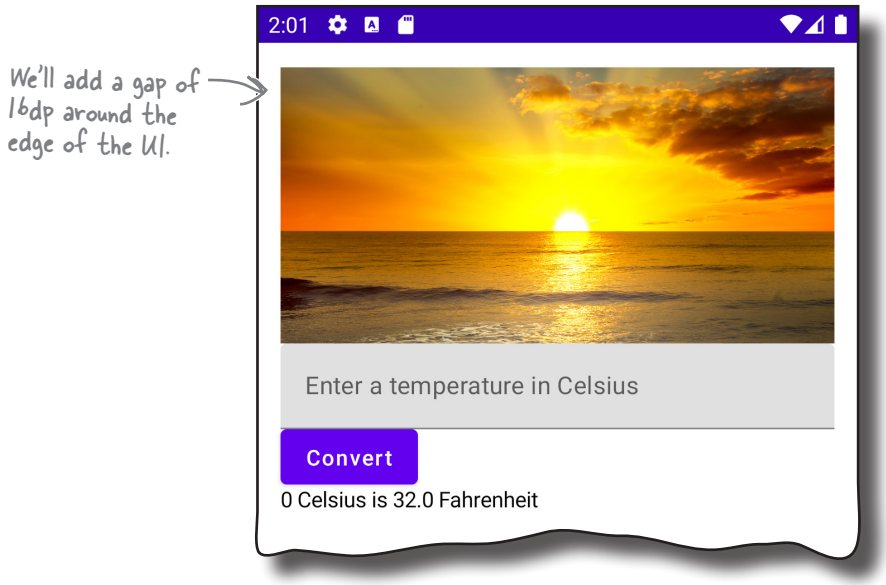
Here's what the new version of the app will look like:




We'll make these tweaks over the next few pages. We'll start by adding some space around the edge of the UI.

# Add padding to the Column composable

We want to add a gap between the edge of the screen and the app's components so that it looks like this:



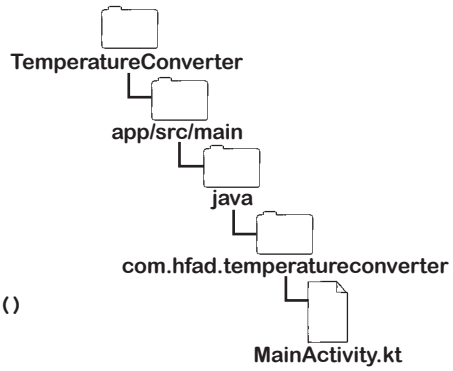

We'll do this by adding some **padding** to the `Column` composable. Applying padding to composables has a similar effect to when you apply it to views; it adds extra space around the component's edges.

You add padding to a composable using a **`Modifier`**. `Modifiers` let you decorate or add extra behavior to composables. To add padding of 16dp to a `Column` composable, for example, you'd use the code:

```
Column(modifier = Modifier.padding(16.dp)) {
    ...
}
```

`Modifiers` are extremely flexible. You may remember, for example, that when we wrote the `Header` composable function, we used a `Modifier` to set the `Image` height and width like so:

```
Image(
    ...
    modifier = Modifier.height(180.dp).fillMaxWidth()
)
```




We'll add padding to the app's `Column` composable a few pages ahead. Before we do that, let's find out how to center the `Button`.

# You can center composables in Columns or Rows

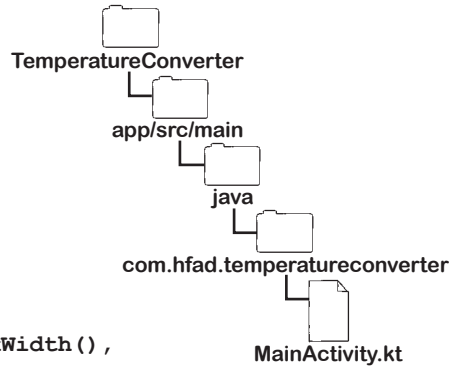
Text
Converter
**Modify**

If you want to center one or more composables, such as a `Button` or `Text`, you can do so using `Columns` and `Rows`. There are a couple approaches you can take, depending on what you want to do.

## Center all of a Column's content

If you want to align the composables in a `Column` so they're all centered horizontally, you can do so by making the `Column` as wide as possible, then setting its **`horizontalAlignment`** argument.

To horizontally center all the composables in the Temperature Converter app, for example, we'd add the code (in bold) to the `Column` composable:

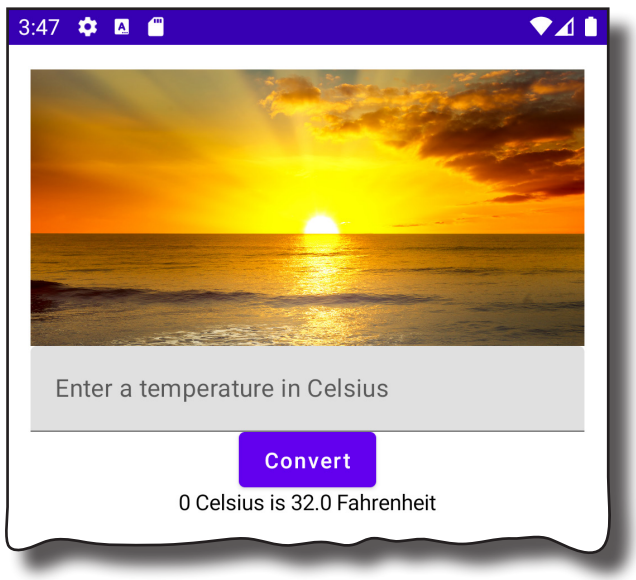



```
@Composable
fun MainActivityContent() {
    ...
    Column(modifier = Modifier.padding(16.dp).fillMaxWidth(),
           horizontalAlignment = Alignment.CenterHorizontally) {
        ...
    }
}
```

This code centers everything in the column horizontally.

Doing so makes the app look like this:

All the components are centered horizontally. It's less noticeable for the Image and TextField because they're set to fill the maximum width.

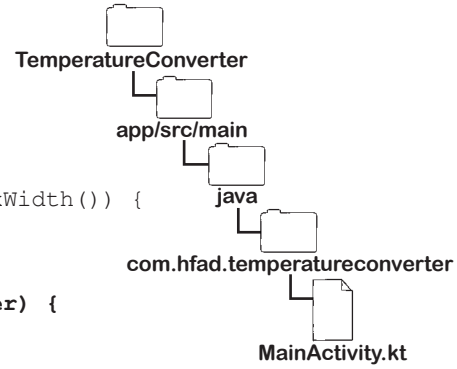



But what if you want to center just one of the composables?

## Center the content of a single Row

If you want to horizontally center a single composable, you can do so by placing it in a `Row`. Simply modify the `Row` to make it as wide as possible, then set its **`horizontalArrangement`** attribute.

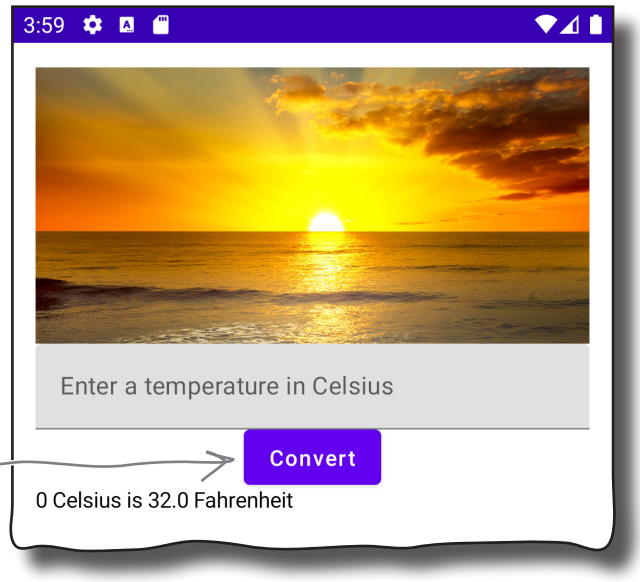
To horizontally center the `ConvertButton` composable, for example, we could put it in a `Row`, and center it like this:




```
@Composable
fun MainActivityContent() {
    ...
    Column(modifier = Modifier.padding(16.dp).fillMaxWidth()) {
        ...
        Row(modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.Center) {
            ConvertButton { ... }
        }
        ...
    }
}
```

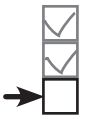
Here, only ConvertButton is centered horizontally.

Doing so centers the button like this:




This time only the Button is centered horizontally.

That's how you use `Columns` and `Rows` to align and arrange composables. Before we update *MainActivity.kt*, there's just one final thing we need to discuss: how to apply themes to composables.



Text
Converter
**Modify**

# Applying themes: revisited

As you learned in Chapter 8, themes let you give your app a common look and feel. You can use one to remove the default app bar, for example, or change the app's colors.

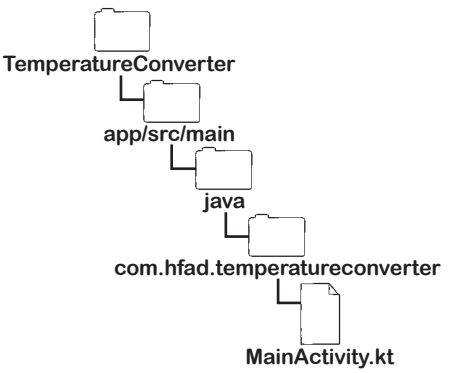
You already know that you can apply a theme by defining one in the app's style resource files, which you then refer to in *AndroidManifest.xml*. Doing so applies the theme to the app, including any views. What this *doesn't* do, however, is apply the theme to any *composables*. To style these, you need to use a different approach.

## How to apply a theme to composables

If you want to apply a theme to a composition, you need to do so using Kotlin code. The following code, for example, uses `MaterialTheme` and `Surface` composables to apply a theme to `MainActivityContent`:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MaterialTheme {
                Surface {
                    MainActivityContent()
                }
            }
        }
    }
}
```

This applies the default Material theme to MainActivityContent.



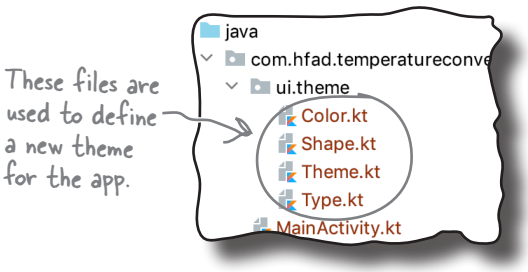

**`MaterialTheme`** is used to apply the default Material theme to composables. In this case, it applies it to `MainActivityContent` and the composables this function calls, such as `TemperatureText` and `ConvertButton`.

The above code also includes a **`Surface`** composable, which is used to style surfaces. It's used to apply things like 3D effects and shadows.

If you want to override the default `MaterialTheme`, you can do so be defining a new theme in Kotlin code. Android Studio usually adds extra theme code for you when you choose the option to create a new project with a Compose activity, so let's take a look at this code and see what it does.

# Android Studio includes extra theme code

When we created the Temperature Converter project, Android Studio added extra Kotlin files to define a new theme. These files are named *Color.kt*, *Shape.kt*, *Theme.kt*, and *Type.kt*, and they're located in the *app/src/main/java* folder in the *com.hfad.temperatureconverter.ui.theme* package.



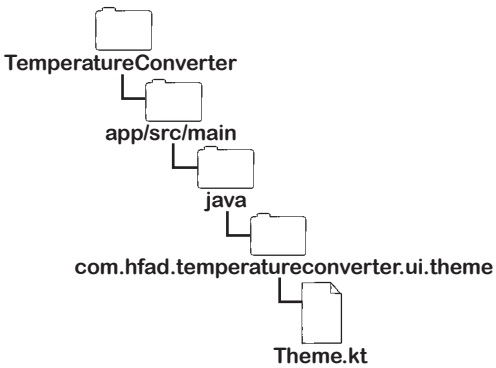

The main file is *Theme.kt*, as it defines a new theme for the app. The theme's name is `TemperatureConverterTheme`, and its code looks like this:

```
package com.hfad.temperatureconverter.ui.theme

import androidx.compose.foundation.isSystemInDarkTheme
import androidx.compose.material.MaterialTheme
import androidx.compose.material.darkColors
import androidx.compose.material.lightColors
import androidx.compose.runtime.Composable

private val DarkColorPalette = darkColors(
    primary = Purple200,
    primaryVariant = Purple700,
    secondary = Teal200
)

private val LightColorPalette = lightColors(
    primary = Purple500,
    primaryVariant = Purple700,
    secondary = Teal200

    /* Other default colors to override
    background = Color.White,
    surface = Color.White,
    onPrimary = Color.White,
    onSecondary = Color.Black,
    onBackground = Color.Black,
    onSurface = Color.Black,
    */
)
```

This code defines light and dark color palettes.

The colors themselves are defined in Color.kt.

Don't worry if your code looks different to ours. We're just showing you the Theme.kt code Android Studio generated for us; it may have generated different code for you depending on your version of Android Studio.




The code continues on the next page.

# Theme.kt continued

**Text Converter Modify**

```
@Composable
fun TemperatureConverterTheme(
    darkTheme: Boolean = isSystemInDarkTheme(),
    content: @Composable() () -> Unit
) {
    val colors = if (darkTheme) {
        DarkColorPalette
    } else {
        LightColorPalette
    }

    MaterialTheme(
        colors = colors,
        typography = Typography,
        shapes = Shapes,
        content = content
    )
}
```

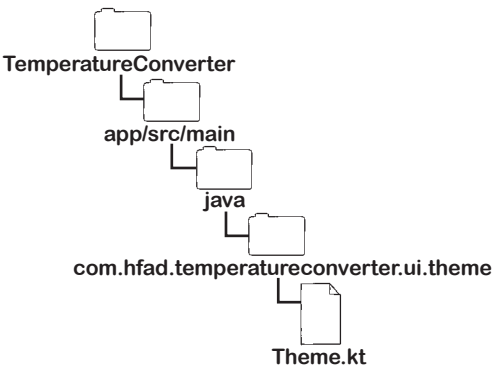
This is a composable function for the app's theme.

See whether the device is using a light or dark theme.

Set the color palette.

The theme's typography is defined in Type.kt.

Shape.kt defines any shapes used by the theme.




As you can see, the theme overrides the `MaterialTheme` colors, typography, and shapes. If you want to tweak any of these, you can do so by updating the files in the *.ui.theme* package.

## How to apply the theme

Once you've defined a theme, you can apply it to the app's composables. To apply `TemperatureConverterTheme` to `MainActivityContent`, for example, you'd use the following code:

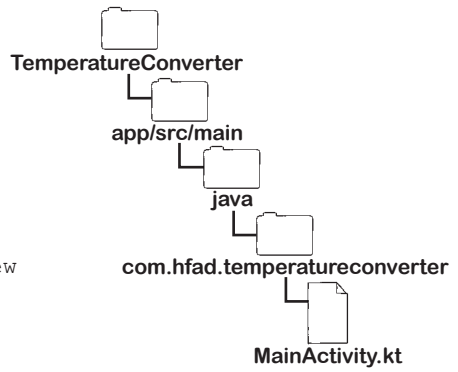
```
...
    setContent {
        TemperatureConverterTheme {
            Surface {
                MainActivityContent()
            }
        }
    }
...
```

This applies TemperatureConverterTheme to MainActivityContent, including its surfaces.

# The full code for MainActivity.kt

You now know everything that you need to tweak the UI's appearance and make it look exactly how we want.

Here's the complete code for *MainActivity.kt*; update the file to include the changes below (in bold):




```
package com.hfad.temperatureconverter

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.runtime.Composable
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.foundation.Image
import androidx.compose.foundation.layout.*
import androidx.compose.material.*
import androidx.compose.ui.Modifier
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.res.painterResource
import androidx.compose.ui.unit.dp
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MaterialTheme {
                Surface {
                    MainActivityContent()
                }
            }
        }
    }
}
```

Add these imports.

Apply the default Material theme to the composition, including any surfaces.

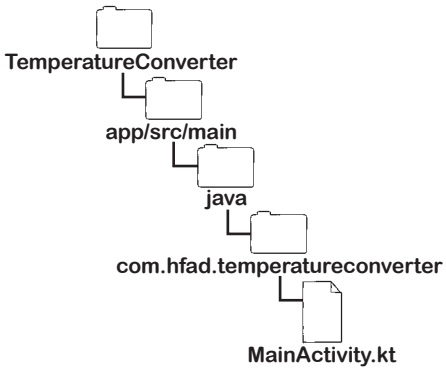
The code continues on the next page.

# MainActivity.kt continued

Text
Converter
**Modify**

```
@Composable
fun TemperatureText(celsius: Int) {
    val fahrenheit = (celsius.toDouble()*9/5)+32
    Text("$celsius Celsius is $fahrenheit Fahrenheit")
}

@Composable
fun ConvertButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Convert")
    }
}

@Composable
fun EnterTemperature(temperature: String, changed: (String) -> Unit) {
    TextField(
        value = temperature,
        label = { Text("Enter a temperature in Celsius") },
        onValueChange = changed,
        modifier = Modifier.fillMaxWidth()
    )
}

@Composable
fun Header(image: Int, description: String) {
    Image(
        painter = painterResource(image),
        contentDescription = description,
        modifier = Modifier
            .height(180.dp)
            .fillMaxWidth(),
        contentScale = ContentScale.Crop
    )
}
```

You don't need to change any of the code on this page.




The code continues on the next page.

# MainActivity.kt continued

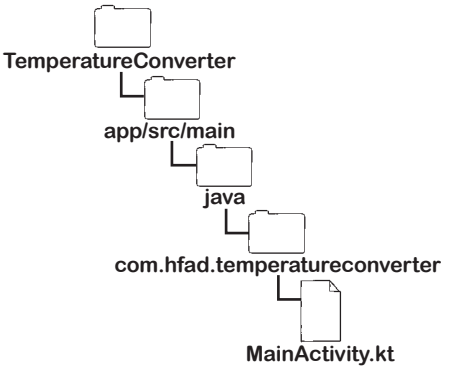
```
@Composable
fun MainActivityContent() {
    val celsius = remember { mutableStateOf(0) }
    val newCelsius = remember { mutableStateOf("") }

    Column(modifier = Modifier.padding(16.dp).fillMaxWidth()) {
        Header(R.drawable.sunrise, "sunrise image")
        EnterTemperature(newCelsius.value) { newCelsius.value = it }
        Row(modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.Center) {
            ConvertButton {
                newCelsius.value.toIntOrNull()?.let {
                    celsius.value = it
                }
            }
        }
        TemperatureText(celsius.value)
    }
}

@Preview(showBackground = true)
@Composable
fun PreviewMainActivity() {
    MaterialTheme {
        Surface {
            MainActivityContent()
        }
    }
}
```

Put padding of 16dp between the UI and the edge of the screen.

Add a Row to center the Button horizontally.

Add a closing curly bracket for the Row.

Apply the default Material theme.

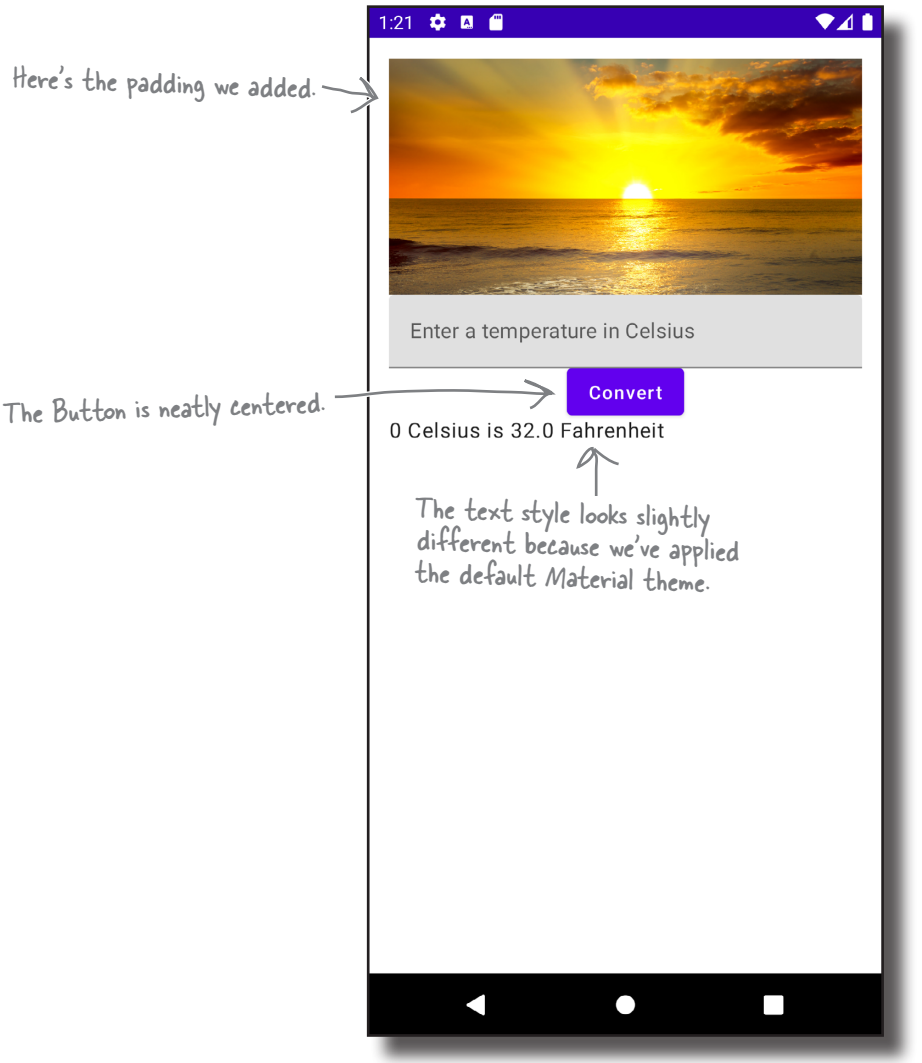



Let's take the app for a test drive.

# Test Drive

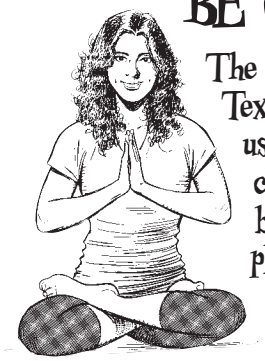
Text
Converter
**Modify**

When we run the app, `MainActivity` is displayed, and the UI looks exactly how we want it to. There's a gap between the UI and the edge of the screen, the button is centered horizontally, and it uses the default Material theme.




Congratulations! You've now learned how to build a UI using Jetpack Compose instead of views.

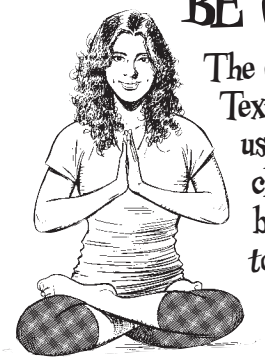
In the next chapter, you'll take this knowledge even further by learning how to integrate composables into an existing `View`-based UI.



# BE Compose

The composable function below includes a TextField, a Button, and a Text. When the user enters a name in the TextField and clicks the Button, the new name should be displayed in the Text. Your job is to play like you're Compose, and say if the function works as intended. If not, how would you change it?

```
@Composable
fun ChangeHello() {
    val name = mutableStateOf("friend")
    val newName = mutableStateOf("")

    TextField(
        value = newName.value,
        label = { Text("Enter your name") },
        onValueChange = { name.value = it }
    )

    Button(
        onClick = { name = newName }
    ) {
        Text("Update Hello")
    }

    Text("Hello ${name.value}")
}
```



# BE Compose Solution

The composable function below includes a TextField, a Button, and a Text. When the user enters a name in the TextField and clicks the Button, the new name should be displayed in the Text. Your job is to play like you're Compose, and say if the function works as intended. If not, how would you change it?

Code changes are needed to make the function work as intended. We've added these to the code below.

```
@Composable
fun ChangeHello() {
    val name = remember { mutableStateOf("friend") }
    val newName = remember { mutableStateOf("") }

    Column {
        TextField(
            value = newName.value,
            label = { Text("Enter your name") },
            onValueChange = { newName.value = it }
        )

        Button(
            onClick = { name.value = newName.value }
        ) {
            Text("Update Hello")
        }

        Text("Hello ${name.value}")
    }
}
```

name and newName need to be stored in memory.

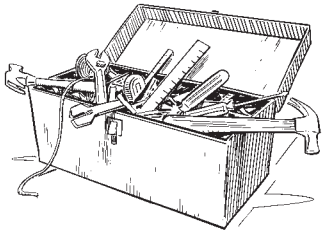
Without this, the composables would be stacked on top of one another.

The TextField needs to use newName for its value.

Update name.value when the user clicks on the button.

Closing curly bracket for the Column.

# Your Android Toolbox



**You've got Chapter 18 under your belt and now you've added Jetpack Compose to your toolbox.**

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

## Bullet Points

- Jetpack Compose lets you build native UIs using pure Kotlin code.
- The UI is built using composables instead of views and layout files.
- Compose activities extend `ComponentActivity`.
- Any composables run from `setContent()` will be displayed to the activity's UI.
- A composable function is a function that uses one or more composables. It needs to be annotated with `@Composable`.
- You can preview composable functions with no arguments using the `@Preview` annotation.
- Use `Text` to display text.
- Use `Row` and `Column` to arrange composables in rows and columns.
- Use `Image` to display an image.
- Use `Button` to display a button.
- A composable is only recomposed when any values it depends on are updated.
- Use `remember` to store an object in memory.
- Use `mutableStateOf()` to create a `MutableState` object whose values can be updated.
- Use `TextField` to display a text field.
- Use a `Modifier` to decorate a composable or add behavior.
- Use `Columns` and `Rows` to center composables.
- Use `MaterialTheme` to apply the default Material theme to composables.
- You can override the default theme by creating a new one in Kotlin code, and applying it to the UI.
- Use `Surface` to style surfaces and apply 3D effects.

# 19 integrating compose with views

# 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**.

# You can add composables to View-based UIs

In the previous chapter, you learned how to implement a Compose UI by building a brand new Temperature Converter app. Instead of adding views to a layout file, you created the UI by calling composables in the activity's Kotlin code.

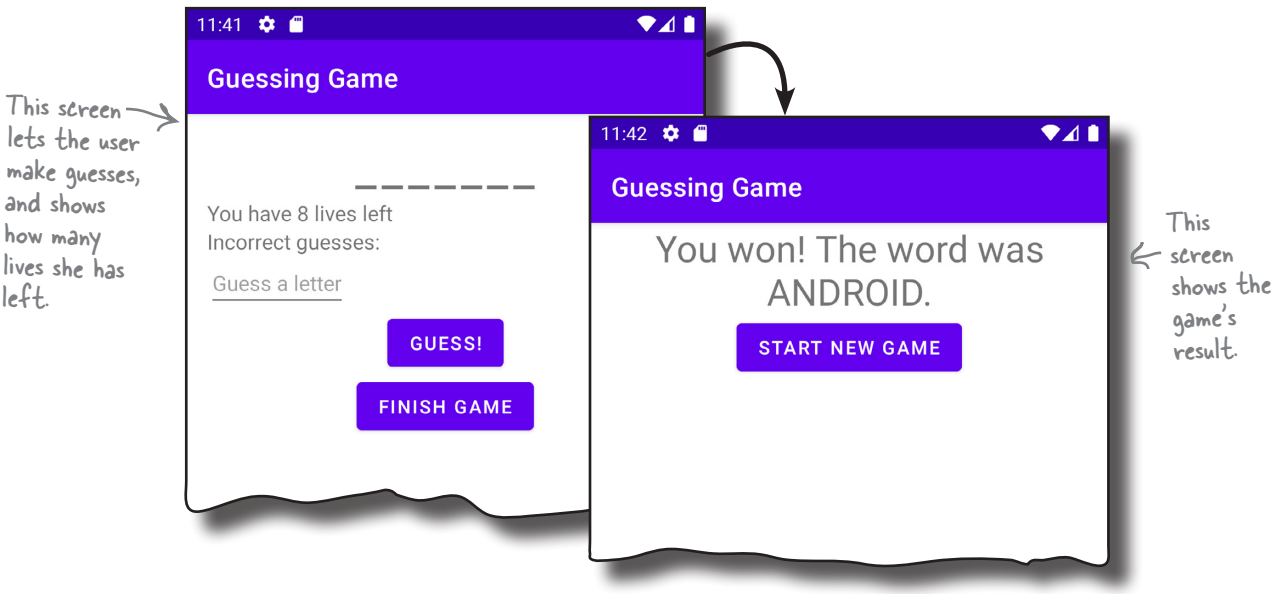
There are times, however, when you might want to use views *and* composables in the same UI. This may be because you want to use components that are only available as views or composables, for example, or you want to migrate parts of your app to Compose.

The great news is that you can add composables to a UI that's defined in a layout file. We'll show you how this is done by taking the Guessing Game app we created earlier in the book, and migrating it to Compose.

## The Guessing Game app revisited

As we're sure you remember, the Guessing Game app lets the user guess which letters are included in a secret word. She wins the game if she guesses all the letters, and loses if she runs out of lives.

Here's what the game currently looks like:




Before we start replacing the app's views with composables, let's quickly review how the app is constructed.

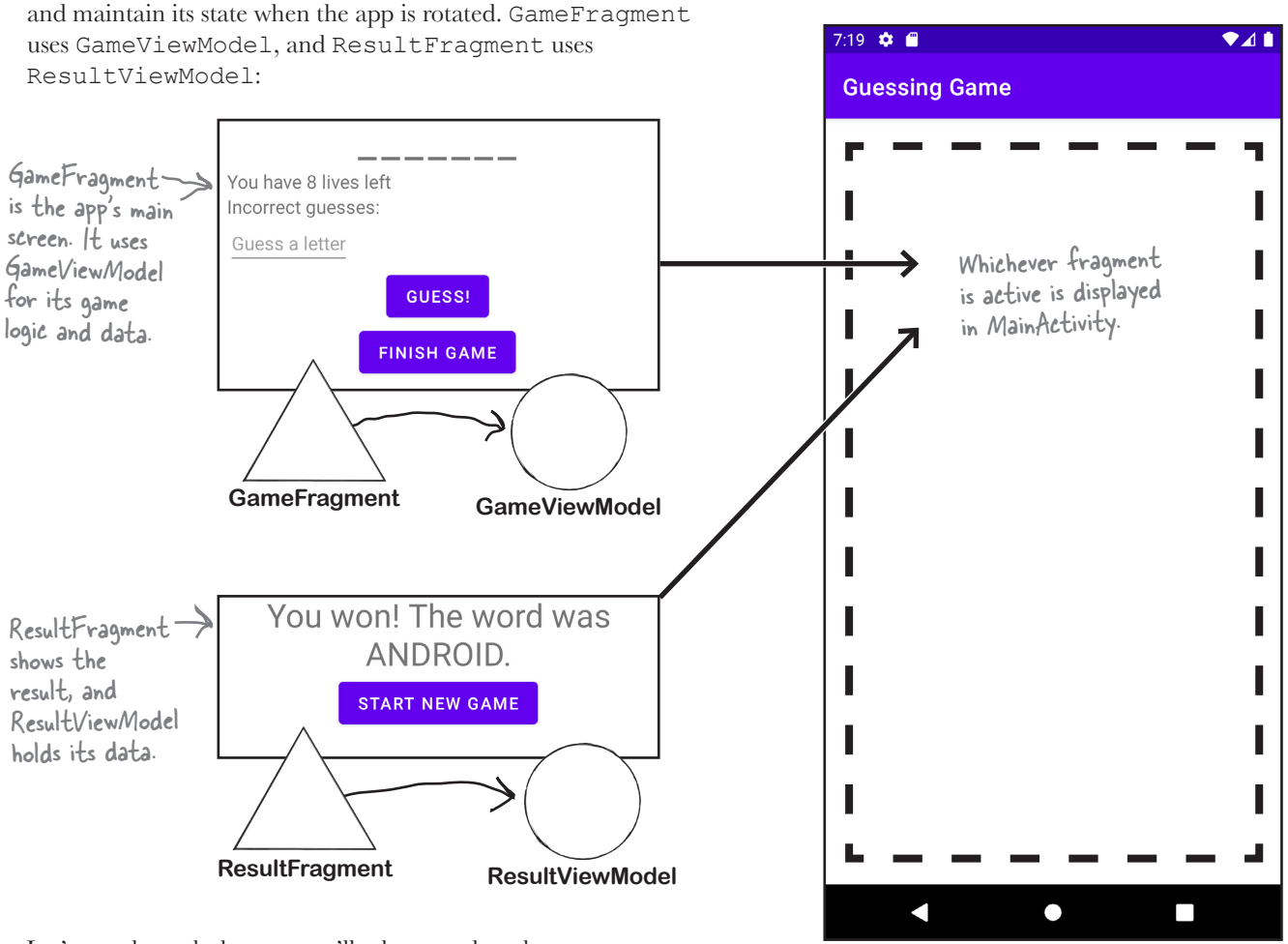
# The Guessing Game app structure

The Guessing Game app uses two fragments for its UI: `GameFragment` and `ResultFragment`.

`GameFragment` is the app's main screen, which the user interacts with to play the game. It displays information—such as the number of lives left and any incorrect guesses the user has made—and lets the user make guesses. It also includes a button that, when clicked, lets the user immediately finish the game without making any more guesses.

`ResultFragment` is displayed when the game is over. It shows the user if she has won or lost the game, along with the secret word.

The app also include two view models—`GameViewModel` and `ResultViewModel`—which hold the app's game logic and data, and maintain its state when the app is rotated. `GameFragment` uses `GameViewModel`, and `ResultFragment` uses `ResultViewModel`:



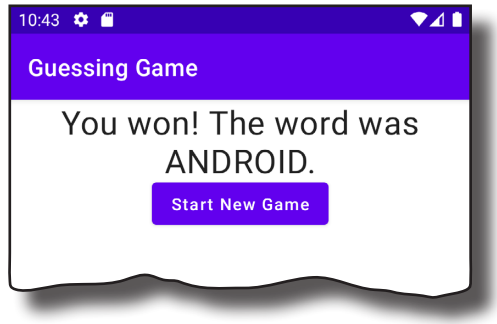

Let's run through the steps we'll take to update the app.

# Here's what we're going to do

We're going to replace the Guessing Game app's views with composables in two main steps:

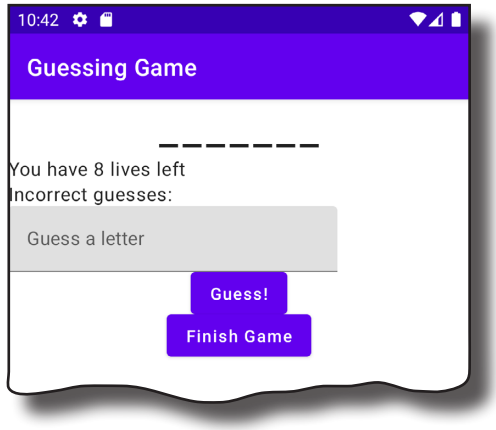
1. **Replace ResultFragment's views with composables.**
We'll add the Compose libraries to the app's *build.gradle* files, and then add composables to `ResultFragment`'s layout to replicate the current views. When we're satisfied that the composables do what we want, we'll remove the views from its UI.

This is what the Compose version of ResultFragment's UI will look like.




2. **Replace GameFragment's views with composables.**
We'll then go through a similar process for `GameFragment`. We'll add composables to its layout to reproduce its current views, and when we're certain that they work in the way we want, we'll remove the views from its UI.

This is what the Compose version of GameFragment will look like.



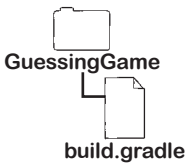

**Don't forget!**

You're going to modify the Guessing Game app you created in Chapters 11 to 13, so make sure you open the project for this app.

Let's start by adding the Compose libraries to the app's *build.gradle* files.

# Update the project build.gradle file...

We'll begin by specifying which version of Compose to use. Open the file *GuessingGame/build.gradle*, and add the lines below (in bold) to the appropriate sections:



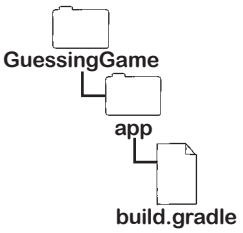

```
buildscript {
    ext.compose_version = "1.0.1"
}
dependencies {
    ...
    classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:1.5.21"
}
```

We're using this version of the Compose libraries.

Use this version of the Kotlin Gradle plug-in.

## ...and update the app build.gradle file, too

In the app's *build.gradle* file, we need to add a bunch of Compose options and libraries, and make sure the minimum SDK is 21. Open the file *GuessingGame/app/build.gradle*, and add the following lines (in bold):




```
android {
    defaultConfig {
        minSdk 21
        ...
    }
    buildFeatures {
        ...
        compose true
    }
    composeOptions {
        kotlinCompilerExtensionVersion compose_version
    }
}
```

Make sure your file includes this line.

Enable Compose.

Add this section.

```
dependencies {
    ...
    implementation("androidx.compose.ui:ui:$compose_version")
    implementation("androidx.compose.ui:ui-tooling:$compose_version")
    implementation("androidx.compose.foundation:foundation:$compose_version")
    implementation("androidx.compose.material:material:$compose_version")
    implementation("androidx.compose.material:material-icons-core:$compose_version")
    implementation("androidx.compose.material:material-icons-extended:$compose_version")
    implementation("androidx.compose.runtime:runtime-livedata:$compose_version")
    ...
}
```

You need to add all of these Compose dependencies (yes, really).

Once you've made these changes, click on the Sync Now option.

**ResultFragment**
GameFragment

# We'll replace ResultFragment's views with composables

Now that we've updated the *build.gradle* files to include Compose, we can start replacing the app's views with composables. We'll begin with `ResultFragment`, as this is the simplest fragment.

As you may recall, `ResultFragment`'s layout includes a `TextView` to display the game's result, and a `Button` to let the user start a new game. It looks like this:

Notice how the default Material theme styles composables and views slightly differently. You can change this behavior by overriding the theme, or changing component properties.

We can replace these views with composables by using a `Text` composable instead of the `TextView`, and a `Button` composable instead of the `Button` view. This is what the new UI will look like:

We'll build the new UI bit by bit so that at first, `ResultFragment` will use both views and composables. Let's begin by finding out how to add composables to a layout file.

# A ComposeView lets you add composables to a layout

If you want to add composables to a `View`-based UI, you can do so by adding a **`ComposeView`** element to the layout file. This is a type of view that can display composables, and the code looks like this:

```
<androidx.compose.ui.platform.ComposeView
    android:id="@+id/compose_view"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
```

This is all the code you need to add a ComposeView to a layout file. Put it wherever you want any composables to appear in the UI

You can think of a `ComposeView` as a type of view that acts as a placeholder for any composables you want to add to the UI in your Kotlin code. When the app runs, it displays the layout's views, and populates the `ComposeView` with composables.

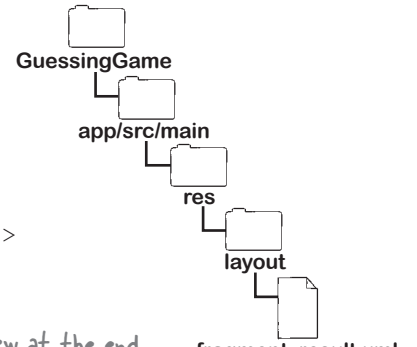
**A ComposeView is a View that acts as a placeholder for composables. It lets you use Compose in a View-based UI.**

## We'll add a ComposeView to fragment_result.xml

We want to add `Text` and `Button` composables to `ResultFragment`'s UI, so we need to add a `ComposeView` to its layout file.

Here's the updated code for *fragment_result.xml*; update the file to include the changes below (in bold):

```
<?xml version="1.0" encoding="utf-8"?>
<layout...>

    <data>
        <variable
            name="resultViewModel"
            type="com.hfad.guessinggame.ResultViewModel" />
    </data>

    <LinearLayout...>
        ...
        <androidx.compose.ui.platform.ComposeView
            android:id="@+id/compose_view"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"/>
    </LinearLayout>
</layout>
```




Add this ComposeView at the end of ResultFragment's linear layout.

Now that we've added the `ComposeView`, let's add some composables to it.

**ResultFragment**
GameFragment

# Add composables using Kotlin code

Once the layout includes a `ComposeView`, you can add composables to it in the fragment's `onCreateView()` method using code like this:

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
): View? {
    _binding = FragmentResultBinding.inflate(inflater, container, false).apply {
        composeView.setContent {
            //Add composables
        }
    }
    ...
}
```

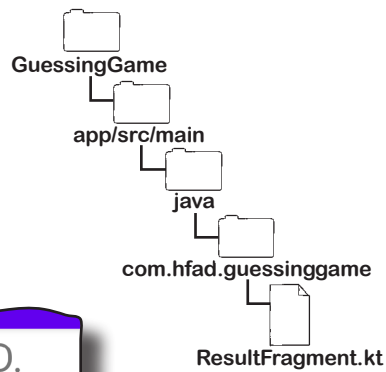
This applies any composables to the UI.

This tells the ComposeView what composables to include.

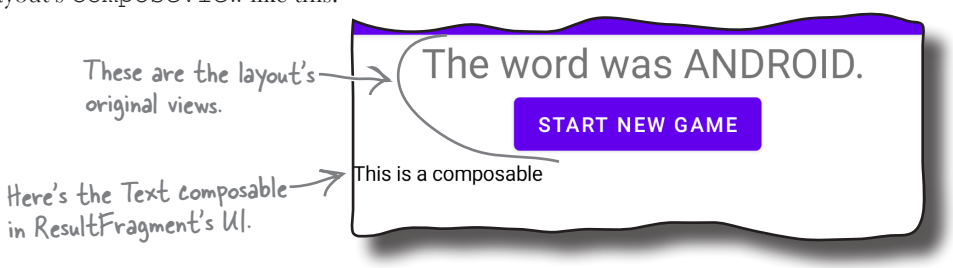
The code calls `setContent()` on the layout's `ComposeView`, which tells it what composables it needs to include. It then applies this to the fragment's inflated layout. If you wanted to add a `Text` composable to `ResultFragment`'s layout, for example, you'd use the code:

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
): View? {
    _binding = FragmentResultBinding.inflate(inflater, container, false).apply {
        composeView.setContent {
            Text("This is a composable")
        }
    }
    ...
}
```

This adds a Text composable to ResultFragment's layout.




Running this code would display the `Text` composable in the layout's `ComposeView` like this:



# Add a composable function for the fragment's content

Now that you've learned how to add composables to a `ComposeView`, let's use the one we've added to *fragment_result.xml* to replicate its existing views.

We'll begin by adding a new composable function to *ResultFragment.kt* named `ResultFragmentContent`, which we'll use for the fragment's UI. We'll call it from `setContent()` so that any composables we add to it will run when `ResultFragment` gets displayed.

Here's what the new code looks like; we'll add it to *ResultFragment.kt* a few pages ahead:

```
...
import androidx.compose.material.MaterialTheme
import androidx.compose.material.Surface
import androidx.compose.runtime.Composable

class ResultFragment : Fragment() {
    ...
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentResultBinding.inflate(inflater, container, false).apply {
            composeView.setContent {
                MaterialTheme {
                    Surface {
                        ResultFragmentContent()
                    }
                }
            }
        }
        ...
    }
    ...
}

@Composable
fun ResultFragmentContent() {
}
```

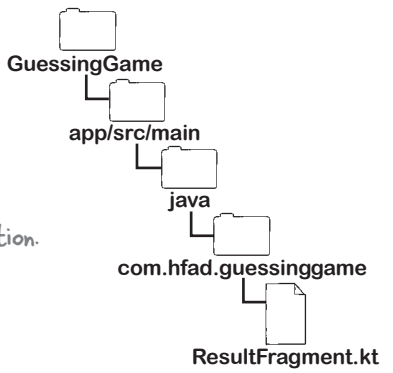
Import these classes.

This adds ResultFragmentContent to the layout's ComposeView, and applies the default Material theme.

This applies composables to the UI.

The rest of the code doesn't need to change.

This is the ResultFragmentContent composable function.



# Reproduce the Start New Game button

Now that we've added the `ResultFragmentContent` composable function to *ResultFragment.kt*, we can use it to add composables to the UI. We'll begin by adding a button.

`ResultFragment` currently includes a button `View` labeled "Start new game" that uses the following `OnClickListener` to navigate to `GameFragment` when clicked:

```
binding.newGameButton.setOnClickListener {
    view.findNavController()
        .navigate(R.id.action_resultFragment_to_gameFragment)
}
```

The button runs this code when it's clicked.

We can reproduce the button in Compose by creating a new composable function named `NewGameButton` that we'll run from `ResultFragmentContent`. We'll add a lambda argument to `NewGameButton` so that `ResultFragmentContent` can tell it what to do when it's clicked.

Here's what the new code looks like; we'll add it to *ResultFragment.kt* a couple pages ahead:

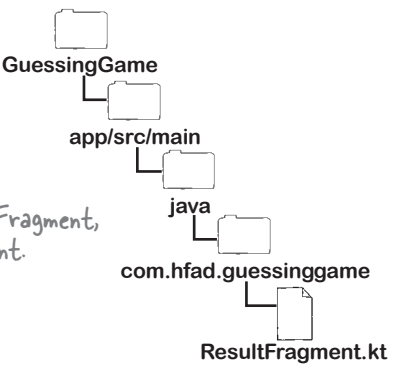
```
@Composable
fun NewGameButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Start New Game")
    }
}

@Composable
fun ResultFragmentContent(view: View) {
    Column(modifier = Modifier.fillMaxWidth(),
        horizontalAlignment = Alignment.CenterHorizontally) {
        NewGameButton {
            view.findNavController()
                .navigate(R.id.action_resultFragment_to_gameFragment)
        }
    }
}
```

This is the NewGameButton composable function.

NewGameButton needs a View for it to navigate to GameFragment, so we're adding a View argument to ResultFragmentContent.

We're using a Column to center the button horizontally.

This adds NewGameButton to the UI, and makes it navigate when clicked.

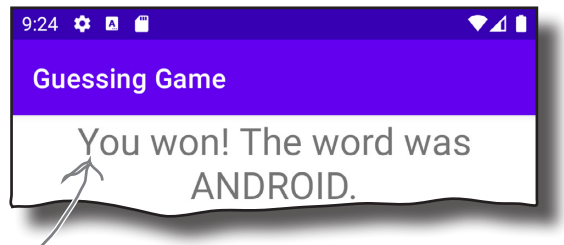



That's the button sorted. What about the layout's text?

# Reproduce ResultFragment's TextView

As you may recall, `ResultFragment` uses a `TextView` in its layout to display the game's result. It's defined using this code:

```
<TextView
    android:id="@+id/won_lost"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:gravity="center"
    android:textSize="28sp"
    android:text="@{resultViewModel.result}" />
```




The TextView's text comes from ResultViewModel's result property.

Instead of using a `TextView` to display the text, we can define a new `ResultText` composable function that displays the text in a `Text` composable. We'll add a `String` argument to `ResultText` that `ResultFragmentContent` will use to pass it the text.

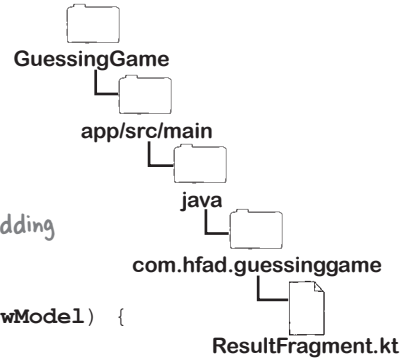
Here's the new code; we'll add it to *ResultFragment.kt* on the next page:

```
@Composable
fun ResultText(result: String) {
    Text(text = result,
         fontSize = 28.sp,
         textAlign = TextAlign.Center)
}

@Composable
fun ResultFragmentContent(view: View, viewModel: ResultViewModel) {
    Column(modifier = Modifier.fillMaxWidth(),
           horizontalAlignment = Alignment.CenterHorizontally) {
        ResultText(viewModel.result)
        NewGameButton {
            view.findNavController()
                .navigate(R.id.action_resultFragment_to_gameFragment)
        }
    }
}
```

This is the ResultText composable function.

ResultText gets its text from the view model, so we're adding a ResultViewModel argument to ResultFragmentContent.

Add ResultText to the UI, and get it to use the view model's result property for its text.

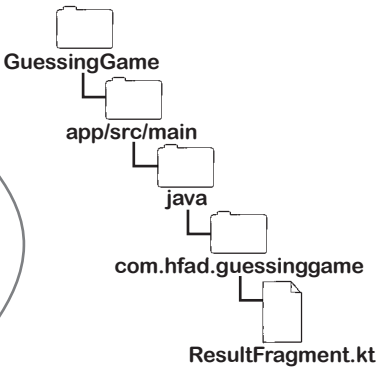



That's all the code that we need to reproduce all of `ResultFragment`'s views with composables. Let's see what the code looks like.

**ResultFragment**
GameFragment

# The updated code for ResultFragment.kt

Here's the updated code for *ResultFragment.kt*; update the file to include the changes below (in bold):




```
...
import androidx.compose.material.MaterialTheme
import androidx.compose.material.Surface
import androidx.compose.runtime.Composable
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.material.Button
import androidx.compose.material.Text
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.unit.sp

class ResultFragment : Fragment() {
    private var _binding: FragmentResultBinding? = null
    private val binding get() = _binding!!
    lateinit var viewModel: ResultViewModel
    lateinit var viewModelFactory: ResultViewModelFactory

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentResultBinding.inflate(inflater, container, false).apply {
            composeView.setContent {
                MaterialTheme {
                    Surface {
                        view?.let { ResultFragmentContent(it, viewModel) }
                    }
                }
            }
        }
        val view = binding.root
```

Import these classes.

This applies composables to the UI

Add these lines.

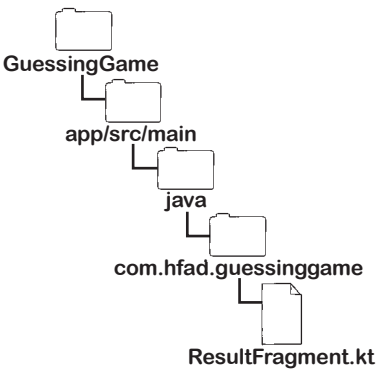
Here, view calls the fragment's getView() method, which returns the root view.

The code continues on the next page.

# ResultFragment.kt continued

```
        val result = ResultFragmentArgs.fromBundle(requireArguments()).result
        viewModelFactory = ResultViewModelFactory(result)
        viewModel = ViewModelProvider(this, viewModelFactory)
            .get(ResultViewModel::class.java)
        binding.resultViewModel = viewModel

        binding.newGameButton.setOnClickListener { ... }
        return view
    }

    override fun onDestroyView() { ... }
}
```

We've left out this code as it hasn't changed.




Add the ResultText composable function, which shows the result.

```
@Composable
fun ResultText(result: String) {
    Text(text = result,
        fontSize = 28.sp,
        textAlign = TextAlign.Center)
}
```

Add NewGameButton to display the button.

```
@Composable
fun NewGameButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Start New Game")
    }
}
```

Add this composable function, which we're using for the fragment's main content.

```
@Composable
fun ResultFragmentContent(view: View, viewModel: ResultViewModel) {
    Column(modifier = Modifier.fillMaxWidth(),
           horizontalAlignment = Alignment.CenterHorizontally) {
        ResultText(viewModel.result)
        NewGameButton {
            view.findNavController()
                .navigate(R.id.action_resultFragment_to_gameFragment)
        }
    }
}
```

Add ResultText and NewGameButton to ResultFragmentContent, arranged in a column.

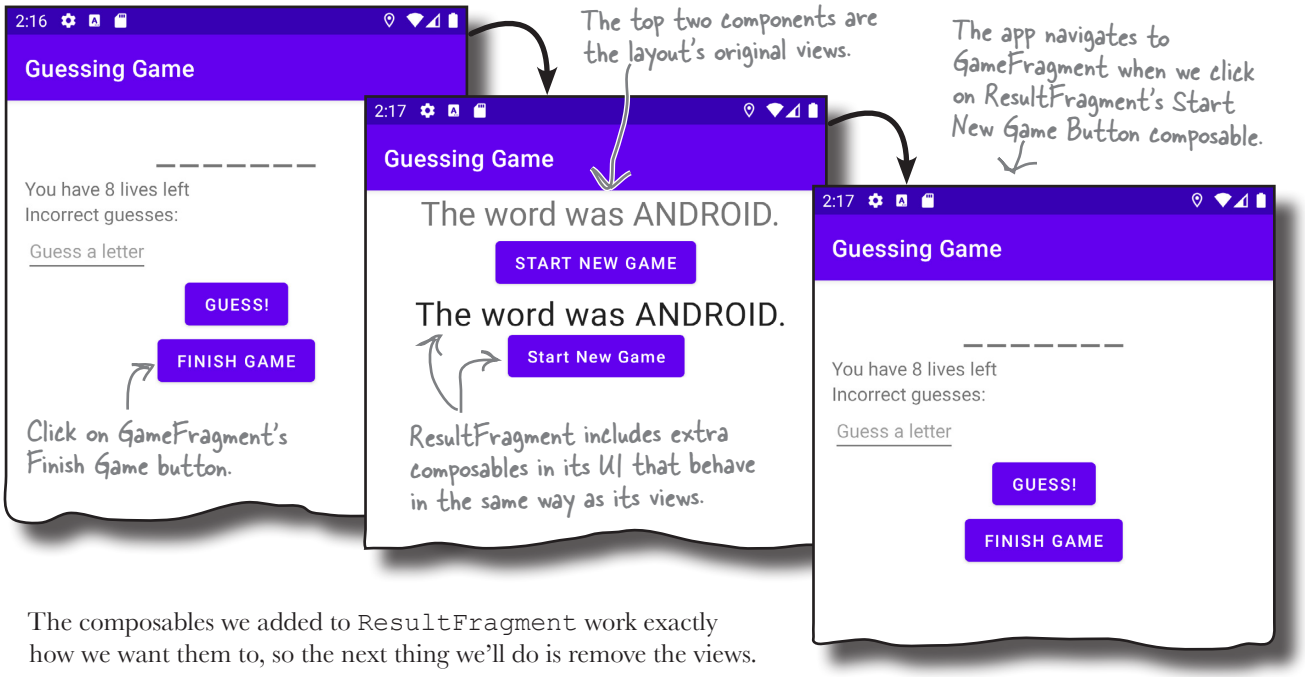
Let's take the app for a test drive.

# Test Drive

**ResultFragment**
**GameFragment**

When we run the app, `GameFragment` is displayed. When we click on its Finish Game button, it navigates to `ResultFragment`.

`ResultFragment` displays the original views, and the composables we just added. The `Text` composable displays the correct text, and the `Button` composable navigates to `GameFragment` when it's clicked.




The composables we added to `ResultFragment` work exactly how we want them to, so the next thing we'll do is remove the views.

## there are no Dumb Questions

**Q: So `ResultFragment`'s views and composables are both using the same view model. Is that right?**

A: Yes. The view model has a property for the game's result, and the `TextView` and `Text` composable both refer to it.

**Q: Do composables usually use a view model?**

A: They often do, yes. Just like with a `View`-based UI, view models provide a useful way of managing the state of any composables so they don't get reset when the user rotates the screen.

**Q: You're passing a view model to `ResultFragmentContent` as an argument. Can't the function just get a reference to the view model?**

A: Composable functions can get a reference to a view model using the `viewModel()` function, but at the time of writing, the Compose library it's in isn't stable. You can find out more about this—and other features—in the appendix.

# We need to remove ResultFragment's views...

You can remove a fragment or activity's views in a couple ways:

1. **By removing any views you don't need from the layout file**
   This approach is useful if the UI contains a mix of views and composables.

2. **By removing the entire layout file**
   If the UI *only* includes composables, you can delete the layout file, and remove any references to it in the activity or fragment's Kotlin code.

In the Guessing Game app, we've reproduced all of `ResultFragment`'s views with composables, so we no longer need any of its original views. This means that we can delete its layout file *fragment_result.xml* so that the UI only includes composables.

## ...and update ResultFragment.kt

Before we delete the layout file, we first need to remove any references to its views from *ResultFragment.kt*, and stop using view binding.

View binding gives your activity and fragment code an easier way to access a layout file's views. As we're removing ResultFragment's layout file, it no longer needs to use view binding.

We also need to tweak the fragment's `onCreateView()` method so that instead of inflating a layout file, it adds the composables to the fragment's UI.

As you learned in Chapter 18, you can do this with an *activity* by simply calling `setContent()` from its `onCreate()` method like this:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            //Code that runs composables
        }
    }
}
```

This is Compose activity code. We need to use Compose fragment code instead.

When you're dealing with a fragment, however, you need a slightly different approach. To find out what this is and why, let's revisit the fragment's `onCreateView()` method.

**ResultFragment**
GameFragment

# onCreateView() returns the UI's root view

As you know, a fragment's `onCreateView()` method gets called when an activity needs to display the fragment's UI.

When the UI is defined with a layout file, code in the `onCreateView()` method inflates the layout into a hierarchy of views, and returns the root view. The root view gets added to the activity's layout, which displays the fragment's UI.

But what if there's no layout file?

## Return a ComposeView for Compose UIs

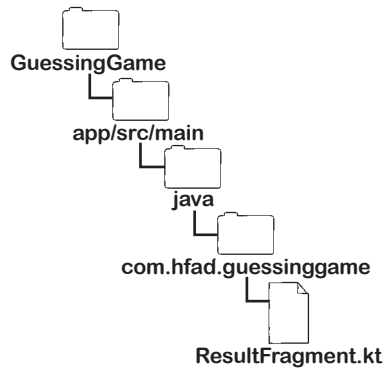
If the fragment's UI is just made up of composables and has no layout file, **the `onCreateView()` method must still return a `View?`, or the code won't compile**.

You deal with this by making the method return a `ComposeView` that includes all of the UI's composables. Here's what the code for this looks like:

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
): View? {
    //Any code that doesn't use composables

    return ComposeView(requireContext()).apply {
        setContent {
            //Any Compose code that defines the UI
        }
    }
}
```

Return a ComposeView.

Add composables to the ComposeView.




When the activity needs to display the fragment's UI, it calls the fragment's `onCreateView()` method as before. The method returns a `ComposeView` that includes the fragment's composables, which are then displayed by the activity.

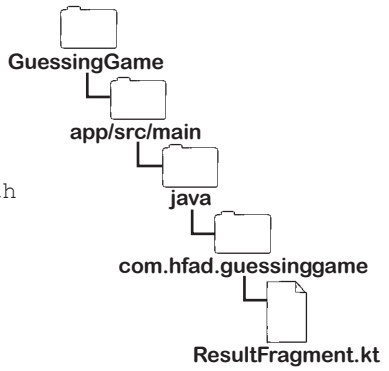
That's everything you need to know to complete the code for *ResultFragment.kt*. We'll show you the full code over the next few pages.

# The full code for ResultFragment.kt

Here's the complete code for *ResultFragment.kt*; update the file to include the changes below (in bold):

```
package com.hfad.guessinggame

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
~~import com.hfad.guessinggame.databinding.FragmentResultBinding~~
import androidx.navigation.findNavController
import androidx.lifecycle.ViewModelProvider
import androidx.compose.material.MaterialTheme
import androidx.compose.material.Surface
import androidx.compose.runtime.Composable
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.material.Button
import androidx.compose.material.Text
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.unit.sp
import androidx.compose.ui.platform.ComposeView

class ResultFragment : Fragment() {
    ~~private var _binding: FragmentResultBinding? = null~~
    ~~private val binding get() = _binding!!~~
    lateinit var viewModel: ResultViewModel
    lateinit var viewModelFactory: ResultViewModelFactory

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        val result = ResultFragmentArgs.fromBundle(requireArguments()).result
        viewModelFactory = ResultViewModelFactory(result)
        viewModel = ViewModelProvider(this@ResultFragment, viewModelFactory)
            .get(ResultViewModel::class.java)
```

Remove this line.




Import this class.

Delete these lines. We're no longer using views, so we don't need to use view binding.

Create the view model before returning the ComposeView.

The code continues on the next page.

**ResultFragment**
GameFragment

# ResultFragment.kt continued

```
~~_binding = FragmentResultBinding.inflate(inflater, container, false).apply {~~
return ComposeView(requireContext()).apply {
    ~~composeView.~~setContent {
        MaterialTheme {
            Surface {
                view?.let { ResultFragmentContent(it, viewModel) }
            }
        }
    }
}
~~val view = binding.root~~

~~val result = ResultFragmentArgs.fromBundle(requireArguments()).result~~
~~viewModelFactory = ResultViewModelFactory(result)~~
~~viewModel = ViewModelProvider(this, viewModelFactory)~~
    ~~.get(ResultViewModel::class.java)~~
~~binding.resultViewModel = viewModel~~

~~binding.newGameButton.setOnClickListener {~~
    ~~view.findNavController()~~
        ~~.navigate(R.id.action_resultFragment_to_gameFragment)~~
~~}~~
~~return view~~
}

~~override fun onDestroyView() {~~
    ~~super.onDestroyView()~~
    ~~_binding = null~~
~~}~~
}

@Composable
fun ResultText(result: String) {
    Text(text = result,
        fontSize = 28.sp,
        textAlign = TextAlign.Center)
}
```

Remove this line.

Return a ComposeView.

Remove this composeView reference as it refers to the ComposeView in the layout file.

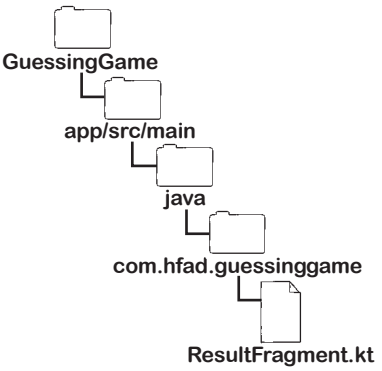
We're no longer using view binding, so delete this line.

Delete these lines because we're getting the view model earlier in the code.

We're not using the button View any more, so delete these lines too.

Make sure you don't delete this.

Delete this method, because we're no longer using view binding.

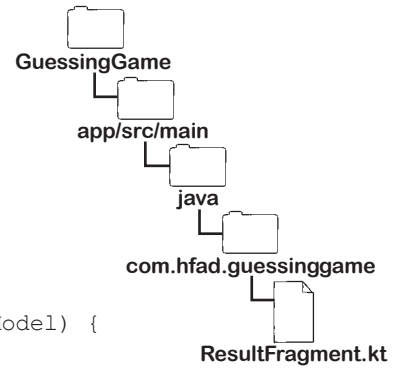



The code continues on the next page.

# ResultFragment.kt continued

```
@Composable
fun NewGameButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Start New Game")
    }
}

@Composable
fun ResultFragmentContent(view: View, viewModel: ResultViewModel) {
    Column(modifier = Modifier.fillMaxWidth(),
           horizontalAlignment = Alignment.CenterHorizontally) {
        ResultText(viewModel.result)
        NewGameButton {
            view.findNavController()
                .navigate(R.id.action_resultFragment_to_gameFragment)
        }
    }
}
```

You don't need to change any of the ResultFragment.kt code on this page.



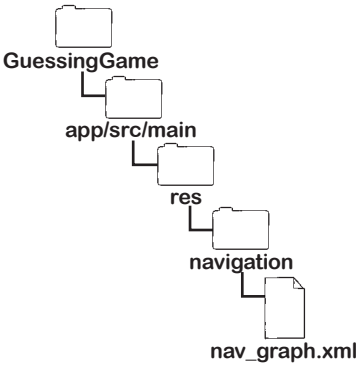

## We can delete fragment_result.xml

Now that `ResultFragment` is no longer using its layout file, we'll remove any references to *fragment_result.xml* from the navigation graph, and delete the layout file.

First, open *nav_graph.xml* in the *app/src/main/res/navigation* folder, and remove the reference to `"@layout/fragment_result"` from the `ResultFragment` section like this:

```
...
<fragment
    android:id="@+id/resultFragment"
    ...
    tools:layout="@layout/fragment_result" >
...
```

Remove this reference from the navigation graph.




Then right-click on *fragment_result.xml* in the explorer, choose Refactor, and then select the Safe Delete option. When you click on the OK button and choose the option to do the refactor, the file is deleted.

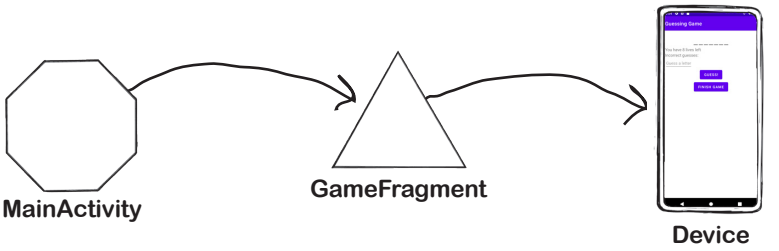
Let's go through what happens when we run the app.

**ResultFragment**
GameFragment

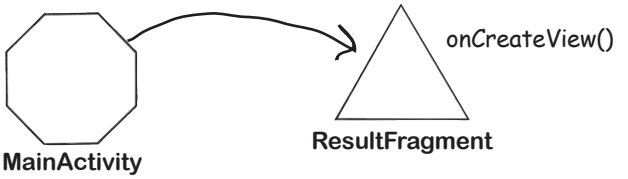
# What happens when the app runs

The following things happen when the app runs:

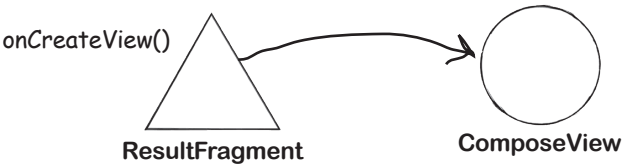
1. **MainActivity launches and displays GameFragment in its layout.**



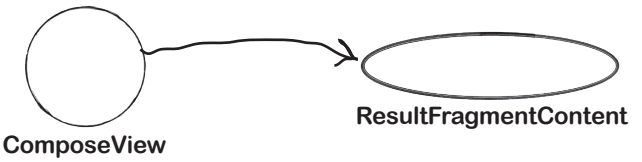

2. **When the user clicks on the Finish Game button, or wins or loses the game, the app navigates to ResultFragment and calls its onCreateView() method.**




3. **The onCreateView() method creates a ComposeView.**

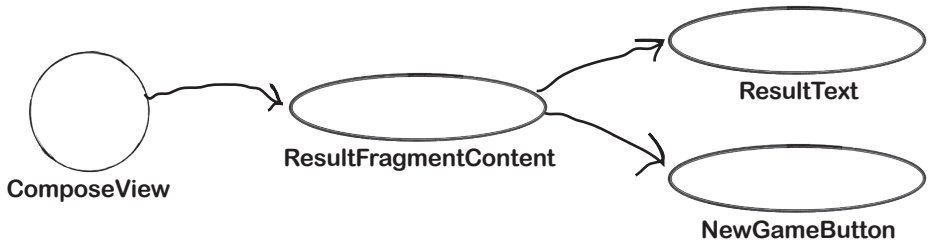



4. **The onCreateView() method sets the ComposeView's content to ResultFragmentContent.**

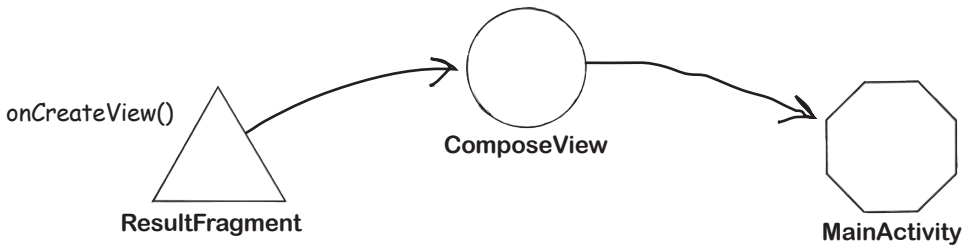


# The story continues

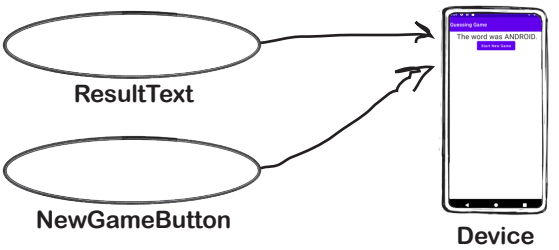
**5** **When ResultFragmentContent runs, it calls the ResultText and NewGameButton composables.**
The composables are added to the `ComposeView`.




**6** **The onCreateView() method returns the ComposeView to MainActivity.**




**7** **MainActivity displays the ComposeView in its layout.**
The `ComposeView`'s composables are displayed on the device.

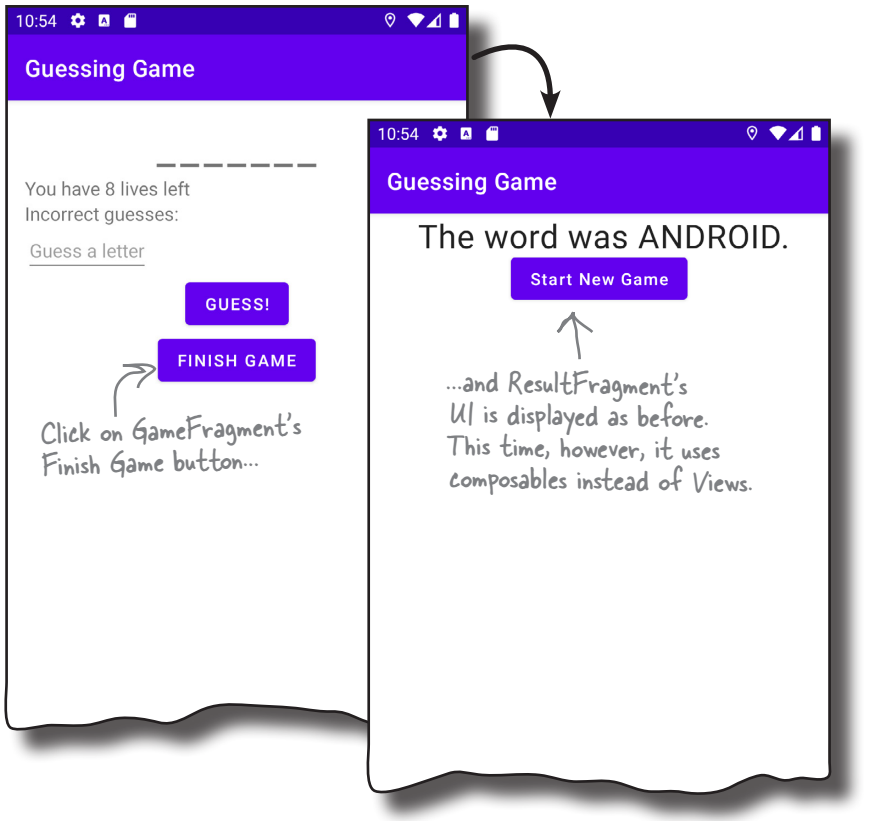



Let's take the app for a test drive.

# Test Drive

**ResultFragment**
GameFragment

When we run the app and click on `GameFragment`'s Finish Game button, it navigates to `ResultFragment` as before. This time, however, the UI is just made up of composables.



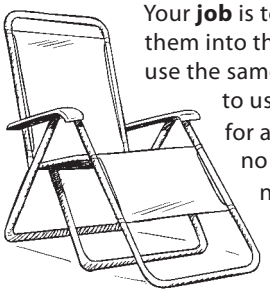

That's everything we need to make `ResultFragment` use a Compose UI. Before we get to work on `GameFragment`, have a go at the exercise on the next page.

## there are no Dumb Questions

**Q:** **You said that a `ComposeView` is a type of `View` you can add composables to. Is there a type of composable you can add `Views` to?**

**A:** Great question! Yes, you can use an `AndroidView`, which is a type of composable that returns a `View`. It's useful if you have a Compose UI, and you want to add a component to it that's only available as a `View`, and not as a composable.

# Pool Puzzle



Your **job** is to take code snippets from the pool and place them into the blank lines in the code below. You may **not** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to write the code for a fragment named `MusicFragment` that has no layout file and uses a composable function named `MusicFragmentContent` for its UI. A theme named `MusicTheme` needs to be applied to the UI, including any surfaces.

```
class MusicFragment : Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        return ..........
            ..........
                ..........
                    ..........
                        MusicFragmentContent()
                    }
                }
            }
        }
    }
}
```

**Surfaces**

**{ {**

**Surface**

**ComposeView**

**setContent**

**ComposableView**

**.apply**

**makeItSo**

**this**

**View { {**

**requireContext()**

**( )**

**MaterialTheme**

**MusicTheme**

**Note: each thing from the pool can only be used once!**

Answers on page 857.

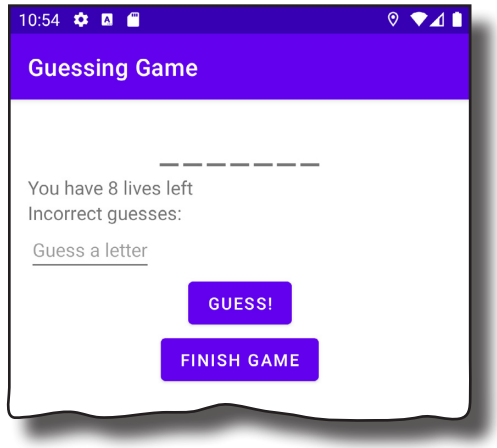
**ResultFragment**
**GameFragment**

# Next we'll make GameFragment use composables too

Now that we've made `ResultFragment` use composables instead of views, we can do a similar thing for `GameFragment`.

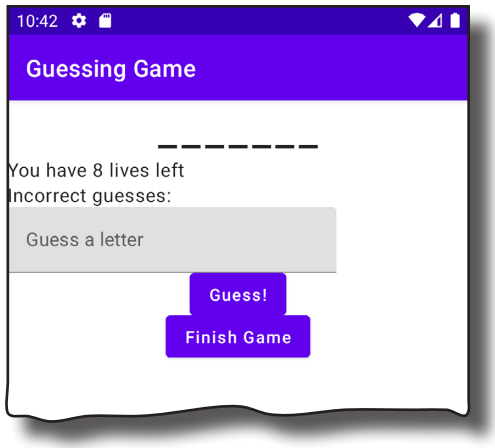
As you may recall, `GameFragment`'s layout includes `TextViews`, `Buttons`, and an `EditText` to let the user play the guessing game. Here's a reminder of what it looks like:

GameFragment currently uses all these Views for its UI.




Over the next few pages, we're going to replace these views with composables. Here's what the new UI will look like when we've finished:

We'll replace the Views with these composables.

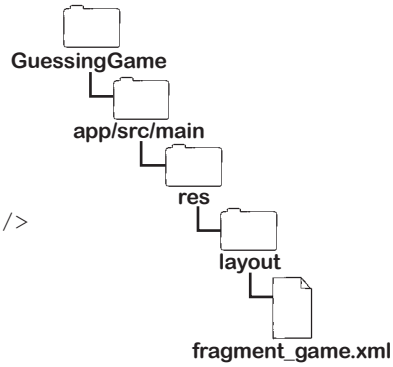



As before, we'll begin by adding new composables to `GameFragment`'s UI, which means we need to add a `ComposeView` to its layout. We'll show you the code for this on the next page.

# We'll add a ComposeView to fragment_game.xml

Here's the code to add a `ComposeView` to `GameFragment`'s layout; update the file *fragment_game.xml* to include the changes below (in bold):

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

    <data>
        <variable
            name="gameViewModel"
            type="com.hfad.guessinggame.GameViewModel" />
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:padding="16dp">

        ...

        <androidx.compose.ui.platform.ComposeView
            android:id="@+id/compose_view"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"/>
    </LinearLayout>
</layout>
```




Add this ComposeView to the layout file at the end of the linear layout. We'll use it to add composables to GameFragment's UI.

Now that we've updated the layout file, let's start adding composables to it.

**ResultFragment**
**GameFragment**

# Add a composable function for GameFragment's content

Just as we did with *ResultFragment.kt*, we'll add a new composable function to *GameFragment.kt* that we'll use for the fragment's UI. We'll name the function `GameFragmentContent`, and call it from `setContent()` so that any composables we add to it will run when `GameFragment` gets displayed.

Here's what the new code looks like; we'll add it to *GameFragment.kt* a few pages ahead:

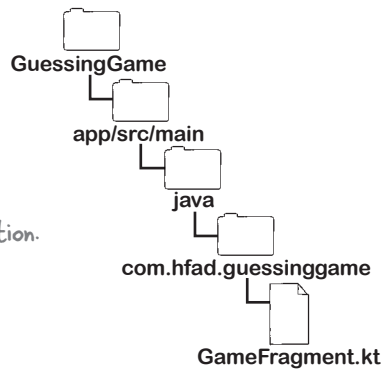
```
...
import androidx.compose.material.MaterialTheme
import androidx.compose.material.Surface
import androidx.compose.runtime.Composable

class GameFragment : Fragment() {
    ...
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentGameBinding.inflate(inflater, container, false).apply {
            composeView.setContent {
                MaterialTheme {
                    Surface {
                        GameFragmentContent()
                    }
                }
            }
        }
        ...
    }
    ...
}

@Composable
fun GameFragmentContent() {
}
```

Import these classes.

This applies composables to the UI.

This adds GameFragmentContent to the layout's ComposeView, and applies the default Material theme.

This is the GameFragmentContent composable function.



# Reproduce the Finish Game button

Just as we did with `ResultFragment`, we'll add composables to the `GameFragmentContent` composable function so that they're displayed in `GameFragment`'s UI. We'll begin by reproducing the Finish Game button.

The Finish Game button is defined using the following code in the fragment's layout:

```
<Button
    android:id="@+id/finish_game_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:text="Finish Game"
    android:onClick="@{() -> gameViewModel.finishGame()}" />
```

This code defines GameFragment's Finish Game button.

Clicking the button calls this method.

As you can see, when the button is clicked, it calls `GameViewModel`'s `finishGame()` method.

We can reproduce this in Compose by creating a new composable function named `FinishGameButton` that we'll run from `GameFragmentContent`. Here's what the new code looks like; we'll add it to *GameFragment.kt* a few pages ahead:

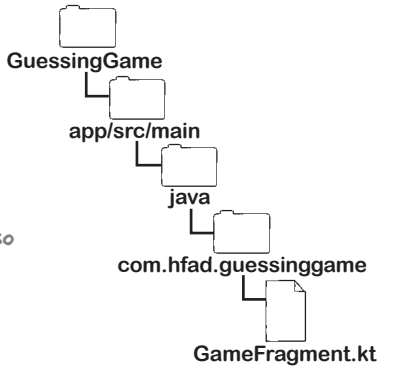
```
@Composable
fun FinishGameButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Finish Game")
    }
}

@Composable
fun GameFragmentContent(viewModel: GameViewModel) {
    Column(modifier = Modifier.fillMaxWidth(),
        horizontalAlignment = Alignment.CenterHorizontally) {
        FinishGameButton {
            viewModel.finishGame()
        }
    }
}
```

This is the FinishGameButton composable function.

FinishGameButton needs access to a GameViewModel object, so we're adding an argument for it to GameFragmentContent.

We're using a Column to center the button.

This adds FinishGameButton to the UI, and makes it call the view model's finishGame() method when it's clicked.



**ResultFragment**
**GameFragment**

# Reproduce the EditText with a TextField

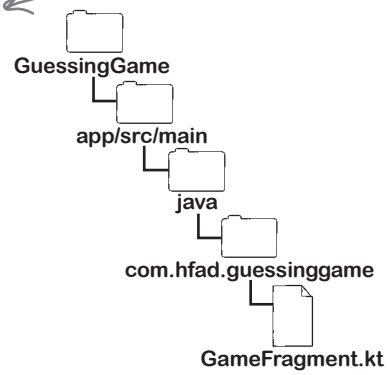
The next view we'll reproduce is the `EditText` that lets the user enter a letter. We'll create a new composable function named `EnterGuess` that uses a `TextField`, and accepts two arguments: a `String` for the user's guess, and a lambda that specifies what should happen when the value changes.

We'll run the `EnterGuess` function from `GameFragmentContent` so that it's added to the fragment's UI. We'll also add a `MutableState` object named `guess` to `GameFragmentContent`, which we'll use to manage the `TextField`'s state.

Here's what the new code looks like; we'll add it to *GameFragment.kt* a couple pages ahead:

The EnterGuess function accepts a String for its value, and a lambda that specifies what should happen when the user updates it.

```
@Composable
fun EnterGuess(guess: String, changed: (String) -> Unit) {
    TextField(
        value = guess,
        label = { Text("Guess a letter") },
        onValueChange = changed
    )
}

@Composable
fun GameFragmentContent(viewModel: GameViewModel) {
    val guess = remember { mutableStateOf("") }

    Column(modifier = Modifier.fillMaxWidth()) {
        EnterGuess(guess.value) { guess.value = it }

        Column(modifier = Modifier.fillMaxWidth(),
               horizontalAlignment = Alignment.CenterHorizontally) {
            FinishGameButton {
                viewModel.finishGame()
            }
        }
    }
}
```




The EnterGuess composable function will use this for its TextField's value.

This adds EnterGuess to the UI.

We're using an extra Column for EnterGuess so that it's not centered.

# Reproduce the Guess button

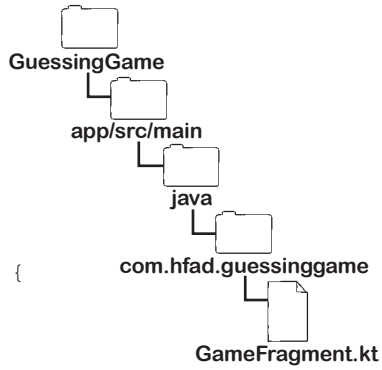
Now that we've added a `TextField` to the UI that lets the user enter a letter, we'll add a `Button` composable that passes the letter to the view model's `makeGuess()` method.

We'll add the `Button` using a new composable function named `GuessButton` that we'll run from `GameFragmentContent`. Here's what the new code looks like; we'll add it to *GameFragment.kt* on the next page:

This is the current Guess button. We'll reproduce it with a Button composable.

The GuessButton composable function has a lambda argument that specifies the Button's behavior.

```
@Composable
fun GuessButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Guess!")
    }
}

@Composable
fun GameFragmentContent(viewModel: GameViewModel) {
    val guess = remember { mutableStateOf("") }

    Column(modifier = Modifier.fillMaxWidth()) {
        EnterGuess(guess.value) { guess.value = it }

        Column(modifier = Modifier.fillMaxWidth(),
               horizontalAlignment = Alignment.CenterHorizontally) {
            GuessButton {
                viewModel.makeGuess(guess.value.uppercase())
                guess.value = ""
            }
            FinishGameButton {
                viewModel.finishGame()
            }
        }
    }
}
```




We're adding GuessButton to the second Column composable so it's centered horizontally.

When clicked, the Button will call the makeGuess() method, and set the value of the guess variable to "". Doing so clears the text in the EnterGuess TextField.

We've now reproduced three of `GameFragment`'s views with composables. Before we tackle the remaining ones, let's update *GameFragment.kt* and take the app for a test drive.

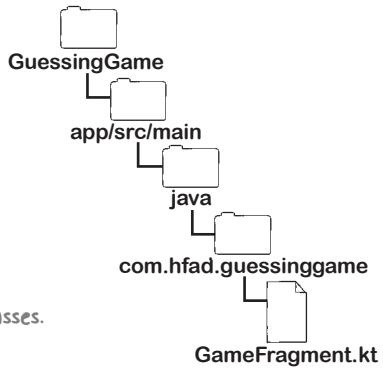
ResultFragment
**GameFragment**

# The updated code for GameFragment.kt

Here's the code so far for *GameFragment.kt*; update the file to include the changes below (in bold):

```
...
import androidx.compose.runtime.Composable
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.material.*
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier

class GameFragment : Fragment() {
    ...
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentGameBinding.inflate(inflater, container, false).apply {
            composeView.setContent {
                MaterialTheme {
                    Surface {
                        GameFragmentContent(viewModel)
                    }
                }
            }
        }
        ...
        return view
    }
    ...
}

@Composable
fun FinishGameButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Finish Game")
    }
}
```

Import these classes.




Add these lines.

This passes the view model to GameFragmentContent.

This adds the composables to the UI.

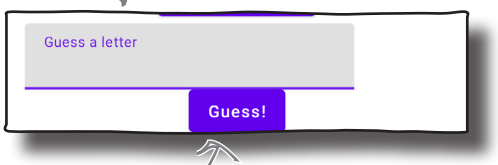
Add the FinishGameButton composable function.

The code continues on the next page.

# GameFragment.kt continued

```
@Composable
fun EnterGuess(guess: String, changed: (String) -> Unit) {
    TextField(
        value = guess,
        label = { Text("Guess a letter") },
        onValueChange = changed
    )
}

@Composable
fun GuessButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Guess!")
    }
}

@Composable
fun GameFragmentContent(viewModel: GameViewModel) {
    val guess = remember { mutableStateOf("") }

    Column(modifier = Modifier.fillMaxWidth()) {
        EnterGuess(guess.value) { guess.value = it }

        Column(modifier = Modifier.fillMaxWidth(),
            horizontalAlignment = Alignment.CenterHorizontally) {
            GuessButton {
                viewModel.makeGuess(guess.value.uppercase())
                guess.value = ""
            }
            FinishGameButton {
                viewModel.finishGame()
            }
        }
    }
}
```

Add the EnterGuess composable function.




Add the GuessButton composable function too.

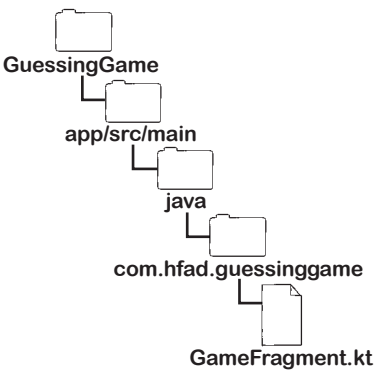
Add GameFragmentContent, which we'll use for the fragment's content.

Define the Guess variable.

Add EnterGuess to a Column.

Use another Column to center GuessButton and FinishGameButton.

Run GuessButton and FinishGameButton.



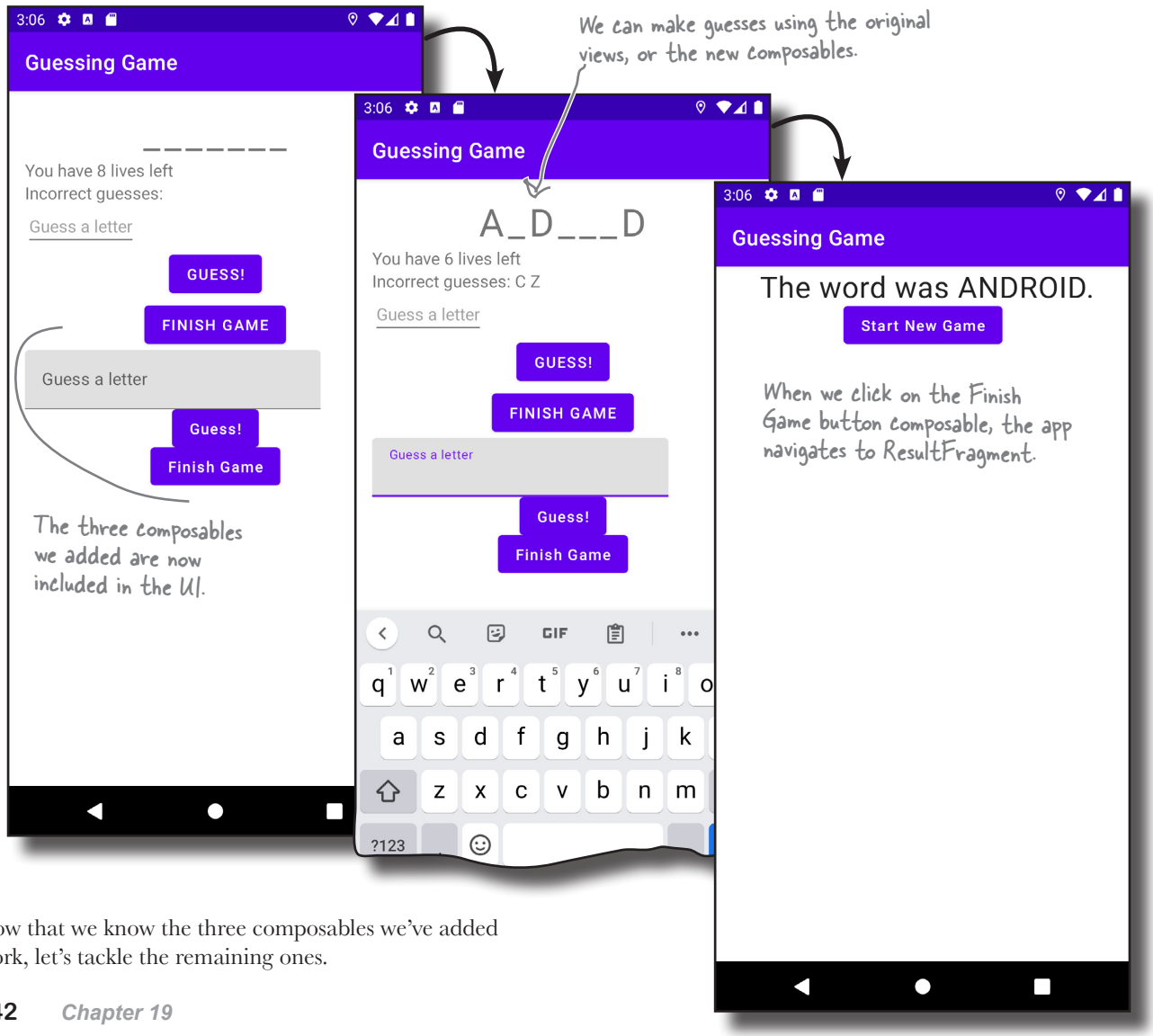

Let's take the app for a test drive.

# Test Drive

**ResultFragment**
**GameFragment**

When we run the app, `GameFragment` is displayed. It includes all its original views, along with three extra composables that let us make guesses and finish the game.

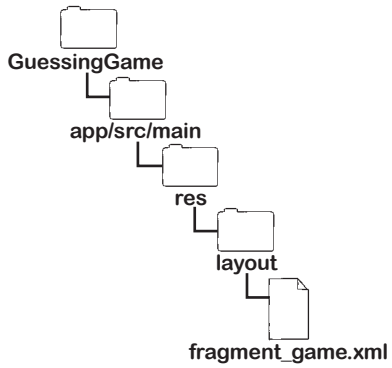
When we use the `EnterGuess` and `GuessButton` composables to guess which letters the secret word contains, the app registers each guess. If we make a correct guess, the letter is added to the secret word display, and if we make a wrong guess, the number of lives left is updated and the letter is added to the list of incorrect guesses.




Now that we know the three composables we've added work, let's tackle the remaining ones.

# We'll display the incorrect guesses in a Text composable

The next view we'll reproduce is a `TextView` that uses live data to display the user's incorrect guesses. Each time the user makes a wrong guess, it gets added to the `incorrectGuesses` property in the view model, and the `TextView` responds by updating the text it displays.




Here's a reminder of the `TextView`'s code:

```
<TextView
    android:id="@+id/incorrect_guesses"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="16sp"
    android:text="@{@string/incorrect_guesses(gameViewModel.incorrectGuesses)}" />
```

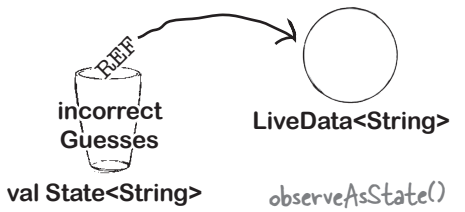
The text view's text is derived by adding the value of GameViewModel's incorrectGuesses property to the incorrect_guesses String resource. As the incorrectGuesses property uses live data, the text stays up-to-date.

We can replace the `TextView` with a `Text` composable so that it displays the same text, but how do we make sure the text gets updated when the `incorrectGuesses` property value changes?

## Respond to LiveData using observeAsState()

As you learned in Chapter 18, composables are recomposed when any `State` or `MutableState` objects they depend on get new values. This *doesn't* happen, however, for `LiveData` objects—like the view model's `incorrectGuesses` property. If you try to use a `LivaData` object's value with a composable, it won't get redrawn when its value changes; it will just keep using the object's initial value, so it won't stay up-to-date.

If you want to make a composable respond to `LiveData` updates, you can do so using the **`observeAsState()`** function. This function returns a `State` version of a `LiveData` object so that any composables that depend on it are recomposed when its value changes.




observeAsState() lets composables respond to LiveData updates as though they're State objects.

You use the `observeAsState()` function with code like this:

```
val incorrectGuesses = viewModel.incorrectGuesses.observeAsState()
```

This defines a variable (whose type is `State`) that observes the view model's `incorrectGuesses LiveData` property. It means that when its value changes, composables can respond.

Let's put this into practice by reproducing `GameFragment`'s incorrect guesses `TextView` in Compose.

**ResultFragment**
**GameFragment**

# Create an IncorrectGuessesText composable function

To reproduce the incorrect guesses `TextView`, we'll define a new composable function named `IncorrectGuessesText`. The function will accept a `GameViewModel` argument, observe its `incorrectGuesses` property, and use a `Text` composable to display its value in the UI. Each time the property's value gets updated, the `Text` will be recomposed and display the updated text.

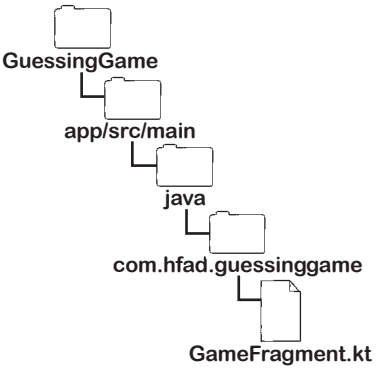
Here's what the `IncorrectGuessesText` function looks like:

The function accepts a GameViewModel argument.

```
@Composable
fun IncorrectGuessesText(viewModel: GameViewModel) {
    val incorrectGuesses = viewModel.incorrectGuesses.observeAsState()
    incorrectGuesses.value?.let {
        Text(stringResource(R.string.incorrect_guesses, it))
    }
}
```

stringResource() is a function that lets you use String resources with composables.

As you can see, the `Text` composable in the above code uses a function named `stringResource()` to set its text. This function lets you use `String` resources with composables, and pass arguments to them.



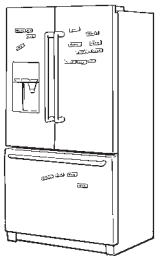

## Run IncorrectGuessesText from GameFragmentContent

Just like the other composable functions we've created, we'll add `IncorrectGuessesText` to `GameFragment`'s UI by running it from the `GameFragmentContent` composable function. The new code looks like this:

```
@Composable
fun GameFragmentContent(viewModel: GameViewModel) {
    ...
    Column(modifier = Modifier.fillMaxWidth()) {
        IncorrectGuessesText(viewModel)
        ...
    }
}
```

We'll pass the view model to the new function.

We'll update *GameFragment.kt* a few pages ahead. First, see if you can piece together the code for the remaining two composables by having a go at the following exercise.



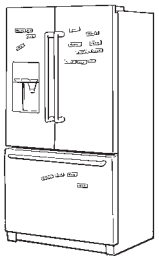
# Compose Magnets

Someone used fridge magnets to create two new composable functions (named `SecretWordDisplay` and `LivesLeftText`) that reproduce `GameFragment`'s two remaining views. Unfortunately some of the magnets fell off when someone unleashed the kitchen kraken. Can you piece the code back together again?

The `SecretWordDisplay` function needs to display the `secretWordDisplay LiveData<String>` property from `GameViewModel`. The `LivesLeftText` function needs to display the `lives_left String` resource, and pass it `GameViewModel`'s `livesLeft LiveData<Int>` property. Both functions need to be able to respond to live data updates.

```
@Composable
fun SecretWordDisplay(viewModel: GameViewModel) {

    val display = viewModel.secretWordDisplay_____

    display_____ {

        Text(_____)
    }
}

@Composable
fun LivesLeftText(viewModel: GameViewModel) {

    val livesLeft = viewModel.livesLeft_____

    livesLeft_____ {

        Text(_____)

    }
}
```

You won't need to use all the magnets.



# Compose Magnets Solution

Someone used fridge magnets to create two new composable functions (named `SecretWordDisplay` and `LivesLeftText`) that reproduce `GameFragment`'s two remaining views. Unfortunately some of the magnets fell off when someone unleashed the kitchen kraken. Can you piece the code back together again?

The `SecretWordDisplay` function needs to display the `secretWordDisplay LiveData<String>` property from `GameViewModel`. The `LivesLeftText` function needs to display the `lives_left String` resource, and pass it `GameViewModel`'s `livesLeft LiveData<Int>` property. Both functions need to be able to respond to live data updates.

```
@Composable
fun SecretWordDisplay(viewModel: GameViewModel) {

    val display = viewModel.secretWordDisplay.observeAsState()

    display.value?.let {

        Text(it)

    }
}
```

Observe the view model's secretWordDisplay LiveData property as though it's a State object.

Display the value of the display variable so long as it's not null.

```
@Composable
fun LivesLeftText(viewModel: GameViewModel) {

    val livesLeft = viewModel.livesLeft.observeAsState()

    livesLeft.value?.let {

        Text(stringResource(R.string.lives_left, it))

    }
}
```

Observe the livesLeft LiveData property as though it's a State object.

Use the lives_left String resource for the text, passing it the value of the livesLeft variable (so long as it isn't null).

You didn't need to use these magnets.

`toState()` `toState()`

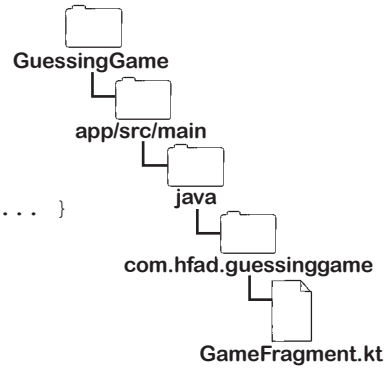
# The updated code for GameFragment.kt

You now know how to reproduce all of `GameFragment`'s views with composables, so let's add them to *GameFragment.kt*. Update the file to include the changes below (in bold):

```
...
import androidx.compose.runtime.livedata.observeAsState
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.unit.em
import androidx.compose.ui.unit.sp
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Arrangement

class GameFragment : Fragment() {
    ...
}

@Composable
fun FinishGameButton(clicked: () -> Unit) { ... }

@Composable
fun EnterGuess(guess: String, changed: (String) -> Unit) { ... }

@Composable
fun GuessButton(clicked: () -> Unit) { ... }

@Composable
fun IncorrectGuessesText(viewModel: GameViewModel) {
    val incorrectGuesses = viewModel.incorrectGuesses.observeAsState()
    incorrectGuesses.value?.let {
        Text(stringResource(R.string.incorrect_guesses, it))
    }
}

@Composable
fun LivesLeftText(viewModel: GameViewModel) {
    val livesLeft = viewModel.livesLeft.observeAsState()
    livesLeft.value?.let {
        Text(stringResource(R.string.lives_left, it))
    }
}
```

Import these classes.

Add these composable functions.




The code continues on the next page.

ResultFragment
**GameFragment**

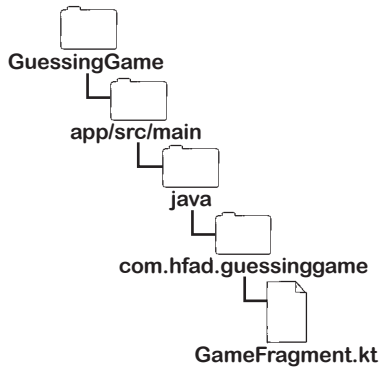
# GameFragment.kt continued

Add this function.

```
@Composable
fun SecretWordDisplay(viewModel: GameViewModel) {
    val display = viewModel.secretWordDisplay.observeAsState()
    display.value?.let {
        Text(text = it,
             letterSpacing = 0.1.em,
             fontSize = 36.sp)
    }
}

@Composable
fun GameFragmentContent(viewModel: GameViewModel) {
    val guess = remember { mutableStateOf("") }

    Column(modifier = Modifier.fillMaxWidth()) {
        Row(modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.Center) {
            SecretWordDisplay(viewModel)
        }
        LivesLeftText(viewModel)
        IncorrectGuessesText(viewModel)
        EnterGuess(guess.value) { guess.value = it }

        Column(modifier = Modifier.fillMaxWidth(),
            horizontalAlignment = Alignment.CenterHorizontally) {
            GuessButton {
                viewModel.makeGuess(guess.value.uppercase())
                guess.value = ""
            }
            FinishGameButton {
                viewModel.finishGame()
            }
        }
    }
}
```

Center SecretWordDisplay in a Row.

Add LivesLeftText and IncorrectGuessesText to the UI.

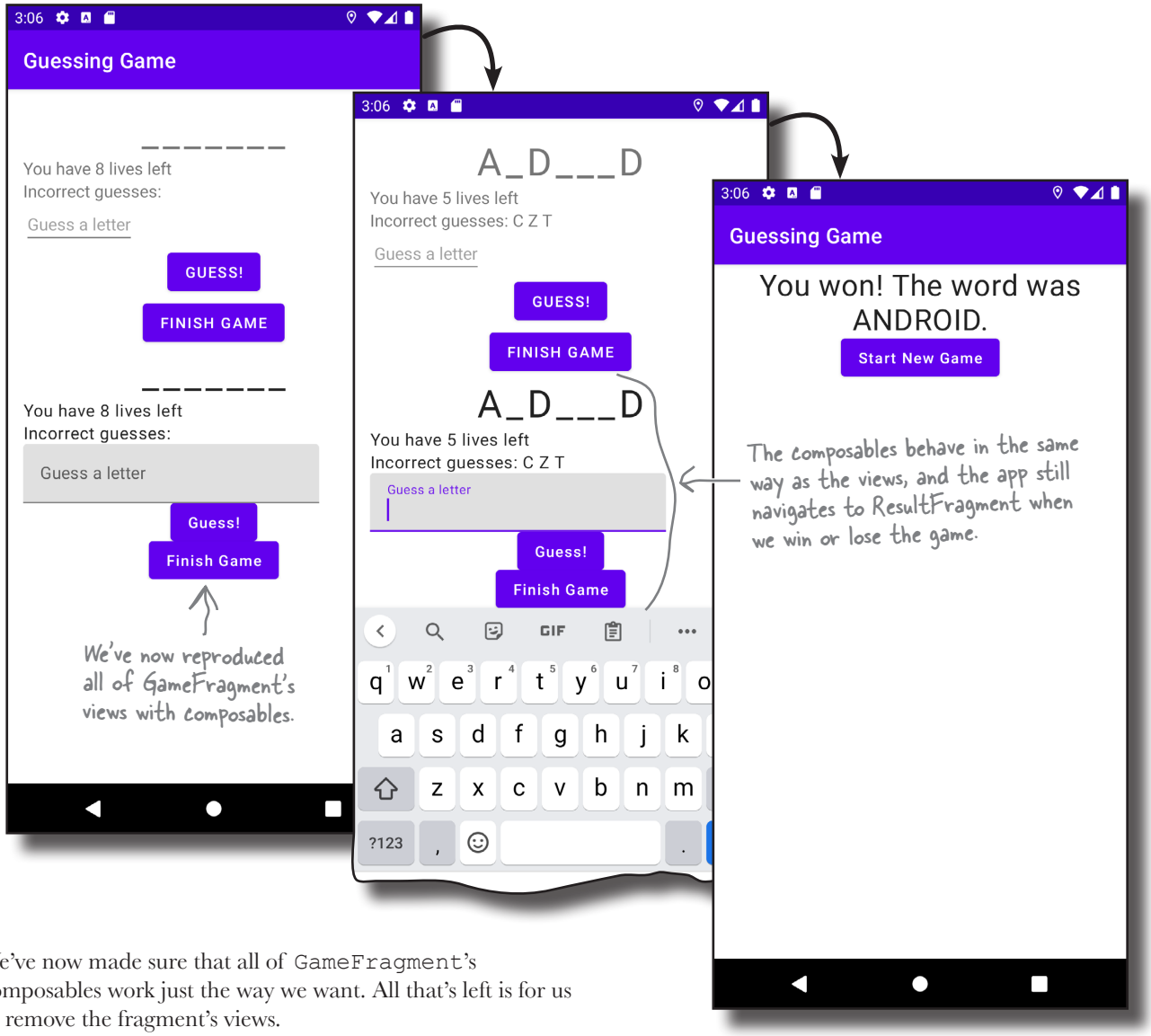



Let's take the app for a test drive.

# Test Drive

When we run the app, `GameFragment` includes all its original views, and their Compose equivalents.

When we try to guess which letters the secret word contains, the text in the `SecretWordDisplay`, `LivesLeftText`, and `IncorrectGuessesText` composables is automatically updated.




We've now made sure that all of `GameFragment`'s composables work just the way we want. All that's left is for us to remove the fragment's views.

**ResultFragment**
**GameFragment**

# Remove views from GameFragment.kt

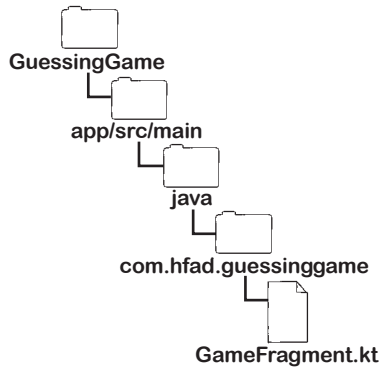
Just as we did with `ResultFragment`, we'll remove `GameFragment`'s views by deleting its layout file. Before we can do this, however, we need to remove any references to the views from *GameFragment.kt*. We'll also stop using data binding, as this is no longer needed.

Here's the full code for the file; update *GameFragment.kt* to include all the changes shown below (in bold):

```
package com.hfad.guessinggame

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.guessinggame.databinding.FragmentGameBinding
import androidx.navigation.findNavController
import androidx.lifecycle.ViewModelProvider
import androidx.lifecycle.Observer
import androidx.compose.runtime.Composable
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.material.*
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.runtime.livedata.observeAsState
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.unit.em
import androidx.compose.ui.unit.sp
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.ui.platform.ComposeView
```

Remove this line.

Import this class.




The code continues on the next page.

# GameFragment.kt continued

```
class GameFragment : Fragment() {
    private var _binding: FragmentGameBinding? = null
    private val binding get() = _binding!!
    lateinit var viewModel: GameViewModel

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        viewModel = ViewModelProvider(this).get(GameViewModel::class.java)

        viewModel.gameOver.observe(viewLifecycleOwner, Observer { newValue ->
            if (newValue) {
                val action = GameFragmentDirections
                    .actionGameFragmentToResultFragment(viewModel.wonLostMessage())
                view?.findNavController()?.navigate(action)
            }
        })

        _binding = FragmentGameBinding.inflate(inflater, container, false).apply {
        return ComposeView(requireContext()).apply {
            composeView.setContent {
                MaterialTheme {
                    Surface {
                        GameFragmentContent(viewModel)
                    }
                }
            }
        }

        val view = binding.root
        viewModel = ViewModelProvider(this).get(GameViewModel::class.java)

        binding.gameViewModel = viewModel
        binding.lifecycleOwner = viewLifecycleOwner
```

Remove these lines.

Move the code to get the view model.

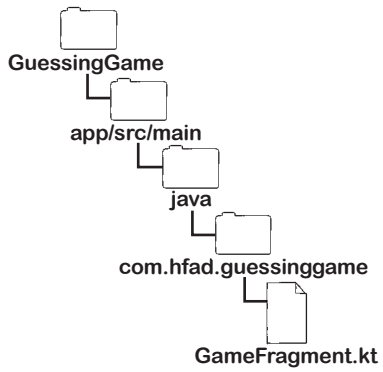
Move the code to navigate to ResultFragment when the game has finished.

The extra ?'s are because "view" refers to the fragment's getView() method, which returns its root view.

Delete this line.

Remove the composeView reference.

Delete these lines.




The code continues on the next page.

ResultFragment
**GameFragment**

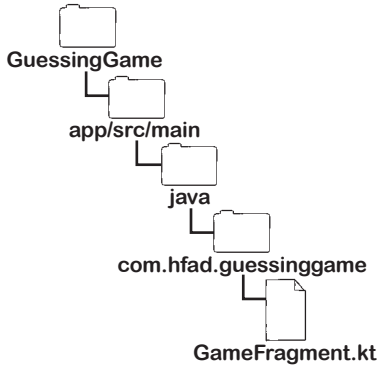
# GameFragment.kt continued

```
        viewModel.gameOver.observe(viewLifecycleOwner, Observer { newValue ->
            if (newValue) {
                val action = GameFragmentDirections
                    .actionGameFragmentToResultFragment(viewModel.wonLostMessage())
                view.findNavController().navigate(action)
            }
        })

        binding.guessButton.setOnClickListener() {
            viewModel.makeGuess(binding.guess.text.toString().uppercase())
            binding.guess.text = null
        }
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}

@Composable
fun FinishGameButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Finish Game")
    }
}

@Composable
fun EnterGuess(guess: String, changed: (String) -> Unit) {
    TextField(
        value = guess,
        label = { Text("Guess a letter") },
        onValueChange = changed
    )
}
```

Delete all of these lines so that we're no longer referring to any views.

Don't delete this.

Delete this method since it's no longer needed.

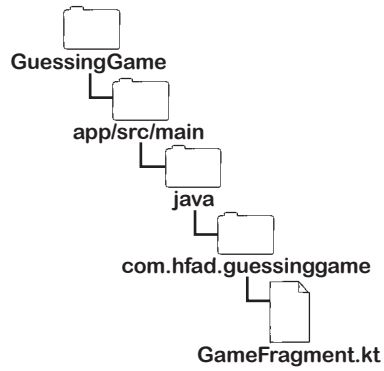



The code continues on the next page.

# GameFragment.kt continued

```
@Composable
fun GuessButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Guess!")
    }
}

@Composable
fun IncorrectGuessesText(viewModel: GameViewModel) {
    val incorrectGuesses = viewModel.incorrectGuesses.observeAsState()
    incorrectGuesses.value?.let {
        Text(stringResource(R.string.incorrect_guesses, it))
    }
}

@Composable
fun LivesLeftText(viewModel: GameViewModel) {
    val livesLeft = viewModel.livesLeft.observeAsState()
    livesLeft.value?.let {
        Text(stringResource(R.string.lives_left, it))
    }
}

@Composable
fun SecretWordDisplay(viewModel: GameViewModel) {
    val display = viewModel.secretWordDisplay.observeAsState()
    display.value?.let {
        Text(text = it,
             letterSpacing = 0.1.em,
             fontSize = 36.sp)
    }
}
```

You don't need to update any of the code on this page.

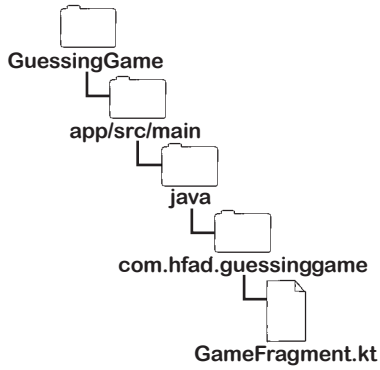



The code continues on the next page.

**ResultFragment**
**GameFragment**

# GameFragment.kt continued

```
@Composable
fun GameFragmentContent(viewModel: GameViewModel) {
    val guess = remember { mutableStateOf("") }

    Column(modifier = Modifier.fillMaxWidth()) {
        Row(modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.Center) {
            SecretWordDisplay(viewModel)
        }
        LivesLeftText(viewModel)
        IncorrectGuessesText(viewModel)
        EnterGuess(guess.value) { guess.value = it }

        Column(modifier = Modifier.fillMaxWidth(),
            horizontalAlignment = Alignment.CenterHorizontally) {
            GuessButton {
                viewModel.makeGuess(guess.value.uppercase())
                guess.value = ""
            }
            FinishGameButton {
                viewModel.finishGame()
            }
        }
    }
}
```




You don't need to update any of the code on this page either.

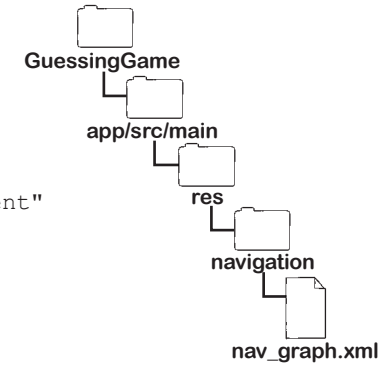
Those are all the changes we need to make to *GameFragment.kt* so that it doesn't inflate its layout or reference any of its views.

As the fragment no longer needs to have a layout file, we can go ahead and delete it after we've removed any references to it from the navigation graph. We'll do this on the next page.

# Delete fragment_game.xml

The navigation graph includes a reference to the layout file *fragment_game.xml*. We need to remove this before we can remove the layout file.

Open *nav_graph.xml* in the *app/src/main/res/navigation* folder, and remove the line that references `"@layout/fragment_game"` from the `GameFragment` section like this:




```
<fragment
    android:id="@+id/gameFragment"
    android:name="com.hfad.guessinggame.GameFragment"
    android:label="fragment_game"
    tools:layout="@layout/fragment_game" >
    <action ... />
</fragment>
```

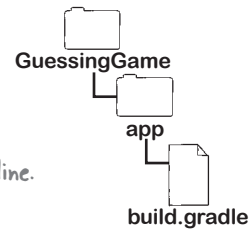
Remove this reference.

Then right-click on *fragment_game.xml* in the explorer, choose Refactor, and select the Safe Delete option. When you click on the OK button and choose the option to do the refactor, the file is deleted.

## We can also switch off data binding

One final change we can make to the Guessing Game app is to disable data binding. As you may recall, we originally enabled data binding so that views in *fragment_game.xml* and *fragment_result.xml* could interact with each fragment's view model. Now that we've deleted the layout files, data binding is no longer required.

To switch off data binding, open the file *GuessingGame/app/build.gradle*, and remove the data binding line from the `buildFeatures` section like this:




```
...
    buildFeatures {
        dataBinding true
        ...
    }
...
```

Delete this line.

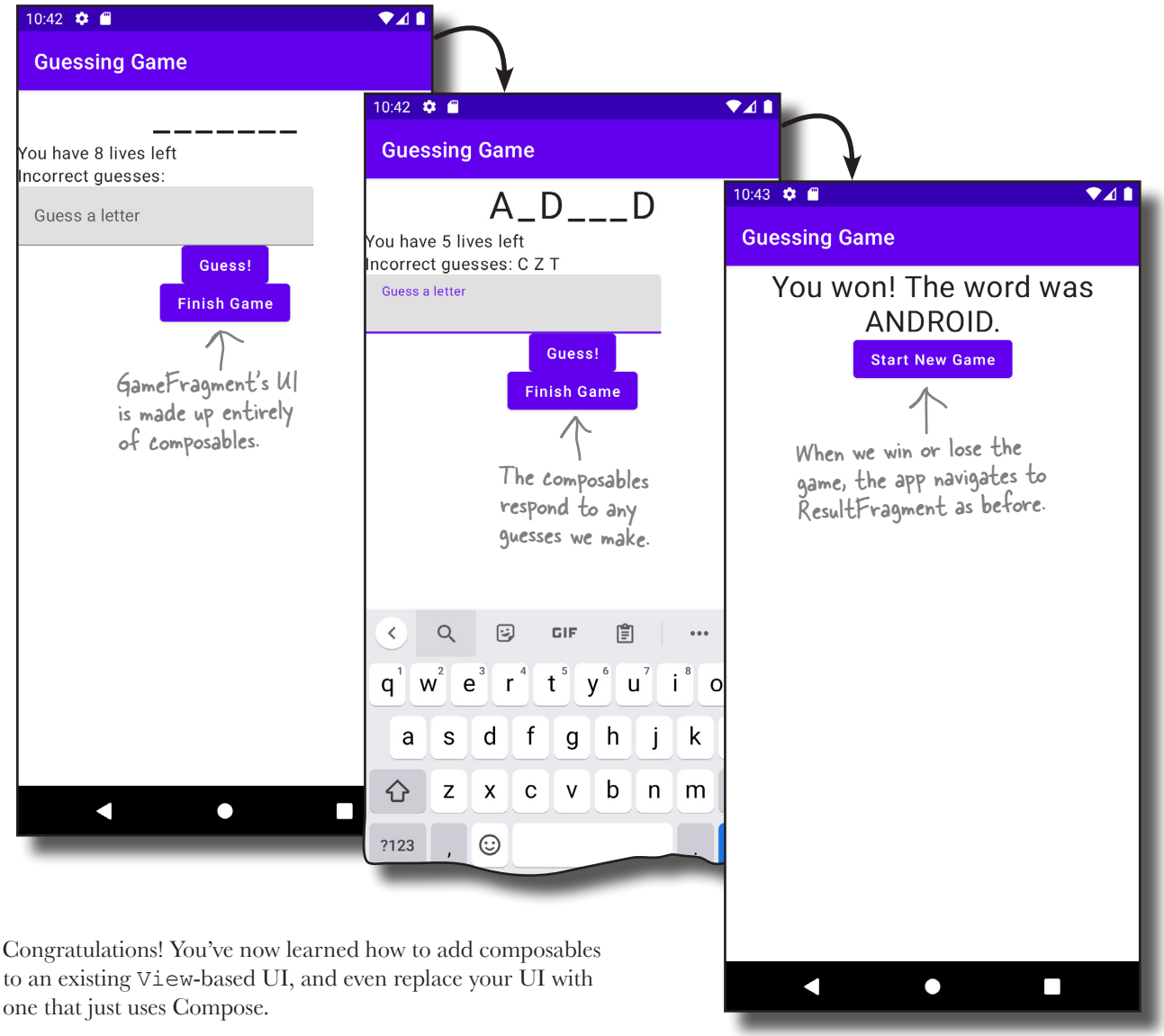
When you've done this, click on the Sync Now option to sync the change with the rest of the project.

Let's take the app for a final test drive.

# Test Drive

**ResultFragment**
**GameFragment**

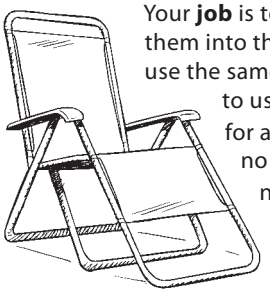
When we run the app, `GameFragment` is displayed. This time, its UI is just made up of composables. They work exactly how we want them to.




Congratulations! You've now learned how to add composables to an existing `View`-based UI, and even replace your UI with one that just uses Compose.

We think Compose has a bright future ahead of it, and you can find out more about it in the appendix. In the meantime, why not try adding composables to some of the other apps you've built in this book?

# Pool Puzzle Solution



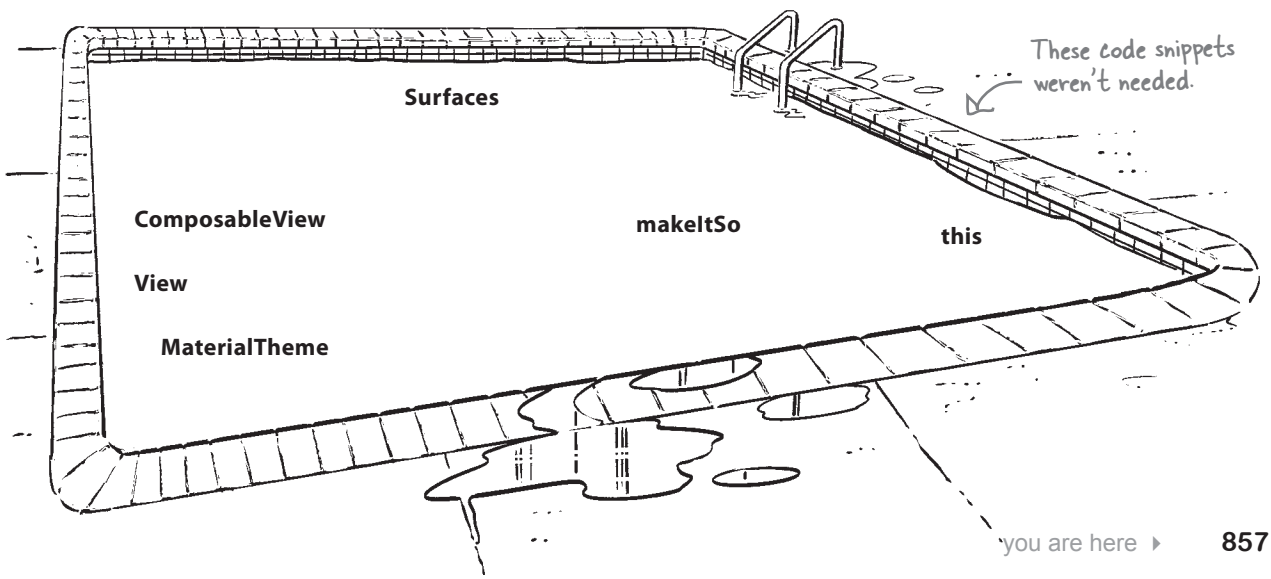
Your **job** is to take code snippets from the pool and place them into the blank lines in the code below. You may **not** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to write the code for a fragment named `MusicFragment` that has no layout file and uses a composable function named `MusicFragmentContent` for its UI. A theme named `MusicTheme` needs to be applied to the UI, including any surfaces.

```
class MusicFragment : Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        return ComposeView ( requireContext() ) .apply {
            setContent {
                MusicTheme {
                    Surface {
                        MusicFragmentContent()
                    }
                }
            }
        }
    }
}
```

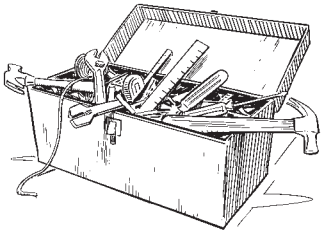
Return a ComposeView.

Use setContent to add composables to the UI.

Apply MusicTheme to the UI, including surfaces.



# Your Android Toolbox



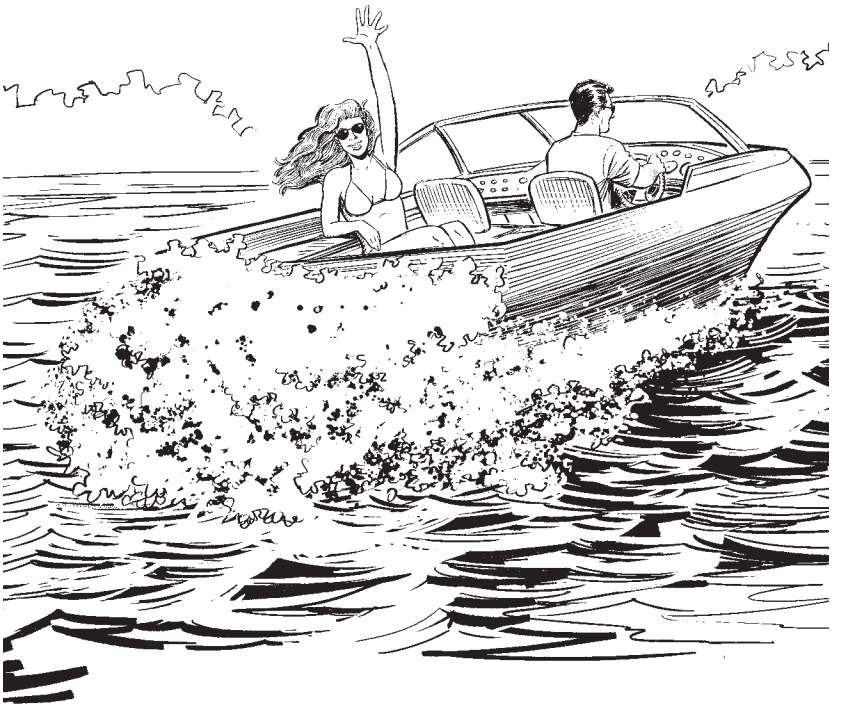
**You've got Chapter 19 under your belt and now you've added integrating Jetpack Compose to your toolbox.**

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

## Bullet Points

- Apps can include a mix of `View` and Compose UIs.
- Compose can be used in existing apps so long as the minimum SDK is 21, and the language is Kotlin.
- Before you can use Compose in an existing `View`-based UI, you need to add Compose options and libraries to the *build.gradle* files.
- A `ComposeView` is a `View` you can add composables to. You can think of it as a placeholder for any composables you want to add to your layout file.
- Composables can interact with a view model's properties and methods.
- Use `observeAsState()` to make composables respond to `LiveData` updates.
- Once you've replaced an activity or fragment's views with composables, you can remove its layout file.

# Leaving town



## It's been great having you here in Androidville

**We're sad to see you leave,** but there's nothing like taking what you've learned and putting it to good use. There are still a few more gems for you in the back of the book and a handy index to study, and then it's time to take all these new ideas and put them into practice. Bon voyage!

# 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**.

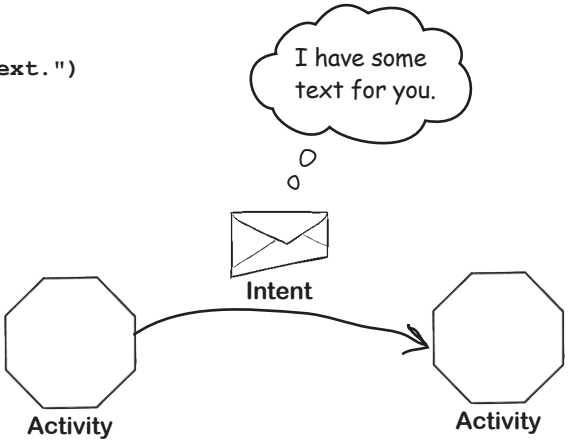
# 1. Sharing data with other apps

If you want to share simple data with another app, you can do so with an **`Intent`**. You can think of an `Intent` as an "intent to do something." It's a type of message that lets you send data to another object—such as an activity—at runtime.

## Sharing data with Android's intent resolver

If you want to pass text to another activity, you can do so using code like this:

```
val sendIntent: Intent = Intent().apply {
    action = Intent.ACTION_SEND
    putExtra(Intent.EXTRA_TEXT, "This is some text.")
    type = "text/plain"
}
startActivity(sendIntent)
```




The code first creates an `Intent` named `sendIntent`. It uses the `type` property to specify the type of data that's being sent (in this case plain text), and uses `putExtra()` to attach the data (here, it's some text).

The `action` property tells Android which types of activity can receive the intent. Here, it's set using:

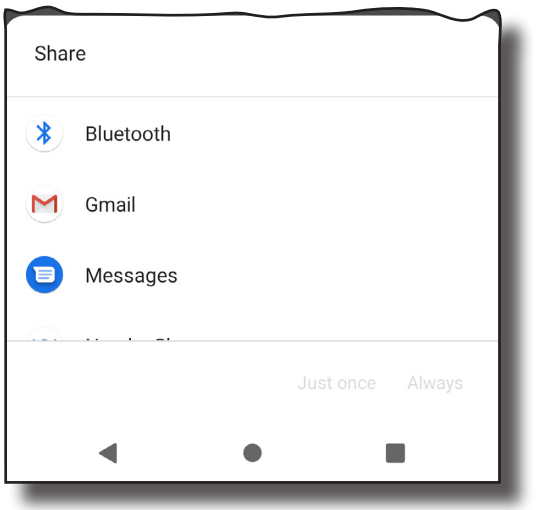
```
action = Intent.ACTION_SEND
```

which means that only activities that can send messages can receive the intent.

The intent is sent using the line:

```
startActivity(sendIntent)
```

Behind the scenes, Android looks for activities across all the device's apps that can accept intents with the specified action and type. If it finds multiple such activities, it displays an **Intent Resolver** screen, which lets the user choose which app they want to share the data with.




This is the Intent Resolver screen. It's showing us a list of apps on our device that can send plain text data.

Android then starts the activity, passing it the data.

## Sharing data with the Android Sharesheet

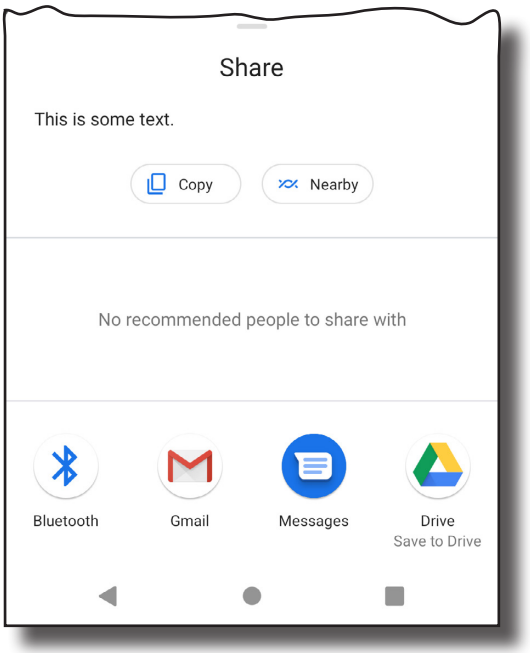
In most cases, you'll want to share data using the Android Sharesheet instead. This lets you specify who you want to share data with and how, and the code looks like this:

```
val sendIntent: Intent = Intent().apply {
    action = Intent.ACTION_SEND
    putExtra(Intent.EXTRA_TEXT, "This is some text.")
    type = "text/plain"
}
val shareIntent = Intent.createChooser(sendIntent, null)
startActivity(shareIntent)
```

The createChooser() method is used to display the Android Sharesheet.

As you can see, the code includes an extra call to the `Intent`'s `createChooser()` method. When this runs, it displays the Android Sharesheet like this:

The Android Sharesheet is more flexible than the Intent Resolver, and gives you more options.




You can find out more about sharing data with other apps—and how to make your app receive data—here:

*https://developer.android.com/training/sharing*

# 2. WorkManager

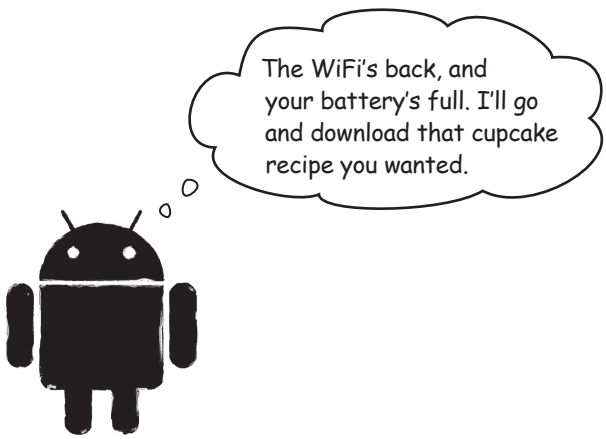
There are times when you'll want your app to process data in the background. This might be because it needs to access storage, for example, or it needs to download a large file.

As you learned in Chapter 14, you can use Kotlin coroutines for tasks that need to be executed immediately. But what if you want to defer the task, or have a long-running task that must continue to run, even if the device restarts?

## Use WorkManager to schedule deferrable tasks

If you want to schedule a task to run in the background, you can use Android's **WorkManager** API. It's part of Android Jetpack, and it's designed for potentially long-running tasks that are guaranteed to run even if the user quits the app or restarts the device.

You can even use WorkManager to run tasks when certain constraints are met, such as when WiFi is available, or chain complex tasks together.




You can find out more about WorkManager—and how to use it—here:

*https://developer.android.com/topic/libraries/architecture/workmanager*

# 3. Dialogs and notifications

In Chapter 9, you learned how to show the user a simple pop-up message using a toast and a snackbar. These are useful for low priority messages that appear in the app, and don't require any user action.

There are times, however, when you need to display a message that prompts the user to make a decision, or that's displayed outside the app's UI. In these situations, you can use **dialogs** and **notifications**.

## Use dialogs to prompt the user for a decision

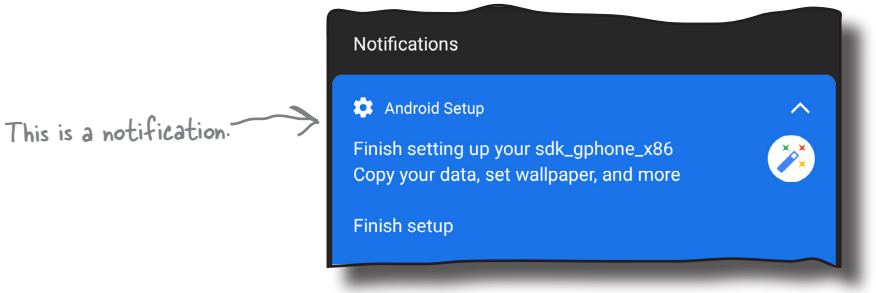
A dialog is a small window that appears in the middle of the screen. They're normally used for situations where the user must make a decision before the app can proceed:

You can find out how to create and use dialogs here:

*https://developer.android.com/guide/topics/ui/dialogs*

## Notifications appear outside the app's UI

If you want to remind the user to do something, or tell them when they've received a message, you can use a notification. Notifications appear on the device status bar and the notification drawer, and may appear on the device lock screen as well:




You can find out how to create and use notifications here:

*https://developer.android.com/guide/topics/ui/notifiers/notifications*

# 4. Automated testing

If you create an app that's intended to be used by thousands or even millions of people, you will quickly lose users if it's flaky or keeps crashing. You can prevent many of these problems, however, using automated testing.

Two popular testing frameworks are JUnit and Espresso. When you create a new Android project, Android Studio usually includes dependencies for these in the app's *build.gradle* file.

Automated testing usually falls into two categories: **unit tests** and **instrumented tests**.

## Unit tests

Unit tests run on the development machine, and they check the individual pieces—or units—of your code. They live in the *app/src/test* folder of your project, and look something like this:

```
package com.hfad.myapp

import org.junit.Test
import org.junit.Assert.*

class ExampleUnitTest {
    @Test
    fun additionIsCorrect() {
        assertEquals(6, 3 + 3)
    }
}
```

## Instrumented tests

Instrumented—or on-device—tests run inside an emulator or physical device, and check the fully assembled app. They live in the *app/src/androidTest* folder of your project.

We'll show you an example of an instrumented test on the next page.

## An instrumented test example

Here's an instrumented test that checks whether a composable displays the correct text:

```
package com.hfad.myapp

import androidx.compose.material.MaterialTheme
import androidx.compose.material.Surface
import androidx.compose.ui.test.junit4.createComposeRule
import androidx.compose.ui.test.onNodeWithText
import androidx.test.ext.junit.runners.AndroidJUnit4
import org.junit.Rule
import org.junit.Test
import org.junit.runner.RunWith

@RunWith(AndroidJUnit4::class)
class HelloTest {
    @get:Rule
    val composeTestRule = createComposeRule()

    @Test
    fun shouldShowHello() {
        composeTestRule.setContent {
            MaterialTheme {
                Surface {
                    Hello("Fred")
                }
            }
        }
        composeTestRule.onNodeWithText("Hello Fred!").assertExists()
    }
}
```

Go here to find out more about using automated testing with Android apps:

*https://developer.android.com/training/testing*

# 5. Supporting different screen sizes

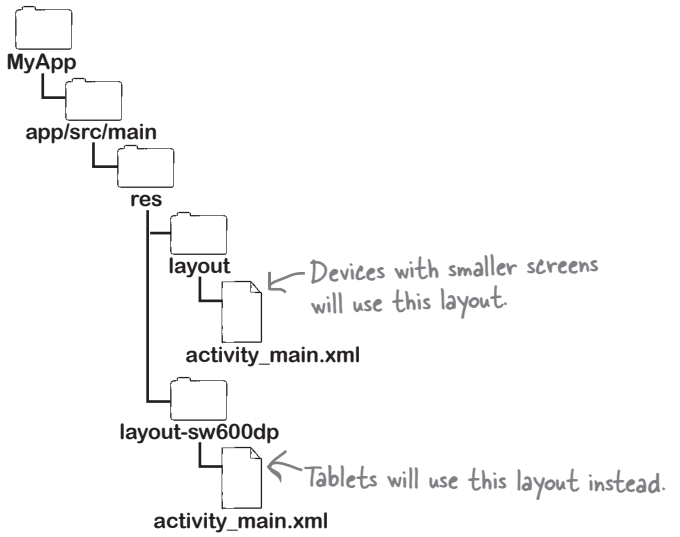
Android apps come in all sorts of shapes and sizes, and you want your apps to look good on all of them. There are several techniques you can use, including constraint layouts (which you learned about in Chapter 4), providing alternate layouts, and sliding pane layouts.

## Provide alternate layouts

Android apps can have multiple versions of the same layout file that cater to different screen specs. This is done by creating multiple layout folders, each one with an appropriate name, and adding a separate layout file to each one.

To have one layout for devices with a width of 600dp or above (such as 7" tablets), and another for smaller devices, add an extra folder to the *app/src/main/res* folder named *layout-sw600dp*. Then put a layout for wider devices to use in the *layout-sw600dp* folder, and one for smaller devices in the *layout* folder.

When the app runs on a phone, it will use the layout in the *layout* folder as normal, and when it runs on a wider device, it will use the one in the *layout-sw600dp* folder instead.




You can find out more about using width qualifiers here:

*https://developer.android.com/training/multiscreen/screensizes#alternative-layouts*

## Use a SlidingPaneLayout

Some layouts include a navigable list of records, where clicking on an item displays its details. On small devices, you might want the details to appear in a separate screen, but on large devices, you might prefer to display the list and details side by side.

To deal with this kind of situation, you can use a `SlidingPaneLayout` to define separate panes for the list and details. The code looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.slidingpanelayout.widget.SlidingPaneLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recycler_view"
        android:layout_width="280dp"
        android:layout_height="match_parent"
        android:layout_gravity="start" />

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/nav_host_fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="300dp"
        android:layout_height="match_parent"
        android:layout_weight="1"
        app:defaultNavHost="true"
        app:navGraph="@navigation/nav_graph" />
</androidx.slidingpanelayout.widget.SlidingPaneLayout>
```

This is the pane for the list (a recycler view).

This is the pane for the details (a FragmentContainerView).

The layout uses the `layout-width` attributes of each pane to figure out whether they'll fit side by side on the device. If the device is wide enough, the layout displays them next to one another, and if it's not, it displays them on separate screens.

You can find out more here:

*https://developer.android.com/reference/androidx/slidingpanelayout/widget/SlidingPaneLayout*

# 6. More Compose features

In Chapters 18 and 19, we introduced you to building UIs with Compose. We think it has a bright future ahead of it, but it's so new that at the time of writing, some of its libraries aren't yet stable.

We didn't want to leave without giving you a glimpse into the future, so here are some of the libraries and features we're most excited about.

## The Compose ViewModel library

In Chapter 19, we passed an existing view model to a composable so it could access the view model's properties and methods. If you use the Compose view model library, this is no longer necessary; the composable can get the view model for itself.

Add the following dependency to the app's *build.gradle* file:

```
implementation "androidx.lifecycle:lifecycle-viewmodel-compose:1.0.0-alpha07"
```

Then add code like this to the composable:

```
val viewModel: ResultViewModel = viewModel(
    factory = ResultViewModelFactory(result)
)
```

This line is only necessary if the view model needs to be generated using a view model factory.

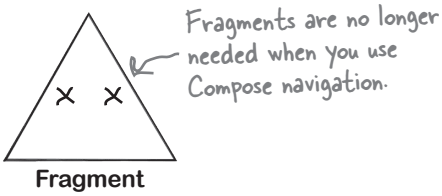
## Compose constraint layout library

In Chapter 18, we showed you how to arrange composables in rows and columns. If you need something a little more flexible, however, there's the Compose constraint layout library. As you might expect, this library lets you position composables using constraints.

You can find out more about this library, and how to use it, here:

*https://developer.android.com/jetpack/compose/layouts/constraintlayout*

## The Compose Navigation component




As you already know, you navigate between screens in a `View`-based UI using the Navigation component. You define a separate fragment for each screen, and the Navigation component decides which fragment to display.

When you use the Compose Navigation component, you no longer need to use fragments. Instead, you define a *composable* for each screen, and the Navigation component decides which one to display.

To use the Compose Navigation component, add the following dependency to the app's *build.gradle* file:

```
implementation("androidx.navigation:navigation-compose:2.4.0-alpha08")
```

Then use code like this for your activity:

```
package com.hfad.myapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.material.Text
import androidx.compose.material.Button
import androidx.compose.material.MaterialTheme
import androidx.compose.material.Surface
import androidx.compose.runtime.Composable
import androidx.navigation.NavController
import androidx.navigation.compose.NavHost
import androidx.navigation.compose.composable
import androidx.navigation.compose.rememberNavController

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MaterialTheme {
                Surface {
                    MainActivityContent()
                }
            }
        }
    }
}
```

The code continues on the next page.

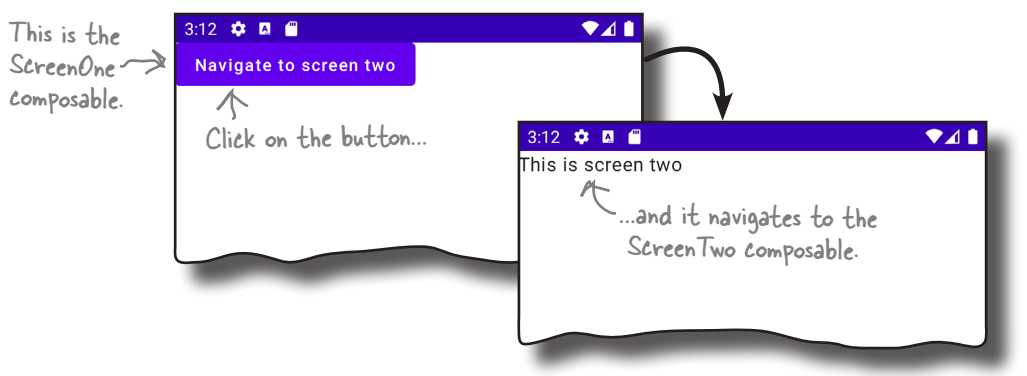
## Compose navigation code continued

```
@Composable
fun ScreenOne(navController: NavController) {
    Button(onClick = { navController.navigate("two") }) {
        Text("Navigate to screen two")
    }
}

@Composable
fun ScreenTwo() {
    Text("This is screen two")
}

@Composable
fun MainActivityContent() {
    val navController = rememberNavController()
    NavHost(navController = navController, startDestination = "one") {
        composable("one") { ScreenOne(navController) }
        composable("two") { ScreenTwo() }
    }
}
```

ScreenOne displays a Button that navigates to ScreenTwo.

ScreenTwo displays a Text.

When the app runs, it looks like this:




You can keep up-to-date with Compose here:

*https://developer.android.com/jetpack/compose*

# 7. Retrofit

In Chapter 14, you learned how to persist your data in a Room database held on the device. But what if your data is held remotely instead?

In this situation, you can use Retrofit. This is a third-party REST client for Android, Java, and Kotlin that lets you make network requests, interact with REST APIs, and download JSON or XML data.

You can find out more about Retrofit here:

*https://square.github.io/retrofit/*

You can also find out more about including Retrofit in your app's architecture by going here:

*https://developer.android.com/jetpack/guide*

# 8. Android Game Development Kit

If you're interested in developing games for Android apps, we suggest you check out the Android Game Development Kit (AGDK). This is a range of libraries and tools that let you develop, optimize, and deliver Android games.

The Android Game Development Kit includes game libraries for C/C++ game development. There's a `GameActivity`, for example, which inherits from `AppCompatActivity` and uses a C API. There's also a way of using the software keyboard in C, and handling input from game controllers.

[Note from Marketing: Make sure you mention your book Head First C.]

You can read more about the Android Game Development Kit here:

*https://developer.android.com/games/agdk*

# 9. CameraX

If you want to use the device camera in your app, there's a Jetpack library you can use called CameraX. This gives you a consistent API that works across most Android devices, and resolves device-compatibility issues. You can use it to preview, capture, and analyze images. There's even an Extensions add-on that lets you access features used by the native camera app that ships with the device.

Read more about CameraX here:

*https://developer.android.com/training/camerax*

# 10. Publishing your app

Once you've developed your app, you'll probably want to make it available to other users. You'll likely want to do this by publishing your app through an app marketplace such as Google Play.

There are two stages to this process: preparing your app for release, and then releasing it.

## Preparing your app for release

Before you can release your app, you need to configure, build, and test a release version of it. This includes tasks such as deciding on an icon for your app, removing any logging code, and modifying *AndroidManifest.xml* so that only devices you want to be able to run your app can download it.

Before you release your app, make sure that you test it on *at least* one tablet and one phone to check that it looks the way you expect and its performance is acceptable.

You can find further details of how to prepare your app for release here:

*http://developer.android.com/tools/publishing/preparing.html*

## Releasing your app

This stage includes publishing your app, selling it, and publicizing it.

Before you publish your app we suggest you go here:

*https://developer.android.com/distribute/best-practices/launch*

It includes checklists and tips to help you publish and manage your app.

For ideas on how to best target your app to your users and build a buzz about it, we suggest you explore the documents here:

*https://developer.android.com/distribute/best-practices/engage*

and here:

*https://developer.android.com/distribute/best-practices/grow*

# Index

## A


abstract class, for database 586

actionMessageFragmentToEncryptFragment() method, Directions 271

actions
- in app bar 317
- connecting fragments with 242, 270
- snackbar 395

activities 2, 3, 28
- in Compose 762, 763–768, 774–777
- default 41
- layout inflation 107
- MainActivity.kt 17–18, 64
- view binding 409–417, 418, 426, 431
- view model 439

Activity class 2, 183

activity lifecycle 169–218, 421
- activity magnets exercise 191–192
- inherited methods 183
- properties and methods 170
  - Chronometer 173–175
  - exercise 200–203
  - onCreate() and onDestroy() 181–185
  - onStart(), onStop(), onRestart() 194
  - summary table 214
- states in 181–190, 193–212
- Stopwatch app 170–172

ActivityMainBinding class 410, 416, 420

activity_main.xml file 13, 14, 28, 41

adapter, recycler view 624, 627–633
- attaching to recycler view 636–637
- creating file 627
- data type property 628
- exercises 646–649, 698–701, 720–723
- view holder 629–632

addTask() method
- TaskDao 595
- TasksFragment 592
- TasksViewModel 600

AGDK (Android Game Development Kit) 873

aligning views 146–148, 155

Android apps. *See also* specific app projects
- activity states 180–182
- building 7–10
- development environment 5–6
- editors 15–19
- folder and file structure 13–14
- interactive 37–80
- minimum SDK for 10–11
- package name for 87
- publishing 874
- refining (editing) 28–35
- running 19–27

Android devices 2
- adapting apps for different sizes of. *See* fragments
- API level on. *See* Android SDK (API level)
- rotation of 178–180, 184–185
- running app on 19

Android Game Development Kit. *See* AGDK

Android Jetpack 125
- AppCompatActivity. *See* AppCompatActivity class
- CardView. *See* CardView
- Compose. *See* Compose
- constraint layout. *See* constraint layout
- constraint layout. *See* constraint layout
- data binding. *See* data binding
- fragments. *See* fragments
- live data. *See* live data
- Navigation. *See* Navigation component
- recycler views. *See* recycler views
- Room. *See* Room persistence library
- view binding. *See* view binding
- view models. *See* view models

AndroidManifest.xml file 14
- language settings 86
- location of 87
- theme in 300, 800

Android platform
  getting started 1–36
  versions of. *See* Android SDK (API level)
Android SDK (API level) 5
  device emulator 20
  libraries for 13
  list of 11
  specifying for AVD 22
Android Sharesheet 863
Android Studio 4–36
  adding Google repository reference 127
  alternatives to 7
  Compose 760
  creating projects 8–10
  extra theme code 801
  fixing hard-coded text 51–53
  installing 5–6
  system requirements 6
Android virtual devices. *See* AVDs
API level. *See* Android SDK (API level)
APK files 25
app bar 295
  attributes in manifest file for 300–302
  default 298
  label for 300
  layout for toolbar animation 366
  removing with theme 303
  replacing with toolbar 303–306, 362
  themes required for 300–301
AppBarConfiguration class 320, 347
AppBarLayout 366
AppCompatActivity class 64
  in activity lifecycle 183
  versus ComponentActivity 763
  Jetpack 125
  setSupportActionBar() method 363
app folder 14
apps. *See* Android apps
areContentsTheSame() method, DiffUtil 678
areItemsTheSame() method, DiffUtil 678
Args class 273
argument constructor and view model factory 470
@array reference 60
array resources, Strings and 59–62
?attr reference 304
automated testing 866–867
AVD Manager 21–24
AVDs (Android virtual devices) 20–24

# B

back behavior, Secret Message app 282–287
backing property. *See* private properties, marking
back stack 284–286
barrierDirection attribute 150
barriers, constraint layout 149–151
base property, Chronometer view 173
Beer Adviser app 38–80
  array resources 59–63
  creating project 40–41
  getBeers() method 74–79
  responding to clicks 63–79
  String resources 51–57
  updating layout 41–49, 68
Beighley, Lynn
  Head First SQL (O'Reilly) 584
bias, views with 137
binding expression exercise 562–564
binding property 416
  ActivityMainBinding 411–412
  fragment 425, 526
_binding property, fragment 425
binding variable, TaskItemBinding 690
bind() method, TaskItemViewHolder 631, 687, 710
Bits and Pizzas app 357–402. *See also* Material design system; view binding
  adapter magnets exercise 646–649
  chip buttons 383–384
  collapsing toolbar 373–378
  exercises 664–667, 720–723
  FAB (floating action button) 385–386
  radio button view 381–382
  responding to clicks 392–399
  scrolling toolbar 360–370
blueprint tool. *See* constraint layout

bottom navigation bar 295, 327–335
activity code for 332, 334
linking to navigation controller 333–334
menu resource file 330
replacing with navigation drawer 337
SentItems Fragment 328–330
as view type 331
BottomNavigationView class 331
buildFeatures section, build.gradle file 523
build folder 14
build.gradle file 127, 239–244
Bundle 184–190
get methods to retrieve values 184
onCreate() method 185
onSaveInstanceState() method 185, 436
put methods to add values 184
saving and retrieving activity state 186–190
Bundle? parameter 227
Button composable 778–779, 839
Button element. *See also* FAB
Button view 43–44
adding to constraint layout 129–132
calling method with data binding 557, 566
CompoundButton 382–383
finishing the Guessing Game 549–556
radio buttons 381
replacing view with composable 820
by syntax 786

# C

CameraX 873
cardCornerRadius attribute, CardView 652
cardElevation attribute, CardView 652
CardView 651–662
adding library to 651
creating 652–653
exercise 663
CatChat app 296–354. *See also* navigation UI
bottom navigation bar 327–335
navigation drawer 336–350
toolbar 298–306
toolbar navigation 307–325
celsius variable, MainActivityContent 781
centering composables 798
centering views, constraint layout 133
chains, constraint layout 154–161
checkableBehavior attribute 341
CheckBox button 382
checkedRadioButtonId property 381, 393
chip, compound button 383–384
chip group, compound button 384
Chronometer view, Stopwatch app 171–177
click response
Beer Adviser app 63–79
Bits and Pizzas app 392–399
Button composable 778
data binding 552
Navigation component 246
recycler view navigation 710–719
Stopwatch app 174
Tasks app 730
code editor 15, 29, 44–46
CollapsingToolbarLayout 373–378, 386
color resource file, toolbar theme 302
Column composable 766, 797–799
@ColumnInfo annotation 580
ComponentActivity class 763, 765
@Composable annotation 765
composable functions 765–766
converter composables 773–795
exercises 771–772, 787–788, 807–808, 845–846
GameFragmentContent 836
replicating views 819–823
composables 763
adding to layouts 817–818
adding to view-based UI 833, 811–860
Button 778–779, 839
Column 766, 797–799
hierarchical tree structure 780
Image 775, 776
Row 766, 799
Text 764–776, 778, 816, 818, 821
TextField 789–795, 838
versus views 779

Compose 756–810
activity code in 763–768
constraint layout library 870
converter composable functions 773–795
exercises 756–757
libraries for 815, 870–872
using with views 811–860
ViewModel library 870
Compose Navigation 871–872
ComposeView 817–818, 826, 834–835
compositions 769, 780
CompoundButton subclass 382–383
configuration changes, view model's handling of 436–437
constraint layout 121–168
adding button 129
adding to layout file 128
aligning views 146–148
barriers 149–151
chains 154–161
code for 136, 152–153, 165–166
Compose library 870
connecting multiple views 144–145
deleting constraints 134–135, 155
exercise 140–143
flows 162–167
Gradle 127
guidelines 147–148
Jetpack 125
positioning views 130–133
single and multiple views 126
views with bias 137
constraint widget tool 135
contentDescription argument, Image composable 775
contentDescription attribute, FrameLayout 110
contentScrim attribute, CollapsingToolbarLayout 375
Context class 183
ContextWrapper class 183
ConvertButton composable function 778–779, 781, 790
converter composables 773–795
Button 778–779, 839
header image 775
TextField 789–795, 838
CoordinatorLayout 365
coroutines, for data access in background 593–595, 742
createChooser() method, Intent 863
create() method, ResultViewModelFactory's override of 472

# D

@Dao annotation, database access 582, 587
DAO (Data Access Object) 582
dark app bar 301
database. *See* Room persistence library
@Database annotation, TaskDatabase class 586
data binding 519–568
calling methods 549–556
code for 528–529, 553–554, 693–694
switching off 855
TasksFragment 571, 599–602, 611–612
using with recycler views 687–697
versus view binding 558–559
data binding variable, setting
gameViewModel 542
resultViewModel 525–526
secretWordDisplay 537
Task 688
viewModel 599
data element
GameFragment layout 536
ResultFragment layout 525
data property
List<Task> 679
recycler view adapter 628, 631, 638, 675
@Delete annotation, database access 584
delete() coroutine, TaskDao 742
delete() method, database access 584
deleteTask() method, EditTaskViewModel 742
density-independent pixels. *See* dp
deriveSecretWordDisplay() method, SecretWordDisplay 488
design editor 15, 29
constraint layout 124
opening 41
XML changes reflected in 48

Design option, composable preview 770
destinations, navigation graph 241, 271
device configuration, StopWatch app 180–183
dialogs 865
DiffUtil.ItemCallback abstract class 678–680
DiffUtil utility class 676–688
  implementing 678–687
  purpose of 676
Directions class, fragment 271–272
distributing apps 874
dp (density-independent pixels) 85
drawable resource 111
DrawerLayout 344–346, 347

# E

EditTaskFragment 708
  code for 748–749
  creating 726
  displaying task ID 735–736
  EditTaskViewModel 741–747
  navigation graph 726–727
  TasksFragment code to navigate to 732–734
  updating layout 745–749
  updating task records with 740–752
EditTaskViewModel 744
EditTaskViewModelFactory 745
EditText, LinearLayout 90, 838
Elvis operator 488
EncryptFragment 221, 259–266
  back behavior 282–287
  code for 262, 274
  creating 261–262
  Directions classes 271–272
  navigating from MessageFragment to 265–266
  passing message from MessageFragment to 270–272
  retrieving value from MessageFragment 273–274
  String argument for 268–269
EncryptMessage 270
enterAlways value, layout_scrollFlags attribute 367
EnterGuess composable function 838
EnterTemperature function, TextField composable 790
entities attribute, @Database annotation 586
@Entity annotation, data class 580
entries attribute, Spinner 60
events
  on-click 65–67
  toolbar response to scroll events 367
exercises
  Be the . . . 105–106, 140–143, 325–326, 215–216, 288, 468, 686
  crosswords 562–564
  Magnets 97, 75–77, 191–192, 278–281, 418, 479–480, 515–516, 533, 646–649, 698–701, 720–723, 845–846
  Mixed Messages 617, 619
  Pool Puzzle 233–234, 399–400, 507–508, 557, 663, 833
  Sharpen your pencil 31–32, 117–118, 57–58, 207–208, 289, 499–500, 548, 589–590
  What's My Purpose? 17–18, 311–312
  Who am I? 606, 618
exportSchema attribute, @Database annotation 586
ExtendedFloatingActionButton 385

# F

FAB (floating action button) 356
  Bits and Pizzas app 385–386
  responding to clicks 392–396
  snackbar 399–400
  view binding exercise 430, 432
fillViewport attribute, ScrollView 116
findNavController() method 248
findViewById() method 66
  Fragment class's lack of 246–247
  TaskItemViewHolder 687
  versus view binding 403–406
FinishGameButton composable function 837
finishGame() method, GameViewModel 550–556, 837
Fireside Chats, view binding versus data binding 559
floating action button. *See* FAB
flow_horizontalStyle attribute 164
flows, constraint layout 162–167
flow_wrapMode attribute 163, 164

focus, activities having 204–213
formatTasks() method 609
Fragment class 226, 246
FragmentContainerView 229
  navigation host 244–245, 314, 728
  OrderFragment display 361
FragmentOrderBinding class 420
FragmentOrderBinding? type 425
fragments 221–256. *See also* specific fragments for apps
  adding navigation host to display 445
  CatChat app 296
  code for 226, 228–230, 249
  creating for Guessing Game app 442
  layouts for 226–228, 236, 371–372
  lifecycle 422–423
  linear layout exercise 233–234
  live data 487–498
  MessageFragment 235–242
  Navigation component 222–223, 238–245
  passing data between 257–292
  updating views in layouts directly 521
  updating views versus view model 484
  view binding 419–429
  view model 439, 515–516
  WelcomeFragment 225–232
FrameLayout 109–114
  height 109
  OrderFragment 360
  order of views in 112
  scroll view 115–119
  view attributes 110
  width 109
fromBundle() method, Args class 273

# G

GameFragment 439
  adding views to 446–450
  code for 461–462, 493–494, 511–512, 553–554, 847
  composables instead of views 834–855
  creating 442
  data binding 535–547
  function in game 520
  GameViewModel 457–465
  live data with 485–498
  passing of gameOver control to GameViewModel 506–510, 511–512
  removing views 850–854
  response to view model property updates 491–493
  restricting properties 501–505
  tasks of 447
GameFragmentContent composable function 836, 837, 839, 844
_gameOver backing property, GameViewModel 506, 549
gameOver property, adding to GameViewModel 506–510
GameViewModel 440
  adding gameOver to 506–510
  code for 458–459, 509–510
  creating 457
  data binding 535–547
  finishGame() method 550–556
  function in game 520
  incorrectGuessesText composable function 844
  linking to fragment 460–465
  live data with 485–492, 496–498, 542
  protecting properties 501–505
  ViewModelProvider 460
gameViewModel data binding variable 542
getAll() method, TaskDao 607–608, 640
getBeers() method, Beer Adviser app 74–79
getBoolean() method, Bundle 184
getInstance() method, database instance 587, 597
getItemCount() method, TaskItemAdapter 628, 679
getLong() method, Bundle 184
get() method, TaskDao 742
Google repository, adding reference to 127
Gradle build system 14, 127
gravity attribute, view 95–96, 100
GridLayoutManager 658–661
group element, navigation drawer 341
GuessButton composable function 839
Guessing Game app 438–442, 484–518, 520–568
  adding button 550–556
  composables instead of views for 812–815
  data binding. *See* data binding
  gameOver property 506–510
  live data. *See* live data

protecting properties 501–505
structure 812–813
view models. *See* view models
guess, MutableState 838
GUI components, inherited from View class 44
guidelines, constraint layout 147–148, 163

# H

Header composable function 776
headerLayout attribute, NavigationView 344
header layout, navigation drawer 343
Head First C (O'Reilly) 873
Head First SQL (O'Reilly) 584
HelpFragment, CatChat app 307
activity navigation to 317
adding to menu 316–319
creating 309–310
navigation graph 313
Help item, CatChat toolbar 307
hint attribute, EditText 90
horizontalAlignment argument, Column composable 798
horizontal barrier, constraint layout 150
horizontal chain, constraint layout 154–155
horizontal constraints 130

# I

icon attribute, menu item 316
icons
floating action button 385
navigation drawer 339, 346
id attribute
menu item 316
view 44
IDE, not using 7
IDs, action 242
image resources
adding 110
collapsing toolbar 375
Image composable 775, 776
ImageView 110, 343, 375
navigation drawer 343
system image for AVD 22
WebP image format 110
InboxFragment, CatChat app 307, 308, 313, 319
incorrect guesses, GameFragment 538
incorrectGuesses property, GameViewModel 487, 843
IncorrectGuessesText composable function 844
incorrect_guesses text view 537, 539
inflateFrom() method
TaskItemBinding 690
TaskItemViewHolder 630
inflate() method
ActivityMainBinding 411
LayoutInflater 227
inflaters, layout 227
inputType attribute, EditText 90
@Insert annotation, database access 583
insert() method, tasks into database 583, 592–594
instrumented tests 866
Intent message 862
Intent Resolver 862
internationalization. *See* localization, String resources
isChecked property 382, 384
isLost() method 488
item elements
menu 316
styles 302

# J

java folder 14
Jetpack. *See* Android Jetpack

# K

Kotlin 2, 4

# L

label attribute
application 300
TextField composable 789

layout_anchor attribute, FAB 386
layout_anchorGravity attribute, FAB 386
layout_behavior attribute, NestedScrollView 368
layout_collapseMode attribute, toolbar 374, 375
layout_constraintTop_toBottomOf attribute 151
layout element
  adding to task_item.xml 688
  data binding 532
  GameFragment 536–541
  ResultFragment 525
layout_gravity attribute
  floating action button 385
  view 100–101
layout_height attribute
  collapsing toolbar 374
  FrameLayout 109
  LinearLayout 85
  view 47, 46, 44
LayoutInflator class 227
layout manager, recycler views 634, 658–661
layout_margin attributes, view 102
layouts 2, 3, 28, 81–120
  activity_main.xml 13, 14, 28, 30, 41
  adding composables to 817–818
  adding views to fragment 446–452
  code for 30, 44–46
  in Compose 762
  constraint. *See* constraint layout
  creating buttons 43–44
  data binding. *See* data binding
  default 41
  editing 41–49
  exercises 31–32, 351–352, 371–372, 379, 401
  for fragments. *See* fragments
  FrameLayout 109–114
  inflation of 107–108
  nested 114, 122–123
  providing alternate 868–869
  removing 825
  Task record display 626
  tasks in creating 82
  view binding 410, 532
  warnings in 50
  XML updating 47–49
layout_scrollFlags attribute, toolbar 367, 374
layout_weight attribute, view 92–94, 139
layout_width attribute
  FrameLayout 109
  LinearLayout 85
  SlidingPaneLayout 869
  view 44, 46, 47
LENGTH options for snackbar 395
libraries. *See also* Room persistence library
  CardView 651
  Compose 870–872
  location in project 13
  Material library 359
  recycler view 625
LinearLayout 41, 43, 45
  attributes 85
  building 84–106
  code for 103
  editable text view 90
  exercise 97
  gravity for view contents 95–96, 100–101
  height of views in 93
  margins for view contents 102
  order of views in 91
  orientation 86
  weight of views in 92–94
List<Task> data property, TaskItemAdapter 628, 679
ListAdapter class 679–680, 686
listener binding 552
live data 483–517
  versus coroutines 593
  GameViewModel 542
  protecting properties 501–505
  recycler view navigation 730–731
  setting up fragments' use of 487–498
LiveData<Boolean> property, EditTaskViewModel 743
LiveData<List<Task>> object 608–609
LiveData<String> version of tasks property 609
LiveData<Task> property, EditTaskViewModel 742–748
LiveData type 501
livesLeft property, GameViewModel
  updating text 492
  updating value property 489, 501

using live data 487
value options 488
lives_left String resource 538, 539
lives text view 537
localization, String resources 51
logcat, viewing 467
Log class 467
Log.i() method 467
Log.wtf() method 467
Lollipop 19


# M


MainActivityContent composable function 774–777, 790–792, 800–801
MainActivity.kt file 13, 14
makeGuess() method
GameFragment 453
view model 839
make() method, snackbar 395
makeText method, toast 394
margins, view 102, 135
Material design system 299. *See also* navigation UI
adding Material library 359
elements applied to 356
OrderFragment. *See* OrderFragment
replacing app bar with toolbar 362
styling Temperature Converter app 796
toolbar 304
MaterialTheme, composables 800
MaterialToolbar class 304
Material You 299
menu. *See* Navigation component
menu attribute
BottomNavigationView 331
NavigationView 344
menu resource file
bottom navigation bar 330
exercise 326–327
navigation drawer 339
toolbar 315–316, 317
menu root element, menu resource file 316
MessageFragment 235–242
code for 272
Directions classes 271–272
EncryptFragment 259, 270–274
MessageFragmentDirections class 271
methods, creating 75–77
Modifiers, Compose 775, 797
multiple composables, arranging 766
multi-screen apps. *See* Navigation component
MutableLiveData<Type> 487
MutableLiveData properties 492, 501–503, 506
MutableState object 781, 789, 790, 838
MVVM (Model-View-ViewModel) architecture 572, 606, 618
My First App 7–36
building 7–15
project 8–11
refining (editing) 28–35
running 19–27


# N


name attribute
FragmentContainerView 229
style 301
NavHostFragment 244, 728
navigate() method, navigation controller 248
_navigateToList backing property, LiveData 743
navigateToList property, LiveData 743
navigateToTask property, TasksViewModel 730–731, 732
Navigation component. *See also* navigation UI
activity code for 245
adding to CatChat project 310
AppBarConfiguration class 320
CatChat app exercise 311–312
code for 286
Compose Navigation 871–872
fragments 222–223, 238–245
ID attribute 316
navigation controller 238
findNavController() 248
fragment navigation 270
linking bottom navigation bar 333–334

linking navigation drawer 347
onCreate() 320
navigation graph 238, 443–444
creating 240–243
EditTaskFragment 726–727
EncryptFragment 263–264
exercises 278–281, 289
onCreate() 320
pop behavior 285–286
SentItemsFragment 329
toolbar 313
XML resources 243, 829, 855
navigation host 238
adding to layout 244
adding to MainActivity layout 445
FragmentContainerView 314
OnClickListener for WelcomeFragment 246
running app with 250–252
updating for Tasks app interactivity 725
navigation drawer 295, 336–350
activity code 345–346, 348–349
adding to main activity 344–346
configuring toolbar icon 347
header 343
highlighting selected item 341
layout 338
linking to navigation controller 347
menu items 339–342
replacing bottom navigation bar with 337
support section 340–342
navigation UI 293–350. *See also* recycler view navigation
activity code 345–346, 348–349
bottom navigation bar 327–335
navigation drawer. *See* navigation drawer
theme 298–302
toolbar. *See* toolbar
NavigationView 344–345
nested layouts 114, 118–119, 122–123
NestedScrollView 368
newCelsius, Mutable State object 790
NewGameButton composable function 820
newTaskName property, TasksViewModel 600
notifications 865
notifyDataSetChanged() method, TaskItemAdapter 628, 675


# O


observeAsState() function, State object 843
observe() method, MutableLiveData property 492
Observer class 492
offset property, Chronometer view 174
onBindViewHolder() method, TaskItemAdapter 631, 679
onCleared() method, view model's overriding of 466
onClick argument, Button composable 778
onClick attribute, adding to button 552
on-click event 65–66
OnClickListener 65–66
adding to Task items 710
floating action button 393, 394
Stopwatch app 174
onCreate() method, activity 66
activity lifecycle 181, 182, 214
AppCompatActivity 64
Bundle 185
Navigation component 320
toolbar 306
view binding 411, 416
onCreateOptionsMenu() method, activity 317, 318, 322
onCreateViewHolder() method, TaskItemAdapter 630
onCreateView() method, fragments 227
with composables 818, 825–826
data binding 526
OnClickListener 246
overriding 226–227
TasksFragment 636
view model properties 473
onDestroy() method, activity lifecycle 181, 182, 214
onDestroyView() method, fragments 422
onOptionsItemSelected() method, activity 318
onPause() method, activity lifecycle 204–213, 214
onRestart() method, activity lifecycle 194, 196, 197, 214
onResume() method, activity lifecycle 204–213, 214
onSaveInstanceState() method
activities 185, 436
fragments 436
onStart() method, activity lifecycle 193, 194, 196–199, 214

onStop() method, activity lifecycle 193, 194, 195–199, 214
onTaskClicked() method, TasksViewModel 730, 732
onTaskNavigated() method, TasksViewModel 730
onValueChange attribute, TextField composable 789
OrderFragment 357, 360–370
  adding toolbar 362
  building layout 387–390
  code for 388–390, 397–398, 427–428
  collapsing toolbar 373–378
  creating 360
  radio button views to make choices 381–392
  scrolling toolbar 364–370
  snackbar 395–396
  toast pop-up to respond to click 392–394
orientation attribute, LinearLayout 45

# P

package name 87
packed chain style, constraint layout 159
padding, applying to composables 797
padding attribute, LinearLayout 88–89
paddingEnd attribute, LinearLayout 88
paddingStart attribute, LinearLayout 88
paddingVertical attribute, LinearLayout 88
painter argument, Image composable 775
parent attribute, style 301
paused activity 204–213
pop behavior, navigation graph 285
pop-up messages, interactive 356
@Preview annotation, composables 769
PreviewMainActivity composable function 769–772, 774
@PrimaryKey annotation, database table 580
private properties, marking 501–503
projects. *See also* specific app projects
  adding Google repository reference 127
  creating 8–10, 40–41
  files in 12–14, 34
properties
  Chronometer 173
  MutableLiveData 492, 501–503, 506
  protecting view model 501–505
  view model 487–494
protecting view model properties 501–505
publishing apps 874
putBoolean() method 184
putLong method 184

# Q

@Query annotation, database access 584

# R

RadioButton 381
RadioGroup 381
RecyclerView.Adapter 679
recycler view navigation 705–754
  click response 710–719
  navigating to new fragment 724–737
  updating Task records with EditTaskFragment 740–752
recycler views 621–670. *See also* recycler view navigation
  adapters 624, 627–633
  adding library 625
  card view 651–662
  DiffUtil 676–688
  exercise 646–649
  flexibility of 650
  versus formatted String 624
  review of process 672–675
  task list display 626–645
  TasksFragment display 634–645
releasing apps 874
remember, storing variable values in Compose 781
res folder 14
resource files 2
  array resources 59–62
  folders for types of 13–14
  image resources. *See* image resources
  menu resources. *See* menu resource file
  String resources. *See* String resources
resources element 55
responding to click. *See* click response

ResultFragment 439
adding views to 451–452
code for 474–475, 528–529, 827–829
composables instead of views 816–833
creating 442
function in game 520
in gameOver with live data 506
linking ResultViewModel to 470–475
removing views 825
updating text in layout 524–531
view model for 469–478
ResultFragmentContent composable function 819–820
result property, data binding approach 524, 527
ResultText composable function 821
ResultViewModel 440, 469–478
function in game 520
live data 495
ResultViewModelFactory class 472
resultViewModel property, data binding variable 525–529
Retrofit 873
reversed() method, String 273
RoomDatabase class 586–588
Room persistence library 569–620
accessing Task records 741
app setup 573–575
database 578–590
getting records 607–616
inserting records 591–606
SQLite 572
rotation of device
configuration changed by 178–180, 455
saving activity state for 184–191
Row composable 766, 799
Run command 24–27
running property, Chronometer view 174

# S

Safe Args plug-in 259
Args class 273
Bits and Pizzas app 725
Directions class 271
exercises 288
Guessing Game app 441
safe 290
Secret Message app 267–277
saveOffset() method, Chronometer view 175
scalable pixels. *See* sp
scaleType attribute, FrameLayout 110
scheduling deferrable tasks 864
screens. *See also* fragments
choosing best layout for 83
layouts for. *See* layouts
supporting different sizes 180–183, 868–869
scrolling toolbar, CoordinatorLayout 364–370
scroll value, layout_scrollFlags attribute 367
ScrollView 115–119
SDK. *See* Android SDK (API level)
Secret Message app 220–256, 258–292
back behavior 282–287
EncryptFragment. *See* EncryptFragment
fragments and navigation. *See* Navigation component
MessageFragment 235–241
Safe Args plug-in 267–277
WelcomeFragment 225–232
secretWordDisplay property, GameViewModel 487, 537
SentItemsFragment 329, 328–330
setAction() method, snackbar 395
setBaseTime() method, Chronometer view 175
setContent() function
ComponentActivity 763, 765
ComposeView 818
setContentView() method, activity 64
binding object 411
converting view to object 107
fragments and navigation 227
overriding 181
setOnClickListener() method, activity 66, 710, 711
setSupportActionBar() method, toolbar 306, 363
sharing data with other apps 862–863
showAsAction attribute, toolbar 316
SlidingPaneLayout 869
snackbar, popup message 356, 392
actions 395–396
exercises 399–401

spanCount attribute, GridLayoutManager 658
Spinner 47, 59, 60, 67
Split option, composable preview 770–771
spread chain style, constraint layout 159
spread inside chain style, constraint layout 159
sp (scalable pixels) 85
SQLite database 572, 578–590
  annotated interface 582–585
  inserting records 591–606
  tables 579–585, 589–590
  TaskDatabase abstract class 586–588
src attribute, FrameLayout 110
src folder 14
StaggeredGridLayoutManager 658
startButton property, ActivityMainBinding 410
start() method, Chronometer view 173
State objects, composables and live data 843
stop() method, Chronometer view 173
Stopwatch app. *See also* activity lifecycle
  Chronometer view 171–177
  device configuration 180–183
  managing state changes 193–212
  saving activity state 184–191
  view binding 408–417
stopwatch property, Chronometer view 174
string-array element 59
string element 55
String, formatted versus recycler view 624
@string reference 53
stringResource() function, Text composable 844
String resources 14, 51
  adding and using 54
  Compose 762
  extracting 52–53
  for GameFragment's extra text 538–541
  random number exercise 57–58
  Secret Message app 224
  Stopwatch app 171
  strings.xml 52, 55
  text display exercise 548
strings.xml file 52, 55
style attribute, chip 383
style element 301
styles, defining in style resource files 301–302
submenu, creating 340
submitList() method, TaskItemAdapter 681
support section, navigation drawer 340–342
supportsRtl property, AndroidManifest.xml 86
Surface composables 800
suspend, marking TaskDao's data access methods with 594–596
Switch button 382
SystemClock.elapsedRealtime(), Chronometer view 173
system image, specifiying for AVD 22

# T

table name property 580
TaskDao interface 582–585
  accessing methods for Task record editing 741
  code for 594
  coroutines 742
  getAll() method 607–608, 640
  get() method 742
  taskDao property 587
  TasksViewModel 591
  TasksViewModelFactory 597
taskDao property, TaskDatabase 587, 597
TaskDatabase abstract class 586–588
Task data class 579–581
TaskDiffItemCallback class 677, 678, 681–682
taskDone property, database column 580
taskId property, as primary key 580
TaskItemAdapter
  adding data to layout view 631
  CardView layout update 654–656
  code for 655–656, 691–692, 713
  connecting to data source 638–642
  creating 627
  data type property 628, 675
  extending ListAdapter 679–680
  passing of Toast to TaskItemViewHolder via 711–717
  passing Task records to 681–682

summary 633
telling recycler view to use 636–637
updating with DiffUtil 677
view holder 629–630
TaskItemBinding class 688–692
TaskItemViewHolder inner class 629, 630
bind() method 631
CardView layout update 654–656
data binding 687, 689–692
OnClickListener 710
passing of Toast to 711–717
task_item.xml file 653, 693
taskName property, database column 580, 631
Task records
in database 741
layout file 626
passing to TaskItemAdapter 681–682
recycler view 626–645
updating with EditTaskFragment 740–752
Tasks app 570–620, 622–670, 672–704, 706–754
adapters and task list 627–645
database for 578–590
data binding 689–697
DiffUtil 676–688
getting records 607–616
grid of cards 650–662
inserting records 591–606
setup 573–575
structure 571–573, 707
view models. *See* view models
TasksFragment 571
activities 577
code for 601–602, 613, 635, 641, 733
creating 575–577
data binding 576, 599–602
displaying records 607–616
EditTaskFragment navigation 726, 732–734, 743
ListAdapter update 682
passing Toast code in lambda 711–717
recycler view display 634–645
tasksString property 611–612
TasksViewModel 592, 596–597
TasksFragmentDirections class 729
tasks property, TasksViewModel 608, 638
tasks, scheduling deferrable 864
tasksString property 609
TasksViewModel 571, 591–600
connecting tasks list to TaskItemAdapter 638–640
creating 592, 596–597
getting tasks 608
inserting records 600
launching TaskDao insert() as coroutine 595
passing task ID to EditTaskFragment 730–732
tasks and tasksString property additions 610
TasksViewModelFactory class 596–598
Task type, data binding variable 688, 690
Temperature Converter app 756, 773
code for 803–805
creating and configuring project 760–761
modifying appearance 796–806
TemperatureConverterTheme 801–802
TemperatureText composable function 777
ConvertButton 778
newCelsius object 790
updating text displayed 779
updating value of argument 781
testing, automated 866–867
text attribute
Button 44, 53
TextView 30, 68
Text composable 764–776
for Button 778
replacing TextView with 816, 818, 821
TextField composable 789–795
activity code for 791–792
attributes for 789
EnterGuess function 838
EnterTemperature function 790
text variable, TextField composable 789
TextView 30
Chronometer view 173
code for 30–31, 44–47
editing text in 68
LinearLayout 41, 46
recycler view adapter 629
replacing with Text composable 816, 821, 843
theme attribute, navigation 300, 366
Theme.kt 801–802

themes
applying to composables 800–802
applying to project 298–302
color resource file 302
removing default app bar using 303
title attribute, menu item 316
toast, popup message 392–394, 711
ToggleButton 382
toolbar
code for 305–306, 321
collapsing 373–378, 386
creating 298–306
navigation drawer icon 347
replacing app bar with 303–310, 362
scrolling 364–370
using to navigate 307–325
configuring to show Help 319–320
HelpFragment 309–310
Help item for 316–318
InboxFragment 308
menu resource file 315–316
navigation graph 313
navigation host 314
Transformations.map() method 609

# U

unit tests 866
@Update annotation, database access 583
update() coroutine, TaskDao 742
update() method, database access 583
updateScreen() method, GameFragment 453
updateTask() method, EditTaskViewModel 742
Up navigation versus Back navigation 319
USB debugging 19

# V

val and var, for live data properties 489
value attribute, TextField composable 789
value property, MutableLiveData 488–492
making private 501
as nullable 488
updating as val not var 489
value property, MutableState 781
variable element, ResultFragment layout 525
version attribute, @Database annotation 586
vertical chain, constraint layout 161
vertical constraints 131–132
view binding 407–429
activities 409–417
code for 413–415, 427–428
versus data binding 532, 558–559
fragments 419–429
Guessing Game app 441
stopping for code with composables 825
switching off when data binding will do 560
View class 44. *See also* views
ViewGroup? parameter 227
ViewGroup subclass 113
view holder, recycler view adapter 629–632
viewLifecycleOwner argument 492
ViewModel class 457, 460, 466, 469
view model factory 471–475, 596–598, 745
viewModelFactory property 473
ViewModel library, Compose 870
viewModel property 460, 473, 599
ViewModelProvider class 460–462, 471, 472
view models 435–478
data binding to get values from 521–522
EditTaskViewModel 741–747
exercises 468
GameFragment 457–465
Guessing Game app 438–442
live data with 485–487
reasons for using 437, 456
ResultFragment 469–478
TasksViewModel 591–600
views. *See also* view binding
aligning 146–148, 155
bottom navigation bar as type of view 331
versus composables 779
connecting multiple views 144
constraint layout. *See* constraint layout
CoordinatorLayout 365

data binding 525
editable text 90
findViewById() method 66, 403–406
FrameLayout 109–114
GameFragment 446–450
height 100
hierarchy of views 123
ID 44
LinearLayout 84–86, 90–96, 100–102
navigation drawer 338
NestedScrollView 368
OrderFragment's content 381–392
removing from Guessing Game app 825, 850–854
ResultFragment 451–452
scroll view 115–119
toolbar as type of view 304

# W

WebP image format 110
WelcomeFragment 221
  activity code for 249
  adding to project 225–232
  back behavior to return to 283–287
  OnClickListener 246
won_lost ID, ResultFragment 524
won_lost text view, ResultFragment 527
WorkManager API 864

O'REILLY®

Learn from experts.
Become one yourself.

Books | Live online courses
Instant Answers | Virtual events
Videos | Interactive learning

Get started at oreilly.com.

©2022 O'Reilly Media, Inc. O'Reilly is a registered trademark of O'Reilly Media, Inc. 175.8 x 9.25